ABSTRACT

JOHNSON, BRITTANY ITELIA. A Tool (Mis)communication Theory and Adaptive Approach for Supporting Developer Tool Use. (Under the direction of Emerson Murphy-Hill and Sarah Heckman.)

Software development and maintenance is an important part of the software engineering process. It is during these phases of the software engineering process that developers complete actions to ensure quality software, such as finding and resolving software defects and refactoring. Many of the actions involved in writing and maintaining software are manual, tedious and, as a result, error-prone. Program analysis tools, including those that find and resolve defects, provide feedback to and automate tasks for developers. Research has found, however, that developers use these tools infrequently in practice. Multiple studies have explored developers' usage of program analysis tools and cumulatively found that developers encounter various challenges when attempting to understand and resolve problems in their code.

The goal of this research is to improve the communication between developers and static analysis tools by providing theories and approaches that identify and utilize the differences between developers' information needs, based on their knowledge, that influence developers' ability to resolve tool notifications. To achieve this goal, I explored developer tool usage and information needs when understanding and resolving notifications. In this thesis, I propose a theory that developers have difficulty with tool output because of gaps and mismatches between how much developers know and how tools communicate. I evaluate this theory by building on other existing research to determine factors that influence developer knowledge and information needs of developers, based on their knowledge, when resolving defects.

My thesis consists of four consecutive studies. I first conducted a study to explore the reasons developers have for using and not using static analysis tools. I categorized interview responses from professional developers about their usage of tools and identified five reasons developers have for not frequently using the tools available to them. The results suggest that one major barrier to use is the ability for developers to understand the textual and visual notifications provided by tools. The second study took a deeper look at the difficulties developers have when understanding tool

notifications by observing developers as they attempt to explain textual and visual notifications presented by various tools. Based on the data collected and analyzed, I proposed a theory of tool communication based on the 10 kinds of challenges identified in the data. All of the challenges identified in this study stemmed from gaps in developers' knowledge that notifications did not fill and mismatches between knowledge participants had accrued through their experiences and the information provided by tools.

After determining an explanation for developer difficulties with tool notifications, the next study evaluated the possibility of operationalizing and evaluating that theory. The theory, and previous research, suggests software development experiences influence developer knowledge. Therefore, the third study used developer source code contributions, one of the primary experiences for developers, and concept inventories, a validated educational assessment, to classify developers based on their knowledge of programming concepts. The models created from this study classified developers' knowledge of variables, exception handling, and generics with 60–80% accuracy. Results from this study support the possibility for tools to automatically ascertain how much developers know about the defects they encounter in their code using the source code they have written.

Finally, because it is possible to classify developer knowledge using their code contributions, the fourth study evaluated the effectiveness of using developer knowledge classification to customize the information presented to developers by program analysis tools. I used existing research on problem solving to adapt notifications communicating about defects pertaining to the concepts of variables, exception handling, and generics. I presented developers classified as novices and experts in the concepts of interest with notifications aligned and misaligned with their knowledge classification to compare performance and preference. I found that most often, it took participants less time to resolve notifications aligned with their knowledge classification; there was a significant difference between the time it took novices and experts in a given concept to resolve aligned versus misaligned notifications. I also found that on average, novices in a given concept presented with aligned notifications made significantly fewer attempts at resolving notifications than those presented with misaligned notifications, and that developers tended to prefer information provided in aligned notifications. For experts, aligned notifications led to a higher percentage of resolved notifications.

Although not all differences in observations were statistically significant, results from this study suggests that presenting information in tool notifications based on developer software development experiences can improve communication between developers and their tools. I conclude this thesis with a discussion of future research directions that will stem from this research.

A Tool (Mis)communication Theory and Adaptive Approach for Supporting Developer Tool Use

by
Brittany Itelia Johnson

A dissertation submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Doctor of Philosophy

Computer Science

Raleigh, North Carolina

2017

APPROVED BY:

Emerson Murphy-Hill Co-chair of Advisory Committee	Sarah Heckman Co-chair of Advisory Committee
Tim Menzies	Tiffany Barnes

Eric Rotenberg

DEDICATION

To my parents, Engadine and Valerie, who made me the strong black woman I am.

To my sister, Chelsea, who continues to motivate and inspire me through everything.

To the love of my life, Anthony, who has been supportive and by side the whole ride.

To my first mentor, Dr. James Bowring, who helped me find my passion and urged me to pursue it.

BIOGRAPHY

Brittany Itelia Johnson was born in a small town called Sumter, South Carolina to Engadine and Valerie Johnson on November 25, 1988. She graduated from Sumter High School in 2007. During her time in high school, Brittany was active in the marching and jazz bands. After high school, she continued on to pursue her undergraduate degree at the College of Charleston (CofC), where she studied Computer Science. While at CofC, Brittany was a member of SCAMP, an organization focused on increasing minority participation in STEM research. This led to her participation in undergraduate research, which informed her passion to pursue a Ph.D. She was also a member of the Pep Band, where she let off some academic steam, and an officer in the CofC Chapter of the National Society of Collegiate Scholars. Brittany obtained her Bachelor of Arts in Computer Science in Spring 2011. She was selected as the feature student graduating from College of Charleston with the Class of 2011 for her involvement and performance in academics, research, and extra-curricular activities. Following her time at CofC, Brittany began her journey to her Ph.D. at NC State University in Fall 2011 under the direction of Dr. Emerson Murphy-Hill. As she began her Ph.D. studies, she accrued a co-advisor, Dr. Sarah Heckman. During her time at NC State, aside from her research, she participated in numerous outreach and mentoring initiatives as she discovered her passion for mentoring others like her. Brittany aspires to have a career in academia, where she can incorporate both her passion for research and her passion for mentoring.

ACKNOWLEDGEMENTS

I would like to thank my advisors, my committee, the Developer Liberation Front, RealSearch, and AltCode for their help.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER

1

INTRODUCTION

Software implementation and maintenance are important parts of the software development process [**krishnan2000empirical**]. It is during these phases that developers complete actions to ensure quality software, such finding and resolving software defects and refactoring. Many of the actions involved in writing and maintaining software are manual, tedious and, as a result, error-prone. Tools, known as *program analysis tools*, exist for developers to automate some of theses tasks and reduce the effort involved in tasks like defect finding and resolution [**bruckhaus1996impact**]. Some program analysis tools, known as static analysis tools, allow developers to find and resolve defects early in the development process and occasionally to find defects previously overlooked [**ayewah2010google**; **Ayewah:2008:FindBugs**].

Research has found, however, that developers use these tools infrequently in practice [**Ayewah:2008:FindBugs**; **ge2012reconciling**; **smith2015questions**]. The research in this dissertation provides direction for tool developers and designers by exploring developer tool use and providing insights into the potential improvements we can make to developer-tool interactions. To provide concrete motivation for this research, consider Valerie. Valerie is a hypothetical software developer at a start-up company. She primarily writes Java code, though she did not learn to program in Java, and uses the Eclipse Integrated Development Environment (IDE). In her spare time, she builds her knowledge of Java programming concepts by contributing to open source software. While contributing to the Sun

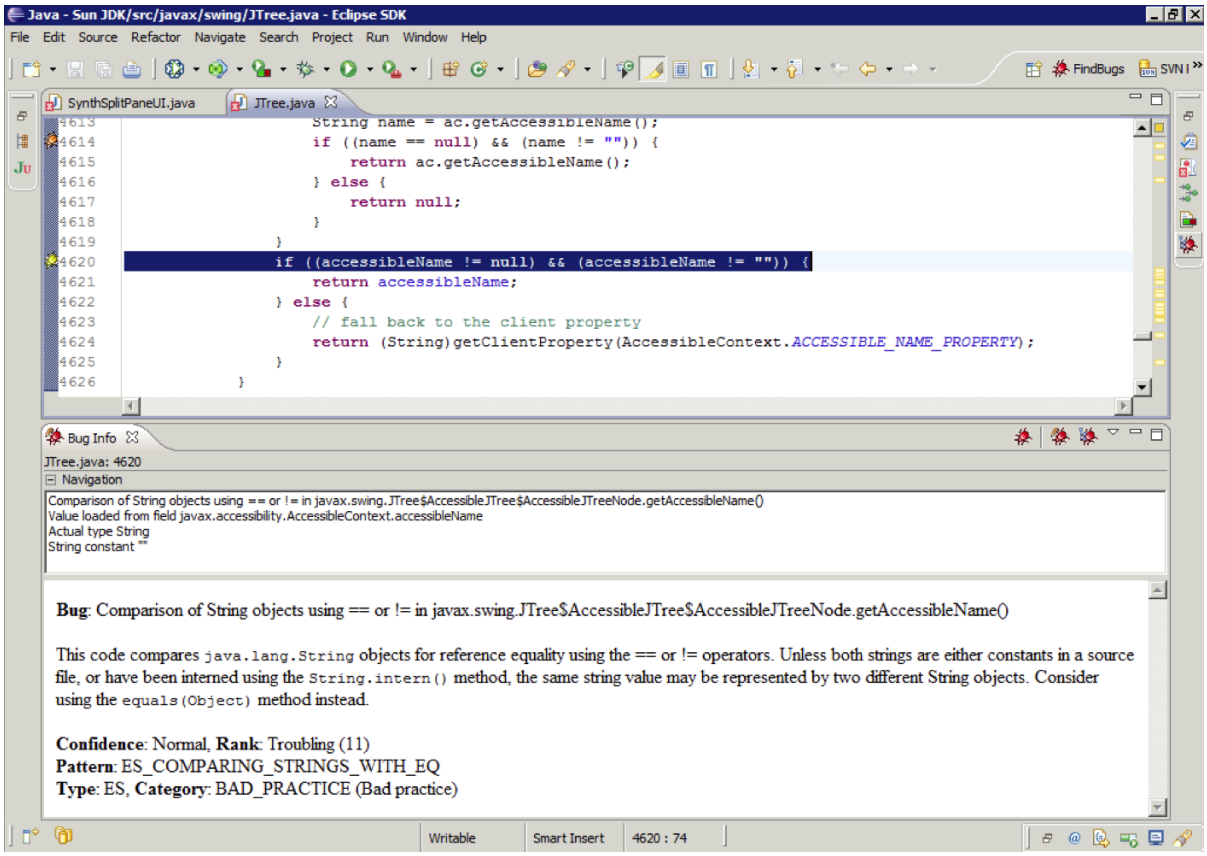


Figure 1.1 Findbugs notification in the Eclipse IDE on checking string equality.

JDK, a fellow developer suggested she install and use FindBugs. Although she found herself slightly overwhelmed by the volume of output provided, she decided to narrow in on the more important defects in her current file.

Although Valerie's goal when using tools like FindBugs is to find and resolve defects, which requires the ability to interpret the notifications provided by the tools, a secondary goal is to learn more about Java programming concepts. The first few defects Valerie encountered she was able to understand and resolve. However, she quickly realized that some notifications are better at communicating problems while contributing to knowledge than others.

For example, when Valerie encountered the notification in Figure 1.1, she was able to understand and resolve the notification. Along with her existing knowledge, the notification in Figure 1.1 informed her of *why* what she was doing was wrong and *how* she can fix it. One the other hand, when Valerie comes across the notification in Figure 1.2, she realizes that despite her experience

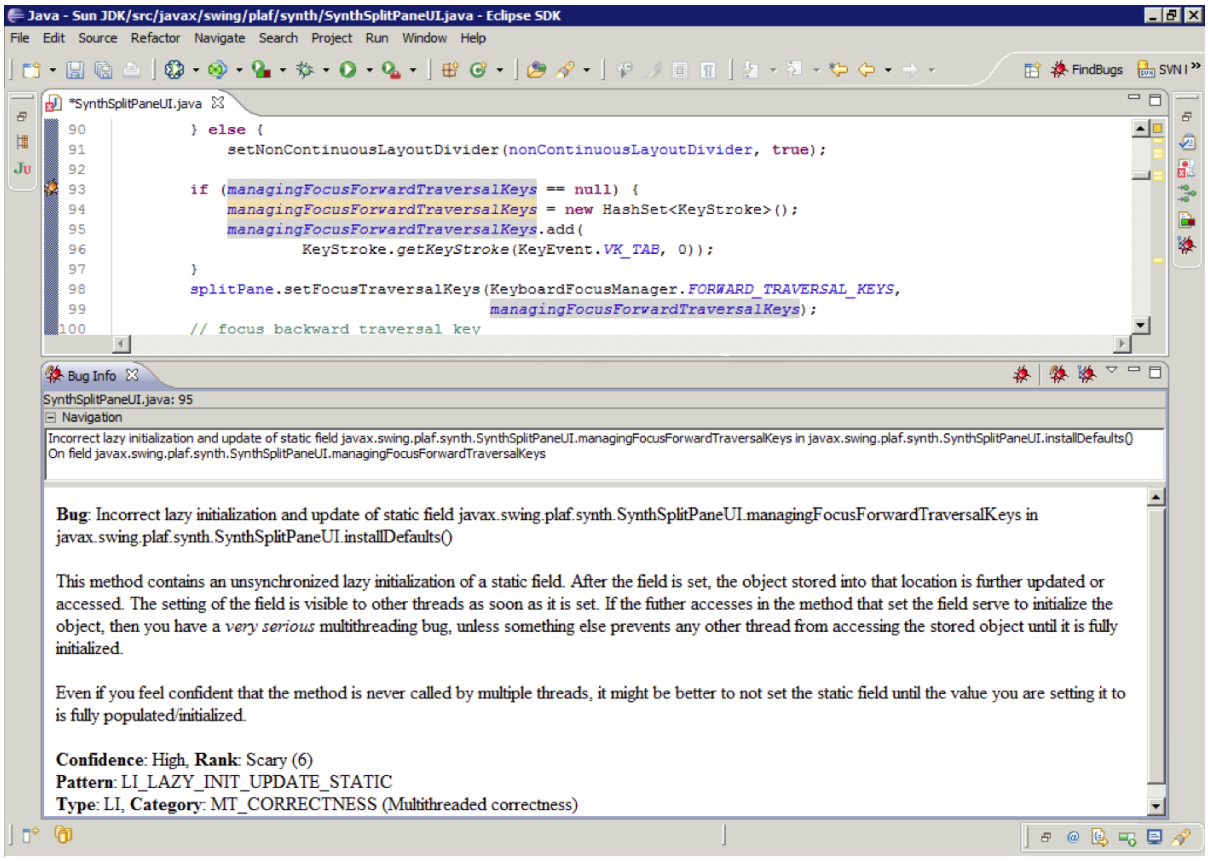


Figure 1.2 Findbugs notification in the Eclipse IDE concerning multi-threading.

with FindBugs, she is having difficulty determining how to resolve the notification. From previous bugs, and some research on the web, she knows that an orange bug icon indicates a *scary* bug and that by clicking the bug icon she can gain access to more information about the bug.

She first attempts to use what knowledge she does have regarding multi-threading, which she accrued from struggling with and resolving compiler synchronization warnings, to better understand the problem. However, she is unfamiliar with the concept central to the notification in Figure 1.2 (lazy initialization). Though the notification tells her that the problem relates to multi-threading, she is unable to make a connection between her knowledge regarding multi-threading and the message FindBugs is attempting to communicate and therefore cannot resolve the notification without outside help. As done previously with compiler synchronization notifications, she toggles between the web and her IDE to understand and resolve the notification.

Because the tools Valerie uses have no notion of what she does and does not know, some

notifications communicate in a way that she is able to understand the problem, while others are not, leading to miscommunication. *The goal of this research is to improve how program analysis tools communicate to developers by providing theories and approaches that identify and utilize the differences between developer information needs, based on their knowledge, that influence their ability to resolve tool notifications.*

In the following chapters, I will discuss the research I conducted to explore challenges like those encountered by Valerie and research for developing techniques, frameworks, and tools to mitigate these kinds of challenges. Chapter 2 grounds my research in existing research in Computer Science and other fields. Study 1, outlined in Chapter 3, answers the following research questions:

RQ_1 : *What reasons do developers have for using or not using static analysis tools to find bugs?*

RQ_2 : *How well do current static analysis tools fit into the workflows of developers? We define a workflow as the steps a developer takes when writing, inspecting and modifying their code.*

RQ_3 : *What improvements do developers want to see being made to static analysis tools?*

Based on findings from Study 1, which identified tool notification understandability as a barrier to use, Chapter 4 answers the research question *why do developers encounter challenges when interpreting tool notifications?*. I used the findings from Study 2 to formulate the boxed explanatory theory above regarding how knowledge affects developers' ability to interpret and resolve notifications.

Based on these two initial studies, I formulated and evaluated the following thesis:

> Program analysis *tool use* is a form of *communication* and *inability to interpret and resolve* notifications is a result of *miscommunication* caused by *knowledge gaps* and *knowledge mismatches*; therefore we can improve tool design by providing the means for tools to *classify* individual developer's *conceptual knowledge* and adapt notifications accordingly, leading to the potential for *reducing time* required for developers to resolve tool notifications, *increasing ability to resolve*, and *decreasing attempts made when resolving*.

Based on the theory proposed in Chapter 4, Chapters 5 and 6 present Study 3, which answers the following research questions:

RQ_1 : *Is source code a good predictor of how much developers know about programming concepts?*

RQ_2 : *Does concept-specific source code increase the ability to classify how much developers know about programming concepts in comparison to a naive model?*

In Chapter 5, I outline an approach used to assess developer concept knowledge in preparation for the work done in Chapter 6 to predict developer concept knowledge. Finally, based the ability to

predict conceptual knowledge, I posit and evaluate the following hypotheses and research question in Chapter 7:

H_1 : *Knowledge-based notifications can decrease the time required for notification resolution.*

H_2 : *Knowledge-based notifications increase developer likelihood of resolving notifications.*

H_3 Knowledge-based notifications decrease developer code churn when resolving notifications.

RQ : *Do developer adaptation preferences match expectations from existing problem solving literature?*

Findings from Study 4 suggest the possibility of improving developers' ability to quickly resolve notifications. Though the differences observed in this study are small, it is a first step in the direction of better understanding how to improve tool use for developers. To conclude this dissertation, Chapter 8 outlines possible directions for future work that builds the foundations set by this research in this thesis.

The contributions of this dissertation are as follows:

- A categorized list of reasons developers have for not using program analysis tools, accompanied by tool design suggestions provided by developers.
- An explanatory theory for the challenges that developers encounter when interpreting information provided by tool notifications.
- An approach for assessing developer depth of programming concept knowledge that builds on an existing validated approach for assessing breadth of knowledge.
- An approach, developed using existing problem solving research, for adapting tool notifications based on developer concept knowledge.

CHAPTER

2

RELATED WORK

2.1 Program Analysis Tools

Program analysis tools are designed to aid developers when developing software by automating the writing, analysis, and modification of source code. Often, program analysis is discussed as synonymous with static analysis [**nielson2015principles**]. For the purpose of my research, I define a program analysis tool as *a recommendation system that performs program analysis, whether it be static or dynamic analysis, and provides information to developers regarding the source code being analyzed* [**robillard2014recommendation**]. Examples of program analysis tools include, but are not limited to, static code analyzers, code coverage tools, code smell detectors, and refactoring tools [**adolph2011using**; **Murphy-Hill:2010:Ambient**; **ge2012reconciling**]. Program analysis tools can be used in integrated development environments (IDEs) as well in most text editors that can be used for programming, such as Vim [1] or Emacs [2]. In the following sections, I will define and discuss static analysis and dynamic analysis tools separately; the reader should note that although I discuss static and dynamic analysis separately, it is not uncommon to find program analysis tools that combine static and dynamic analysis [**ernst2003static**].

[1] `http://www.vim.org/`
[2] `https://www.gnu.org/software/emacs/`

2.1.1 Static Analysis Tools

Static analysis tools are designed to aid developers when developing software by statically analyzing source code, pre-runtime, and providing the developer with feedback about the state of their code [**ernst2003static**]. Typically, static analysis works by examining the current state of the program, predicting how the program may react in that state at runtime, and reporting any information they deem necessary to the developer. Static analyses are often more conservative than dynamic analyses; this is to reduce the potential for false positives, as in most cases static analysis cannot say with 100% certainty what will happen during run-time [**ernst2003static**]. Examples of static analysis tools include defect detectors, such as FindBugs, compilers, code smell detectors, and refactoring tools.

Let's use the example of FindBugs,[3] an open source static analysis tool, to better understand how static analysis tools work. FindBugs statically analyzes code to report potential defects. FindBugs determines the potential for defects using *bug patterns*. Bug patterns are code idioms that map to errors, found in Java *bytecode*. Bytecode, in Java, represents the compiled Java class files. Because FindBugs analyzes code without executing it, there is a heightened risk for *false positives*. False positives are defects detected that will never manifest during run-time. When FindBugs finds a potential defect, it alerts the developer using notifications that provide information regarding the defect. I will discuss tool notifications in more detail in Section 2.1.3.

2.1.2 Dynamic Analysis Tools

Dynamic analysis tools are designed to aide developers when developing software by analyzing source code during run-time and providing the developer with feedback about runtime behavior [**ernst2003static**]. Dynamic analysis works by executing the program and then making observations about program execution; because dynamic analysis runs the code, it is typically more precise than static analysis. Though dynamic analysis can produce more precise results in a similar amount of time as static analysis, dynamic analysis execution is less likely to generalize to future executions since it is based on a set of inputs that can, and probably will, change for each execution. Examples of dynamic analysis tools include testing, code coverage, and profiling tools.

Let's use the example of Cobertura,[4] an open source dynamic code coverage tool, to better understand how dynamic analysis tools work. Cobertura executes source code using JUnit test cases and reports to the user what parts of the code got covered during execution and what parts did not. Because Cobertura executes the source code, it can communicate precisely regarding the flow of

[3] `http://findbugs.sourceforge.net/factSheet.html`
[4] `http://cobertura.github.io/cobertura/`

the program during run-time. A static code coverage tool could speculate how much of a code base would be covered based on test cases, and possibly even a set of inputs; however, it would require more effort and be more likely to produce false positives than a dynamic code coverage tool. On the down side, the test suites a developer writes may not be characteristic of all possible executions of the program, thereby lowering the generalizability of dynamic analyses.

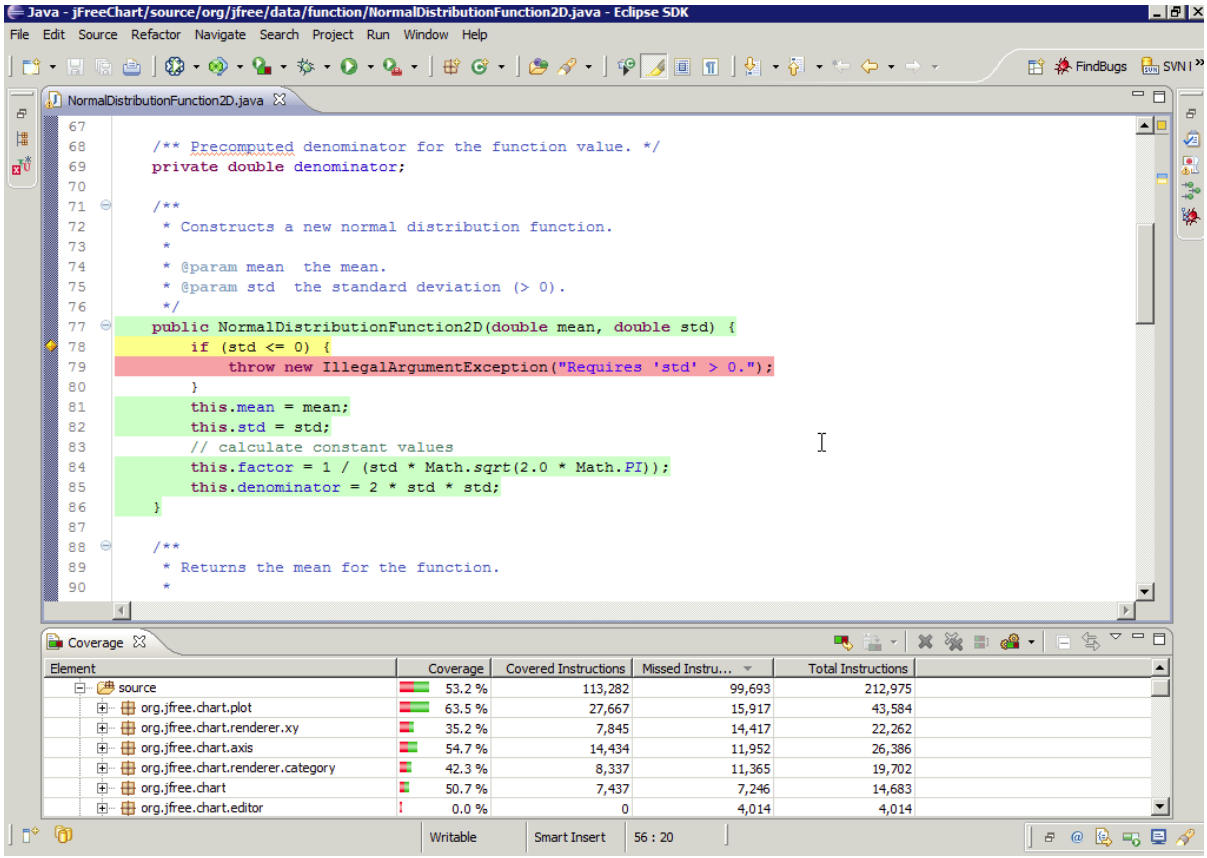


Figure 2.1 EclEmma notifications in the Eclipse IDE.

2.1.3 Communication via Notifications

One common thread between program analysis tools like FindBugs and Cobertura is that they use *notifications* to communicate with the developer. Figure 1.2 and Figure 2.1 provide examples of tool notifications. A notification, when speaking in terms of program analysis tools, is typically a

combination of visuals and text used to communicate a message to its user; for program analysis tools, the user is the developer. Text editors like Vim and Emacs rarely include any visual components, however, because most IDEs include text editors and text-based notifications. Therefore research on notifications in IDEs can apply to notifications developers receive in both text editors and IDEs. Therefore, I focus my research on notifications inside IDEs.

Notifications across tools vary; some provide lots of text (like FindBugs) with few visual aides, some use primarily visual means of communications (like Cobertura). Notifications can also have different goals, which may influence how developers design notifications. For example, the goal of a notification from Coverity, [5] another static analysis tool, is to explain a potential defect in the developer's source code and, ideally, help the developer make a decision about the defect (i.e. whether and how to resolve). Because Coverity's goal is to explain, we expect to see textual notifications that provide that explanation. The goal of code coverage notifications is to statically show dynamic program behavior and help the developer determine the effectiveness of her test suite. EclEmma, another code coverage tool, uses colors applied directly to the source code to communicate as opposed to text. Though EclEmma also uses text to communicate code coverage (i.e. `1 of 2 branches missed` on a partially covered `if` statement), this does not allow the developer to scan the program for areas in most need of attention. Therefore, EclEmma uses other visuals, such as the bar visuals in the Coverage View (Figure 2.1) to show coverage on a given package or class.

The list of tools and notifications tools use can go on and on, but for the purposes of my research, the general definition I will use for a notification is *a combination of visual and textual interfaces used by a program analysis tool to communicate information to developers about their source code.* Notifications can vary regarding what information and how much detail they provide, however, there are commonalities across tool notifications that informed this definition, which I will discuss next.

2.1.4 Typical Notification Components

One reason I talk about program analysis tools as a type of recommendation system is because they provide information to developers completing software engineering tasks [**robillard2014recommendation**]. Another reason is that program analysis tools use the same strategies defined by Robillard as typical of recommendation systems: 1) strategies for getting the user's attention and 2) descriptive interfaces. Program analysis tools use these strategies when communicating with developers via notifications.

[5] `http://www.coverity.com/`

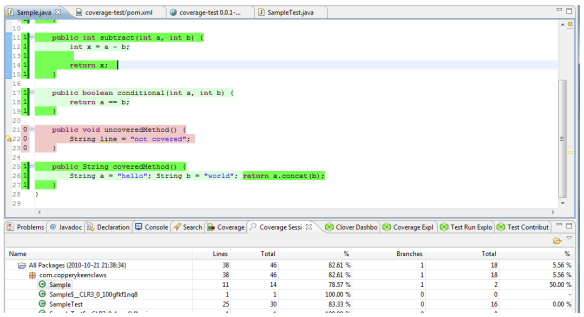


Figure 2.2 Notifications provided by Cobertura regarding code coverage. Figure from

There are a variety of ways that a tool can get the attention of its user [**robillard2014recommendation**]. Program analysis tool notifications get the attention of developers in their IDEs in one or more of the following ways: icons, dashboards, pop-ups, affordance overlays, annotations, or email notifications. Once the tool has the developer's attention, program analysis tool notifications provide descriptive interfaces that convey information about a developer's source code. Information is conveyed using some combination of textual, visual, and sometimes transformative descriptions.

Some tools, like FindBugs and most IDE compilers, use icons to get developers' attention. Using the same icons, developers can access more information either by hovering over or clicking the icon. Although the icons are visual, most of the description provided by these tools is textual. As stated previously, these kinds of notifications are most common with static analysis tools as they typically need to be more descriptive. However, dynamic analysis tools like Veracode, [6] which communicate about defects similar to the ones reported by FindBugs and Coverity, also use icons and text descriptions to pass along information to the developer.

Some tools use affordance overlays or annotations to both get the attention of the developer and for the descriptive interface. For example, Cobertura and JSlice [7] use affordance overlays in the form of source code highlighting, as shown in Figure 2.2 and Figure 2.3, to alert the developer of and communicate about dynamic behavior. Coverity Dynamic Analyzer, [8] which is similar to Veracode, uses annotations, such as the one in Figure 2.4 in the editor to communicate about defects in the code.

A small subset of tools use dashboards, such as the one shown to the right in Figure 2.5. Stench-Blossom, a code smell detection tool gets and maintains a developer's attention using an ambient dashboard [**Murphy-Hill:2010:Ambient**]. At anytime the developer is interested in the information

[6] `http://www.veracode.com/products/dynamic-analysis-dast/dynamic-analysis`
[7] `http://jslice.sourceforge.net/`
[8] `http://www.coverity.com/library/pdf/Coverity-Dynamic-Analysis.pdf`

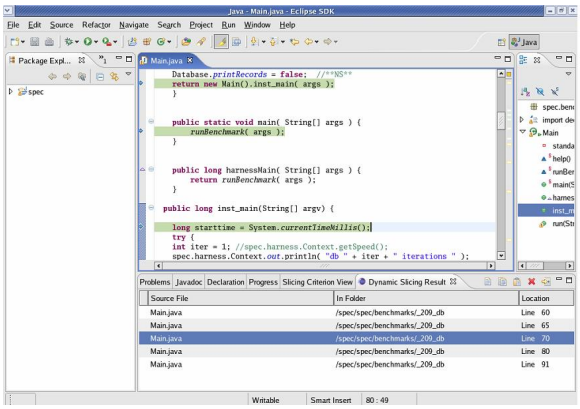


Figure 2.3 Notifications provided by JSlice regarding a dynamic slice of the program.

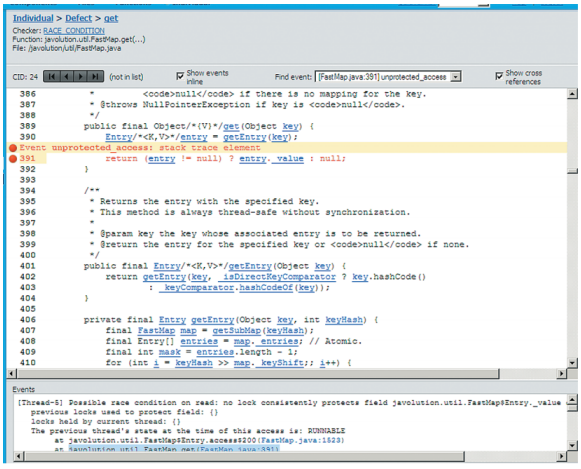


Figure 2.4 A notification provided by Coverity regarding a race condition.

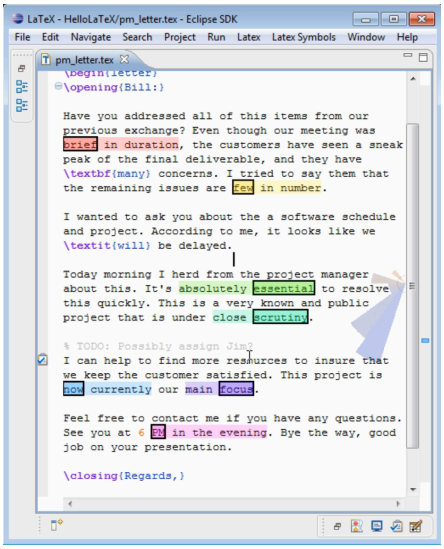


Figure 2.5 Notifications provided by StenchBlossom regarding code smells. Figure from [**Murphy-Hill:2010:Ambient**].

being provided by the tool, the developer can use options in the dashboard to explore code smells present in their code base. The description is visual, using color overlays that map to each type of code smell.

Finally, an even smaller subset of program analysis tools provide transformative descriptive interfaces. Transformative interfaces provide the developer with some idea of how the suggestion being made would affect the task at hand. For example, WitchDoctor, a refactoring tool, detects refactorings and then makes the developer aware of their refactoring by offering to complete the refactoring for the developer. To inform this process, WitchDoctor provides information regarding the assumed refactoring by showing the developer within their text editor what will happen if the refactoring is applied, as shown in Figure 2.6.

2.1.5 Breaking Down the Code Developers Write & Tools Analyze

The goal of program analysis tool notifications is to communicate some information to the developer about her source code about the task at hand. The source code a developer writes is a runnable mani-

```
public static int fullSize()
{
    int size = 0;

    for(String string : list)
    {
        size = findSize(size, string);
    }

    return size;
}
private static int findSize(int size,String string){
    int string_size=string.length();
    size+=string_size;
    return size;
}
```

Figure 2.6 A notification provided by WitchDoctor regarding a refactoring that's taking place. Figure from [**murphy2014recommendation**].

festation of programming-oriented and human-oriented concepts [**van2004concepts**; **biggerstaff1994program**]. At the lowest, most fundamental level, are *programming-oriented concepts*, which relate to how the source code maps to programming language concepts. For simplicity, I will refer to these concepts simply as programming concepts. Programming concepts can be as simple as the means for storing and passing data, such as variables, or as complex as the means for structuring data, such as generics [**jazayeri1997programming**]. At a more abstract level, *human-oriented concepts* relate to the high level requirement of the source code, such as "complete transaction".

A given notification can be associated with one or more human-oriented concept, though not all tools communicate about human-oriented concepts. For example, refactoring tools do not communicate about requirements such as "acquire target." These kinds of tools typically focus on programming concepts; refactoring tools attempt to communicate about programming concepts such as variables and modules. On the flip side, there can be more than one notification pertaining to a one human-oriented concept and one notification can communicate about more than one programming concept.

Consider, for example, the following source code in Figure 6.2. The requirement, or human-oriented concept, at play is "write to log file". If a notification was attached to this code, it would be telling the developer something about the code she wrote to "write to log file". There are multiple programming concepts at play, which aligns with the types of notifications the developer could get. In the process of writing this code, the developer could get a notification regarding any number of programming concepts (buffered streams, exception handling); for example, tools like FindBugs, Sonar, and IntelliJ's built-in static analyzers notify developers when they have opened a stream (`BufferedWriter` in the above example) and there is a possibility the stream that is writing to the file is not closed. A developer may also get a notification regarding exception handling. Here, the developer has written code to catch an `IOException` if it occurs. However, if she did not

```
1  public class LoggerRegistry<T extends ExtendedLogger> {
2      private static final String DEFAULT_FACTORY_KEY =
3          AbstractLogger.DEFAULT_MESSAGE_FACTORY_CLASS.getName();
4      private final MapFactory<T> factory;
5      private final Map<String, Map<String, T>> map;
6
7      public LoggerRegistry() {
8          this(new ConcurrentMapFactory<T>());
9      }
10
11     public LoggerRegistry(final MapFactory<T> factory) {
12         this.factory = Objects.requireNonNull(factory, "factory");
13         this.map = factory.createOuterMap();
14     }
15
16     private Map<String, T> getOrCreateInnerMap(final String factoryName) {
17         Map<String, T> inner = map.get(factoryName);
18         if (inner == null) {
19             inner = factory.createInnerMap();
20             map.put(factoryName, inner);
21         }
22         return inner;
23     }
24
25 }
26
```

Figure 2.7 A source code example for writing to a log file.

implement code to deal with the potential for an `IOException` she would get a compiler notification communicating the need to do so.

Research related to mine falls under the following categories: evaluating and improving tool usability, improving notification understanding and resolution, predictive user models, and developer knowledge representation.

2.2 Program Analysis Tool Usability

There have been many studies on program analysis tools, many of which focus on their correctness and functionality [**Ayewah:2008:FindBugs**; **Bessey:2010:Coverity**; **dugan2000developing**; **luk2005pin**]. Unlike existing work, which typically focuses on one type of program analysis tool, my

work focuses on developers' perception on using different types of program analysis tools, what may have caused their perceptions, and how we can build on or improve their perceptions. Perception plays an important role in when considering human and computer interactions [**Dastani:2002:Perception**] and can be influenced by a number of things, such as the subjective preferences of the user.

Pettit and colleagues studied the effect of enhanced compiler messages for students [**pettit2017enhanced**]. They found no significant improvement, based on likelihood of successive compiler errors, compiler error occurrence, and progress students make towards successful project completion. Their enhancements were designed without rationale, only focusing on students. Also, the information the was same no matter how much students already knew. Enhancements were made based somewhat on assumed knowledge; the adaptations I am proposing would be based on actual knowledge. I also assessed other types of enhancements, such as inclusion of examples and links to external information.

Hamou-Lhadj and Lethbridge surveyed trace exploration tools to determine how these tools can be used by developers and potential improvements to trace exporation tools [**hamou2004survey**]. Based on their evaluation of the tools, they found that areas for improvement for these type of tools include visibility during trace removal and automated suggestions. Pacione and colleagues conducted a case study on five dynamic visualization tools that perform either static or dynamic code analysis and observed the output provided [**pacione2003comparative**]. Pertaining to usability, they found that level of abstraction can make a difference when explaining large scale problems.

Storey and colleagues ran user experiments with 12 developers on two approaches for presenting software structure in a reverse engineering system [**storey1997rigi**]. They found that certain interfaces are better for low-level tasks and that users prefer reverse engineering tools with consolidated interfaces, rather than multiple windows. Bennett and colleagues conducted interviews and user experiments to evaluate the usability and potential improvements of their sequence diagram creation and exploration tool [**bennett2008survey**]. Sequence diagram tools create sequence diagrams using static analysis, dynamic analysis, or both. Much of the feedback developers provided suggested search and selection highlighting are both useful feature for sequence diagram tools, though there are improvements that can be made to existing implementations.

Ayewah and Pugh conducted a study where they claimed that static analysis tools should help engineers find bugs as early as possible in the development cycle, when they are cheap to fix [**Ayewah:2008:FBSurvey**]. They interviewed 12 FindBugs users by phone and conducted a controlled study with 12 students to see how they use FindBugs and handle defects that are labeled "not a bug". Ayewah and Pugh also conducted a study on using checklists for triaging bug reports [**Ayewah:2009:Checklists**]. In their study, they asked students to complete a checklist based off the warnings produced by FindBugs in order to identify warnings that are most important to the

users. The checklist gave different scales used to measure warning severity and relevance and was used for 13 different warnings.

Khoo et al. examined and focused on the interface of static analysis tools and how the interface could be improved [**Khoo:2008:PathProjection**]. They developed a user interface toolkit called *Path Projection* that uses program visualizations to help developers walk through the error reports produced by static analysis tools. *Path Projection* was designed to improve and simplify the process of triaging bug reports, or labeling bugs as a false or true positives, by utilizing checklists to systematically label bugs. This research is similar to my work in that they look at improving the static analysis tool user experience. My research builds on this work by investigating not only improving the user experience, but also finding out why these improvements need to be made from the developers who use them.

Heckman and Williams conducted research in an attempt to develop a benchmark, FAULT-BENCH, that would help developers compare and evaluate static analysis alert prioritization and classification techniques [**Heckman:2008:Faultbench**]. The overall goal of their research was to make using static analysis tools easier and more useful to developers. My work is related in that I am also looking for ways to improve the current state of tools for developers. Layman et al. recruited 18 participants to investigate factors that developers may consider when deciding whether to address a defect when notified of it [**Layman:2007:FaultFix**]. This study is related to my work in that they are also interested in learning more about how developers use tools available to them and how usage can be made easier. My work builds on these works by focusing on various aspects of using program analysis tools, including how users interact with the tools.

2.3 Aiding Notification Resolution

Existing research has focused on easing the process of understanding and resolving notifications [**Hartmann:2010:Su**
Mucslu:2012:Speculative; **pham2015automatically**; **fritz2014developers**] from one particular tool. Rather than studying program analysis tools separately, we believe it is more fruitful to understand the challenges developers encounter across multiple program analysis tools. As we describe in this section, existing studies that examine multiple tools typically either focus on tools of the same type (i.e. multiple compilers) or helping developers make informed choices among tools. My work is related in that our findings can be used to improve the design of tools to better support developers. My work differs in that we investigate different types of tools to identify general challenges developers encounter when interpreting notifications across tools.

Much of the research on improving developers' ability to interpret tool notifications has focused on compiler notifications [**Hartmann:2010:Suggestions**; **Traver:2010:Messages**; **barik14**].

Hartmann and colleagues developed a social recommender system, HELPMEOUT, to better assist novices with understanding and resolving compiler notifications [**Hartmann:2010:Suggestions**]. They found their tool provides useful fixes about half of the time. Traver investigated why developers have difficulty with compiler notifications and ways to improve compiler notification design [**Traver:2010:Messages**]. Based on his findings, Traver developed compiler notification design principles, which includes using consistent messages and including more visual aids.

Nienaltowski and colleagues studied novice developers' ability to identify errors in their programs and how we can better support that process [**Nienaltowski:2008:Compiler**]. Rigby and Thompson studied novices' use of Eclipse and Gild, a customized version of Eclipse featuring "novice-friendly" compiler notifications [**Rigby:2005:Novice**]. Muşlu and colleagues developed QUICK FIX SCOUT, an extension to Eclipse Quick Fix, to ease the process of determining an optimal fix [**Mucslu:2012:Speculative**]. They found programmers could more quickly assess and apply quick fixes when able to easily reason about fix trade-offs. Following up on work with QUICK FIX SCOUT, Muşlu and colleagues explored the possibility of improving IDE recommendations, and the ability for developers to determine the best fix for their code, by considering the whole code base rather the local context of the notification [**mucslu2012improving**]. Barik and colleagues studied how developers reason about compiler notifications to improve tool support for understanding and resolving tool notifications [**barik14**]. Compiler notifications are not the only type of notifications a developer might encounter, further supporting the need for cross-tool investigations. Studying tool notifications across tools, as we have, increases the likelihood our findings can generalize to a variety of tools.

Cross-tool studies that do exist focus on helping developers decide what tools to use rather than tool improvement. Mettrey evaluated five expert systems tools on factors such as performance, to aide developers in selecting one for their projects [**mettrey1991comparative**]. Wagner and colleagues compared two analysis tools that detect defects to evaluate their efficiency [**wagner2008evaluation**]. Other tool evaluations have had the same goal [**roy2009comparison**; **zheng2006value**].

Though to our knowledge there are no studies that explore the applicability of communication theory to tool use, there are studies that explore the applicability of other theories to tool use [**barik14**; **xiao2014social**; **riemenschneider2001explaining**]. One is my prior work on how developers visualize compiler messages; we found that self-explanation theory can be used to explain how developers work through compiler error messages [**barik14**]. In other prior work, we used Diffusion of Innovation theory to explore factors that influence security tool adoption [**xiao2014social**]. Similarly, Rienmenschneider and Hardgrave explored why tools do not get used using the Technology Acceptance Model, based largely on the Theory of Reasoned Action [**riemenschneider2001explaining**]. Lawrance and colleagues used information foraging theory to propose a theory of information

foraging for how programmers navigate code when debugging [**lawrance2013programmers**].

2.4 Predictive User Models

Existing research has explored the idea of creating and using predictive user models both in the design of intelligent tutoring systems (ITS) and adaptive user interfaces (AUI). Research also exists that uses code as a proxy for knowledge [**fritz2010degree**]. While related, this research proposes models that predict knowledge of source code; my research proposes models that predict knowledge of programming concepts for adapting tool notifications.

Both ITS and AUI use models of user knowledge to adapt to their users, based on the user's experiences [**murray1999authoring**]. ITS pose questions to students to model knowledge of concepts and adapt lesson plans. In contrast, I used source code history to model knowledge for adaption of tool notifications. Amershi and Conati explored using machine learning rather than knowledge-based user models to deal with the drawback of using knowledge-based user models, as most ITS do [**amershi2007unsupervised**]. Stamper and Barnes proposed a method for using student data, such as the code they write, to improve ITS with adaptive hints for programming mistakes [**stamper2009unsupervised**]. Similarly, my research explores using source code and machine learning to predict user knowledge but for adaptive tool notifications.

There are a variety of AUI used in different Human-Computer Interaction contexts, many of which use task or domain models [**schlungbaum1996model**]. Most relevant to our models in the context of AUIs are works that build and apply user models, such as FUSE, which creates user models based on static and dynamic properties of the user to assist with user interface development [**lonczewski1996fuse**]. Most relevant to our research are the models proposed by Zou and colleagues for adaptive menus in Eclipse [**zou2008adapting**]. This AUI's models are built based on how often developers use menu items to remove menu items that are used infrequently; we built models based on the concept-specific code a developer writes to adapt notifications to their experience.

Using source code to represent knowledge relates closely to the degree-of-knowledge models proposed by Fritz and colleagues to determine how familiar a developer is with a particular portion of a codebase [**fritz2010degree**]. Their models predict how much a developer might know about a given piece of source code in a codebase based on how often they have visited or edited that part of the code. Other tools exist that make use of developer source code to make predictions without models. Most relevant is Stylos and colleagues' tool Jadeite which determines API usage examples to provide to a developer based on code other developers have written [**stylos2009improving**]. Along the same lines, Perscheid and colleagues explored the notion that expertise is a good metric for determining

who would understand a fault or failure when testing best in a code base [**perscheid2012test**].

Other research has explored how user actions can be used to represent knowledge, mostly in the context of games and learning. Eagle and colleagues conducted research to explore the relationship between the interactions students make in games and understanding science concepts embedded in those games [**eagle2015measuring**]. Hicks and colleagues developed an approach to modeling student interactions when using programming tutors and educational games, like BOTS, for predicting the best hints to provide when they are having trouble [**hicks2014building**].

CHAPTER

3

STATIC ANALYSIS TOOLS USE

Despite the benefits of using static analysis tools to find bugs, consistent usage of these tools is not very frequent [**Ayewah:2008:FindBugs**; **ge2012reconciling**]. There have been studies to investigate ways of improving static analysis tools [**Bessey:2010:Coverity**; **Khoo:2008:PathProjection**]. However, none look at what the tools do for a developer, what could be improved *and* why. The study outlined in this chapter provided an improved understanding of why software developers are not using static analysis tools and how current tools could be improved to increase usage based on developer feedback. To answer the question *why do developers not use program analysis tools*, I conducted interactive interviews with 20 professional developers to better understand why they do not use static analysis tools, one common type of program analysis tool, to find bugs when writing code [**johnson2013don**]. For this study, I focused on static analysis tools used to finds bugs. This includes tools like FindBugs, Lint [**Johnson:1978:Lint**] , IntelliJ [**IntelliJIDEA**] (which includes built-in static analyzers), and PMD [**PMD**]. FindBugs will be referenced the most as it is the tool I chose to use during the interviews.

Table 3.1 Descriptive statistics reported by participants.

Participant	Open Source Tools	Closed Source Tools	Local
Abby	FindBugs	IntelliJ	Yes
Adam	CheckStyle, FindBugs, PMD	IntelliJ	Yes
Andy	FindBugs, Lint	Jtest	Yes
Chris	CheckStyle, FindBugs, Lint	Coverity	Yes
Cody	Dehydra	-	Yes
Frank	-	-	Yes
Gordon	Lint, CheckStyle, FindBugs	-	Yes
Jake	FindBugs, Lint	FlexLint, Klocwork Insight, Visual Studio	Yes
James	CheckStyle, FindBugs, Lint	Visual Studio	Yes
Jason	FindBugs, Lint	-	Yes
John	CheckStyle, Copy/Paste Detector(CPD), FindBugs, Lint, PMD & CodePro	-	Yes
Jordan	CheckStyle, FindBugs, PMD	JTest	Yes
Josh	FindBugs, Lint	Coverity	No
Lee	CheckStyle, FindBugs, Lint	Visual Studio	Yes
Matt	Lint	FlexLint, PyCharm	Yes
Phil	-	-	Yes
Ray	CheckStyle, FindBugs	-	Yes
Ryan	FindBugs, Ling	Coverity	Yes
Steve	CheckStyle, FindBugs, Lint, CPD	IntelliJ	Yes
Tony	CheckStyle, FindBugs, Lint, PMD, CPD, cpplint, Splint	Coverity	Yes

3.1 Exploring Developer Tool Use

For this study, I conducted interviews with software developers. Each semi-structured interview lasted approximately 40-60 minutes and, with the participant's consent, was recorded. By conducting "semi-structured" interviews, I aimed to achieve the flexibility needed to get as much detailed information as possible [**Hove:2005:Interview**]. I prepared a script of questions for the interview, but would add or omit questions on the fly depending on how detailed a participant was in their responses. We created and modified the script as we conducted trial interviews; any changes made to the script was based on the responses we got from our 4 trial participants [**Johnson:2012:PreFFSAT**].

Upon completion, I manually transcribed each session. I performed qualitative analysis [1] on the transcripts by "coding" the transcriptions. This process is discussed in detail in Section 3.1.6.

3.1.1 Participants

I conducted this study with a group of 20 participants. Although this seems like a small sample, I followed a similar methodology to that of Layman et. al.'s study that only had 18 participants [**Layman:2007:FaultFix**]. I recruited participants using an electronic recruitment flyer that was sent out to industry contacts to then be sent to developers within their company. Sixteen participants were, at the time, professional developers at a large company and 4 were graduate students at North Carolina State University with previous industry experience. Participants' years of development experience ranged from 3 to 25 years. We did not explicitly ask participants about their experience building static analysis tools, however, based on conversations approximately 2 participants had tool building experience. We interviewed two participants remotely, one by phone and one by video chat, due to location differences. Each participant filled out a short questionnaire used to collect demographic information prior to their interview.

Table 3.1 shows the statistics and background information gathered from the questionnaire and interviews. The first column lists the participants' pseudonyms, given for confidentiality purposes. The second and third columns show the open-source tools and closed-source tools that they have used to find bugs. If a space has a "-", it indicates no response from the participant.

3.1.2 Research Questions

For this study, I answered the following research questions:

RQ_1 : What reasons do developers have for using or not using static analysis tools to find bugs?

[1] All study materials including interview scripts and coding categories are available at `http://www4.ncsu.edu/~bijohnso/ffsat.html`

RQ_2 : How well do current static analysis tools fit into the workflows of developers? We define a workflow as the steps a developer takes when writing, inspecting and modifying their code.

RQ_3 : What improvements do developers want to see being made to static analysis tools?

I asked these questions because answers to these questions are important to the progression through my dissertation and can give toolsmiths and researchers areas for future work and improvement in the area of static analysis tool usability. Research has shown that the way a tool interrupts a developer's workflow is important therefore I also wanted to specifically investigate this aspect of tool usage [**Robertson:2004:Interruption**; **Gluck:2007:Attentional**]. The interviews focused on developers' experiences with finding defects using static analysis tools. Learning developers' relevant experiences and observing how they use static analysis tools to find bugs may shed some light on why these tools are underused. I organized the interviews into into three main parts: Questions and Short Responses (Section 3.1.3), Interactive Interview (Section 3.1.4), and Participatory Design (Section 3.1.5).

3.1.3 Part I: Questions and Short Responses

During part 1, Question and Short Response, I asked developers questions related to their general usage, understanding, and opinion of static analysis tools in order to answer RQ1.

Some of the questions I asked include:

- Can you tell me about your first experience with a static analysis tool?
- Can you remember anything that stood out about this experience as easy or difficult?
- Have you ever used a static analysis tool in a team setting? Was it beneficial and why?
- Have you ever consciously avoided using a static analysis tool? Why or why not?
- What in your opinion are the critical characteristics of a good static analysis tool?

3.1.4 Part II: Interactive Interview

The second part is what I called the "interactive interview". The goal behind the interactive interview is to be able to observe developers actually using a static analysis tool. This allowed for the gathering of more detailed information as to how developers are using their tools. I used the information obtained during this portion to address RQ2. I asked participants to explain what they are doing out loud [**Lewis:1982:ThinkAloudProtocol**] so I could get a better understanding of their workflow and

thought process. Practice interviews before this study revealed that using the interactive interview portion produced more detailed information regarding when and how developers use their static analysis tools [**Johnson:2012:PreFFSAT**].

Some of the questions asked during this portion include:

- Now that you have run your tool and gotten your feedback, what is your next move(s)?
- Do you configure the settings of your tool from default? If so, how?
- Does this static analysis tool aid in assessing what to do about a warning?
- Do you feel that "quick fixes" or code suggestions would be helpful if they were available?[2]

For confidentiality reasons, not all participants could use their own workstation for this part of the interview. For those who could not, I provided 6 open source projects in Java, such as log4j [**log4j**] and Ant [**ANT**], and asked each participant to run FindBugs on one of them. I chose FindBugs because it is one of the most popular and mature open source static analysis tools for Eclipse. Due to technical difficulties, the remote interviews could not fully experience the "interactive" portion. Each remote participant was given a scenario of static analysis tool usage and asked to, first, explain their thought process in walking through that particular scenario. I then asked the same questions as I would have asked if they had been local.

3.1.5 Part III: Participatory Design

The last part of the interview allowed participants to make design suggestions for improving static analysis tools. I utilized a concept called *participatory design* [**Spinuzzi:2005:Participatory**], which involves getting stakeholders (in this case, the participants) involved in the design process by allowing them to show what they want instead of saying it. In order to promote creativity, each participant was given a blank sheet of paper and asked to sketch what they wanted their tool to look like and describe how it should work [**Johnson:2012:PreFFSAT**]. I did not require participants to draw something, but 6 of them did. The rest of the participants gave verbal descriptions of tool features they desired.

3.1.6 Coding Interview Responses

After completing the interviews, I manually transcribed each interview. Then, I *coded* the transcriptions. Coding is a process that is meant to make referencing transcriptions quicker and eas-

[2] Participants were only asked about quick fixes and code suggestions being useful when they mentioned, either during the Question and Answer or Interactive Interview, that they either a) find quick fixes useful, b) felt that the tool should be more helpful or c) did not understand how to fix the defect we presented them with.

ier [**Gordon:1998:Coding**]. I used Gordon's basic steps to code the interview transcripts and used the codes to help organize the Results (Section 3.2). According to Gordon's steps, before coding an interview, "coding categories" need to be defined. These should be general enough for relevant information to be grouped together but detailed enough that a concrete example only falls under one category. Because of this, it is possible to have "emergent" categories that may need to be defined after reading the transcriptions. I developed and used the following coding categories:

- *Tool Output*: anything related to the output produced by the tool (for example, false positives).
- *Supporting Teamwork*: anything about using static analysis tools in a team or collaborative setting
- *User Input and Customizability*: points made about the customizability of the static analysis tools (for example, modifying rule sets)
- *Result Understandability*: anything said about the ability or inability to understand or interpret the results produced by a static analysis tool
- *Workflows*: anything related to the steps a developer takes when writing, inspecting and modifying their software (for example, tool integration)
- *Tool Design*: the proposed tool design ideas from our participants.

Examples of each of these categories from the transcriptions are as follows:

Tool Output

Jason: *"...like I mentioned with FlexLint it gives you so many warnings and sifting through them is so, arduous that whenever I just look at it I'm like ehhh forget this."*

User Input/Customizability

Andy: *"...it's like is this list prioritized by you know what's important to me? No. You know? And there may be a default listing that should be prioritized because like this one's inefficient."*

Supporting Teamwork

John: *"The only reason I like the batch results is to communicate, broadcast to the team a sense of progress or lack of progress."*

Result Understandability

Matt: *so now I wanna know why raising a string exception is bad. Like what should I be doing instead? Since it thinks it's a problem. And so none of these really help me.*

Workflows

Mike: *"Clang is my favorite. It's built into the compiler. You don't have to invoke anything special."*

Tool Design

Chris: *"I don't mind the idea of the actual source code itself having some plasticity . . . let's say the fourth line there was some error here. . . having the 5th line drop down and having the content expand with maybe all sorts of annotations about my code."*

The next step in Gordon's methodology is to assign "category symbols" to each category for easier indexing and processing of information. Gordon then suggests finding and classifying the relevant information in the transcriptions using the category symbols. In my set of codes, each coding category had its own color as a "symbol"; if a portion of a participant's transcription fell into one of the categories, the text would be highlighted the same color as its respective category. A participant's coded interview could contain multiple categories or even multiple data items for one category. To ensure consistency, only I was responsible for coming up with the coding categories and "symbols" and going through the transcriptions to apply them. The last step is to check the reliability of the codes. For this study, once the coding was complete, I passed the documents over to other contributors to look over. If there were any discrepancies we discussed and resolved them as a group. This includes items that could fall into more than one category; in this situation, either a new, more specific, category or a "sub-category" was created for the item. The purpose of the categories are to organize the data in a relevant and useful manner; they are not meant to directly correlate with the research questions. The next section discusses findings from this study.

3.2 Barriers to Tool Use

In this section, I will discuss the results obtained from the interviews. I answered my research questions by linking the questions to coding categories and interview parts.

Based on the data collected, the following was true:

- My first research question (RQ1) was answered by observing the results categorized under "Tool Output," "Supporting Teamwork," "User Input and Customizability," "Result Understandability" and "Developer Workflows"; the information collected in these categories could be reasons why developers are or are not using static analysis tools.
- My second research question (RQ2) was answered by observing the results categorized under "Developer Workflows."

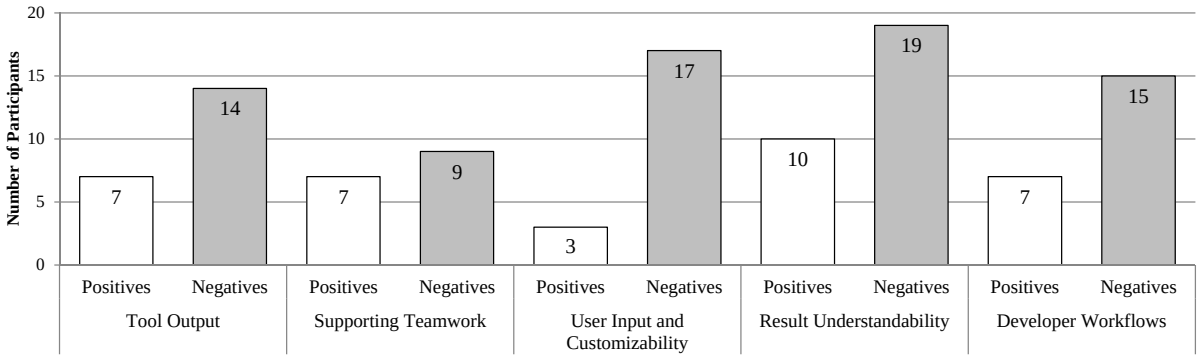


Figure 3.1 The number of participants in each category expressing the good and the bad about static analysis tools they have used.

- My third research question (RQ3) was answered by observing the results categorized under "Tool Design"; most of these results are from the Participatory Design portion.

In each category, I expected there to be negative and positive remarks about current tools, both of which are equally important in answering my research questions; anything positive could be a reason for use while anything negative could be a reason to discontinue use. For each coding category, I separated the relevant statements into positive statements and negative statements; if something good is said about a static analysis tool it is considered a *positive* comment and vice versa for a *negative* comment. In Figure 4.6, we can see that the majority of participants have had problems with tool output, customizability and workflow integration, and all but one of participant has had problems with understanding the results of tool analyses. Tool design is not included because this category was defined to capture the developers' ideas for improving static analysis tools. Their reasons for wanting the features are captured in the other categories.

3.2.1 RQ1: Reasons for Use and Underuse

The interviews revealed that there are a variety of reasons developers may have for choosing to use a static analysis tool to find bugs in their code. One of the obvious reasons is because too much time and effort is involved in manually searching for bugs. Five out of 20 participants felt that because static analysis tools can automatically find bugs, they are worth using. During his interactive interview, Jason told us "*anything that will automate a mundane task is great.*" In other words, one reason for using static analysis tools is that they automate the process of finding bugs.

Another reason developers might use a static analysis tool is if it is already available in the development environment and ready to be used. For 3 participants, this was the case. Development

environments such as IntelliJ and PyCharm come with built-in static analyzers, which requires little extra effort on the developer's part. Two participants, Matt and Adam, used PyCharm and IntelliJ regularly and liked the fact that static analysis was already integrated. For 7 participants, a good reason to use static analysis tools is to support team development efforts. According to Josh and Andy, static analysis tools do this by raising awareness of the potential problems, or "dumb mistakes," in the code earlier in the development process. For Cody and Ray, static analysis tools are useful for communicating and enforcing coding standards and styles on development teams. Some developers enjoy using the static analysis tools they use to find bugs because of the level of customizability. Three participants fit into this category. According to James, the customizability of a tool can play a large part in the volume and quality of output developers get.

Although some participants could find reasons to use static analysis tools to find bugs, most of our participants brought up conflicting concerns that could make the decision to adopt and regularly use static analysis tools less obvious.

Tool Ouput. Tool output was a popular discussion topic. Out of the 20 developers I interviewed, 14 expressed the negative impacts of poorly presented output. Static analysis tools are known to produce false positives and these false positives can "outweigh" the true positives in volume [**Shen:2011:EFindBugs**]. Another known fact is that, especially with larger projects, the number of warnings produced by a tool can be high, sometimes in the thousands [**Ayewah:2010:GFF**]. Some participants felt, however, that false positives and large volumes of warnings would be less burdensome if the way the output is presented was more user-friendly and intuitive. Cody, who likes using Dehydra, finds himself frustrated at times because the results are dumped onto his screen with no distinct structure causing him to spend a lot of time trying to figure out what needs to be done. Jason wishes that his tool's output would be a "slice" that shows what the problem is and what else could be affected in order to more quickly assess what is or is not important. This "slice" should be taken from the entire project, using call hierarchies, to show which parts are affected by each defect. During his interactive interview he commented on a previous experience with FindBugs. He had a large list of warnings to scroll through but without there being any context to the problems it just seemed like "a bunch of junk to sift through," which made him not want to bother using it. It may be worth investigating how valuable an output like this would be.

Collaboration. In industry, software development is often a team effort. For 9 participants, lack of or weak support for teamwork or collaboration was one reason that teams, as well as individual developers, may not adopt or regularly use static analysis tools. According to John, although static analysis tools are useful for trying to enforce coding standards, there is no easy way to share the settings with other people on the team so it ends up being a cumbersome manual process and causing confusion when the standards need to be changed. Many participants mentioned the desire

for a way to easily communicate and collaborate when using their static analysis tool, especially in a team setting. Although static analysis tools can be beneficial in team settings, current tools are not collaborative enough for some developers. Newer versions of FindBugs offer a cloud storage feature that can be used store, share and discuss warning evaluations [**FindBugsCloud**]. Although a feature like this does make it easier to communicate and share warning evaluations between developers, to add a comment to a bug or current evaluation a web browser is needed. This takes the developer out of context and out of the development environment which could demotivate some individuals from checking them when they should.

Customizability. For 17 participants, customizability was important however many tools are not trivial to configure and do not accommodate the customizations that developers want. False positives and large volumes of warnings are well-known downsides to using static analysis tool to find bugs. However, Frank told us he believes that the way a tool is configured plays a large part in the output a developer gets. John stated during his interview that "*many tools are so hard to configure, they prevent you from doing anything.*" Sometimes it is difficult just to get to the menu where the options for configuring a particular feature are, which participants Matt and Josh agree with. One participant, Jake, found himself in an interesting situation during his interactive interview where he could not figure out how to customize his tool and wound up having to search the web to locate the tool's preferences. A common problem expressed by most of the participants is the inability to temporarily ignore or suppress certain warnings. Although some static analysis tools allow developers to turn off certain filters, not all developers are comfortable with turning warnings completely off. Matt, for example, is afraid that he may not remember to turn it back on. The notion of dismissing or ignoring static analysis warnings may be too coarse; as Jordan noted, he would prefer if static analysis tools offered a way of recording his judgement about a given warning. More sophisticated judgements may include things like "this warning isn't a problem now, but may be in the future if the following conditions are met…".

Result Understandability. The main objective when using a tool like FindBugs is to learn what defects are in the code so that problems can be removed. A developer not being able to understand what the tool is telling her, according to participants, is a definite barrier to use. Nineteen out of 20 participants felt that many static analysis tools do not present their results in a way that gives enough information for them to assess what the problem is, why it is a problem and what they should be doing differently. James told us during his interview that "*it's one thing to give an error message, it's another thing to give a useful error message.*" When talking about the Eclipse Python plug-ins, he also stated, "*I find that the information they provide is not very useful, so I tend to ignore them.*" A few participants felt that it would be helpful to have links to more details or examples in the error reports. In some situations more information is needed to understand exactly what the problem is

and why it is a problem; understanding why a defect is a problem can help the developer better assess whether the error is a false positive and try to avoid repeating the same problem. Ryan told us during his interactive interview that a start would be using "real words," or a more natural language, to explain the problem.

The most frequently mentioned difficulty when using static analysis tools was lack of or ineffectively implemented quick fixes. Most participants expressed interest in having their tool provide code suggestions or quick fixes that assist them when attempting to fix a bug; Abby proclaimed "*if you can tell me it's an error, you should be able to tell me how to fix it.*" Jordan strongly agrees; he loves tools that have quick fixes and hates tools that do not. According to the interviews, these fixes do not have to be automatic; some prefer that code suggestion previews be used or possibly using examples to get a better understanding of how to fix the problem. Some participants expressed interest in but skepticism toward integrating quick fixes into static analysis tools. For example, during Jordan's interactive interview, he noted that sometimes when using multiple tools, they may have conflicting quick fixes or solutions. In Frank's past experiences with automated code changes, he has had to do manual refactorings because something was done wrong; because of this, he prefers to use find and replace to make his own changes. Another participant, Adam, was concerned with knowing whether the semantics of his code would be preserved after applying a quick fix. Most static analysis tools, if they offer quick fixes, leave it to the developer to figure out exactly what has been done after it has been done. Almost all of the participants agreed that effectively designed quick fixes can help them to better understand the problems tools tell them about, leading to a better sense of productivity for the developer.

3.2.2 RQ2: Workflow Integration

The most common topic during the interviews was tool and environment integration. Sometimes a developer's process includes running a static analysis tool, but more often it is not part of a developer's workflow to stop and run a tool in the middle of working on some code or a specific task; she usually prefers finding a "stopping point" in her code to run the tool [**Layman:2007:FaultFix**]. Analysis of these interviews revealed that while this is true, there are many different ways that developers may want their tool to fit into their development workflow. For example, some developers prefer that the tool run in the background; it is easier for them to figure out what is wrong if they are in the process of doing it and do not have to think about invoking the tool. On the other hand, some developers do not use IDEs, so if they are to use a static analysis tool, compiler integration is very important. Nineteen of the 20 developers I interviewed expressed the importance of workflow integration to them and how these needs have been or should be met.

For some participants, there are features of static analysis tools they have used that helped the tool better integrate into their workflow leading to increased usage of the tool. In fact, John felt that static analysis tools can be used to help organize a workflow, based on the results it produces. For example, running a static analysis tool on some code for the first time can be a good indicator of the kinds of bugs the tool finds and that may be present; this can give an idea as to how detailed of an analysis the tool provides, possibly giving a better idea of when it would be best to run it. Of all the tools Adam has used in the past, he much preferred to use IntelliJ and its built-in static analysis to find bugs; they are tightly integrated making it seem more "real time". For these participants, as well as a few others, integration with the development environment plays a major role in their decision to use or continue using a static analysis tool. Common standalone static analysis tools like FindBugs and PMD have the ability to integrate with IDEs like Eclipse and NetBeans which becomes especially important when using more than one static analysis tool at a time, as we learned from discussing a past experience of Steve's where he used 3 different static analysis tools at once. Jordan and Chris like how FindBugs, PMD and CheckStyle fit into their development processes; for Jordan, it is an integral part of his workflow. For the majority of participants, however, current static analysis tools are not doing enough to effectively integrate into their development process.

One of the biggest demotivational forces on a developer when it comes to using a static analysis tool to find bugs is when it is what Tony calls a "disjoint process." Many participants, especially those who do not use IDEs, do not like when they have to go out of their coding environment to use a tool or view the results produced by the tool. For example, Frank, Lee, James and Andy commented on how "painful" it was during their interactive interview to have to switch perspectives in FindBugs to explore the complete listing of bugs. According to Lee, having to open another perspective to know what is going on is a guarantee that unmotivated people will not do it. For Frank, although it is nice that the results are hidden so that you are not overwhelmed, having to go back and forth and drill down to see the bugs requires extra effort and is disruptive to his workflow. Other tools participants had similar complaints about was Coverity and Lint for C/C++ projects. For Ryan and Tony, the biggest downside to using Coverity is that it is not capable of being integrated into their coding environment, leading to a lot of clicking back and forth between their editor and the static analysis tool. Phil does not like using Lint because of the fact that he has to "go out of his way" to do so.

Some participants made it clear, however, that even if the tool is integrated with their development *environment*, it is still possible that the tool does not integrate well into their development *process*. For example, Mike does not use IDEs so using a tool that integrates well with an IDE does not fit well into his development process; he likes using Clang because it can be tied into his compiler which does not require a "development environment". According to Gordon, one of the key problems with static analysis tools is that at times they can prevent him from being productive.

One way this can happen is when the tool slows the developer down by taking a long time to run, which was a common complaint amongst participants in this study. From Jason's experience, he believes that "*if it disrupts your flow, you're not gonna use it.*" Jason's statement rang true among other participants as well, like Steve who had used various tools in his past but did not like to use FindBugs because, even though it is IDE integrable, it runs slow. IntelliJ, which contains built in static analyzers, utilizes *idle time* when reporting bugs in an attempt to prevent the problem of interrupting the developer's workflow but for Matt, it can still be bothersome. Jason believes that the problem with current static analysis tools is that they are not capable of running well on larger code bases, leading to a break in his "development flow" as he waits for the tool to catch up.

In terms of workflow, participants valued using static analysis both to fix bugs once they are introduced into the program, but also later in the development process. From a workflow standpoint, it is valuable to fix potential bugs when they are entered into a program because the necessary context to understand the bug is already in the developers' working memory. In contrast, fixing bugs later is difficult because a developer must recall the context to analyze the corresponding static analysis warning. This contrast is similar to the difference between "floss refactoring" and "root canal refactoring," where the former involves restructuring code as it is being worked with and the latter involves refactoring by finding the "worst code" and dealing with that first [**Murphy-Hill:2008:RefactoringTools**]. Root canal refactoring is a discouraged practice and its analog in static analysis – finding the most severe static analysis warnings in a whole codebase and dealing with those first – may also be a wasteful practice. Research has shown that many static analysis warnings in working systems do not actually manifest as program failures [**Ayewah:2010:GFF**].

3.2.3 RQ3: Tool Design

My main research goal to determine how we can improve static analysis tools for developers. The best way to do this is to find out how developers want their tool to be designed. Most of the proposed designs are for warning notification and manipulation or quick fix display. Participants made some other interesting proposals which will also be presented.

Quick Fix Design. Ten participants made a suggestion related to the way in which a quick fix should be displayed. Most participants wanted to be able to preview the fix and how it is going to change their code before they apply it. Abby and Tony recommended splitting the code editor to show a diff of the code, using highlighting to show what code has changed or been added to their code. On one side there would be the code now and on the other the code once the fix is applied. Some felt that you should be able to see the fix before applying it, but then also manually apply it so that you know the fix is being applied without introducing any new problems. One participant,

Mike, prefers not to have quick fixes at all because he feels the error messages are enough to assess what to do about an error.

One interesting quick fix design idea, which came from Ryan during his interactive interview, was to have what he called a "*three option dialog box*" available when applying a quick fix. This dialog box would pop up upon a click to fix the bug and there would be three choices: apply the entire fix (default option), do not apply the fix or step by step apply the solution allowing the developer to decide which parts of the solution they would like to keep. Static analysis tools like FindBugs and IntelliJ offer some quick fixes. However, they do not give a full context preview of the changes that will be made, leaving it to the developer to manually ensure that the fix was applied correctly and to their liking.

Warning Notification and Manipulation Design. All 20 participants told us when and how they wanted to be notified of errors in their code. The theme in this category is "fast." Developers want tools that provide faster feedback in an efficient way that does not disrupt their workflows. For some participants, this meant running the tool in the background of the IDE so that feedback occurs as soon as a problem is detected. For other participants, this meant running the tool at build time or compile time. In this way, the results are presented when the developer is at a predefined "stopping point." [**Layman:2007:FaultFix**].

Overall, participants found that current static analysis tools are not fast enough when providing them with feedback; this quickness should be accompanied with discretion as the developer does not want the tool to break their thought process.

Participants also thought it would be beneficial to have the ability to easily make "judgements" about defects, such as setting it aside to view later, save these judgements and share them with other developers. Many participants suggested that static analysis tools should allow developers to ignore specific defects and move them to their own list for later viewing, a form of *temporary suppression*. Most tools, if they allow the developer to ignore specific warnings, only allow the developer to turn off or suppress a bug category for particular line of code using a comment-like annotation, which Gordon told us makes the code "smell". Developers would like to have the option to ignore each individual defect in case they either do not want to fix it and do not want to be bothered by it again or do not want to be bothered with it at that particular time but would like to come back to it later.

Other Design Ideas. Participants also came up with creative design ideas. One participant, Chris, suggested giving the editor "plasticity". When he is given a warning and would like to get more information, the tool should move the code surrounding the warning to embed this information into the editor. A couple of participants thought it would be useful to have visual output, possibly a pie-style diagram of the project and the bugs in it, instead of standard list and tree outputs to make it easier to go back and forth between warnings and code. During Frank's participatory design

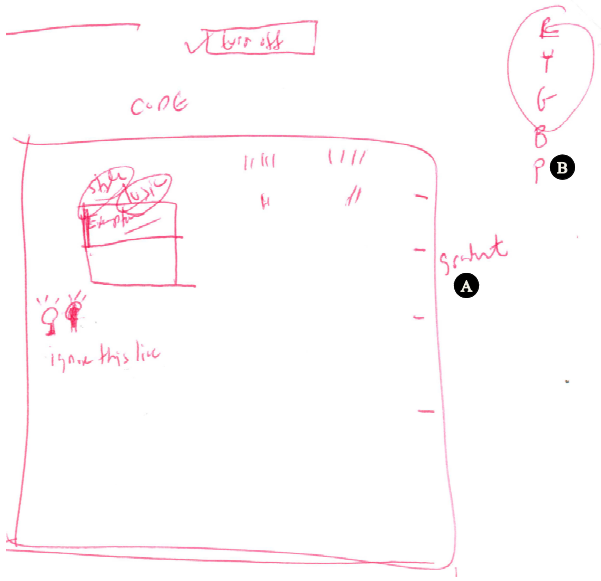


Figure 3.2 One of our participant, Matt's, Participatory Design drawing; (A) shows where Matt wants the gradient colors and (B) shows the way his current tool represents severity.

session, he suggested a potential solution; a parts-to-a-whole corpus view of the project as a *"heat map"*. The heat map would use colors to show where the errors are and how severe the problems are. It would start with an overall "view" of the project and as you drill down you can see the condition at each level to see where the most attention is needed. This is similar to the concept behind Khoo's toolkit *Path Projection* in that the toolkit is meant to visualize output that is usually, if not always, textual and difficult to understand [**Khoo:2008:PathProjection**].

An interesting suggestion made by a couple of participants was to represent the severity of the defects using gradients of one color instead of multiple different colors; the darker the color the more important or urgent the bug is. Figure 3.2 depicts a drawing one participant, Matt, drew during his participatory design; he labeled the side of the editor "gradient" (A) where he would like to see his severity representation. In the top right corner, Matt also lists the colors that his current tool uses (B); for example, "R" means red. The idea behind this is not new; other studies have focused their attention on using colors for error representation [**Oberg:1992:Gradients**; **Murphy-Hill:2010:StenchBlossom**].

3.2.4 Threats to Validity

There exists threats to the validity of this study; here I categorize each threat as a threat to external, internal, or construct validity.

External. One limitation to the generalizability of this study is the sample size. Although I obtained valuable information from the 20 interviews, due to time constraints (and busy developers) they may not be representative of the larger population that use static analysis tools. Although we would have liked more participants, having a large number of interviews to transcribe and code could lead to less accurate analysis. The study conducted by Layman et al. [**Layman:2007:FaultFix**], which we discussed earlier as utilizing a similar methodology, had a participant pool of similar size (18). Another possible threat is that we only interviewed developers who have used static analysis tools. In some cases it may be that static analysis tools are not being used for other reasons, such as lack of awareness. It should also be noted that some participants had experience building static analysis tools, giving them somewhat of a biased opinion of the usage of these tools.

Internal. Another threat to the validity of this study is the way in which I conducted the remote interviews. I did not thoroughly prepare for what I would do if the technology I wanted to use did not work or was not available. Therefore, the interactive interview and participatory design in remote interviews had to be conducted differently than local interviews. Despite this, there was still value in the results obtained from the remote participants; they could still give useful insights from their previous experiences. Only 2 of the interviews fell into this category, so this helps limit the impact of this threat.

Construct. The objective for using the interactive interview was to get more accurate information on how developers use their tools. One limitation here is that some developers were not as familiar with the code or environment they had to use in the interviews as they would be with their own code in their own development environment. This could have caused some developers to take different actions than they would have in their own environment. Ideally it would have been better to have been able to observe participants working in their own environment; however, for confidentiality reasons, we could not view participants' proprietary code. In an effort to compensate for this threat, the open source projects and tool we chose are well-known, frequently used open source projects. Another threat to the validity of this work is that I did not originally consider that I may have said things in the consent form or session script that would have given unintended "hints" to participants concerning my research expectations. One example of this is my outlining the research goals in the introductions I gave prior to beginning each session. This could have led to what is called "hypothesis guessing" where participants respond to questions based on what they think the researcher wants to hear [**Threats**]. In retrospect, I helped alleviate this threat in the interviews by asking participants

experience questions.

3.3 Next Steps to A Solution

3.3.1 Notification Resolution Solutions

For many developers in the study above, it was important that their tools help them with resolving the defects found in their code. Current static analysis tools may not give enough information for developers to assess what to do about the warnings produced and very seldom offer a fix to what it claims is an issue. If static analysis tools offered quick fixes, giving a potential solution and applying it to the problem may help developers assess warnings more quickly and ultimately save time and effort. My results indicate that FindBugs, for example, would be more useful if it had more informative messages and offered quick fixes.

At the same time, quick fixes do not appear to be a universally applicable mechanism to help developers resolve static analysis warnings because many static analysis warnings do not have a small set of solutions. For example, FindBugs warns developers when two method names in the same class differ only by capitalization; no quick fix for this problem is likely to satisfy a developer. Instead, interactive quick fixes that enable easy access to refactoring and code modification tools may be able to semi-automatically help developers resolve static analysis warnings. Barik and colleagues developed and evaluated an approach for interactively resolving defects based on these findings [**barik2016quick**].

On the negative side, quick fixes could also cause developers to be hasty in fixing their code, which could potentially lead to more problems, such as the introduction of new defects [**Mucslu:2012:Speculative**]. There are also challenges related to implementing usable interactive quick fixes. We have not yet investigated what theses challenges are or how to address them as they are out of scope for this particular study.

3.3.2 Notification Understandability Solutions

According to the findings above, although the point of using tools is to help identify and resolve defects, developers do not find the lack of quick fixes to be as much of a barrier to use as inability to interpret the notifications provided. The tool, and any available quick fixes, become less useful if the developer does not understand the problem as it is being communicated. Three out of four findings regarding tool use pertain to the notifications tools use and how tools present information to the developer. Discovering that tool notifications are one of the reasons developers do not use program analysis tools is useful, however, not actionable. The next piece of information needed to make

these findings actionable is to discover why developers have difficulty interpreting tool notifications. Only then can we start to explore ways to mitigate these barriers and potentially increase usage of these tools. By focusing on improving developer ability to interpret tool notifications, I propose we can also improve developer ability to resolve tool notifications. In the next chapter, I discuss a follow-up study that explores why developer have difficulty with tool notifications.

CHAPTER

4

THEORY OF (MIS)COMMUNICATION

In the previous chapter, I discussed findings from a study that explored reasons developers have for not using program analysis tools. One reason that almost all developers in that study did not frequently use static analysis tools was due to difficulty interpreting the output provided by their tools.

could agree on is that the output provided by their tools are sometimes difficult to interpret. Therefore, this chapter presents research that took a deep dive to explore the challenges developers encounter when interpreting tool notifications.

To answer the question *why do developers encounter challenges when interpreting program analysis tool notifications?*, I observed 26 developers with varying backgrounds while they used three different program analysis tools: Eclipse Java compiler, FindBugs, and EclEmma. I presented participants with and asked them to interpret notifications from each of the three tools. To identify challenges, I examined tool use through the lens of communication theory [**bowman1987modeling**]. Building on an existing model of (mis)communication [**mustajoki2008modelling**], I identified 12 kinds of challenges developers encounter when interpreting tool notifications.

Based on the challenges participants encountered when interpreting tool notifications, I proposed a tool miscommunication theory that I later use to inform the design of program analysis tool notifications. Experts in qualitative research suggest that rather than presenting a set of disparate

findings, qualitative researchers should instead produce an explanatory theory, a "skeleton or framework that explains why things happen" [**corbin2014basics**]. While explicitly putting forward theories is rare in software engineering [**hannay2007systematic**], one example is Lawrance and colleagues' theory of how programmers navigate code during debugging [**lawrance2013programmers**]. In the same way that Lawrance and colleagues' build on information foraging theory [**pirolli1999information**], my theory builds on communication theory [**bowman1987modeling**]. I summarize my theory as:

> *The challenges developers encounter when interpreting program analysis tool notifications are caused by gaps and mismatches between developer knowledge and how notifications communicate information.*

In the following sections, I will discuss the methodology I used to identify developer challenges, the 12 kinds of challenges I identified that informed this theory, and ways in which this theory can be operationalized.

4.1 Identifying Challenges

In the study outlined in Chapter 3, I asked developers to recall experiences with static analysis tools and briefly use FindBugs. I found that some developers do not use program analysis tools due to difficulty interpreting the notifications tools use to communicate. To find out how tools could better communicate with developers, this study was designed to answer the question: *Why do developers encounter challenges when interpreting program analysis tool notifications?* Using Hannay and colleagues' guidelines [**hannay2007systematic**], I framed my question as *why* rather than *what* to support the building of a theory that explains the challenges developers encounter.

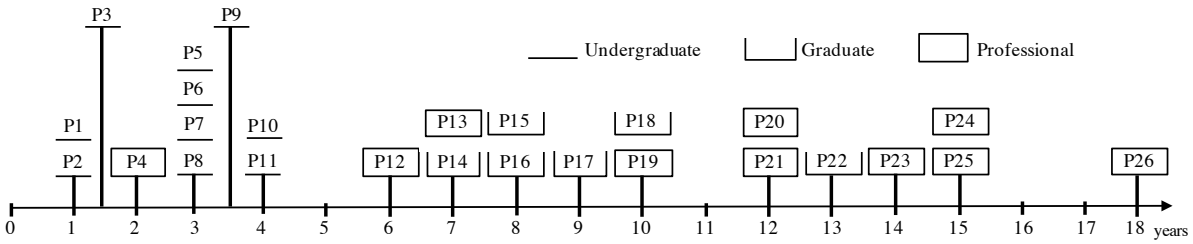


Figure 4.1 Distribution of participants based on years of programming experience.

4.1.1 Participants

I recruited twenty-six participants using mailing lists, classroom recruitment, and personal contacts. Participants included undergraduate students, graduate students, and professional developers, with varying amounts of development and tool usage experience. Figure 4.1 shows the distribution of participants' development experience, based on self-reports in a pre-study questionnaire. Increasing participant numbers indicate increasing software development experience, and throughout this chapter, I use boxes or partial boxes to indicate participant job roles (professional, graduate, and undergraduate respectively). For example, the figure indicates that P24 is a professional developer with fifteen years of development experience. Three graduate students (P15, P18, P22) reported having industry experience. Ten participants had prior experience using EclEmma. Nineteen participants had prior experience with FindBugs. All participants had experience with the Eclipse Java compiler.

4.1.2 Program Analysis Tools Investigated

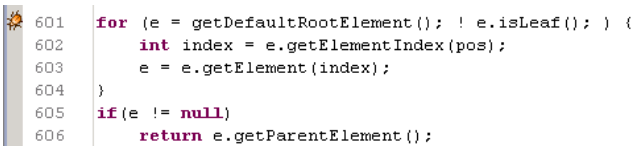
This study focused on tools that can be used in the Eclipse Integrated Development Environment (IDE) [**EclipseIDE**]. I chose Eclipse because it is one of the most widely used IDEs [**Goth:2005:Beware**], making it easier to recruit qualified participants, and because it is compatible with a variety of tools. I selected FindBugs, the Eclipse Java Compiler, and EclEmma as mature, popular tools.

FindBugs

FindBugs (version 2.0) notifications communicate with the developer about defects in her code based on code patterns. Bug icons ( ) in the gutter are colored red to indicate the "scariest" code patterns, orange for "scary" patterns, yellow for "troubling" patterns, and blue for "of concern." Text descriptions are available by hovering over or clicking the icon as seen in Figure 4.2.

Eclipse Java Compiler

Eclipse Java compiler (JDT version 3.8) notifications communicate with developers when their program cannot compile and provide warnings about suspicious code [**EclipseCompiler**]. Notifications are typically shown as squiggly underlines in the editor. Like FindBugs, the compiler uses color to represent severity; errors are shown as red underlines, warnings as yellow underlines. Underlines are augmented with gutter icons ( ), as shown in Figure 4.3 at line 159. When the developer mouses over the underlined code or the icon, the notification displays a text description (Figure **??**). Unlike



```
601  for (e = getDefaultRootElement(); ! e.isLeaf(); ) {
602      int index = e.getElementIndex(pos);
603      e = e.getElement(index);
604  }
605  if(e != null)
606      return e.getParentElement();
```

(a) Source Code

Nullcheck of e at line 605 of value previously dereferenced in `javax.swing.text.DefaultStyledDocument.getParagraphElement(int)`

(b) Short Description

A value is checked here to see whether it is null, but this value can't be null because it was previously dereferenced and if it were null a null pointer exception would have occurred at the earlier dereference. Essentially, this code and the previous dereference disagree as to whether this value is allowed to be null. Either the check is redundant or the previous dereference is erroneous.

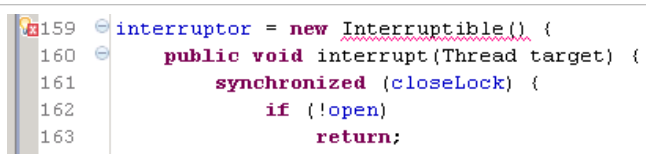
(c) Full Description

Figure 4.2 A notification of a previous null check from FindBugs (FB4).

FindBugs, clicking the gutter icon does not provide a detailed description. Instead, clicking the icon sometimes provides possible fixes that can be automatically applied to the code called quick fixes.

EclEmma

EclEmma (v2.2) is a code coverage tool that executes a program, typically with JUnit as the driver [**JUnit**], to communicate with the developer about code paths that did and did not get exercised. Although EclEmma communicates about one particular execution, as with the other tools it provides information to the developer regarding code (during runtime rather than compile-time). EclEmma



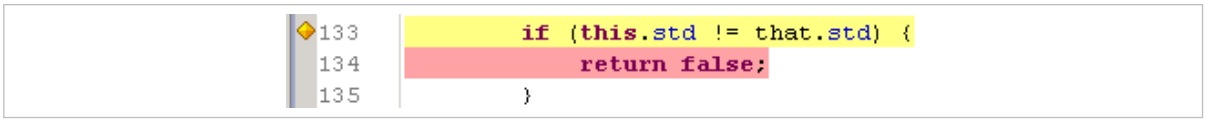
```
159  interruptor = new Interruptible() {
160      public void interrupt(Thread target) {
161          synchronized (closeLock) {
162              if (!open)
163                  return;
```

(a) Source Code

The type `new AbstractInterrruptibleChannelInterruptible()` must implement the inherited abstract method `new AbstractInterruptibleChannel.Interruptible.interrupt()`

(b) Text Description

Figure 4.3 An Eclipse compiler notification about unimplemented methods (CMP5).



(a) Source Code with Highlighting

1 of 2 branches missed

(b) Text Description

Figure 4.4 An EclEmma notification about partial branch coverage (ECL3).

uses highlighting to indicate code execution; code highlighted in green was executed, red was not executed, and yellow was partially executed. Figure 4.4 shows an example of coverage reported by EclEmma on an `if` statement. When the developer mouses over the icon, the tool notifies her of how many paths got executed on the associated branch statement at line 133 (Figure **??**).

These tools may seem quite different, but I chose them specifically to identify challenges developers experience *across* tools. Despite the differences, these tools attempt to communicate similar concepts to developers using similar textual and visual notifications. For example, both FindBugs and EclEmma communicate information about control flow, and both FindBugs and the Eclipse Java Compiler communicate about data flow. All three tools use color codes in a largely consistent manner, such as using red to indicate the highest level of urgency. And as a final example, most notifications communicate information about program elements, such as methods and classes, and information about program execution, be it potential or actual.

4.1.3 Study Protocol

Each session with a participant lasted approximately one hour. Prior to each session, participants filled out a consent form and pre-questionnaire.[1] Each session consisted of seventeen tasks.

Source code for the tasks came from OpenJDK [**OpenJDK**] and JFreeChart [**JFreeChart**]. I chose Open JDK because it has a large code base from which I could easily find bugs using their publicly available FindBugs cloud report [**FindBugsCloud**]. I chose JFreeChart because it is a large code base with working JUnit test cases that exhibit less-than-perfect code coverage.

For each task, I presented participants with and asked them to interpret one or more notifications from a given tool. I disallowed the use of a web browser to isolate the challenges developers encounter to the notifications used by the tools and to exclude challenges caused by outside tools or resources. Allowing use of the browser would have added data that does not help answer the current research

[1] This work was approved under IRB No. 2787.

Table 4.1 Notifications used in our study

Notification	Tool	Problem	Category
FB1	FindBugs	String comparison using == or !=	Pointers/References
FB2	FindBugs	Incorrect Lazy Initialization	Multi-threading
FB3	FindBugs	Synchronize on mutable field	Multi-threading
FB4	FindBugs	Redundant null check	Null/Pointers/References
FB5	FindBugs	Possible null pointer dereference	Null/Pointers/References
CMP1	Eclipse Compiler	Unused code	Dead Code
CMP2	Eclipse Compiler	Unchecked Conversion, Raw Type	Generics
CMP3	Eclipse Compiler	Unimplemented methods	Inheritance/Polymorphism
CMP4	Eclipse Compiler	Serializable class needs serial ID	Serialization
CMP5	Eclipse Compiler	Unimplemented methods	Inheritance/Polymorphism
CMP6	Eclipse Compiler	Method not applicable for arguments	Inheritance/Polymorphism
ECL1	EclEmma	Red class with red class header	Class/test coverage
ECL2	EclEmma	Red class (constructor only)	Class/test coverage
ECL3	EclEmma	Simple if statement	Branch/test coverage
ECL4	EclEmma	Return statement with branches	Branch/test coverage
ECL5	EclEmma	Try/Catch/Finally (coverage varies)	Test coverage, Exception handling
ECL6	EclEmma	Nested if statements	Branch/test coverage

```
46    private Vector<String> _contexts;
47
48    public ContextListImpl(org.omg.CORBA.ORB orb)
49    {
50        _orb = orb;
51        _contexts = new Vector(INITIAL_CAPACITY, CAPACITY_INCREMENT);
52    }
```

(a) Source Code

- Type safety: The expression of type Vector needs unchecked conversion to conform to Vector<String>.
- Vector is a raw type. References to generic type Vector<E> should be parameterized.

(b) Text Description

Figure 4.5 A notification from the compiler about generics (CMP2).

question. I also wanted to see if developers could interpret tool notifications without the aid of outside resources. During many tasks, and at least once for every participant, participants discussed or completed notification resolution. I did not require them to do so as the focus of this study was on the ability to interpret, not to resolve. As participants explained the notifications, the first author asked follow-up questions as necessary.

Table 7.1 shows a list of the notification tasks participants encountered during each session. A more detailed listing, with screenshots, of the notifications used in this study can be found in Appendix B. For each task, I chose notifications to represent the types of notifications developers may encounter when programming. For FindBugs and the Eclipse compiler, I chose notifications that appeared frequently in the OpenJDK project. I chose EclEmma notifications from JFreeChart to exercise a range of its coverage scenarios. Because EclEmma's documentation did not specify the range of notifications it uses, I manually went through JFreeChart's codebase after running the tool and took note of each new coverage scenario encountered. I then included an example of every coverage scenario in the EclEmma tasks.

For FindBugs, each task during the session corresponded to a single notification. All but one compiler task corresponded to a single notification; because the two notifications on CMP2 (Figure 4.5) contribute to the same problem on the same line, I presented them as one task. Each EclEmma task consisted of participants explaining coverage notifications for the entire class.

4.1.4 Data Collection

I recorded audio and the screen in each session for analysis. I then created transcripts from the audio, and included descriptions of actions that a participant performed that were relevant to interpreting the notification. For example, if a participant navigated to different parts of the code but did not explicitly describe it, I added a description of that navigation to the transcript.

4.1.5 Data Analysis

I analyzed each session using open and selective coding [**corbin2014basics**] to discover participant challenges. To identify a challenge, I needed concrete criteria. I proposed that tool use is a form of communication, and therefore that challenges when interpreting a notification can be seen as ineffective communication. Existing research on how computers should talk to people suggests that if an explanation is required for a message to be understood, the message was not effective [**dean1982computer**].

I used this logic to determine when a challenge occurred, using three criteria for inclusion:

1. The participant explicitly stated a challenge.
2. The participant was unable to explain the notification.
3. The participant had to take steps, outside of reading the notification, to deduce the problem.

Whether an observation met a criterion is independent of whether the participant was able to explain the notification.

I and a collaborator individually used open coding on each transcript, labeling portions that mapped to a challenge. We then reconvened to merge our codes. The criteria above guided this process; if we could not agree that a code fit our criteria, we removed it from our data set. Of the 404 codes we originally extracted, we disagreed on 82 (20%) from twenty-six sessions. To resolve our disagreements, we referred to our criteria; if we could not come to an agreement regarding the code fitting the criteria, we removed the statement from our data set. For four sessions, we had no disagreement. In the end, we identified 322 codes. We put each code onto a note card, along with the participant and tool being used.

Next, I used a card sorting methodology similar to that of Muşlu and colleagues [**Muslu:2014:Transition**]. The goal of this card sort was to identify themes based on the identified codes. I used five of the eight authors on the publication of this work and completed the card sort in three phases. In phase 1, we sorted all cards into high-level themes; each card could only go in one theme. s Phase 2 focused on determining where high-level themes could be broken down into lower-level themes. In phase 3, we focused on making sure that each card was in the best fitting theme. During this phase, we also clarified theme definitions and made note of example statements to represent each theme.

Because one of the criteria is participant inability to explain a notification, any actions or statements made surrounding that occurrence was included in the card sort. Upon reflection, some emergent themes took the form of consequences rather than challenges, such as notification resolution without understanding and lack of trust in the tool, therefore I will not discuss them in

this chapter. I likewise will not discuss the emergent theme of tool feature requests. These excluded themes are available with the other on-line research materials.

4.1.6 Study Credibility & Findings Validation

There are inherent threats to the validity of empirical research [**onwuegbuzie2007validity**]. Despite these inherent threats, prior research suggests there are ways we can increase confidence in the credibility and validity of empirical findings [**gasson2004rigor**; **li2004trustworthiness**]. Following the safeguards for conducting empirical research proposed by Li [**li2004trustworthiness**], I ensured the following in the collection, interpretation, and reporting of the data I collected:

- *Voluntary participation and anonymity.* To receive truthful responses from participants, I provided participants up-front with information regarding the purpose of the study, what will happen with the data, and how anonymity will be ensured.
- *Purposeful sampling.* To sample with the purpose of gathering diverse participants and to increase the ability to generalize findings, I recruited participants from academia and industry with varying levels of programming experience.
- *Triangulation.* To increase reliability, I triangulated data from direct observation and think aloud.
- *Prolonged engagement.* To allow participants time to get acclimated to a researcher being present while not getting too fatigued to contribute data, each session lasted about one hour. To increase the effectiveness of this safeguard, the researcher interrupted as little as possible.
- *(Near-) Natural situation.* To increase ecological validity, I set up the study environment and recruited participants familiar with that environment and programming language. I also allowed participants to explore the code as they would if it were their own.
- *Peer debriefing, stepwise replication, and interrater reliability.* To ensure researcher agreement about the findings, two authors separately analyzed the transcripts for statements of interest. I also included multiple researchers throughout the multi-step analysis and reporting process.
- *Member checks.* To ensure validity of the data and our interpretation, I reached out to all participants, providing them with a summary of the findings, a copy of the written report, and a form for providing feedback on the findings.
- *Thick description.* To enable judgment of how my research fits with other contexts, we describe in detail the methods used to collect the data and the setting in which it was collected.

Other safeguards include *Training for subjects*, *Background checks*, and *Refrain from generalizing*. I did not conduct training for think aloud, as it could have affected my ability to recruit participants. I used criteria for participation as a background check and do not generalize outside the context of software developers.

4.2 Knowledge-Related Challenges

Remember Valerie from Chapter 1? Though she is a hypothetical developer, the challenges she faced are not hypothetical. Valerie experienced challenges caused by both knowledge gaps (no knowledge regarding lazy initialization) and knowledge mismatches (expecting an explicit connection to synchronization). Because research suggests experience matters when understanding vulnerabilities, and that experiences affect knowledge, I speak about knowledge here and throughout this thesis as the culmination of experiences [**johnson1989mental**; **argote2011organizational**; **baca2009static**]. Using that definition, a *knowledge gap* occurs when there is a gap between what the developer knows, based on her experiences, and how the tool communicates; a *knowledge mismatch* occurs when what the developer knows and expects from the tool, based on her experiences, does not match the notification the tool uses.

The challenges that comprise my theory are shown in Figure 4.6. Vertical lines represent the tasks and the horizontal bars indicate challenges. The area of the dots indicate how many participants encountered challenges with that notification in that theme. Diagonal lines map participants to the challenges interpreting that notification. When opened in Adobe Acrobat, clicking "Hide/Display Details" interactively toggles between showing and hiding this mapping. I describe each challenge type, and validation of the findings, in detail in the remainder of this section.

4.2.1 Knowledge Gaps

Knowledge gaps occurred when there was a gap between what participants know and the information provided by the notification. Knowledge gap challenges occurred when participants did not have existing knowledge of software development activities relevant to a given notification. However, we found it is not as simple as "beginners battle and experts excel", but instead that challenges faced by developers can occur regardless of programming or industry experience. In this subsection, I will describe general knowledge gap challenges, followed by four specific kinds of knowledge gaps identified in this study.

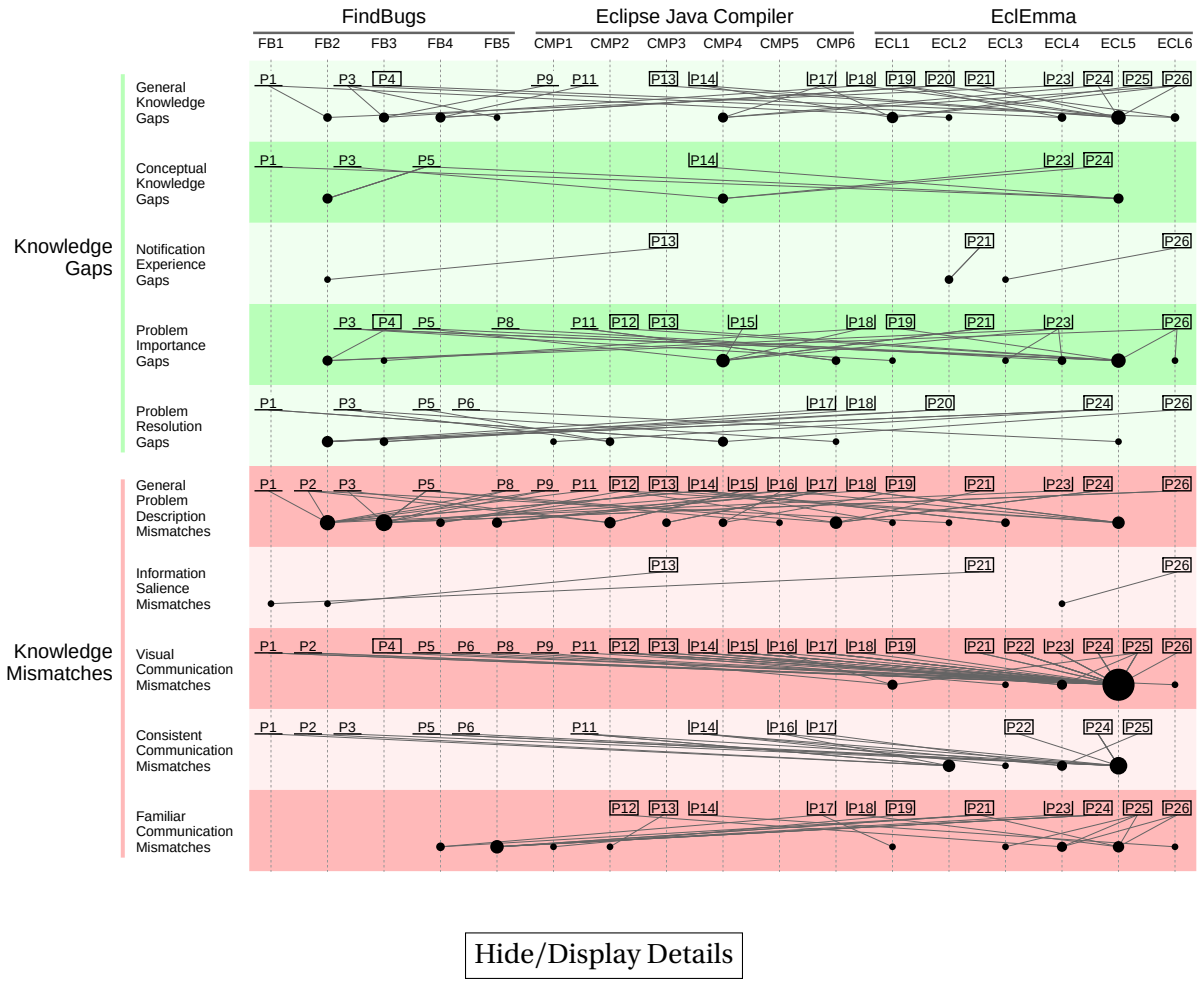


Figure 4.6 Distribution of challenges encountered and notifications that caused them.

4.2.1.1 General Knowledge Gaps

General knowledge gap challenges occurred when there was a gap between the general software development knowledge participants had relevant to the notification and the information provided by the notification. When participants did not provide enough information to map a challenge to a more specific kind of knowledge gap, I placed that challenge in this theme. Participants experienced knowledge gap challenges across all three tools (Figure 4.6).

FindBugs was more dominant in this theme than the compiler, with 9 and 2 participants encountering challenges respectively. This was the case, despite the compiler using less text than FindBugs to communicate. Participants focused on the text to understand the problem, but struggled to un-

derstand what the tool was trying to convey. Despite FindBugs' verbosity, as stated by 4 participants, the tool provided just enough to need to use the web to figure out the problem. For example, P25 struggled to interpret FB3. He made an effort to understand the notification, but then realized the notification did not provide enough for him to feel confident in his explanation, stating:

> *I would definitely want to correct it but I don't get enough info from here to know what to correct or what I did wrong so I would probably take this message and go to Google to see if anybody else is talking or saying something that I understand better.*

Participants who struggled with compiler and EclEmma notifications found themselves in a situation similar to P17 when interpreting CMP4 and notifications in ECL1. When he encountered CMP4 and ECL1, he immediately realized the notifications did not provide enough information for him to come to a conclusion about each. He noted, like P25, that he would need to use Google or documentation to better understand the notification being provided. P17 was unable to come to a conclusion regarding either notification.

These findings confirm that more is not always better [**Nienaltowski:2008:Compiler**] for closing knowledge gaps and that these gaps exist with visual communication as well. Along with general knowledge gap challenges, participants experienced challenges caused by four specific types of knowledge gaps that emerged: *conceptual knowledge gaps*, *notification experience gaps*, *problem importance gaps*, and *problem resolution gaps*.

4.2.1.2 Conceptual Knowledge Gaps

Conceptual knowledge gaps occurred when there was a gap between participants' knowledge of programming concepts, like serialization, present in the notification and the information provided by the notification regarding those concepts. P24, a professional developer with 15 years of experience, attempted to work through CMP4 despite his unfamiliarity with serialization. His guess, based on the notification, was that he was missing a `serialversionUID`; however, beyond that he was unsure how a `serialversionUID` is associated with serialization. This led to the inability for P24 to fully interpret the notification.

P5 encountered challenges interpreting FB2 due to conceptual knowledge gaps regarding multi-threading. The notification spoke about concepts such as lazy initialization, which P5 noted he had not had past experience with. herefore, he could only guess what was wrong with the code.

Conceptual knowledge can also affect visual communication, even when the relevant concepts are not depicted in the notification. Test coverage is the obvious concept necessary to understand test coverage notifications. Some of the notifications participants encountered from EclEmma

required knowledge of other concepts, such as exception handling. Three participants noted they could not confidently explain EclEmma notifications involving `finally` blocks using the visuals provided due to their minimal experience with `finally` blocks.

After completing ECL5, most participants could at least vaguely explain the notifications they encountered. However, P5 still could not definitively conclude anything about the notifications, stating:

> *I don't know what* `finally` *means but it seems like everything inside* `try` *is not getting called…I assume* `finally` *is similar to* `catch` *but I don't really know how* `finally` *works.*

His lack of knowledge regarding `finally` blocks made it challenging for P5, despite his familiarity with other relevant code structures[2]. This, coupled with being his first experience with EclEmma notifications, led to his inability to interpret the notifications in ECL5.

4.2.1.3 Notification Experience Gaps

Notification experience gaps occurred when there was a gap between participants' knowledge gained from experience with a notification they encountered for the first time and the notifications they have previously encountered. Participants' lack of experience with a notification is the knowledge gap that caused challenges in this theme. For example, P21 struggled to interpret the notifications in ECL2 due to the differences in highlighting on uncovered methods and constructors. His comments suggested that he understood the concept of coverage, but stated that the challenge was due to unfamiliarity with the tool. When he first encountered an uncovered method notification, without the signature highlighted like a constructor's signature was, he could not determine whether *lack of highlighting* was equivalent to *red highlighting*. The challenges in this theme are general in that they relate to overall notification knowledge. Some of the challenges that emerged relate to gaps in knowledge regarding notification specifics, such as importance and resolution.

4.2.1.4 Problem Importance Gaps

Problem importance gap challenges occurred when there was a gap between participant knowledge of the importance of the problem and the notification's attempt to communicate importance. As P18 attempted to explain FB3, he realized that although the notification did tell him that he

[2]The reader may also find this confusing, but this was a design decision made by EclEmma's toolsmiths. This confusion arises from a difference between the bytecode representation and the source code representation of `finally` blocks (`https://github.com/jacoco/jacoco/issues/15`). Although this may seem like a design problem, we included the notifications we did, including ECL5, because they are encountered in the wild.

was synchronizing on a mutable field, it did not tell him why that is undesirable. He attempted to determine a reason for why it is undesirable, and though he found the notification's message "unlikely to have useful semantics" helpful, he noted that his reasoning "would not be correct" because he would have to guess.

Without an understanding of why the problem was bad, participants could not confidently interpret the notification; this led to challenges coming up with resolutions. For example, P5 could not confidently resolve CMP4; the notification was clear that a missing `serialversionUID` is the problem, but did not specify why the ID was needed. Though the compiler provided quick fixes, deciding which fix was best for P5 depended on what the ID was used for, which the notification did not specify.

4.2.1.5 Problem Resolution Gaps

Problem resolution gap challenges occurred when there was a gap between what the participant knows about resolving a notification and the resolution suggested by the notification. Most often this gap was present because the notification did not include information specific to notification resolution. When participants did not know how to fix a notification, they had to guess how they might fix it or, as P20 noted, "Google it to make sure" they fully understood the notification and how to fix it. The downside to this approach is that it takes developers into a form of information foraging that involves leaving their working context [**Altmann:2004:Task**], which P13 explicitly stated:

> *Anything that deviates my train of thought from the task at hand... that's the last thing you want when writing code.*

The notifications that did provide a fix description did not provide a clear description of the fix or how to apply it; without the required knowledge, filling this gap was difficult for participants. This was most often the case with the compiler, which provides quick fixes with minimal explanation attached. Two participants struggled with understanding and resolving CMP4. Both appeared confident that something was missing and that they should add the `serialversionUID`. However, neither knew what a `serialversionUID` was or how they should have used it.

Sometimes notifications provided multiple options for resolution but did not provide information regarding which resolution was most appropriate. This left participants with the task of determining the best fix to apply. For example, CMP4 offers multiple fix possibilities, each with its own set of code changes and possible side effects. P26 spent time sorting through and discussing the options for fixing CMP4. Because the tool did not provide information regarding the pros and cons of a each fix, he was unable to explain how to resolve the notification.

4.2.2 Knowledge Mismatches

Knowledge mismatch challenges occurred when there was a mismatch between how participants expected a notification to communicate, based on their knowledge, and how the notification communicated. Unlike knowledge gap challenges, participants had knowledge relevant to the notifications and concepts. However, they encountered challenges when attempting to use their knowledge to interpret the notification. As with knowledge gaps, I describe general mismatch challenges, followed by discussion of four specific kinds of knowledge mismatches identified in this study.

4.2.2.1 General Problem Description Mismatches

General problem description mismatches occurred when there was a mismatch between the way the participant would textually describe the problem and the description provided by the notification. Although other challenges relate to notification text, for General Problem Description Mismatch challenges, it was unclear what about the description participants found confusing. However, it was clear that the text was not communicating in a way that participants could use their knowledge to reconcile. This is related to research on compiler messages conducted by Traver that suggest unambiguity of language is important [**Traver:2010:Messages**]. Similarly, O'Neil discussed the importance of language considerations in data breach notifications [**o2015target**].

Representative of textual communication mismatch challenges was P16's experience interpreting FB5. After he read the text provided by the notification, P16 was unable to come to a definite conclusion regarding the problem, stating:

> *It didn't confirm or deny what I thought because the wording of the [tool tip] was not quite how I would have described it...*

Participants encountered similar challenges with the compiler. P17, for example, went back and forth between the text of CMP3 and information provided via quick fixes as he tried to understand the problem. He was able to guess, based on his knowledge, what the problem might be but moved away from the text of the notification to come to any sort of conclusion about the problem being communicated.

For some participants, the language notifications used was familiar but not something they could quickly recollect. P3, for example, saw the word "mutable" in FB3 and he could not remember what mutable meant. After P5 read the text provided for FB3, he explained that the use of the phrase "useful semantics" may not have been the best choice as, for him, terms like this "have different meanings in computer science and the real world."

Similarly, P5 and P24 struggled due to ambiguity in the language used. P5 found the overall phrasing of CMP6 "weird." P24 was more specific in stating how the language was ambiguous. He found the use of the word "applicable" to be odd in this context and not clearly indicative of the message he assumed the tool was trying to communicate.

Although these textual mismatch challenges encountered are general, specific types of mismatches with the text portion of the notification emerged: *information salience mismatches*, *visual communication mismatches*, *consistent communication mismatches*, and *familiar communication mismatches*.

4.2.2.2 Information Salience Mismatches

Information salience mismatches occurred when there was a mismatch between the information a participant thought was most relevant and the information the notification made salient. This aligns with McCrickard and Chewar's suggestion that general computer users are dissatisfied with notification systems because of mismatched information prioritization [**mccrickard2003attuning**].

Representative of these challenges was P13's attempt to interpret FB2 (Figure **??**). P13 read the tooltip for FB2 but did not find anything useful; he saw "update of static field" but was not certain what the tool was trying to communicate. After digging deeper, he found that the tool eventually elaborated on what is wrong with where and how a field is set when working with threads. This was what he was looking for, as it helped him understand why it was a "*very serious* multi-threading bug," as the notification stated. For him, and the other participants who encountered challenges in this theme, the most easily available information was not so useful, leaving them unsure of what the problem was in the code and why it was a problem. The critical pieces of information, such as that it is a multi-threading problem concerning where synchronization is placed, got buried.

I only observed this phenomena with more experienced developers; this suggests that more experienced developers may have more concrete expectations of what information tools should provide. On the flip side, less experienced developers may not know when important information is buried because they are unsure of what the important information is. Therefore, less experienced developers did not appear to encounter challenges in this theme.

4.2.2.3 Visual Communication Mismatches

Visual communication mismatches occurred when there was a mismatch between how the participant would communicate with other developers about the notifications and the visual elements used by the notifications. For these challenges, it was clear there was a mismatch between what partici-

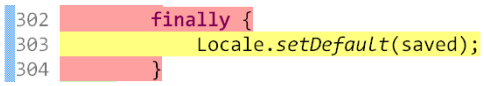


Figure 4.7 A notification from EclEmma regarding `finally` coverage (ECL5).

pants expected and what the tool presented them with, but there was no indication by participants of what specifically caused the mismatch.

For fourteen participants, EclEmma's attempts to communicate `finally` block coverage in ECL5 (Figure 4.7) failed because it was not obvious, based on their mental model of how `finally` blocks work, how a `finally` block can be missed or the code inside a `finally` block could be partially covered. For example, P24 had expectations regarding how EclEmma might communicate coverage of a `finally` based on prior experience with the construct that suggests it always executes. Rather than exploring more, P24 noted he did not understand the way the tool communicated.

Seven participants had expectations regarding how `try` blocks work that did not match how EclEmma reported `try` block coverage (ECL5). For example, as P23 sorted through the notifications in ECL5, he wanted to know which line failed to cause the `try` block to not execute. Attempting to interpret the notification, he stated:

> *In order for the* `catch` *statement to be activated I would imagine that this code had at least been evaluated.*

His expectation, based on his knowledge of the code construct, was that if the `try` did not execute, there was a line of code at fault. However, contrary to his expectations, EclEmma highlights the entire `try` block red if an exception is thrown, which makes it unclear whether the `try` executed at all, and if it did, where an exception was thrown.

Five participants got confused by EclEmma's lack of textual information. Participants probably noticed this because both FindBugs and the compiler provide supplemental textual information when markers, similar to the ones provided by EclEmma, are clicked; in fact, markers and notifications from other tools within EclEmma's interface sometimes distracted participants looking for information regarding code coverage. As P4 accessed the information provided by notifications in ECL6, he noticed and explored the availability of multiple markers that provided information. Some of these markers came from other tools; none provided P4 with "any details about the coverage part," so he was not sure why they were present.

4.2.2.4 Consistent Communication Mismatches

Consistent communication mismatches occurred when there was a mismatch between the consistency expected by the participant and the inconsistencies in how the notifications communicated similar problems. Prior research suggests that within-tool-consistency is an important factor for developers when interpreting and addressing compiler messages [**Traver:2010:Messages**]. The results in this category suggest that experiences affect perception of consistency and that this phenomena generalizes to visually-enriched notifications in other types of tools.

Five participants encountered challenges caused by inconsistencies in how EclEmma reports coverage on branching structures. Under the assumption that yellow highlighting was accompanied by a textual description (i.e. 1 of 2 branches missed), participants often struggled to interpret notifications like the one in Figure 4.7. P6, among others, spent a significant amount of time during her session trying to interpret the notifications in ECL5. When she realized that there were no markers available to better explain partial coverage inside a `finally` block, she began looking at the other similar notifications in ECL5. When she realized that none of the other notifications had what she was looking for, she summarized why she was struggling, stating "I'm not sure what the other option could be…it doesn't have the little yellow diamond on it."

Six participants noticed inconsistencies in how EclEmma reported coverage on non-branching code structures. Five of the six encountered challenges interpreting notifications on methods and constructors. EclEmma highlights the constructor signatures to indicate a missed constructor, however, does not highlight a method signature when it is not executed. For example, during P14's session, he did a lot of back and forth between EclEmma tasks to compare notifications. As he tried to interpret the notifications in ECL3, he reflected on and revisited ECL1 and ECL2, where he recalled there being class, method, and constructor coverage. He remembered the inconsistencies with how ECL1 and ECL2 communicated coverage on these constructs and found it to be confusing. Therefore, he could not give a definite interpretation of any of the three.

4.2.2.5 Familiar Communication Mismatches

Familiar communication mismatches occurred when there was a mismatch between participant familiarity with the methods a notification uses to communicate about programming concepts and the methods the notification used to communicate about programming concepts When participants encountered these challenges, they often noted lack of familiarity or the inability to easily recognize the problem. Participants that noticed unintuitive communication techniques found some for all tools. The majority of participants (five of eleven) stated that EclEmma's dominant use of color to communicate code coverage was not intuitive. For example, participants did find it intuitive to use

yellow for partial coverage in notifications like the ones in ECL3, ECL5, and ECL6. The common problem with the other tools involved association of the notification to the root cause and unintuitive fix descriptions.

4.2.3 Member Check

To assess the validity of how I interpreted the data, and the experiences developers have when interpreting tool notifications, we conducted a member check. Of the seven responses received, two developers agreed with these findings and five strongly agreed. Many found the report "interesting," some noting that although they may not have experienced all of the challenges during the study, they can recall previously encountering such challenges. When asked which challenges they can relate to the most in their experiences with tools, the most common choice was Problem Resolution Gaps (5). The second most common responses (4) include Notification Experience Gaps and Information Salience Mismatches, followed by the third most common responses (3) of Conceptual Knowledge Gaps, Visual Communication Mismatches, and Familiar Communication Mismatches.

4.3 From Theory to Practice

Current tools do not support developer knowledge gaps (Section 4.2.1) and sometimes conflicts with existing developer knowledge (Section 4.2.2). In this section, I discuss several implications of the findings from this study.

4.3.1 Filling Developer Knowledge Gaps

Despite the experience of some participants, every participant encountered at least one notification they did not understand. FindBugs, the Eclipse Java Compiler, and EclEmma attempt to fill knowledge gaps to different degrees and in different ways. FindBugs sometimes provides definitions, examples, and fix suggestions. The compiler provides tooltip descriptions and often an automatic quick fix that developers can apply to learn about notification resolution. EclEmma sometimes provides tooltips to help developers fill knowledge gaps concerning low test coverage.

One straightforward solution is for tools to provide more information to developers to help fill knowledge gaps. For example, for developers that struggled with `finally` block coverage in ECL5, it may have been helpful if the tool provided information regarding `finally` block coverage in EclEmma. Or, for developers who did not know what synchronization is, it may have been helpful to provide a definition or code example of what it means to correctly synchronize an object or method.

Findings from this study suggest tools can fill developer knowledge gaps by consistently providing information about the options for fixing a notification (Section 4.2.1.5) and reasoning for resolution (Section 4.2.1.4). The Eclipse compiler makes a consistent effort to provide fix information, however, it does not make an explicit effort to assist developers with deciding the *best* fix their code nor does it provide rationale for resolution. Muşlu and colleagues provided one potential solution for compiler notifications with QUICK FIX SCOUT [**Mucslu:2012:Speculative**]. Although this approach could be applied to other tools that offer quick fixes, like FindBugs, QUICK FIX SCOUT prioritizes and rationalizes based on one criteria: the number of new notifications introduced by applying the fix. However, other criteria, such as whether the fix uses familiar APIs, may also improve the usability of program analysis notifications.

4.3.2 Matching Developer Expectations

Developer expectations can have an effect on their ability to interpret notification messages (Section 4.2.2). I propose that tools can improve how they communicate to developers if they are able to ascertain developers' knowledge and experiences, which inform their expectations [**dean1982computer**]. For each notification in this study, some developers could interpret the notification and others could not. Therefore, it may be that providing every developer with more information is not the best way to support developers' understanding of tool notifications.

If a tool could know its user's familiarity, or unfamiliarity, with the notifications it provides, or the concepts in those notifications, the tool could determine how to adapt its notifications to better fit the user's expectations. However, tools cannot acquire the knowledge required to build these constructs on their own. One potential solution, modeled after intelligent tutoring systems (ITS) [**tutoringsys**], which I explore later in Chapters 5 and 6, is for tools to use developer knowledge, in the form of their experiences, as a factor when determining the information necessary for a developer to interpret a given notification.

Imagine two developers, D1 and D2; D1 frequently develops in multi-threaded environments while another, D2, is new to multi-threading. For multi-threading experts, like D1, extra information regarding terms and fundamental concepts, such as lazy initialization, may not be necessary. It may be enough to notify her and provide quick access to a suggestion or example for resolving the problem; it may even be distracting having other information available she likely does not need. For multi-threading novices, like D2, all the information provided could be of use; such novices may need even more information.

ITS create student models based on assessments; we imagine IDEs could construct a model of a developer's experience by observing their use of language features, tools, and libraries in the code

they write. This is similar to the design of other kinds of notifications [**mccrickard2003attuning**; **sow2005tasks**; **Zhang:2005**] and aligns with research on recommendation systems that suggests data mining and other knowledge inference techniques can help provide previously-unknown information for task completion [**robillard2014recommendation**]. There may be factors other than their coding experience to consider for accurate models. Other data we can collect include notifications the developer has resolved or portions of the notification text frequently visited or used by the developer.

In order to adapt tool notifications to a given developer's knowledge, there needs to be some notion of how much the developer knows about the concepts in the notification. For the remainder of this document, I use concept to mean programming concepts. I chose to focus on programming concepts because the findings from this study, and existing research conducted by Smith and colleagues, suggests understanding programming concepts affects developers' ability to understand and resolve notifications [**smith2015questions**]. Borrowing from education research and using developer experiences as a concrete representation of knowledge, the remainder of this thesis evaluates the possibility to ascertain and approximately predict developer knowledge of programming concepts.

CHAPTER

5

ASSESSING DEVELOPER KNOWLEDGE

In the previous chapters, I proposed that communication between notifications and developers could be improved if the information provided by notifications could adapt to the developer based on the developer's knowledge. For it to be possible to adapt the information provided by notifications, tools need a way of knowing what concepts the developer knows and does not know. Furthermore, tools also need a way of knowing how well developers know these concepts.

Research in adaptive user interfaces prove models that represent some aspect or characteristic of the user, that tools then use to adapt their interface accordingly [**murray1999authoring**; **schlungbaum1996model**]. The theory posed in Chapter 4 suggests that developer knowledge can affect their ability to effectively use program analysis tools. Therefore, it could be beneficial for tools to have access to models that can predict developer knowledge based on their activity.

In order to make tools aware of developer knowledge, I need a ground-truth way of representing developer knowledge that can be eventually be used as a dependent measure of knowledge in a predictive model. Borrowing from existing computer science education research, I developed concept inventories for knowledge assessment.

A concept inventory, or CI for short, is a validated assessment that uses multiple-choice questions to aid instructors in assessing student understanding of relevant domain or course concepts, while also identifying student misconceptions [**evans2003progress**]. Concept inventories

can cover any range of concepts and originated in the field of physics with the Force Concept Inventory [**hestenes1992force**]. There are guidelines for creating concept inventories, such as avoiding catch-all options like "All of the above" and using empirical methods to validate and improve inventory questions and items. Most often in the natural sciences and engineering, inventories assess breadth of knowledge across the subject or course of interest [**evans2003progress**]. In Computer Science, for example, a range of introductory programming concepts can be represented in a given concept inventory [**tew2010developing**].

Borrowing from concept inventories used in other STEM fields, CS Education researchers and instructors have developed various concept inventories for assessing students' knowledge of introductory Computer Science concepts [**almstrum2006concept**; **krone2010reasoning**; **tew2010assessing**]. Almstrum and colleagues created concept inventories to assess student knowledge of concepts taught in a discrete mathematics course [**almstrum2006concept**]. Tew and Guzdial developed a concept inventory for assessing students' knowledge of introductory concepts in introductory Computer Science courses [**tew2010assessing**; **tew2010developing**]. These inventories used textbooks and experts to create test specifications that they then validated using empirical methods, such have CS Education experts review the specification. Fundamental concepts included on their concept inventory include object-oriented programming and control structures. Similar to other Computer Science concept inventories, Tew and Guzdial's introductory concept inventory assess breadth of knowledge.

Similar to the concept inventories developed by Tew and Guzdial, Zingaro and colleagues developed ConcepTests [**zingaro2010experience**] used in to monitor and assess student understanding of computer science concepts as they were taught. Similar to my concept inventories, these ConcepTests focus on one specific concept at a time except with the goal of encouraging and implementing peer instruction (PI). In contrast, Zingaro and colleagues do not use the validated steps for creating a concept inventory to create their assessments.

Also closely related to my approach and the goal of my inventories is the work of Karpierz and Wolfman, who designed a variety of multiple choice questions for concept inventories on binary search trees and hash tables [**Karpierz2014Misconception**]. Rather than focusing on various Computer Science concepts at once, they focus on developing questions to assess specifically binary search trees and hash tables. Contrary to our approach, they only deployed their concept inventory once – during the last lecture of the course. This eliminates the possibility of comparing how much students knew before taking the course and how much they know now, which I assess with my concept inventories.

Although these concept inventories have been proven useful in various contexts, some instructors believe, and research suggests, we should be moving away from textbook-driven lessons

and assessments, and employ more focused efforts to monitor student learning and understanding [**pellegrino2001knowing**; **ghezzi2005challenges**; **wohlin1999achieving**]. This work builds on existing work by adapting current methods used to develop concept inventories by 1) assessing depth of knowledge regarding a single programming concept, 2) using language specification, tutorials, and other technical documents to create the content of the concept inventories, and 3)incorporating practical software engineering concepts, such as tool use.

5.1 Modified Concept Inventories

Traditionally, concept inventories are used to assess programming knowledge across concepts; in Computer Science, the target audience is typically introductory programming students [**tew2010developing**; **kaczmarczyk2010identifying**]. For the remainder of this document, I will refer to knowledge of programming concepts as *conceptual knowledge*. In order to be able to evaluate a range of expertise with the concept inventories, I evaluated the potential for a depth versus breadth approach to assessing conceptual knowledge, that incorporates software engineering concepts and practices, with the target audience being developers at any stage of expertise. Therefore, I could not borrow directly from research that uses a breadth approach to create concept inventories that assess high level knowledge of concepts taught in an introductory course [**tew2010assessing**]. I extended existing concept inventory research [**tew2010developing**; **nelson1967testing**] by using the new Bloom's Taxonomy to create questions [**scott2003bloom**; **thompson2008bloom**; **starr2008bloom**], subjective resources like language specifications to determine sub-concepts relevant to a single concept, and tool output, including compilation errors and warnings, to integrate practical applications of the programming concepts. The differences between my approach and existing approaches is highlighted in Table 5.1. The final process I have developed for creating general knowledge programming concept inventories that assess depth of knowledge is as follows:

1. Define Conceptual Content for Test Specification
2. Build Bank of Test Questions
3. Pilot Questions
4. Establish Validity and Reliability

Table 5.1 Summary of the differences between my approach and existing approaches.

	Existing CS Concept Inventories	My Concept Inventories
Defining Concept Content for Test Specification	Textbooks; provide definitions to CSEd experts for review	Language specifications, language tutorials, and documentation
Build Bank of Test Questions	Three types of questions (Definitional, Tracing, Code Completion)	Six types of questions (Bloom's Taxonomy), tool output, compilation
Pilot Questions	Collect data for later analysis	Iteratively conduct think aloud and modify
Establish Validity & Reliability	Empirical analysis of responses	Item and distractor analysis

5.2 Defining Conceptual Content

A test specification is a way of formally outlining what will be on the test without having to write any of the questions [**tew2010developing**]. Once I decided on the programming language and concept(s) of interest, rather than using expert review to determine appropriate sub-concepts concepts, use a more objective, practical approach. This is achieved by using language specifications, tutorials, and other documentation surrounding that concept. This is an iterative process meant to increase the depth of assessment on the concept and related sub-concepts.

To determine the conceptual content for each concept inventory, I identified key concepts from the most up to date the Java Language Specification (JLS) [**Gosling:1996:JLS:560667**] and the Oracle Java Concept Tutorials[1] on the concept of interest. The Oracle Java Tutorial was most useful for finding and mapping the relationships between concepts and ancestor concepts; if Lesson X built on top of Lesson Y, we consider Lesson Y an ancestor of Lesson X. For example, based on the generics tutorial, we labeled Upper Bounded Wildcards as an ancestor concept to Wildcards.

Once I had a list of concepts and sub-concepts, I created questions to assess the various concepts, mapping each question to at least one level of Bloom's Taxonomy. I used examples found in the language specifications and tutorials, along with relevant tool output, to create questions for each inventory. For example, the item shown in Figure 5.1 is a question on the generics concept inventory that asks a question to determine the student's ability to **evaluate** a problem pertaining to instantiating a generic type (`Bucket`). According to the new Bloom's Taxonomy, evaluation includes Checking and Critiquing, which involves students making decisions based on criteria. In terms of Computer Science, prior works suggests this can be done by, for example, assessing students' ability

[1] `http://docs.oracle.com/javase/tutorial/java/generics/`

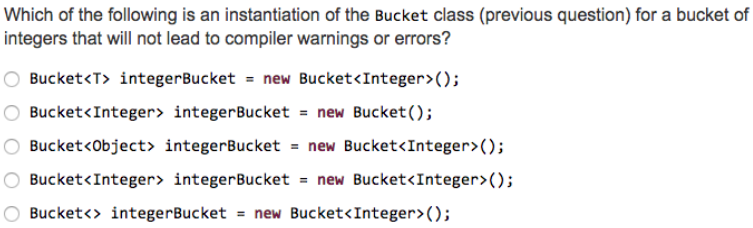


Figure 5.1 Question assessing ability to evaluate generic type instantiation.

to determine if a piece of code satisfies requirements (i.e. compilation) or critiquing the quality of the code based on known standards [**thompson2008bloom**].

5.3 Building A Bank of Questions

Based on existing concept inventory research, the questions on concept inventories should be multiple choice and all questions should have the same number of items to choose from [**tew2010developing**]; typically there are 4–5 items. Existing work also recommends avoiding catch-all items such as "All of the above" and "None of the above" [**tew2010developing**]. While one of the items should be the best right answer, other options, known as distractors, should be plausible permutations that are believable but incorrect.

Next, using the revised Bloom's Taxonomy, which is the most up to date version of the taxonomy, is to derive a bank of test questions. Using Bloom's Taxonomy increases assurance that the questions assess different levels of understanding, which helps assess mastery of a concept [**scott2003bloom**; **thompson2008bloom**]. Each question should map to at least one level of Bloom's Taxonomy [**starr2008bloom**; **khairuddin2008application**]; rigor increases at each level. I found it helpful to compare the kinds of questions asked at each level of Bloom's Taxonomy with the questions in the bank to be used. I believe it is more effective if each level of Bloom's Taxonomy is represented in the bank of questions at least once rather than only focusing on certain levels of the Taxonomy [**scott2003bloom**]. To further incorporate practical aspects of software engineering, for each inventory, create questions that ask about code compilation, writing quality code, and resolving tool output.

5.4 Think Aloud Pilots

Once there is a bank of questions, the next step is to pilot the questions. The goal of this pilot is to determine if there are any obvious ambiguities in the questions or options. Consistent with Parker

and colleagues [**parker2016replication**], we recommend a think-aloud activity where students take each concept inventory and report questions or items that stand out as confusing and why. As participants note ambiguous words or phrases used in the questions or items, immediately make modifications before having any others pilot the inventory. When observing the scores, if all the scores are really high, or really low, this might suggest an assessment that is too difficult or too easy [**nelson1967testing**], which would require revisiting and revising the questions or items. To ensure items contribute to overall effectiveness, use statistical methods to validate the concept inventories.

I piloted my concept inventories with undergraduate and graduate students. For each student, I asked them to take the inventory as they normally would and let us know when one of the following occurs:

- They came across a question where the phrasing did not make sense or was unclear.
- The came across an option for a question that did not make sense or was unclear.
- They encountered a question where they believed more than one option could be correct.
- They encountered a question where none of the options appear to be correct.
- They noticed any typos or discrepancies in the questions, options, or code examples.

I made note during this process of any issues participants encountered, asked them to explain that difficulty, and how it could be improved. Once that participant finished the concept inventory, I immediately integrated the necessary changes into the concept inventory. I iterated this process with all pilot participants.

5.5 Concept Inventory Validation

Validation of an assessment tool can be done in a variety of ways. To test the validity of my concept inventories and the items on each, I conducted *item analysis* and *distractor analysis* using R statistical software [**boopathiraj2013analysis**; **RSoftware**]. I ran these analyses based on data gathered from a different sample than the group used for the initial think aloud pilot.

5.5.1 Item Analysis

The most common form of test validation is item analysis [**gorsuch1997exploratory**]. Item analysis determines if inventory items are valid methods of assessment; most often, the item difficulty and

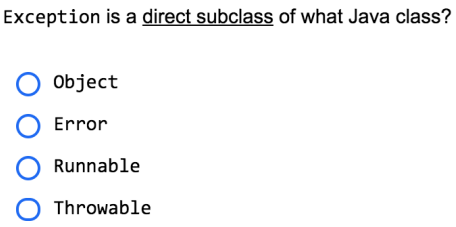


Figure 5.2 Item removed from original concept inventory.

discrimination index are used to assess test quality [**boopathiraj2013analysis**]. For item difficulty, an item is considered too easy if its item difficulty value (Item Difficulty, Table 5.2) is 0.95–1.00 and too difficult if it is less than 0.20. An optimal question has an item difficulty value of 0.50, however, the primary goal is to not have any questions, based on the item difficulty value, that are too hard or too easy.

A question is considered satisfactory if the discrimination index (Point Biserial, Table 5.2) is greater than 0.20. The difficulty and point biserial values can be positive or negative; a negative value suggests an item should be removed or replaced. The threshold value (Item Threshold, Table 5.2) provides the same information as the difficulty value; the lower the value, the easier the question. The biserial value (Biserial, Table 5.2 describes the degree of relationship between two interval scales. For the type of validation needed for the inventories, the threshold and biserial are not relevant.

To clarify the validation process, I present item analysis results from the exception handling concept inventory as an example. This was the concept inventory that underwent the most change as a result of this process. Table 5.2 shows the item analysis output from the initial set of exceptions questions put in thee inventory and piloted with 10 students and developers. I wanted to include developers with industry experience to evaluate the more advanced questions on the inventory. I performed pilots with students and those with industry experience to evaluate how well the concept inventory assessed different levels of knowledge regarding exception handling. Based on the output, one of the questions removed was item 3, which is shown in Figure 5.2. The discrimination index for this item (-0.024) is too low, which means that it was not a good item for discriminating between students who are knowledgeable in exception handling and those who are not. Based on these numbers, I also removed items 4, 8, and 9.

Table 5.2 Exception handling concept inventory item analysis results

	Item Difficulty	Item Threshold	Point Biserial	Biserial
Item 1	0.917	-1.383	0.271	0.489
Item 2	0.333	0.431	0.098	0.127
Item 3	0.667	-0.431	-0.024	-0.032
Item 4	0.917	-1.383	0.271	0.489
Item 5	0.500	0.000	0.553	0.694
Item 6	0.500	0.000	0.484	0.607
Item 7	0.583	-0.210	0.690	0.872
Item 8	1.000	-Inf	N/A	N/A
Item 9	0.500	0.000	0.899	1.000
Item 10	0.833	-0.967	0.495	0.738
Item 11	0.500	0.000	0.899	1.000
Item 12	0.667	-0.431	0.489	0.634
Item 13	0.583	-0.210	0.690	0.872
Item 14	0.667	-0.431	0.342	0.444

5.5.2 Distractor Analysis

I also used distractor analysis to evaluate the changes made during the think-aloud piloting. Distractor analysis determines if the options provided for a given question are fair and if the incorrect options contribute to the quality of the inventory. If the incorrect options are nonsensical or would make no sense as the answer, this takes away from the quality of the concept inventory. For example, if the only item that makes sense is the correct response, the chances that students can guess the right response as opposed to knowing the right response is increased.

The goal is for there to be no distractors that are not being selected and for the high performers (middle to upper range) to most often select the correct response. Because this was the case in the data for all the inventories, I kept all response options for each inventory item.[2] As with previous work, because establishing reliability for one inventory requires data from multiple trials, it may be better to save exploring reliability for future analysis [**tew2010developing**; **tew2010assessing**].

[2]The final version of the concept inventories can be found at the following urls: `http://go.ncsu.edu/null`(null object dereferencing), `http://go.ncsu.edu/generics`(generics), `http://go.ncsu.edu/exceptions`(exception handling).

5.6 Limitations & Challenges

Despite the possibilities for our approach for creating and deploying these concept inventories, there are some limitations.

- **Upfront time commitment.** Although there is value inside and outside the classroom in the process we proposed for depth of knowledge assessment, there is an upfront time cost involved in creating an inventory using our process. In my experience, after creating the first couple of inventories, especially with the up front cost of understanding how to conduct item analysis and what the results mean, the amount of time it takes to create new inventories decreases.
- **Lack of automation for individual response assessment.** Currently, to our knowledge, there is no automated way to observe individual student responses and changes over time, outside of item analysis. Item analysis, which we performed on pilot data from our concept inventories, provide some insights into the effectiveness of each item on the evaluation. However, it does not ease the process of determining which concepts need more attention in class or lab which is one of the potential uses for our inventories.
- **Guaranteeing responses are their own.** Even though we did not deploy our concept inventories for a grade, it is still possible that students used outside sources, or even each other, to help determine answers on the inventory. There is no sure way to guarantee students are not using outside resources. However, we attempted to mitigate this limitation by only allowing so much time for taking the inventory and making it an in class exercise rather than a homework assignment.
- **Keeping concept inventories up to date.** Depending on the language concepts are being taught in, there may be a need to periodically re-assess the material on the inventory for updating. So far, we have not had to make any updates however generics and exception handling are great examples of language features that has evolved since it has been out. Because our concept inventories aim to achieve depth of knowledge assessment, it is important that all relevant sub-concepts are represented in the inventory.

These limitations and challenges are presented for the general application of these concept inventories in a real world setting, such as the classroom or industrial training. I developed these concept inventories to facilitate model building, which I discuss in detail in the next chapter.

CHAPTER

6

DEVELOPER KNOWLEDGE CLASSIFICATION

In Chapter 5, I outlined an approach for assessing developer knowledge of programming concepts. However, it is impractical and possibly infeasible to create a concept inventory for every programming concept and then ask developers to take that inventory before or while using a tool. It would be more practical and feasible if there was a way to use information already available to represent developer knowledge. I propose using developer experiences, in the form of the code they have written, to represent their conceptual knowledge.

Perception of any information provided to a developer is affected by that developer's knowledge and experiences; as a software developer, much of the knowledge accrued comes from experiences writing and modifying source code [**Canas:1994:Mental**; **raju1995differential**; **fritz2010degree**; **argote2011organizational**].

However, currently tools have no way of assessing anything about the developer's knowledge. Existing work in the area of source code mining has focused on measuring and predicting functional and non-functional properties of software [**heckman2009model**; **menzies2007data**; **haapio2011exploring**]. Contrary to much of the work in this area, I used source code mining to predict developer knowledge of programming concepts.

Based on previous research that suggests I can use source code as an indicator of how much a developer knows about a codebase [**fritz2010degree**], I believe source code can also be an indicator of what developers know about programming. Therefore, I designed a study to answer the following research questions:

RQ_1 : *Is source code a good predictor of how much developers know about programming concepts?*

RQ_2 : *Does concept-specific source code increase the ability to classify how much developers know about programming concepts in comparison to a naive model?*

To answer these questions, I built and evaluated models that classify developers' conceptual knowledge. To determine if source code is a good predictor, I compared these models to random chance and a naive model that uses all the source code written by the developer (LOC). The assumption is that if my models can classify developers with greater than 50% accuracy, they perform better than random chance. I collected source code relevant to the Java programming concepts of variables, exception handling, and generics from 19, 35, and 23 developers, respectively. I distributed the concept inventories described in Chapter 5 to validate conceptual knowledge for each model. I trained each model using source code the developers wrote pertaining to each programming concept, or *concept-specific code*, in public GitHub repositories. I used unsupervised learning to determine a model that classifies developers based on their conceptual knowledge. Specifically, I calculated model metrics, such as precision, recall, and false positive rate, to determine and compare model performance.

The contribution of this chapter is a validated approach and set of models for predicting developer conceptual knowledge using public developer source code contributions. For RQ_1, I found that models using source code outperformed a model that randomly assigns developers' expertise. For RQ_2, I found that concept-specific models outperformed models that attempt to assign expertise based on total lines of code (LOC) written.

6.1 Knowledge Acquisition

According to existing research, we acquire knowledge through our experiences [**argote2011organizational**]. For example, a chef's knowledge regarding recipes, best practices, and what tastes good comes from their experiences cooking in both professional and informal settings. Just as a chef gains knowledge from her experiences, as do other professions. Software development is no exception [**fritz2010degree**].

Much of what software developers do involves looking at, writing, or modifying source code. There are a variety of other experiences that come with being a software developer. However, for scoping and proof-of-concept purposes, I focus on the primary task of software development which is writing code. In the following section, I discuss how I used concept inventories and developer source code contributions to classify developer based on their conceptual knowledge.

6.2 Knowledge Classification

To determine the ability to use developer source contributions to predict developer knowledge, I used concept inventories to validate developer knowledge and source code contributions on GitHub to predict validated knowledge classifications. To explain the process used to validate and predict developer knowledge, I will use a hypothetical developer Gabrielle and her experiences writing code as an example.

6.2.1 Knowledge Validation

I conducted knowledge validation using concept inventories I designed from the process described in Chapter 5. The scores from the concept inventories provide data that I used as an independent attribute for each developer. In the following sections I will discuss how I used these scores in the overall process of creating and evaluating knowledge models.

6.2.2 Knowledge Prediction

The goal of this study was to determine the possibility of using developer source code contributions to predict how much developers know about programming concepts in their tool notifications. Using developer source code, along with their concept inventory scores and the independent attributes, I used unsupervised learning to determine the relationship between the two.

Source Code Contributions

To determine the dependent attributes for each model, and answer $\mathbf{RQ_1}$, I analyzed developers' public repositories for code contributions and assigned them to developers using version control (Figure 6.1b).

Consider the code in Figure 6.2, which is a portion of code from the JUnit4[1] repository. Gabrielle contributes to this repository often, which provides rich data regarding her experiences with programming concepts. I analyzed developer source code using the Eclipse JDT ASTParser [**eclipseASTParser**].

[1] `http://junit.org/junit4/`

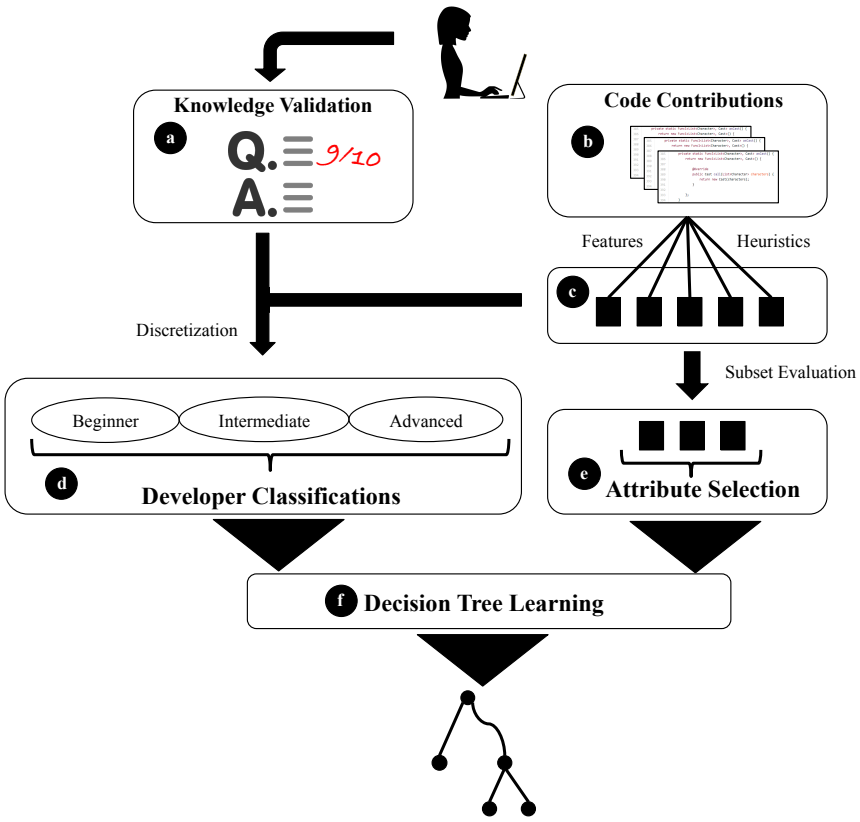


Figure 6.1 An overview of my approach.

Analyzing code statically with the ASTParser detects the presence of concept-specific code. However, it cannot tell us who contributed that code. In order to predict individual developer knowledge, we need to be able to identify that developer's code contributions.

Therefore, I used code bases in repositories so we could determine what developer made what contribution via the commit history. Since I chose the versioning platform Git,[2] I used JGit[3], a Java library that allows for manipulation of Git repositories via Java code. Also, next to SVN, Git is one of the most popular versioning softwares in use today.[4] Along with using ASTParser, I used JGit to analyze for individual developer code contributions.

Previous research suggests time may play a factor in how predictive code contributions can be [**johnson2015bespoke**], therefore I also used JGit to detect when the most recent contribution of each type of concept usage was made. We chose to use GitHub, a social coding site where developers

[2] `https://git-scm.com/`
[3] `http://eclipse.org/jgit/`
[4] `http://www.openhub.net/repositories/compare`

Table 6.1 Source Code Collected for Variables

Source Code	Description	Example
Primitive Types	data type is one of the most basic Java data types (i.e. `int`) and store a value.⋆⋆ (**D**)	`int a = 3;`
Non-Primitive Types	data type is a reference data type and stores a reference to an object.⋆ (**D**)	`Object o = new Object();`
Fields	created to be accessed globally by a class.⋆ (**D**)	`public String s;`
Local Variables	created and assigned a value to be accessed locally by a method or construct.⋆⋆ (**D**)	`public String s = "hi";`
Parameters	used as parameters to pass information into a method the developer created.⋆⋆⋆ (**D**)	`public void foo (String s)`
Public Variables	includes the public modifier.⋆⋆ (**D**)	`public String s = "hi";`
Private Variables	includes the private modifier.⋆⋆ (**D**)	`private String s = "hi";`
Protected Variables	includes the protected modifier.⋆⋆⋆ (**D**)	`protected String s = "hi";`
Static Variables	includes the static modifier.⋆⋆ (**D**)	`static String s = "hi";`
Final Variables	includes the final modifier.⋆⋆⋆ (**D**)	`final String s = "hi";`
Transient Variables	includes the transient modifier.⋆⋆⋆ (**D**)	`transient String s = "hi";`
Volatile Variables	includes the volatile modifier.⋆⋆⋆ (**D**)	`volatile String s = "hi";`

can create and maintain Git repositories,[5] as the source of data because GitHub stores Git repositories and many repositories on GitHub are public. Once a plan was in place for analyzing developers repositories, to answer RQ_1, I next identify source code that is directly related to the programming concepts of interest.

6.2.2.1 Concept-Specific Source Code

To answer RQ_2, I collected concept-specific code and manually collected lines of code (LOC) added to each repository for each developer from GitHub. I define concept-specific source code as code that, according to on-line resources, is relevant to understanding and using the concept in source code. Since much of a developer's experience is writing source code, and experience informs knowledge [**bromme1995fusing**; **argote2011organizational**], I used total LOC as the naive model.

[5] `http://www.github.com`

Table 6.2 Source Code Collected for Exceptions

Source Code	Description	Example
Throws Methods	method that throws an exception in the signature.⋆ (**U**)	`public void foo() throws IOException`
Try Statements	non-empty try block.⋆⋆ (**U**)	`try { ... }`
Catch Blocks	non-empty catch block.⋆ (**U**)	`catch (IOException e){...}`
Multi-Catch Blocks	non-empty catch block that use the multi-catch operator (|).⋆⋆⋆ (**U**)	`catch (IOException | SecurityException e)`
Try-With-Resources	non-empty try block that uses the try-with-resources feature.⋆⋆⋆ (**U**)	`try (BufferedReader br = new BufferedReader())`
Finally Blocks	non-empty finally block.⋆⋆⋆ (**U**)	`finally {...}`
Throw Statements	statement in method body that throws an exception.⋆⋆ (**U**)	`throw new IOException();`
Exception Declarations	creation of a new exception class.⋆⋆ (**D**)	`public class NewException extends Exception`
Catch Exceptions	non-empty catch blocks that catch the generic Exception.⋆⋆ (**U**)	`catch (Exception e) {...}`
Checked Exceptions	statement that uses exceptions that are checked at compile -time.⋆ (**U**)	`FileNotFoundException`
Unchecked Exceptions	statement that uses exceptions that are not checked at compile-time.⋆⋆ (**U**)	`RuntimeException`

Table 6.3 Source Code Collected for Generics

Source Code	Description	Example
Type Argument Methods	method with generic type argument(s).⋆ (**U**)	`public List<String> foo()`
Wildcard Generics	usage of the wildcard type parameter.⋆⋆ (**D**)	`List<?> String`
Generic Type Declarations	creation of new generic class.⋆ (**D**)	`public class Bar<T>`
Type Parameter Fields	field with generic type parameter(s).⋆⋆ (**D**)	`List<T> list;`
Type Parameter Method	method with generic type parameter(s).⋆⋆ (**D**)	`public List<T> foo()`
Diamond Generics	usage of the diamond operator.⋆⋆⋆ (**U**)	`... = new List<>();`
Explicit Method Invocation	method invocation with explicit generic type arguments.⋆⋆⋆ (**U**)	`Collections.<Number, Long>collect(...)`
Implicit Method Invocation	method invocation with implied generic type arguments.⋆ (**U**)	`Collections.collect(...)`
Generic Class Instantiation	instantiation of a generic class.⋆ (**U**)	`Bar<String> b = new Bar<String>();`
Nested Generics	usage of generics within a generic construct.⋆ (**U**)	`Map<Integer, List<String» map = ...;`
Bounded Type Parameters	generics with type bounds.⋆⋆⋆ (**D**)	`List<T extends String> list;`

```
1 public class LoggerRegistry<T extends ExtendedLogger> {
2     private static final String DEFAULT_FACTORY_KEY =
3     AbstractLogger.DEFAULT_MESSAGE_FACTORY_CLASS.getName();                    ——— Sept. 30, 2016 -- Dan
4
5     private final MapFactory<T> factory;                                       ——— Jan. 4, 2017 -- Gabby
6
7     private final Map<String, Map<String, T>> map;                             ——— Sept. 30, 2016 -- Dan
8
9     public LoggerRegistry() {                                                  ——— Sept. 30, 2016 -- Dan
10          this(new ConcurrentMapFactory<T>());                                 ——— Sept. 30, 2016 -- Dan
11    }                                                                          ——— Sept. 30, 2016 -- Dan
12
13    public LoggerRegistry(final MapFactory<T> factory) {                       ——— Jan. 4, 2017 -- Gabby
14          this.factory = Objects.requireNonNull(factory, "factory");           ——— Jan. 4, 2017 -- Gabby
15          this.map = factory.createOuterMap();                                 ——— Jan. 4, 2017 -- Gabby
16    }                                                                          ——— Jan. 4, 2017 -- Gabby
17
18    private Map<String, T> getOrCreateInnerMap(final String factoryName) {
19          Map<String, T> inner = map.get(factoryName);
20
21          if (inner == null) {                                                 ——— May 30, 2016 -- Gabby
22                inner = factory.createInnerMap();                              ——— May 30, 2016 -- Gabby
23                map.put(factoryName, inner);                                   ——— May 30, 2016 -- Gabby
24          }
25
26          return inner;
27    }
28
29 }
```

Figure 6.2 Mapping of developer source code contributions on one class in an open source repository.

I used the same key concepts identified for the concept inventories to determine what concept-specific code to analyze for. For each concept, I used the same resources used to create the concept inventories to determine relevant code to collect from developers' repositories. In the end, I collected 11 examples of generics usage, 10 examples of exception handling usage, and 12 examples of variables usage.[6] All of the code I collected, along with a description and example for each, is shown in Tables 6.1– 6.3. I manually checked the added lines reported by GitHub on each developer's repositories to determine LOC for the naive model.

Because Gabrielle took the variables and generics concept inventories, I analyzed her repositories for code that she contributed that related specifically to variables and generics. For example, looking at the code in Figure 6.2, in the class `LoggerRegistry<T extends ExtendedLogger>`, Gabrielle contributed both variables (lines 5 and 13) and generics code (lines lines 5, 13, and 23).

The output of the analyzer for each repository is an occurrence count for all contributed concept-specific code and when the most recent contribution for each occurred. For example, looking at Gabrielle's contributions in `LoggerRegistry<T extends ExtendedLogger>`, Gabrielle added two variables (a private final field at line 5 and a final parameter at line 13). For each variable she contributed, the analyzer would also note that her most recent contribution for each was in the past couple months. Prior to dividing the data by features or applying heuristics, I used the list of

[6]The repository that holds the analyzer used can be found at: www.github.com/brittjay0104/APATIANproto

concept-specific source code outlined in Table 6.1 (Variables), Table 6.2 (Exception Handling), and Table 6.3 (Generics) to determine levels of source code usage.

6.2.2.2 Source Code Usage Hierarchy

I analyzed data from 19, 35, and 23 GitHub developers for variables, exception handling, and generics (respectively). I used all the output from analyzing developers' code repositories to determine which types of concept usage might be more advanced than others. Based on frequency of source code usage across repositories and developers for each concept (Figure 6.2), I created a hierarchy of source code usage that outlines what source code collected might be considered basic, intermediate, or advanced. Under the assumption that the more something is used the less difficult (and more foundational) it is, I determined which features fit into which category by observing what source code developers used most often versus source code developers use least often. For example, only 27% on the source code we collected included Bounded Type Parameters (Figure 6.2) as opposed to the 99% of source code included Generic Type Declarations. Therefore, I consider Type Declarations basic usage and Bounded Type Parameters advanced usage. I determined thresholds by a leap in the total count of source code usage of more than 100. An example concept source code usage hierarchy, along with a more detailed description of the creation process and usage, can be found on-line with the other materials.

6.2.2.3 Data Features and Heuristics

To provide a wider range of generalizable attributes for answer our research questions, I characterized the dependent attributes (Figure 6.1c) for the models by grouping concept-specific code collected above based on features of the data.

For example, a feature of type parameter fields and methods (Figure 6.2, lines 5 and 13) that groups them together is Gabrielle wrote new generic code (Declarations) for use by other developers, rather than using existing generic code (Usages) as she did on line 23. The set of features identified and used among the data are as follows:

- **Levels of Concept Usage:** I computed the Levels of Concept Usage by adding together counts from the types of each concept that would be considered on a basic, intermediate, or advanced level of usage. I used the concept source code usage hierarchy discussed previously. In Tables 6.1– 6.3, Basic usage has one star ($\star$) at the end of its description, Intermediate two stars ($\star\star$), and Advanced three stars ($\star\star\star$).

- **Declarations:** I computed Declarations for each concept by adding together counts from the

types of concept usage where the developer wrote new concept code for use by themselves or others (i.e. type declarations or new exception type). In Tables 6.1– 6.3, Declarations are labeled with a **(D)** at the end of the description.

- **Usages**: We computed Usage by adding together counts from the types of concept usage where the developer is using code someone else wrote (i.e. method invocations), as opposed to contributing to a new type. In Tables 6.1– 6.3, Usages are labeled with a **(U)** at the end of the description.

Because I currently only collect variable declarations, all variable concept-specific code collected for this study falls under Declarations.

Along with the above features, I also defined two heuristics to apply to the feature groups:

- **Recency:** The recency heuristic takes each of the initial attribute values and multiplies each value by 1.0 if the most recent contribution was made in the last week, 0.8 if between one week and one month, 0.6 for 1–6 months, 0.4 for one year, and 0.2 for more than one year. For example, if, the total number of declarations made by the developer (Declaration heuristic) is 198 and the most recent declaration was written between one week and one month, the Type Declaration Recency (DeclRecency) heuristic value would be 158.4 (198×0.8).

- **Natural Log:** This heuristic calculates natural log of each feature group before and after the recency heuristic is applied.

I defined a recency heuristic because previous analyses suggested time may be a factor to consider when modeling knowledge [**johnson2015bespoke**]. For Gabrielle, if I only analyzed the class in Figure 6.2, Gabrielle contributed a field, private variable, final variable, and parameter; all would be assigned the same recency value (0.6) based on when she made the contribution(s). Gabrielle contributed 3 pieces of generic code: a type parameter field (line 5), a type parameter method (line 13), and an implicit method invocation (line 23). However, as shown in Figure 6.2, Gabrielle contributed type parameter fields and methods more recently than she contributed implicit method invocations. Therefore, while Gabrielle's type parameter field and method recency value is 0.6, her implicit method invocation recency value would be 0.4.

I applied natural log to the data following the reasoning of Fritz and colleagues, who used natural log in their models to account for the potential for large differences in attribute values [**fritz2010degree**]. For example, some repositories returned counts in the thousands for class instantiations but counts of zero for explicit method invocations; this might cause the model to put more weight on the contribution of feature groups and heuristics that include class instantiations than it truly contributes.

6.2.2.4 Developer Knowledge Classifications

Once the attributes have been calculated, I performed discretization to convert the continuous concept inventory score values into intervals of values, or *classes* (Figure 6.1d) [**fayyad1993multi**]. This step became particularly important once I decided to use decision tree learners to answer the research questions – I discuss the decision to use decision trees later in this section.

Discretization involves iteratively comparing attribute values for each instance and finding the "best cut" in the data; each cut is considered a class of the data. A distinguishing feature of this data set is its small size (just a few dozen rows). Hence, I cut the concept inventory scores for each model into two and three classes to answer the research questions. Two classes comes from cutting on the median. I chose three to account for the potential transition between classes [**dreyfus2004five**]. For the three way split, with the small data set, I tried to maintain relatively even size chunks while making sure there is some consistency across models (i.e. someone with a score of 7 or above is never a beginner).

The generics data yielded ternary discretization (three classes) and the variables and exceptions data yielded binary discretization (two classes). For data sets with three classes, I labeled each developer as either Beginner, Intermediate, or Expert. For data sets with two classes, I labeled each developer as either Beginner or Expert. Based on Gabrielle's concept inventory scores and source code data, she is labeled Expert for both generics and variables. We will talk about what the differences in the discretization output might mean in Section 6.4.

6.2.2.5 Attribute Selection

Once I assigned classifications to each developer, I used developer knowledge classifications to perform Correlation-based Feature Selection (CFS) in Weka, software that provides a collection of machine learning algorithms for data mining [**Hall:2009:WDM:1656274.1656278**]. CFS helps to determine the attributes best suited for each model (Figure 6.1e) [**hall1999correlation**] by pruning irrelevant attributes, leaving only the attributes that correlate the most with the developer's classification.

This analysis runs k-fold cross validation using the attributes passed in; to lower the potential for a model with overestimation bias, I maintained even and sizable chunks by using 4-fold cross validation. CFS evaluates each attribute on its predictive ability and uses cross-validation to indicate how stable the best subset of variables is based on how many folds the variable appeared in. For increased model stability, I used the selection criteria that the attribute appear in two or more folds to be included in decision tree learning.

6.2.2.6 Decision Tree Learning

To answer the research questions, I used decision tree learning. More specifically, I used Weka's J48 classifier [**witten1999weka**] to create decision trees. Wolpert and colleagues [**wolpert1997no**] caution that one should not expect any particular algorithm to work best for all possible inputs. Hence, when exploring new data, is it necessary to conduct some experimentation to find useful settings for that data. Accordingly, I ran different analyses and learners to determine which learner was most effective for the data and research goals. Because there are so many machine learning algorithms availble, I neeeded to narrow down the learners I would experiment with. I chose to compare regression, Naive Bayes, and Decision Trees based on prior work with similar goals of making software development-related predictions [**menzies2004assessing**; **menzies2007data**; **heckman2009model**; **fritz2010degree**]. I found decision trees to be optimal for answering the research questions along with being more human readable.

Another motivation for using decision trees as opposed to regression or Naive Bayes is the relatively heightened error tolerance provided by decision tree learners. Because I only used public repository contributions, the data collected may have attributes with no data. For example, upon analyzing a developer's repositories I may find no occurrences of generic type declaration due to the fact that the developer has written generic type declarations in a context not being analyzed (i.e. local software projects). Using decision trees allows us to see meaningful relationships in the data, regardless of any data that may be erroneous or inaccurate.

6.3 Knowledge Models

Based on developer classifications and the full set of attributes, CFS identified the following subset of attributes that fit my selection criteria:

- **Variables** – Public Variables
- **Exceptions** – Advanced Exceptions, Try Statements Recency, Finally Blocks Recency, and Throws Methods
- **Generics** – LOC, Generic Type Declarations, and Generic Type Declarations Recency

I created a decision tree model for each concept with all the concept-specific attributes above to answer RQ_1. To answer RQ_2, I created a LOC only model, when LOC met the attribute selection criteria, to compare to the concept-specific models. The values for each model's *precision*, *recall*, *F-Score*, *TP (True Positive) Rate*, and *FP (False Positive) Rate* are shown in Tables 6.4, 6.5, 6.6, and 6.7. The resulting decision trees can be found in Figures 6.3, 6.4, and 6.5.

Table 6.4 Variables Model Accuracy

Public Variables			
	Beginner	**Advanced**	**Total**
Precision	0.571	0.667	0.627
Recall	0.500	0.727	0.632
F-Score	0.533	0.696	0.627
TP Rate	0.500	0.727	0.632
FP Rate	0.273	0.500	0.404

Table 6.5 Exceptions Model Accuracy

Advanced, Try Statement Recency, Finally Block Recency, Throws Method			
	Beginner	**Advanced**	**Total**
Precision	0.789	0.750	0.771
Recall	0.789	0.750	0.771
F-Score	0.789	0.750	0.771
TP Rate	0.789	0.750	0.771
FP Rate	0.250	0.211	0.232

Table 6.6 Generics Model Accuracy (Non-LOC)

Declarations and DeclRecency (Generics)				
	Beginner	**Intermediate**	**Advanced**	**Total**
Precision	0.667	0.727	0.875	0.729
Recall	0.333	0.889	0.875	0.739
F-Score	0.444	0.8	0.824	0.715
TP Rate	0.333	0.889	0.875	0.739
FP Rate	0.059	0.214	0.133	0.146

Table 6.7 Generics Model Accuracy (LOC only)

	LOC (Generics)			
	Beginner	**Intermediate**	**Advanced**	**Total**
Precision	0.5	0.636	0.875	0.684
Recall	0.333	0.778	0.875	0.696
F-Score	0.4	0.7	0.875	0.683
TP Rate	0.333	0.778	0.875	0.696
FP Rate	0.118	0.286	0.067	0.166

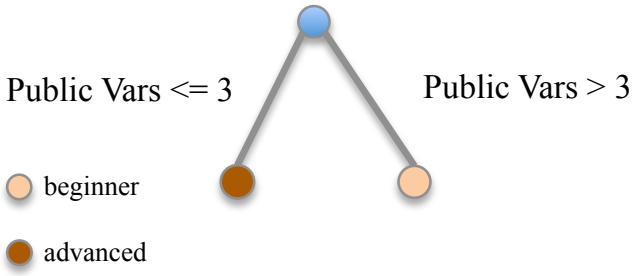


Figure 6.3 Variables decision tree model.

6.3.1 RQ_1 Findings

All four decision tree models had total precision, recall, and F-Score of at least 60%, suggesting these models correctly classified developers at least 60% of the time. Even the model built using LOC only (Table 6.7) provided accurate predictions. This supports existing research that suggests all of our experiences contribute to our overall knowledge [**argote2011organizational**]; my findings quantify this notion, showing it is possible to use developer experience to predict their knowledge. The models also exhibit low FP rates, with a median FP rate of 19.9%. I observed the highest FP rate with the variables model (Figure 6.3); I will discuss limitations that may have caused the difference in FP rate for the variables model in Section 6.5.

Based on the attributes used in the decision trees shown in Figure 6.5 and Figure 6.4, how recently concept-specific code was contributed can also affect conceptual knowledge. Based on the recency heuristic, for example, a higher DeclRecency (Figure 6.5) value suggest more recent code contributions related to declaring generic types. This suggests that determining a developer's knowledge of generics would involve observing both declarations and recent experience with declarations. For exceptions, recency matters for some attributes, such as `try` statements and

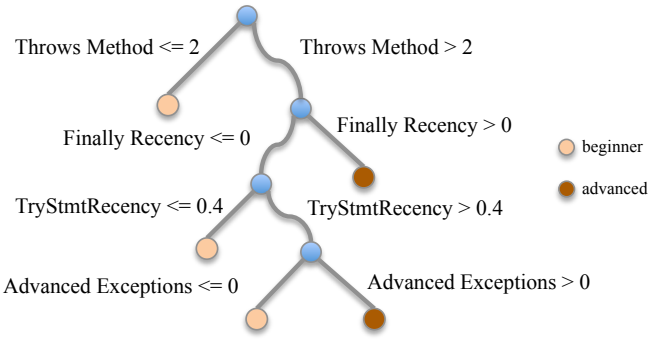


Figure 6.4 Exceptions decision tree model.

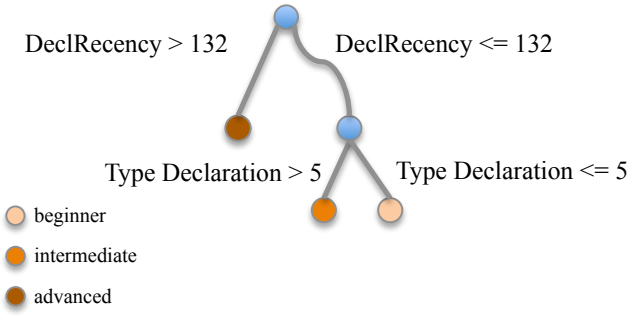


Figure 6.5 Generics decision tree model.

`finally` blocks, while it is not as important for others, such as usage of advanced exceptions language features. For variables, recency also did not appear to improve classification – the variables model only includes the declaration of variables, which may have affected the ability to build an accurate and convincing model.

These findings suggest that source code can be used to classify how much developers know about programming concepts better than random chance.

6.3.2 RQ_2 Findings

All three of the models that we trained using concept-specific attributes classified developers with overall accuracy better than random chance (> 50%). As for comparing the concept-specific models to a more naive model, after using CFS during data analysis, I was able to identify relevant attributes before running cross-validation during tree building. For two of the three concepts evaluated, LOC did not appear in enough folds during attribute selection to be considered during decision tree

learning. Even with the variables data, which only included various attributes pertaining to variable declaration, the most meaningful attributes were concept-specific. The only data set in which LOC appeared in more than two folds was the generics data set. This suggests that in some cases, LOC is not predictive of developer concept-specific knowledge.

Upon closer examination of the generics models, overall model performance is higher in the Type Declaration model (74%) than the LOC model (70%), with increased precision and F-Score for beginner (+0.167, +0.04) and intermediate (+0.09, +0.1) classification and increased total precision (+0.05), recall (+0.04), and F-Score (+0.03). The only scenario where the LOC model outperformed the Declarations model was when classifying developers with advanced generics knowledge (Table 6.6). These findings suggest that although both LOC and concept-specific code can both be used to predict conceptual knowledge, concept-specific code increases overall model accuracy and precision.

These findings suggest that concept-specific code improves model performance when compared to a naive model.

6.4 Implications

The ability to predict developer conceptual knowledge using attributes collected from source code opens the doors for exploring the possible applications, including the ability to adapt tool notifications to developer knowledge. I discuss the potential for adapting tool output and some of the other possible applications for these findings in detail below.

6.4.1 Program Analysis Tool Output

This study was motivated by the study reported in Chapter 4, which posed a theory regarding the challenges developers encounter when attempting to understand, and eventually resolve, tool notifications [**johnson2016cross**]. The theory stated that the challenges developer encounter stem from gaps and mismatches between their knowledge and how tools communicate. One way to assess this theory is to explore the ability, and effectiveness, of adapting tool notifications to the knowledge of the developer using them.

The ability to accurately classify developers' conceptual knowledge is the first step towards being able to provide meaningful notification adaptations. If a tool can automatically ascertain how much a developer knows about the concepts relevant to a given notification, it can better tailor the information provided to the information needed by that developer. The next step is to determine

what a meaningful adaptation for a given developer classification looks like; this work is described in detail in Chapter 7.

Another way that developer conceptual knowledge can be utilized by program analysis tools is to improve developers' experience when first using notifications used by the tool [**johnson2015bespoke**]. One of problems that have been identified with program analysis tools, especially static analysis tools, is the high volume of notifications that are sometimes presented [**Johnson:2013:Why**]. We can use developer conceptual knowledge to determine the notifications to prioritize. A common way current tools prioritize notifications is by problem severity, however, such strategy may not be enough to ensure a positive first experience. Also, findings from Chapter 4 suggest developers do not always agree with the prioritization used by tools [**johnson2016cross**]. Tools could improve the first experience by first presenting developers with notifications the tool knows they can solve based on the concepts they understand best.

6.4.2 Industry & Education Practices

For any given defect, there are one or more programming concepts relevant to understanding and being able to resolve that defect. Similar to work on code review assignment [**balachandran2013reducing**], another potential application for my proposed approach for predicting conceptual knowledge is to assign the best developer to resolve a defect or complete a code review. Although knowledge of the code base is important [**fritz2010degree**], my research suggests knowledge of concepts relevant to the defect or code of interest is also important [**johnson2016cross**]. My approach can be combined with other approaches, such as those that look at the developer's familiarity with the code base [**fritz2010degree**], to assign defects to developers that are most likely to be able to resolve them.

The ability to predict developer knowledge also opens the door for the potential to more effectively assign teams in industrial and educational settings. For example, if the design and implementation of a project or piece of functionality requires specific conceptual knowledge, analysis of developers' existing source code can yield information for ensuring someone with the necessary knowledge is on that team. Along the same lines, my approach can be useful for determining effective pair programming pairs. Pair programming is an effective way of transferring knowledge [**plonka2015knowledge**] and fostering tool discovery [**murphy2011peer**], both of which can aide in defect resolution. Knowledge transfer is more likely to occur when the developers paired together differ in experience (i.e. one is novice and one is expert). Furthermore, a previous study on pair programming found that the productivity provided by pair programming can drop substantially when it comes to problem solving if both programmers have experience with the problem

at hand; this is especially true if the experiences are recent and has not had a chance to be forgotten [**lui2006pair**]. When it comes to pairing developers for a specific task, our approach can be useful for determining which developer is more expert in the concepts relevant to the task at hand.

6.5 Lessons Learned

Even with a glimpse into the possibility of modelling conceptual knowledge, there are limitations to my current approach and challenges to overcome.

6.5.1 Limitations

One limitation to my approach is that it relies solely on source code available in public repositories; not all developers have public repositories and usually not all the code a developer has written is stored in one. Along the same lines, while analyzing developer source code can provide insights into conceptual knowledge, this may not be entirely representative of everything the developer knows about a given concept, or more importantly the notifications they encounter, which could lead to initially inaccurate models.

Another limitation to the findings presented in this chapter is that I built the models with a relatively small numbers of developers. Typically machine learning is done with large data sets, however, due to the nature of the data collected I ended up with smaller data sets (20–30 data points). I mitigated this limitation by using decision tree learning, which has been found to work well with atypical data sets [**zhang2005missing**; **kotsiantis2007supervised**].

One goal as I developed this approach was to provide a range of attributes relating to each concept to increase the likelihood that we are being exhaustive of relevant and possibly predictive code features. A limitation to my approach in this regard is how I dealt with a fundamental concept like variables, which can include a large range of relevant features. In this work, I decided to focus on various aspects of declaring a variable, such as data types and variable visibility, to narrow down the space of attributes being analyzed. I believe this led to the less-than-compelling model for variables conceptual knowledge found in Section 6.3. Although the model is not as compelling or complex as the others, the variables model still suggests that using source code to predict concept knowledge is better than random chance or a naive model. Another limitation is that for a small subset of developers, at least one repository was not 100% Java. Repositories that fit into this category ranged from 42% - 97% Java. Currently, my approach has only been evaluated on developers' knowledge of Java concepts, therefore a LOC model including non-Java code might have affected the results.

6.5.2 Challenges

Mirroring the small data set limitation, one challenge to using my approach is getting enough data from the developer, especially if the developer does not have code in public repositories. One way to deal with this challenges is to modify the approach to work in on local copies and in real-time to detect developer concept-specific contributions in projects not stored in repositories.

Another challenge that comes with using my approach is determining how to deal with both fundamental and nuanced programming concepts. The findings in this chapter suggest that it may be more difficult to build an exhaustive and accurate model for fundamental concepts than it is for nuanced concepts. This may suggest that there are ways that concepts group together that can may affect how one should approach determining relevant code to collect. For example, the code we collected for variables matches the information provided on the primary sources of variables-related concepts used for the concept inventory. However, because declaring variables is necessary in most programs, there may be a need to incorporate other information such as notifications pertaining to variables the developer encountered, to get the best idea of what developers really know about variables.

Along the same lines, another challenge for my current approach is the ability to generalize across concepts. For the most exhaustive models, we can see similarities such as the fact that recency is generally important. However, it is not obvious beyond that if there is a way to general what is important to focus on when classifying developers based on their code contributions. A good starting point for work in this area would be to determine the possibility of generalizing across a subset of concepts with similar characteristics, such as splitting concepts by simple, fundamental concepts, such as variables, methods, and classes, and more complex, nuanced concepts, such as generics, exception handling, and concurrency.

Despite the limitations and challenges associated with my proposed approach, the results of this study provided insights that can be used to further my research on improving the notifications tools use. In the next Chapter, I discuss how we can use developer knowledge to improve communication between developers and their tools.

CHAPTER

7

KNOWLEDGE-BASED COMMUNICATION

Based on findings from previous studies, tools do not always effectively communicating with developers. One reason this occurs is because of differences between developer knowledge and the information provided by notifications, which I discovered in the study outlines in Chapter 4.s Perhaps if tools could ascertain how much developers know about the concepts present in the information they provide, tools could provide feedback to better support developers in understanding and resolving defects. The previous chapter suggested the ability to use developer source code contributions to predict how much they know about the concepts in tool notifications.

Given the ability to classify developers based on their knowledge of programming concepts, I propose improving the communication between tools and developers by providing knowledge-based notifications. Knowledge-based notifications would present information to developers based on how much they know about the concepts relevant to the defect. To my knowledge, there are no existing studies that have explored adapting the information provided by tool notifications.

Despite notification adaptation being a new research area, adaptive user environments are becoming pervasive in research and practice [**zou2008adapting**; **amershi2007unsupervised**; **stamper2009unsupervi**
Adaptive User Interfaces (AUI) use the experiences of users to adapt to better support the user. Most relevant to this research are the models proposed by Zou and colleagues for adaptive menus in Eclipse [**zou2008adapting**]. Similarly, this research explores using source code and machine learning

to predict user knowledge but for adaptive tool notifications.

In this chapter, I discuss my current research which explores possible notification adaptations and the ability to improve communication by improving the support provided by tools to fill knowledge gaps and match developer expectations. To evaluate this proposal, I borrowed from existing research on expertise and problem solving to determine potential notification adaptations to evaluate and how they map to expertise. I used the same concepts used in previous chapters to create and evaluate knowledge models: variables, exception handling, and generics. I found notifications that communicate primarily about various aspect of these concepts, focusing on notifications that communicate about defects that are found in real software projects [**ayewah2007evaluating**; **zheng2006value**].

I conducted a user study with 14 students and professional developers with various backgrounds and knowledge regarding the concepts of interest to evaluate their ability to resolve notifications designed for their level of expertise. This study was designed to evaluate the following hypotheses and research question:

H_1 Knowledge-based notifications can decrease the time required for notification resolution.

H_2 Knowledge-based notifications increase developer likelihood of resolving notifications correctly.

H_3 Knowledge-based notifications decrease developer attempts to resolve notifications.

H_4 Do developer adaptation preferences match expectations, based on existing literature?

7.1 Proposed Approach

For adaptive tool notifications to be realized, it is necessary to determine how tools should present information to developers. Previous research suggests one thing to consider when presenting information regarding a defect is the knowledge of the developer [**johnson2016cross**]. To evaluate H_1 – H_4, I conducted user studies where I presented developers with adapted notifications and asked them to resolve each. In this section, I will outline the related research I used to create the adapted notifications and the approach I used to evaluate notification adaptations.

7.1.1 Notification Adaptations

Existing research on notification design focuses on the general information needs of all developers, regardless of their knowledge regarding the notification or its concepts [**smith2015questions**;

barik14; **robillard2014recommendation**]. Therefore, there is no existing work that has examined notification design for expertise.

There does exists work in other fields that has examined problem solving and debugging in relation to programmer expertise [**larkin1980expert**; **mckeithen1981knowledge**; **Wiedenbeck:1993:Mental**]. I used this existing research to inform the design of the adapted notifications for each level of expertise, and help answer my research question. Based on existing research, I designed knowledge-based notifications using the following criteria:

- Notifications designed for developers classified as intermediate and expert for a given concept provide brief notifications with goal statements to communicate the problem.
- Notifications designed for developers classified as novice for a given concept provide more detailed notifications with goal statements and subgoals to communicate the problem.
- All notifications include examples and links relevant to the problem and its solution.

The previous chapter on classifying developers suggested that for some concepts, progress to expertise is a three stage progression. Previous research found that there is typically homogeneity within expertise groups (novice, intermediate, expert) but that strategy selection and usage is based on expertise [**mckeithen1981knowledge**]. This research also found that the intermediate and expert groups exhibited similar strategies.

For the user study, I chose to design notifications for a novice and expert split rather than novice, intermediate, and expert. I made this decision because 1) designing 3 notifications per defect would have led to longer sessions, 2) the models created in Chapter 6 suggest only 2–3 knowledge groupings per concept, and 3) existing research suggests that intermediate and expert programmers solve problems in similar ways [**mckeithen1981knowledge**].

The adaptation provided for each classification should be meaningful to that expertise group. There exists research that has explored the differences between novice and expert problem solving in STEM domains [**larkin1980expert**; **Wiedenbeck:1993:Mental**]. Using this research, I determined possible adaptations for novice and expert developers.

Larkin and colleagues found that experts in STEM domains have common foundations when problem solving [**larkin1980expert**]. They found that experts use experience to work from a goal to a solution, while novices required a goal and subgoals to reach a solution. Similarly, Wiedenbeck and colleagues have done extensive work on expertise-based problem solving [**wiedenbeck1985novice**; **Wiedenbeck:1993:Mental**]. They posed five characteristics in expert mental representations, most of which novices lack:

- **It is hierarchal and multi-layered.** This means experts have an understanding of high level program structure and goals as well as low level data structures.
- **It has explicit mappings between layers.** Experts also have mappings between program goals and data structures used to achieve them.
- **It has a foundation.** At the foundation of expert mental representations is the ability to recognize basic patterns or plans.
- **It is internally well-connected.** Experts understand how components in the code work together.
- **It is well grounded in program text.** Experts are able to easily relate the various parts of their mental representations, such as an abstract plan, to the source code.

Most notifications provided by state-of-the-art tools provide a problem statement when communicating about a defect. However, my previous research suggests current notifications do not always communicate effectively. Research in other STEM fields suggests that when problem solving, experts work from a goal to solution [**larkin1980expert**]. Therefore, the notifications modified for experts will provide a goal statement rather than a problem statement. The same research found that novices require a goal and set of subgoals that help them achieve the goal to reach a solution. Therefore, notifications modified for novices included a set of general subgoals for resolving the defects. Novice notification also include the problem statement used by FindBugs and the compiler.

I determined the goal statement for each notification by determining the general solution being suggested by the problem statement. For example, the goal statement in Figure 7.2 is based on the problem statement provided for raw type compiler warnings. However, the goal speaks to resolution, letting the developer know she needs to "Correct the creation and/or initialization for the generic collections." There is also a link present, which is active and links to a top search page on the web providing information regarding generic collections, if needed. I provided links throughout the expert and novice notifications based on previous studies that suggest developers often leave their working context to search the web for information [**johnson2016cross**; **nasehi2012makes**].

I determined the subgoals for each notification by determining a) the possible solutions to the defect, b) how to achieve those solutions, and c) how to generalize all of the above. It is important to maintain a realistic environment with realistic information, therefore the ability to generalize the information to other code snippets with the same defect was the most important. For example, for the notification shown in Figure 7.1, the subgoals provide generally applicable fix options and

Figure 7.1 A notification modified for a developer classified as a novice in exception handling.

Figure 7.2 A notification modified for a developer classified as an expert in generics.

provide steps, links, and examples. Also, most importantly, these fixes would generally apply to this defect no matter what code was implemented.

Developers use examples to understand and resolve problems in code; often they go to sources like StackOverflow [1] to find examples with explanations as most tool notifications do not include examples [**nasehi2012makes**]. My previous research also found that professional developers like having insights regarding possible fixes to the problem [**johnson2013don**; **johnson2016cross**]. Therefore, I included examples in all of the notifications. I primarily used StackOverflow to find code examples for the notifications. I also used the same resources used to create the concept inventories in Chapter 5.

Finally, to help mitigate notification familiarity bias, which was found to be relevant to notification understanding and resolution [**johnson2016cross**], I only presented participants with the modified notifications. This included disabling the relevant compiler notifications, which are familiar notifications for participants. Example modified notifications are shown in Figure 7.1 and Figure 7.2.

7.1.2 Notification Selection

Using the criteria outlined above, I modified 9 program analysis tool notifications from FindBugs and the Eclipse Java compiler, the same tools used in my previous study. I excluded ECLEmma to scope the current problem and solution to textual notifications. For each notification, I created a novice version and an expert version.

To determine the notifications for the user study, I attempted to find pre-existing defects in open source software. To narrow my search, I focused on defects that have been found to frequently appear in real world software projects [**ayewah2010google**]. I narrowed my search down to mature projects that were already Eclipse projects, could be relatively easily imported and compiled, and contains code snippets with libraries that are more likely to be familiar to developers.

Once I imported the projects, I analyzed the projects for compiler errors and warnings pertaining to the concepts of variables, exception handling, and generics. I also ran FindBugs on each project to find defects pertaining to the concepts. I went through the relevant notifications one by one to find code snippets that are not dependent on a large number of other classes and contain libraries and functionality that are familiar to the developer. The latter criteria was to mitigate the threat of developers looking at code that is not their own.

After analyzing a number of projects, I chose to go with the Java Collections library. I was able to find existing notifications from FindBugs and the compiler relevant to the concepts of interest.

[1] `https://stackoverflow.com/`

Table 7.1 Notifications used in the user study.

	Problem	**Tool**
Variables	Unwritten field	FindBugs
	Unused field/parameter	FindBugs
	Incompatible types	FindBugs
Exception Handling	Unhandled exception type	Compiler
	Method may fail to close stream on exception	FindBugs
	New exception not thrown	FindBugs
Generics	Inferred type argument(s) do not conform to the bounds of the type variable(s)	Compiler
	Raw types: references to generic types should be parameterized	Compiler
	Cannot convert generic types	Compiler

Because I needed enough notifications to populate my user study, I sometimes had to inject defects into the source code. When this was done, I chose code snippets where only minimal changes led to the introduction of the defect, error, or warning. Also, much of the code used in the library, and the functionality the library is written for, are familiar to developers regardless of their contributing to that code base. The final set of defects used in the study can be found in Table 7.1. For each notification in the table, I created a novice and expert version, for a total of 18 notifications.

7.1.3 Adaptation Evaluation

Once I had a set of modified notifications, I developed a wizard-of-oz plug-in to evaluate H1, H2, and my RQ. This plug-in presents defects that have been augmented with modified notifications to participants. Below I outline the user study I designed to evaluate my hypotheses, which are:

H_1 Knowledge-based notifications can decrease the time required for notification resolution.

H_2 Knowledge-based notifications increase developer likelihood of resolving notifications correctly.

H_3 Knowledge-based notifications decrease developer attempts to resolve notifications.

H_4 Do developer adaptation preferences match expectations, based on existing literature?

Participants in this study were developers from academia and industry. I attempted to recruit participants using various sources, including departmental mailing lists, social media, and personal

contacts. I was able to obtain 14 user study participants, most of which I obtained through personal contacts in academia and industry. All participants had prior experience with Java, Eclipse, and the Eclipse Java compiler. All but 4 participants had prior experience with FindBugs. Participants ranged from undergraduate students to professional developers with years of industry programming experience.

7.1.3.1 Study Design

To evaluate knowledge-based notifications, I need to know when participants in the study are looking at notifications that are aligned and misaligned with their knowledge. A notification is aligned with a participant's knowledge if the classification the notification was designed for matches the developer's classification. To determine developer classification, each participant completed concept inventories on the concepts relevant to the study. These are the same concept inventories used in Chapter 6.

Each session lasted approximately 1 hour. In this hour, I asked participants to resolve the 9 defects selected. For the user studies I used two set-ups. Rather than having all participants look at the same novice and expert notifications, I evaluated a novice and expert version for each notification across user studies. I alternated between two groups, each group consisting of a different subset of the 18 novice and expert notifications. In other words, all participants saw all 9 defects; but, for example, the novice notifications participants in Group 1 saw, another participant in Group 2 saw the expert version of that same notification.

At the beginning of each user study, I first briefed participants on what they would be doing. I also asked participants to rank their experience with Java, Eclipse, FindBugs, and the Eclipse Java compiler on a scale of 1 to 10 (1 being unfamiliar, 10 being extremely familiar).

During this time, I also included instructions specifically regarding information they can and should be using to resolve the notifications and how to access it. Because I want to assess developers' ability to resolve defects based on the information provided in the notifications, I asked participants not to apply quick fixes. To reduce the temptation to use them, I disabled the information provided by compiler outside of the marker and squiggly underline.

To re-enforce the process participants would be repeating during the study, I incorporated a training exercise into the beginning of the study. During the training exercise, participants did not have to resolve the notification but rather access and acknowledge the presence of the view that provides defect information.

After the training exercise, participants worked their way through each defect in their set of 9. I wanted to assess ability to resolve, therefore it was required that participants approach the problem

as if they were going to resolve it. However, I let participants know that if they felt they would need more time than they had for the user study to explore and resolve the defect they could skip it. I did not want to force participants to work until they resolved the problem; the pilots I conducted suggested that after about an hour participants begin to feel fatigued and thereby less engaged in the study.

After participants resolved, or attempted to resolve, each notification, I debriefed with them to learn more about how they came to the solution they used. The goal of debriefing was to determine if the notification contributed to or hindered their ability to resolve the defect. This was also an opportunity to ask participants about experience with the individual defects.

Some participants were distributed, therefore, I also had a remote set-up that I used to conduct user studies with remote participants. The remote set-up was similar to the local set-up; the only difference was that remote participants remotely controlled my computer to complete the user study while local participants physically used my computer. I recorded audio and the screen for all of the user studies in preparation for data analysis.

7.1.3.2 Data Analysis

The primary data point of interest was the time it took for participants to resolve each defect. Resolution of a defect for the sake of this study began when the participant opened and began to use the notification to work towards a resolution. For participants that attempted to resolve the defect without looking at the notification, time began when they either a) explored nearby code as if to determine the resolution or b) began fixing the defect if they did no exploring.

Along with time to resolve, I also collected data regarding the fixes participants selected to implement. For each defect, a fix was considered correct if it both compiled and would not cause new problems at runtime. For some defects, since they were pre-existing in the code base, I used Java best practices examples to determine the potential solutions. I also kept track of how many fixes participants attempted for each defect.

To supplement the quantitative data collected, and answer my RQ, I also analyzed the audio for statements made regarding the notifications and how well participants felt the notifications supported their resolution efforts. For example, findings from the study discussed in Chapter 4 suggest familiarity with the notification can also affect developers' ability to interpret tool output. Therefore, this is something I want to keep track of as it may be able to help better explain the quantitative findings.

All participants completed all three concept inventories necessary for data analysis, therefore I was able to use all the data provided by participants to evaluate my hypotheses. One participant

Table 7.2 Average time to resolve (seconds) aligned and misaligned notifications for each concept.

	Aligned Novices	Misaligned Novices	p-value	Aligned Experts	Misaligned Experts	p-value
Variables	156.1	98.4	*0.192*	87.8	88.7	*0.970*
Exceptions	46.9	137	*0.019**	42.2	79.1	*0.042**
Generics	235.4	249.8	*0.861*	237	341	*0.628*
Total	148.25	180.7	*0.446*	83	140.2	*0.130*

was unable to finish all the defects in the user study due to work obligations. Although he did not get to all 9 notifications, since he completed the inventories I analyzed data collected from the 7 he did complete.

To determine the significance between the differences calculated, I used R with RStudio [**RSoftware**] to run a two-sample unpaired t-test for set of aligned and misaligned values. I chose run a t-test because it can provide accurate results despite my small sample size. A t-test is also an appropriate test for comparing averages across two samples that are different sizes. I report the findings from these analyses in the next section.

7.2 Adaptation Effectiveness

Using the data from the user studies, I evaluated my hypotheses regarding developers' ability to resolve defects and their notification preferences.

7.2.1 Resolving Adapted Notifications

H_1 Findings

Average resolution time in seconds of defects with aligned and misaligned notifications are shown in Table 7.2 and Table 7.3. Table 7.3 also reports the average attempts made to resolve notifications for each concept. Average attempts and resolution time for each participant are listed in Table 7.4 and Table 7.5. I report a comparison of overall average resolution times and average resolution time by participant and by concept. For each concept, I observed the difference between averages for novices and experts. For each participant, I observed differences in their performance with aligned and misaligned notifications.

Overall, it took participants less time to resolve the notifications aligned with their experience than notifications not aligned with their experience. The difference between resolution time was the

Table 7.3 Totals for aligned and misaligned defect resolution.

		Aligned Novices	**Misaligned Novices**	**p-value**	**Aligned Experts**	**Misaligned Experts**	**p-value**
Variables	**Total Defects Attempted**	9	6	*N/A*	14	11	*N/A*
	% Resolved	78%	83%	*0.8188*	100%	91%	*0.2618*
	Avg. Attempts to Resolution	1	2	0.3558	2	2	*N/A*
Exceptions	**Total Defects Attempted**	13	8	N/A	13	8	*N/A*
	% Resolved	77%	100%	*0.1529*	100%	100%	*N/A*
	Avg. Attempts to Resolution	1	1	*N/A*	1	1	*N/A*
Generics	**Total Defects Attempted**	15	18	*N/A*	3	6	*N/A*
	% Resolved	73%	61%	*0.4655*	100%	83%	*0.4746*
	Avg. Attempts to Resolution	2	3	0.0950	1	1	*N/A*
Total	**Total Defects Attempted**	37	32	*N/A*	30	25	*N/A*
	% Resolved	76%	75%	*0.9238*	100%	92%	*0.1179*
	Avg. Attempts to Resolution	1	3	*0.03703**	1	1	*N/A*

largest for experts. This may have occurred because the notifications designed for novices provided more information to read, which some participants felt obligated to read regardless of needing the information.

Nine of the 14 participants spent less time on notifications aligned with their knowledge than they did on notifications not aligned with their knowledge. For two participants, the aligned notifications significantly reduced the time to resolution (Table 7.5). Also, despite there being more participants presented with aligned notifications (67) than misaligned notifications (57), participants took less time overall to resolve aligned notifications (231.25 seconds) than misaligned notifications (320.9 seconds). The difference was not significant, however, this may be due to the fact that I did not take into account other relevant factors, such as defect and notification experience, when determining developer concept knowledge classification.

The concepts I evaluated include fundamental and nuanced concepts. For exceptions and generics, the two more nuanced concepts, it took participants longer to resolve misaligned notifications. It took participants significantly longer to resolve misaligned notifications on exception handling. This was the case for both novices and experts in exception handling. With variables, the fundamental concept, it took participants more time to resolve the notifications aligned with their experience. This may be because 2 of the 3 notifications on variables I showed participants were a) associated with defects the participant was familiar with so they resolved the defect without reading much, if any, of the notification or b) notifications the participant would typically ignore.

The models I presented in Chapter 6 suggest generics may be the most nuanced of the three concepts. Following that theory, it makes sense that it took participants longer to resolve notifications on average with the generics notifications, regardless of which notification they saw. Another explanation for this phenomenon is the participants' familiarity with the defect, as opposed to the notification, based on their prior experience. Participants were generally more familiar with the defects related to variables and exceptions than the defects related to generics.

Familiarity with the defect affected the time it took for participants to resolve notifications. This is related to my prior research that found notification familiarity affects notification resolution [**johnson2016cross**]. However, because I removed the familiar notifications, these findings suggest familiarity with the defect affects resolution time. The affect of defect experience on resolution time was most evident with the concept of generics. Participants were most familiar with the compiler warning provided regarding raw types, therefore could all quickly provide a resolution regardless of the information provided. Novices in generics credited their ability to recognize and resolve the problem to their prior experiences with the problem. Participants were least familiar with the other two defects related to generics, leading to larger times attempting to understand and determine the best resolution.

Table 7.4 Aligned and misaligned notifications resolved by each participant

Aligned				Misaligned		
Notifications Resolved	**Total Notifications**	**Percent Resolved**	**Participant**	**Notifications Resolved**	**Total Notifications**	**Percent Resolved**
5	5	100%	**P1**	4	4	100%
4	4	100%	**P2**	4	6	67%
4	5	80%	**P3**	4	4	100%
5	6	83%	**P4**	3	4	75%
4	5	80%	**P5**	2	4	50%
4	5	80%	**P6**	3	4	75%
5	5	100%	**P7**	3	4	75%
3	5	60%	**P8**	3	4	75%
5	5	100%	**P9**	4	4	100%
2	3	67%	**P10**	3	4	75%
2	3	67%	**P11**	5	6	83%
4	4	100%	**P12**	5	5	100%
6	6	100%	**P13**	3	3	100%
5	6	83%	**P14**	3	3	100%

These findings suggest that knowledge-based notifications do not always significantly decrease the time it takes for developers to resolve defects, but can significantly decrease the time it takes for developers to resolve defects.

H_2 Findings

Overall, novices and experts in a given concept resolved a larger percentage of notifications aligned with their knowledge than notifications misaligned with their knowledge, as shown in Table 7.4. The difference was the largest for experts (8%), who resolved all of the aligned notifications. This difference existed even though participants encountered more aligned notifications than misaligned.

Six of the 14 participants were able to resolve all the aligned notifications presented to them for each concept (Table 7.4). Two of the 6 participants resolved a higher percentage of aligned notifications than misaligned notifications. The remaining four participants resolved 100% of the defects presented to them, whether aligned or misaligned with their knowledge.

For 5 of the 14 participants, the aligned notifications led to resolution 5–30% more often than misaligned notifications; these participants are highlighted in Table 7.4. Four participants were able to resolve all the aligned and misaligned notifications presented to them, only one of which

Table 7.5 Average resolution times (seconds) for aligned and misaligned notifications by participant.

Participant	Average Time to Resolve (Aligned)	Average Time to Resolve (Misaligned)	p-value
P1	147.6	215.25	*0.6018*
P2	120.5	77.25	*0.3245*
P3	238	338.75	*0.5994*
P4	172.8	179.3	*0.9607*
P5	103.75	69	*0.4511*
P6	224.75	110.33	*0.5086*
P7	124.2	75	*0.4537*
P8	108.33	170.33	*0.3871*
P9	47.8	277.25	*0.1836*
P10	35	180.3	*0.0200**
P11	76	111.4	*0.4862*
P12	47.5	132.6	*0.0518**
P13	75.5	59.3	*0.6903*
P14	64.6	130.33	*0.4116*

Table 7.6 Average attempts made toward resolution by participant.

Participant	Average Attempts to Resolution (Aligned)	Average Attempts to Resolution (Misaligned)	p-value
P1	1.8	1.25	*0.388*
P2	1.5	1.3	*0.537*
P3	1.8	4.25	*0.298*
P4	1	2	*0.055**
P5	1.4	1.75	*0.313*
P6	1.6	2.5	*0.809*
P7	1.6	1	*0.304*
P8	2.8	3.25	*0.374*
P9	1	2	*0.089*
P10	1.67	1.5	*0.495*
P11	76	114	*0.576*
P12	1.25	2.4	*0.056**
P13	1	1	*N/A*
P14	1.83	3	*0.243*

was an expert across all concepts. Participants unable to resolve aligned notifications were often conceptual beginners. For many participants in this group, along with being a concept novice, they mentioned not having recent, or any, experience with the defect being presented. This combined with their lacking conceptual knowledge may have affected their ability to resolve these defects in the context and time limit of the study.

These findings suggest that knowledge-based notifications do not significantly increase developers' overall ability to resolve defects but can significantly decrease the attempts novices make to resolve notifications.

H_3 Findings

In order to resolve a defect, developers have to make changes to the source code. These changes, which in real software projects can lead to code churn, can affect the quality of the software [**munson1998code**; **nagappan2005use**]. The less code churn there is, the fewer opportunities there are for developers to introduce new defects. For novices in a given concept, knowledge-based notifications significantly decreased the attempts made to resolve the notifications. Some novices pointed out during this study that when they are unsure of a fix, they would try something and use any errors that follow to refine the solution. Despite the proclamation of this strategy, it took novices in a given concept significantly fewer attempts to resolve notifications aligned with knowledge, averaging 1 attempt as opposed to the 3 attempts for misaligned notifications. Experts took on average 1 attempt to resolve the defects in the study, regardless of alignment with their knowledge. These findings suggest aligned notifications were generally able to lead novices directly to a fix, where the misaligned notifications led to fishing expeditions for a fix.

These findings suggest that knowledge-based notifications do significantly decrease novice developers' attempts to resolve notifications but do not affect the number of expert developer attempts.

The differences discovered in exploring $H_1 - H_4$ are promising and suggestive, despite being in its infancy. This was a first run of user studies in a research area that has not been explicitly explored. Along with the quantitative data discussed above, I collected qualitative data regarding participants' notification preferences to further explore the usefulness of adapted notifications.

7.2.2 Adaptation Preferences

One explicit question I asked participants was their preference regarding the set of notifications provided for each concept. Eight participants selected a version they preferred for each concept. Six participants could not provide a best notification across concepts, typically due to their varying experience with the defects presented.

When the 8 participants were experts in a given concept they preferred the shorter, more concise notifications that were designed for them. As suggested by prior research, experts typically already had plans for resolving the defects [**larkin1980expert**; **Wiedenbeck:1993:Mental**], so they preferred some combination of the problem statement, goal statement, and examples provided. One expert participant, who was classified as an expert in all concepts for the study, said she wished to have had the subgoals for all the notifications, but just as confirmation.

When the 8 participants were novices in a given concept, 90% of the time they preferred the notifications designed for them, with subgoals that walk them through the problem and its solution. Four of the 8 mentioned preferring even moreso the solution-oriented subgoals (i.e. "do this" or "try that") rather than the exploration-oriented subgoals (i.e. "find this" or "look at that"). The handful of times participants did not prefer the notification designed for them noted preferring the expert notification because their prior experience with resolving the defect made the other information unnecessary. All participants found the goal statements provided useful, especially when they were prescriptive (i.e. "Add missing throw statement").

For 6 participants, it was difficult to pick just one version for each concept; most often, it was when participants were expert in a given concept that they had more than one best notification. In 8 instances, participants had varying levels of experience with the defects they had to resolve, which they stated made it difficult across notifications for primarily exceptions and generics. For these participants, the information they preferred depended on the defect they were trying to resolve. This coincides with my previous research, from Chapter 4, that suggested notification experience affects ability to understand and resolve notifications.

Inability to select a best notification may have also been affected by the grouping I selected for my two group set-up. For the purposes of this study, and based on existing research [**mckeithen1981knowledge**], I grouped intermediate developers with experts. There was one participant who was classified as intermediate in generics, based on her score on the concept inventory, but because I chose a two option study design she was considered aligned with expert generics notifications in the study. During her session, she constantly mentioned missing the subgoals when she did not have them. She noted that it was not that she always needed them but that it was nice to have them there when she did. She is also one of the experts that could not decide across concepts which notifications she preferred.

It may be that there is a difference between intermediate expertise in software development and intermediate expertise in other STEM fields.

All participants found the availability of examples in the notification useful, as suggested by previous research [**nasehi2012makes**]; for many this was considered the most valuable information provided. Participants preferred short examples that closely matched the code they were looking at. Four participants noted being distracted by or having difficulty determining the useful parts of examples that were too long or too different from the code they were working in. Similar to findings from [**nasehi2012makes**] on what makes a good example on StackOverflow, some participants wanted explanations with the code examples provided to ease integration into their own code. Two participants requested the examples be partnered with the appropriate subgoal(s).

These findings suggest that some developer adaptation preferences match expectations, based on existing literature, and some do not.

7.2.3 Threats to Validity

The findings from this study are promising and provide insights into the information needs of developers based on their conceptual knowledge. There are threats to the validity of this study that affected the results, which I discuss next.

7.2.3.1 External Threats

This user study evaluated modifying notifications from two tools amongst a number of tools that are available. We also only focused on Java software projects. However, the defects that we chose are reported by various tools and for different languages, increasing my confidence that these findings would generalize to other languages and tools.

Finally, a common concern with empirical studies is the number of participants. Given the time-intensive nature of this study, I was only able to recruit 14 participants. This is on par with other similar studies [**smith2015questions**; **Layman:2007:FaultFix**], though it may seem like a low number of participants. Along the same lines, there was a large number of participants classified as experts in exceptions which led to expert-heavy data for that concept; the same was true for generics novices.

7.2.3.2 Internal Threats

I had to develop a plug-in that was capable of presenting the information I wanted to evaluate in a timely manner, therefore I was unable to implement a fully functional analysis tool. Rather I

created a tool mock up that simulated the behaviors of a real tool. This introduced the potential for unexpected behavior and technical issues when running the application. I ran multiple pilots prior to the actual user studies to work out as many technical issues as possible so the set up was as realistic as possible.

Another technical issue that introduced a threat was the lag that came with conducting the remote interviews.This sometimes affected the time it took participants to resolve the defects. For international participants, there also appeared to be a difference in the keyboard shortcuts typically used in Eclipse, which also affected some participants resolution time.

Due to time limitations, one participant was unable to address all the variables notifications. This also affected my results regarding H1 and H2. However, rather than throwing out this participant's data, I included it to show where value was provided in other notifications.

7.2.3.3 Construct Threats

There also exists the threat of participants having to resolve defects in code that is not their own. This can affect the time it takes and how confident developers may feel about their fix. To mitigate this threat, I chose code snippets for each defect simple enough to be comprehensible in a short period of time but complex enough to still feel realistic.

I designed the notifications for this study using decades old research from Computer Science and other STEM fields. Having to start with research from other fields, and not specifically pertaining to notification design, may have affected the overall findings in particular. It is unclear whether proposed similarities between problem solving in other fields and Computer Science is still valid and apply to tool output. However, there are improvements these findings suggest can be made to potentially further improve communication between developers and their tools.

In some situations, participants could look at the code and recognize what the problem was or might be based on their prior experience. When this happened, some participants did not look at the notification provided prior to resolving; this was often the case with those classified as concept experts. This affected the ability to get feedback on what was and was not useful in the information provided. I attempted to mitigate this threat by asking those who did not use the notification to look at it after the fact for consistent debriefing.

7.3 From "Pipe Dream" to Reality

Findings from this study suggest that notifications modified and presented based on developer knowledge can be used to improve developer ability to understand and resolve defects. Even better,

notifications that were modified to match developer knowledge sometimes reduced the time it took for resolution, suggesting the possibility of helping developers resolve defects more quickly. Along with the general idea of adapting tool output to the developer's level of knowledge, these findings have other implications which I discuss next.

The theory presented in Chapter 4 suggested that lack of knowledge regarding a defect or notification pertaining to a defect can affect developers' ability to resolve them. The findings from this study support these findings and suggest that defect experience may play a greater role in action than could be realized theoretically. The data analyzed in this study suggests defect experience can sometimes make up for lacking conceptual knowledge, suggesting that concept knowledge coupled with defect knowledge may be more effective when determining developer classification. As I expand this area of my research, I plan to explore the feasibility of collecting and using information regarding defect resolution for improved knowledge classification.

Participants classified as novices in a given concept valued, and most often preferred, notifications that included subgoals that walked them through potential solutions to the defect. This aligns with existing research that suggests this is how novices solve problems [**larkin1980expert**], which provides specific content to the notion of providing more information to novices. For some concepts, there is more than a novice to expert progression. In this study, there was a participant who wanted the subgoals to always be available. One way to potentially account for concepts where there is an intermediate stage is to have subgoals always available but only expanded and visible initially for novices. This would also support experts like some of the participants in this study that occasionally found value in the subgoals being present. Participants classified as concept experts preferred notifications that provided concise feedback and confirmation on the problem in the code. These participants rarely went past, and typically preferred, the problem and goal statement when subgoals were available. The only time experts scrolled past the goal and problem statement was when they could not recall what the resolution looked like and wanted to see examples or to confirmation their process or resolution.

Both concept novices and experts valued examples in the notifications, validating the decision to include them in both notifications. For novices, the examples often served as directions for how to implement a fix. For experts that used the examples, they were used as confirmation of fixes they already had in mind or implemented. Novices and experts also agreed that if examples are going to be present, they should be similar to or match the context of the code the defect is attached to. This would allow them to more quickly map what the example was suggested to the proper resolution for their code, something that research has found to be a desired trait [**nasehi2012makes**]. Some participants suggested using general examples that are similar to the code with the defect, while others suggested using examples out of the code base they are in that match or are similar. Another

example related suggestion made by concept novices was to break down the examples to match up with the subgoals. Perhaps further exploration will find novices and experts have preferences regarding the kind of example presented to them.

7.3.1 Challenges to Overcome

One challenge that must be overcome to fully realize adaptive notifications and tools is the challenge of determining the first kind of notification to show developers, either novice or expert. I have shown we can use source code to classify developers based on their knowledge, but tools and environments would need access to both developers' source code and the attributes in developer's source code that matter for that concept.

Another challenge to overcome is how tools will retrieve information to populate the notifications. My user study used manually modified notifications; although this is useful for a study setting it is less useful, and less practical, to have to manually modify all the notifications a tool provides. Also, having to manually update existing notifications might de-motivate toolsmiths from wanting to invest the time to do it. One area for further research is to explore the possibility of automating notification modification. One way this might be done is by borrowing from relevant StackOverflow posts that include steps to resolve similar defects.

In this proof-of-concept study, I chose notifications that communicate about one concept each. In reality, some notifications communicate about multiple concepts at a time. To fully realize the potential of adapting notifications, there needs to be a way to determine all the concepts in a multi-concept notification and what information in the notification maps to which concept. For example, what should tools do with multi-concept notifications when presenting to a developer who is an expert in one of the concepts but a novice in the rest? Ideally, the tool would modify bits of information that communicate about the concepts the developer is less experienced with.

Finally, findings from this study suggest examples are an important and valued part of developers' defect resolution process. Therefore, when examples are included, it is important to find the best, most relevant examples to display. The data collected from this study suggests the best examples for both novices and experts may be those that are similar to or match the code being fixed, are short, and highlight the important parts of the example if not able to be short. Some participants mentioned wanting examples directly from the code base if possible. More focused research should be done to better understand the examples developers find useful and more importantly how tools can access and display these examples.

CHAPTER

8

CONTRIBUTIONS AND FUTURE WORK

This dissertation outlines and describes the findings of four studies designed to provide an in-depth understanding of how developers use tool and ways to improve how notifications support developers during tool use. Each study built on the previous, revealing more about the state of current tools, improvements that can be made, and how we might make the improvements a reality.

In summary, the contributions of this dissertation are as follows:

- A categorized list of reasons developers have for not using program analysis tools, accompanied by tool design suggestions provided by developers.
- An explanatory theory for the challenges that developers encounter when interpreting information provided by tool notifications.
- An approach for assessing developer depth of programming concept knowledge that builds on an existing validated approach for assessing breadth of knowledge.
- An approach, based on existing problem solving research, for adapting tool notifications based on developer concept knowledge classification.

This dissertation focused on one potential route to improving notifications, via making specific changes to tool output and presenting that output to developers based on how much they know

about the concepts present in the notification. However, findings from the studies discussed in this thesis suggest various routes that can be explored and exploited to improve communication between developers and the notifications used by their tools. Based on the contributions from this thesis, below I briefly discuss some ares for future work that build on the theoretical and technical foundations laid by the work completed in this thesis.

8.1 Future Directions

8.1.1 The Big Picture

The first 7 chapters of the thesis laid out a progression of studies that came together to suggest one possible solution for improving how tools communicate with developers. Based on this proposed solution, I first outline future work toward the big picture of adaptive tool notifications.

8.1.1.1 Notification Adaptations

The final study in this thesis suggests the potential for improving communication with developers by using their knowledge to adapt the information provided. One way to improve and further evaluate the effectiveness of notifications adapted to developer knowledge is to provide more accurate developer classifications that are based on more than just the code they write, as discussed in the previous section. Another way to realize the potential of adaptable tool notifications and the support they can provide is to further explore and generalize the design decisions that inform notification adaptations.

Novice and expert developers find value in examples when writing and maintaining their source code [**nasehi2012makes**], as was supported by the findings in Chapter 7. One direction for further research is determining what makes a good example in the context of resolving notifications and the differences, if they exist, between examples experts and novices consider good. Previous research that explored what makes a good examples did so using votes on the site [**nasehi2012makes**]. I propose research that both explores the usefulness of these examples and differences between examples novices and experts find valuable in practice. For example, some participants from the study in Chapter 7 mentioned preferring examples that are similar to the code they are working on. Does this preference propagate across experts and novices? Are there differences between the examples novices and experts find useful for helping them resolve tool notifications? These are the types of questions this direction of research would explore.

Another area for future work in terms of notification adaptations is to determine the concepts in a notifications and how having multiple concepts, where the developer may be more knowledgeable

in one than the other(s), affects overall classification for a given notification. All of the notifications I evaluated my proposed solution with communicated about one programming concept exclusively, which is not always the case. How can we determine all of the concepts relevant to a given notification? How are multi-concept adaptations different from single concept adaptations? These are the kinds of questions this area of research will answer.

8.1.1.2 Incorporating Automation

Another step necessary to realizing the big picture of adaptive tool notifications that better support developers is to determine how models would be incorporated in the process of developing tools and presenting tool output. The study in Chapter 7 assigned developer classification manually, which is not practical or feasible in a real world setting. Another area for future work is exploring the potential for automating model creation and usage. This could be one research study, or split into multiple studies. One study could explore how to automate the creation of a model based on, for example, developers on a given project. The next study would then explore how tools could automatically populate these models to determine developer classification.

Another automation piece that is necessary is for tools to be able to automatically create and present the appropriate notification to a developer, depending on their classification. The study in Chapter 7 laid a foundation for this by determining the kinds of information notifications may need to present. The next step is to determine what information to present in real time along with how, and from where, this information is gathered and presented. This includes determining the concepts relevant to the notification, which informs what classification(s) should be considered for presentation and the goal, subgoals, and best examples to include.

8.1.1.3 Dealing with Change

Knowledge is not stagnant. It fluctuates with our experiences, or lack there of. Therefore, it is necessary for adaptive tools to have that same kind of flexibility. Another direction for future work is to understand how models and tools can keep up with the changes that happen in developer knowledge across time. Along the same lines, it is not clear how to identify when a developer is transitioning or has transitioned from one stage of expertise to another. This is particularly important for developing tools that are able to consistently support developers, rather than becoming useless once a developer has gained or lost knowledge on a given concept. One solution to explore is the possibility of collecting developer experience in real time to populate knowledge models and allow for constant refinement of their classification.

Because knowledge is ever-changing, there may other ways necessary to accommodate developer

information needs. Future research could take a deeper time into information needs, how they may change over time, and developer knowledge changes map, or do not map, with information need changes. Expandable information and feedback loops are two mechanisms that can be used to provide developers with more control over the information provided while potentially providing tools with information they can use to improve communication later down the line.

8.1.2 Developer Knowledge Acquisition

One contribution from my thesis is the notion that we can use the source code developers write relevant to a given programming concept to determine how much they know about that concept. Findings from the last two studies suggest that outside of the code developers write, there maybe other experiences that affect developers' overall software development knowledge such as the defects they have encountered or resolved. One future direction for this research is to further, and more explicitly, explore factors that affect developer knowledge and feasibility of providing information regarding those factors to tools.

If it is possible to collect and use source code contributions, there are other contributions and activities that we can collect from developers to evaluate the role they play in overall knowledge. For example, this thesis found that developer experiences with defect resolution affect their ability to interpret and resolve defects. The approach I proposed for classifying developer knowledge could be improved by collecting defect resolved or issues opend and closed by a given developer and analyzing the relationship between defects that developer has encountered, resolved, and left unresolved and their knowledge pertaining to the notifications she is presented with. Other factors that future research will consider include developer interactions with specific tools and notification text and frequently visited on-line resources.

Research has also found that sometimes knowledge is acquired informally, sometimes without realizing it, through peer interactions [**forman1989role**; **ge2003scaffolding**; **murphy2011peer**]. Therefore, another area of research to explore is how peer interactions affect knowledge that contributes to developers' ability to understand and resolve notifications. Peer interactions come in various forms, from pair programming to casual conversations, so this research would explore the effects on knowledge produced from both settings. It may be that, for example, knowledge acquired during formal activities like pair programming provide knowledge more specific and useful to notification interpretation and resolution than knowledge acquired during informal activities like casual conversation.

8.1.3 Developer Knowledge Classification

For any given defect, there are one or more programming concepts relevant to understanding and being able to resolve that defect. Similar to work on code review assignment [**balachandran2013reducing**], another potential application for the proposed approach for classifying developers' conceptual knowledge is to assign the best developer to resolve a defect or complete a code review. Although knowledge of the code base is important [**fritz2010degree**], my thesis research suggests knowledge of concepts relevant to the defect or code of interest is also important [**johnson2016cross**]. My approach can be combined with other approaches, such as those that look at the developer's familiarity with the code base, to assign defects to developers that are most likely to be able to resolve them. Another direction for this research would be exploring the relationship between the code contributed by a given developer, the code reviews and issues they have completed, and their fit for completing a pending pull request or closing an open issue/defect. This type of evaluation, like the ones completed for the thesis, can be automated and use publicly available data from developers' code repositories.

The ability to classify developer knowledge also opens the door for the potential to more effectively assign student teams in industrial and educational settings. For example, if the design and implementation of a project or piece of functionality requires specific conceptual knowledge, analysis of developers' existing source code can yield information for ensuring someone with the necessary knowledge is on that team. Along the same lines, our approach can be useful for determining effective pair programming pairs. Pair programming is an effective way of transferring knowledge [**plonka2015knowledge**] and fostering tool discovery [**murphy2011peer**], both of which can aide in defect resolution. Knowledge transfer is more likely to occur when the developers paired together differ in experience (i.e. one is novice and one is expert). Furthermore, a previous study on pair programming found that the productivity provided by pair programming can drop substantially when it comes to problem solving if both programmers have experience with the problem at hand; this is especially true if the experiences are recent and has not had a chance to be forgotten [**lui2006pair**]. When it comes to pairing developers for a specific task, our approach can be useful for determining which developer is more expert in the concepts relevant to the task at hand.

APPENDICES

APPENDIX

A

CHAPTER 3 ARTIFACTS

A.1 Pre-Interview Questionnaire

Research Study Pre-Interview Questionnaire

1. Which of the following open-source tools have you used to perform static analysis? (from http://en.wikipedia.org/wiki/List_of_tools_for_static_code_analysis)

   ___ Copy/Paste Detector (CPD)
   ___ FxCop
   ___ StyleCop
   ___ Antic
   ___ Astrée
   ___ BLAST
   ___ Clang
   ___ Frama-C
   ___ Lint
   ___ Splint
   ___ cppcheck
   ___ cpplint
   ___ Checkstyle
   ___ FindBugs
   ___ PMD
   ___ Soot
   ___ JSLint
   ___ JSHint
   ___ CSS Lint
   ___ Other: _____

2. Which of the closed-source tools have you used to perform static analysis? (from http://en.wikipedia.org/wiki/List_of_tools_for_static_code_analysis)

   ___ Axivion Bauhaus Suite
   ___ Black Duck Suite
   ___ BugScout
   ___ Checkmarx
   ___ Coverity
   ___ DevPartner Code Review
   ___ DMS Software Reengineering Toolkit
   ___ Compuware DevEnterprise
   ___ GrammaTech CodeSonar
   ___ Intel Parallel Studio XE
   ___ JustCode
   ___ Klocwork Insight
   ___ LDRA Testbed
   ___ MALPAS
   ___ Parasoft
   ___ Polyspace

___ ProjectCodeMeter
___ Rational Software Analyzer
___ Understand
___ Veracode
___ Visual Studio Team System
___ CodeIt.Right
___ CodeRush
___ NDepend
___ FlexeLint
___ Green Hills Software DoubleCheck
___ PC-Lint
___ JTest
___ SonarJ
___ ESC/Java or ESC/Java2
___ SofCheck Inspector
___SPARK Toolset
___ Other: _____

3. Have you ever worked on a software development team?

   ___ Yes
   ___ No
4. Do you currently work on a software development team?

   ___ Yes
   ___ No
5. Have you ever had to use a static analysis tool while on a software development team?

   ___ Yes
   ___ No
6. What tool will you be using for in the interview? (Required)
   _____

A.2 Interview Script

Hello. My name is Brittany and this is Yoonki. We are currently doing research on how to improve the usability of static analysis tools. As a part of our research, we are conducting a series of interviews to get feedback from programmers, developers and static analysis tool users. Thank you for taking time out of your schedule to let us interview you.

During the course of this interview, we may ask you especially provocative questions, such as suggesting what you've told us is inconsistent with (quote-unquote) "best practice." We do not mean to to insult or offend you, but instead to try to make you think deeply about why you do what you do. Try not to take anything personal and answer as best you can; there are no right answers.

Questions and Short Responses

For the first part of the interview, we're going to ask you some questions about your experience using static analysis tools. For now, we would like to focus on your own personal experience and opinions as a developer. Later in the interview, you will get the chance to be creative and suggest improvements for static analysis tools. We are dedicating 20 minutes to this section so try and keep your answers short, sweet and to the point.

<u>Experience Questions</u>

- Can you tell me about your first experience with static analysis?

- How easy/difficult was your first experience? Can you remember what made you feel that way?

- Do you have experience working on a software team? Did they use static analysis tools?

→ if yes, did you find it useful? (In your opinion did the tool accommodate teamwork(communicating standards/rule sets to be used)?)

→ if no, do you think a team could benefit from using static analysis tools during development?

- When you first started working at Google, was static analysis a part of the development process?

→ Did you have any difficulties or frustrations with using the static analysis tools provided?

→ Did the process/tools used make it clear what kind of bugs are a priority for Google? If not, how did you come to understand what you should and shouldn't be looking for?

Usage Questions

- When was the last time you used a static analysis tool?

- Have you used any other tools besides the ones you've already mentioned (name them)?

- Do you have a specific role as a developer (if so, what is it?)?

→ Do you feel the tool(s) used benefit what you do as a developer?

- Do you use your own static analysis or style checking tool (outside of the Google process)?

→ What is your tool of choice? Why do you prefer this tool?

→ Is there anything about this tool that you don't like, or find difficult?

- Do you use a static analysis tool for all the software you write? If not, why do you use it for some and not others?

- Are there any aspects of using static analysis tools that you describe as painful or difficult?

- Have you ever consciously avoided using static analysis? (Example?) Why do you think you avoided using it?

- Do you use/prefer static analysis tools that (a) run continuously in your IDE, (b) run in your IDE when you tell them to run, (c) as a part of your build process or (d) other?

Other Questions

- What type of bugs do you feel are, or should be, the priority at Google? (Feel free to specify both if you feel the type that are priority shouldn't be)

→ Do you feel the tools being used are finding these types of warnings?

- How did you find out about the static analysis tool you use?

- Do you use any other methods of debugging/finding errors in your code? If so, why do you prefer your method to using a static analysis tool?

- Some have suggested that static analysis tools should find bugs as early as possible in the development cycle. What would you define as 'earliest'? (When would you like for a static analysis tool to notify you of bugs in your code?)

- What, in your opinion, are the critical characteristics of a good static analysis tool?

- Can you describe the process involving static analysis that code undergoes here at Google?

- What are your goals when using static analysis?

→ Do you feel the tools you use now fulfill these?

→ If not, what's missing?

Interactive Interview (current workflow)

During this portion of the interview we would like to see you in action. We have dedicated 20 minutes to this section as well. (** Maybe see if they can use tool on real code and fix a real defect**)

If the subject has agreed to use their own code/tool:
Can you go ahead and pull up some code? As you're doing this we're going to ask you to do some things and ask some questions pertaining to how you do things.

If subject did not want to use their own code:
Here is the code we told you we would bring...would you mind running the tool on the code? While you do this we are going to ask you some questions.

Feel free to explain what you're doing out loud so we can get a better understanding of your workflow/thought process.

- Do the tools you use undergo any sort of evaluation before or after integration into your process? What about the tools Google uses?

→ Does your methods in any way reflect what you have to do in the Google process?

- Now that you've run the tool and gotten your feedback, what would your next move(s)? (Do you look at all the warnings? some, but not others? Fix all?)

- Do you configure the settings of your tool from default settings? (if so, show me)

→ About how much configuring would you say you have to do before you're satisfied?

- Does your static analysis tool or process help in assessing what to do about a warning?

→ On average, how long would you say it takes you to figure out what to do about the warnings reported from your static analysis tool(s)?

→ FindBugs is in the process of adding Quick Fixes to their tool. Do you feel this would be helpful? What about partial quick fixes? What kind of solutions would you be looking for?

- What would you consider an overwhelming number of warnings?

→ Have you ever had an overwhelming number of warnings come up? (If so, how did you react?)

- Are there any guidelines or methods in place for keeping track of warnings that are new versus warnings that are old but still showing up? Does your process report when bugs are fixed?

→ If not, would this sort of mechanism be beneficial?

Do you feel like you need a break before we begin the last portion of the interview?
(If yes, take 5 minute intermission)

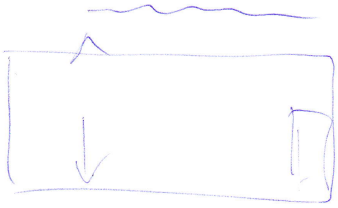
Participatory Design (desired workflow)

Now that we've got you in the mindset of programming using static analysis, we would like to allow you to be creative through a design exercise. We are going to give you a blank sheet of paper and we would like for you to describe to us your ideal workflow using a static analysis tool and draw a simple sketch of how you want to interact with the tool.

Again, we would like to thank you for your time. Do you have any questions for us?
(If no) Do you mind if we keep your drawing for further analysis?

A.3 Participatory Design Sketches

1
2
3
4

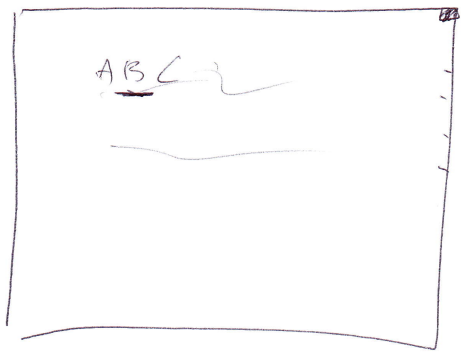


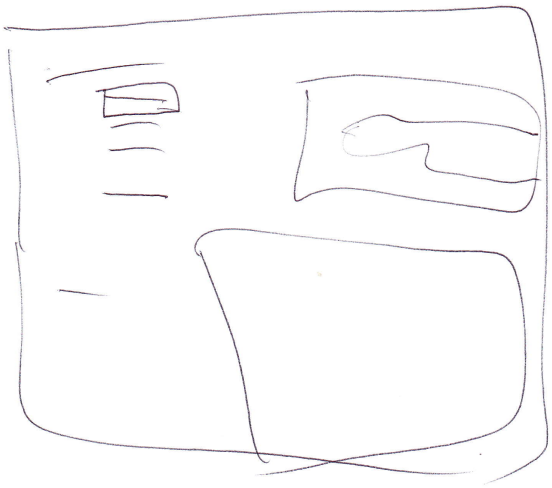
5
6
7
8

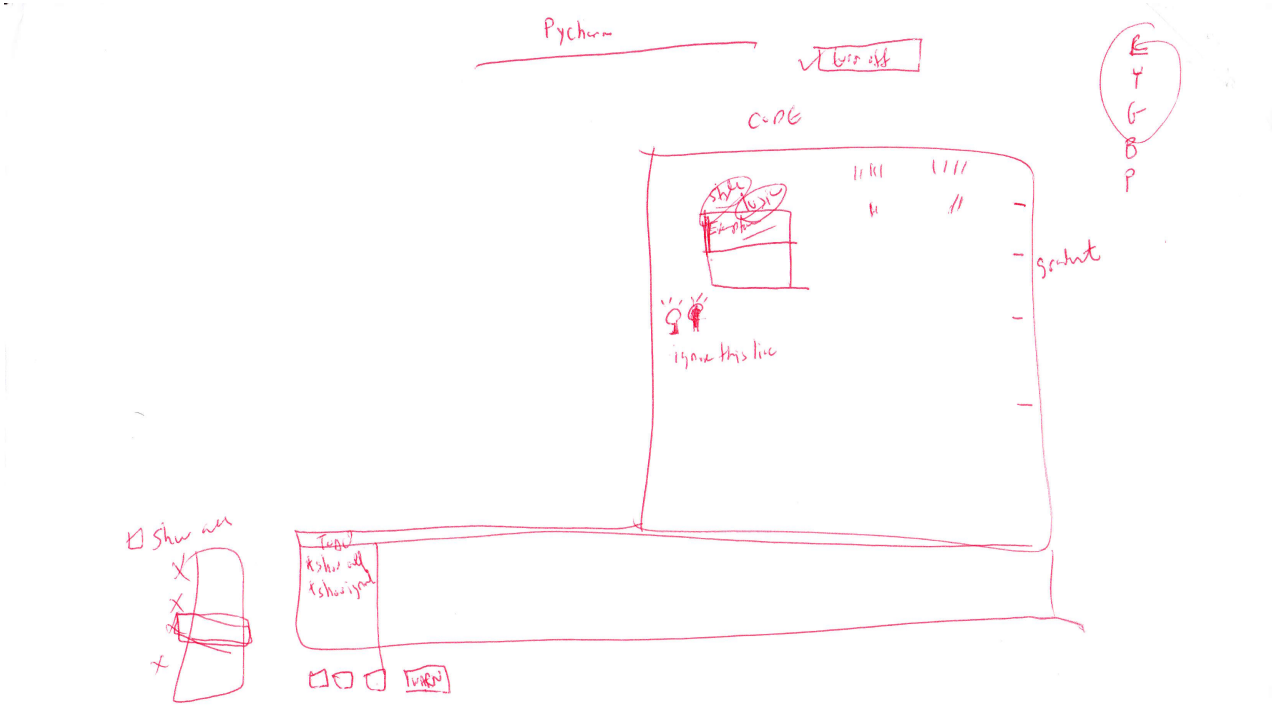
"inline diffs"

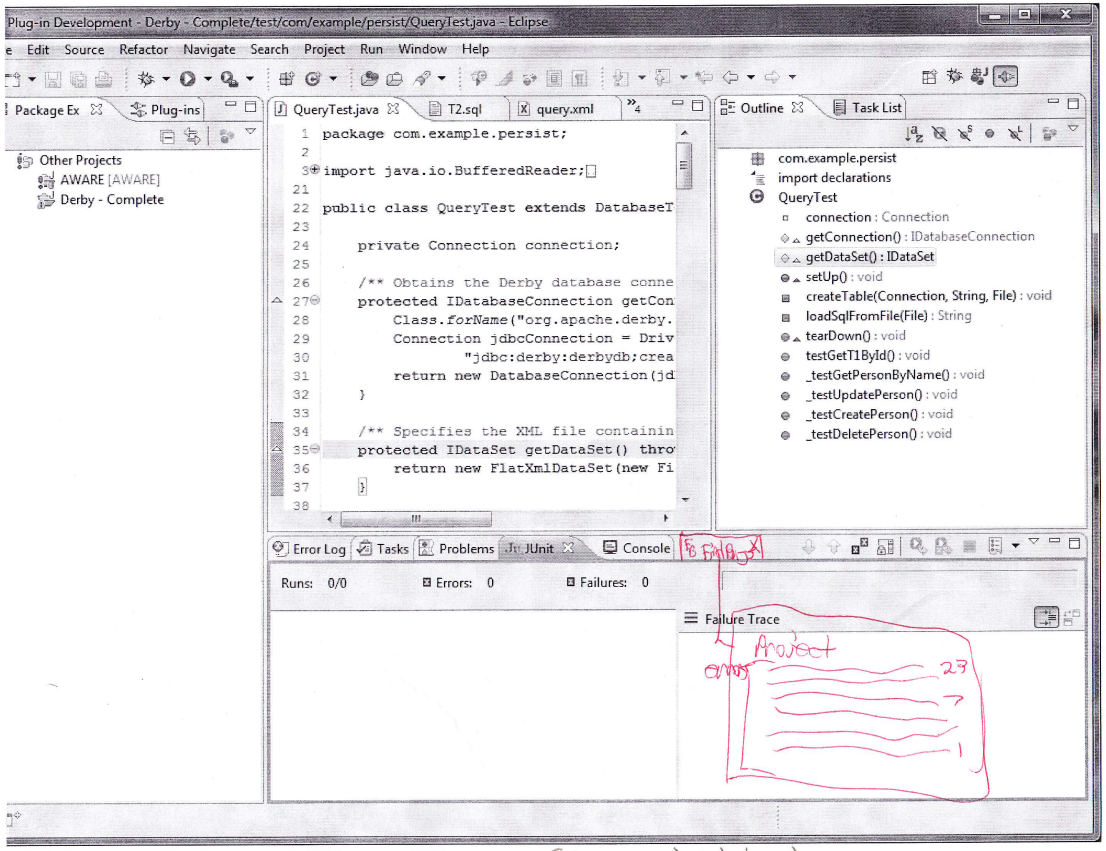
foo.java
@@
-
-
+
+
+

bar.java
@@ 59
-
-
+
+

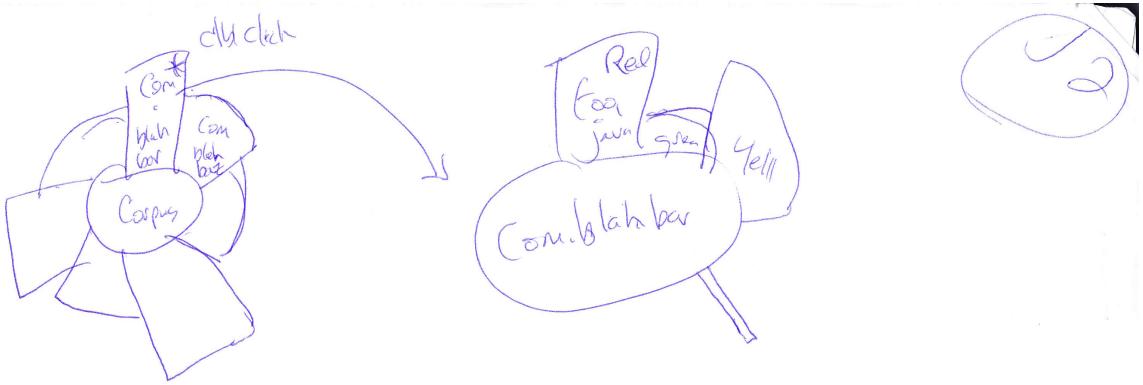

ABC

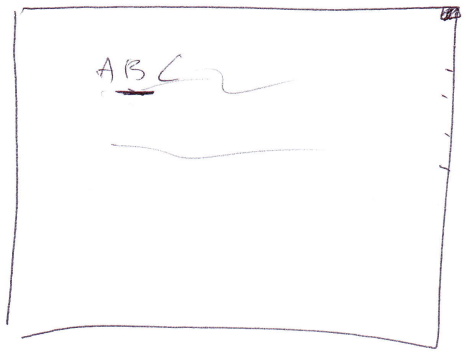


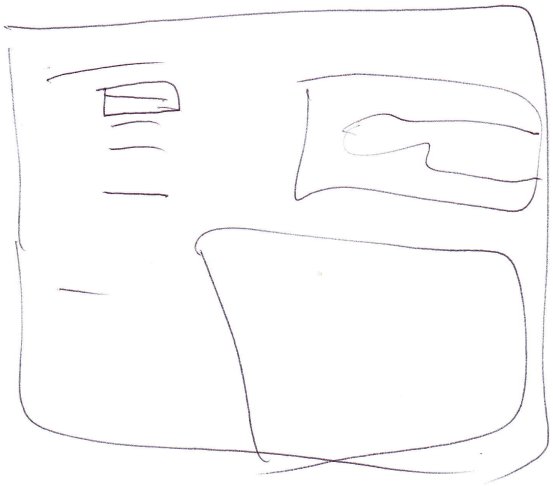

Pycharm
CODE
ignore this line

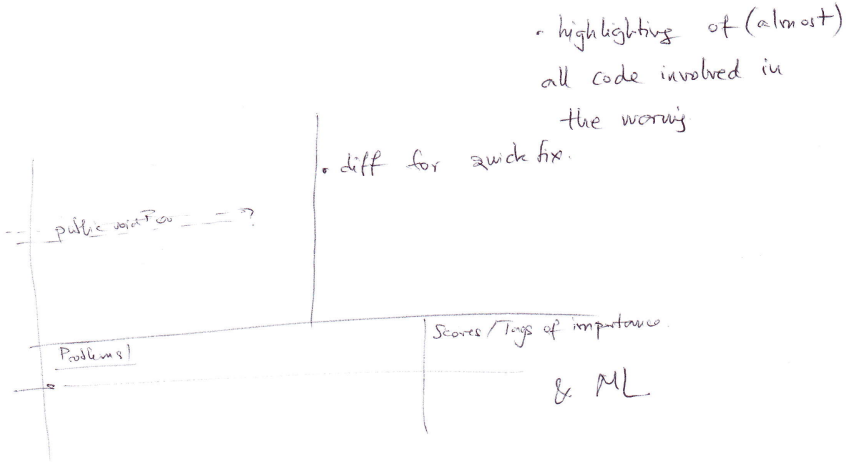


color flags (for priorities)
in Eclipse pane


clik clak
Com
blah
bar
Com
blah
baz
Corpus
Real
Egg
Java
Yell
Com.blah.bar


ABC



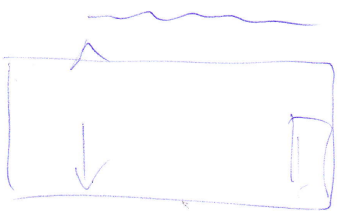


"inline diffs"

foo.java
@@
-
-
+ buffer.append
+ for(...
+

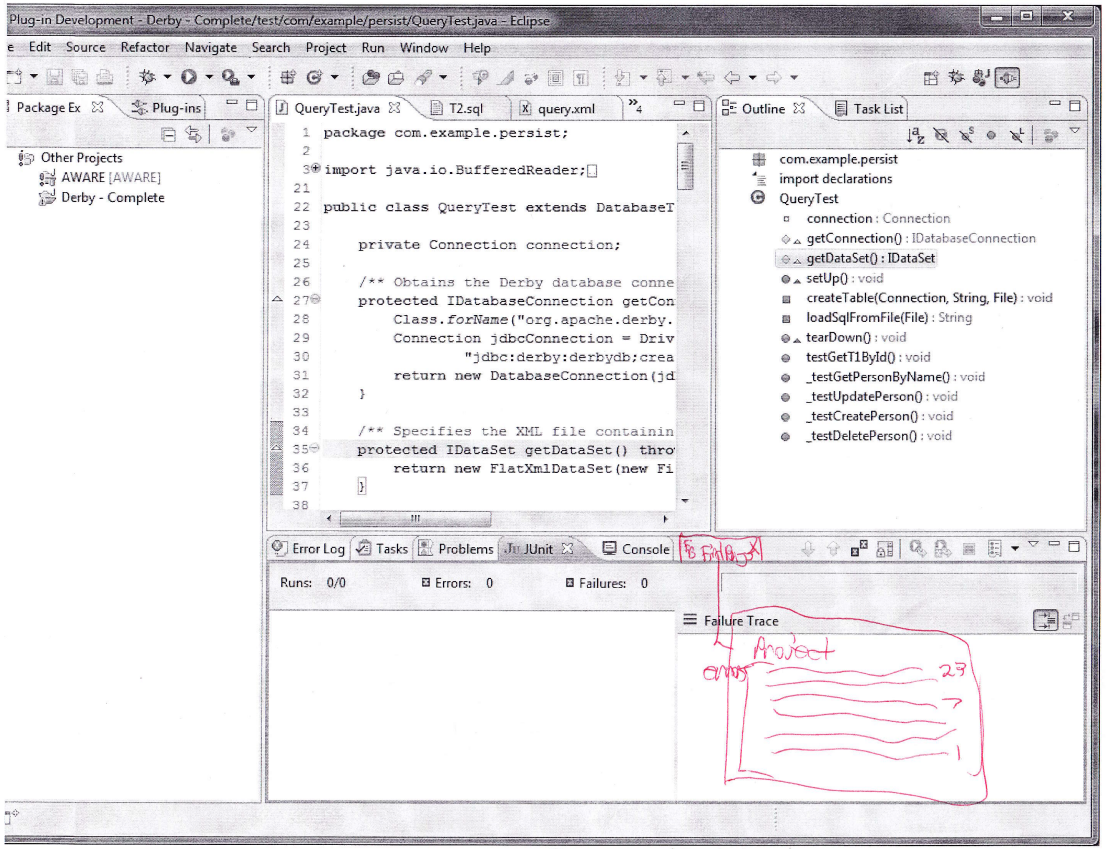
bar.java
@@ -9
-
-
+
+

1
2
3
4



5
6
7
8

- open perspective automatically
- integrated w/ IDE (don't want to have to setup my code)
- highlighting affected code
- Code suggestion / approach
- Ignore this bug button
  - ↳ be able to enter a reason/comment
- Links to more detail
  - educate some developers
    - "as needed training"
- Always



color flags (for priorities) in findBugs pane

APPENDIX

B

CHAPTER 4 ARTIFACTS

B.1 Notification Oracle

Categories:

- **Pointers/References**
- **Multi-threading**
- **Null/Pointers/References**
- **Dead Code**
- **Generics**
- **Inheritance/Polymorphism**
- **Serialization**
- **Test Coverage**

Notification 1 (string comparison using == or !=):

```
4613                    String name = ac.getAccessibleName();
4614                    if ((name == null) && (name != "")) {
4615                        return ac.getAccessibleName();
4616                    } else {
4617                        return null;
4618                    }
```

This code compares java.lang.String objects for reference equality using the == or != operators. Unless both strings are either constants in a source file, or have been interned using the String.intern() method, the same string value may be represented by two different String objects. Consider using the equals(Object) method instead.

You shouldn't compare strings using == or != because it is only comparing the reference not the actual string itself. Comparing strings is done using the .equals() method.

Notification 2 (incorrect lazy initialization):

```
105
106         if (managingFocusForwardTraversalKeys == null) {
107             managingFocusForwardTraversalKeys = new HashSet<KeyStroke>();
108             managingFocusForwardTraversalKeys.add(
109                 KeyStroke.getKeyStroke(KeyEvent.VK_TAB, 0));
110         }
```

This method contains an unsynchronized lazy initialization of a static field. After the field is set, the object stored into that location is further updated or accessed. The setting of the field is visible to other threads as soon as it is set. If the futher accesses in the method that set the field serve to initialize the object, then you

have a very serious multithreading bug, unless something else prevents any other thread from accessing the stored object until it is fully initialized.

Even if you feel confident that the method is never called by multiple threads, it might be better to not set the static field until the value you are setting it to is fully populated/initialized.

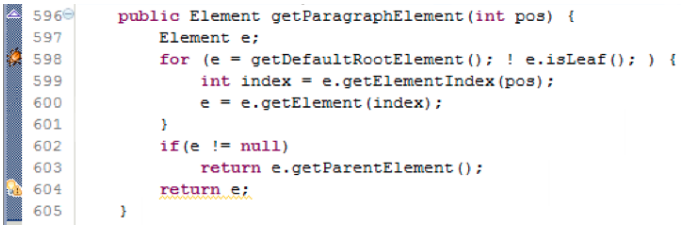
You are initializing a static variable without a synchronizing it, which if you trying to do lazy initialization is incorrect as the way the code is written now more than one of these objects can be created.

Notification 3 (Synchronize on a mutable field):

This method synchronizes on an object referenced from a mutable field. This is unlikely to have useful semantics, since different threads may be synchronizing on different objects.

It is possible for more than one of these objects to have been created or changed before it is actually synchronized on so there is no telling what it is in fact being synchronized.

Notification 4 (Redundant null check):

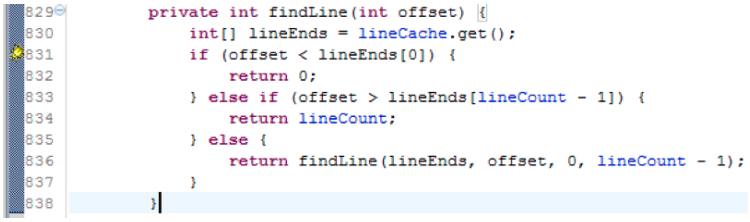


A value is checked here to see whether it is null, but this value can't be null

because it was previously dereferenced and if it were null a null pointer exception would have occurred at the earlier dereference. Essentially, this code and the previous dereference disagree as to whether this value is allowed to be null. Either the check is redundant or the previous dereference is erroneous.

The null check you are doing is not needed or misplaced. If e was null the code would break before reaching the null check. You should consider removing the null check and handing potential exception when e is dereferenced

Notification 5 (Possible null pointer dereference):



The return value from a method is dereferenced without a null check, and the return value of that method is one that should generally be checked for null. This may lead to a `NullPointerException` when the code is executed.

You are trying to access data that may not exist. You should check lineEnds[0] for null before trying to access it.

Notification 6 (Unused code):

You are not using (or reading from) this variable anywhere in this class (it's a private variable so it's not being used outside this class either). You could remove it to get rid of the error and the code would work the same.

Notification 7 (Parameterized/Raw type):

```
46      private Vector<String> _contexts;
47
48      public ContextListImpl(org.omg.CORBA.ORB orb)
49      {
50          // Note: This orb could be an instanceof ORBSingleton or ORB
51          //TODO COMP Task 2
52          _orb = orb;
53          _contexts = new Vector(INITIAL_CAPACITY, CAPACITY_INCREMENT);
54      }
```

```
50          _orb = orb;
51  Multiple markers at this line
52    - Type safety: The expression of type Vector needs unchecked conversion to conform to Vector<String>
53    - Vector is a raw type. References to generic type Vector<E> should be parameterized
```

You created a generic object Vector<String> but did not properly initialize it. The new Vector should be new Vector<String>.

Notification 8 (unimplemented methods):

```
37  class DirectByteBuffer extends MappedByteBuffer
38                         implements DirectBuffer
39
```

```
37  The type DirectByteBuffer must implement the inherited abstract method DirectBuffer.viewedBuffer()
38                         implements DirectBuffer
```

You are implementing a class (DirectBuffer) but not implementing all the required methods (viewedBuffer). If you implement this method the error will go away.

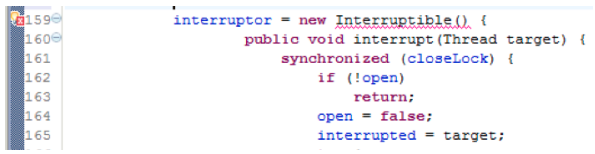
Notification 9 (serializable class needs serial ID):

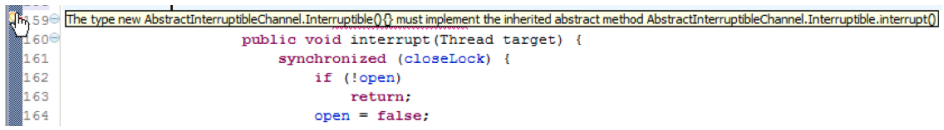
```
44  public class DynAnyBasicImpl extends DynAnyImpl
45  {
46
```

```
44  The serializable class DynAnyBasicImpl does not declare a static final serialVersionUID field of type long
45  {
```

Somewhere down the line of classes/interfaces being implemented/extended from this class, Serializable is being implemented. Proper usage of this interface requires a serialversionUID during deserialization to ensure that the classes loaded are compatible with respect to serialization.

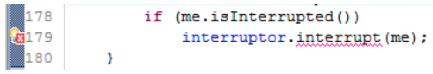
Notification 10 (unimplemented methods):

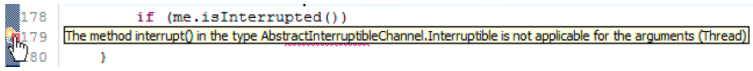




There are methods from the interface you are trying to instantiated as an anonymous class that you are not implementing. You should implement an interrupt method with no parameters or change the method signature in the interface.

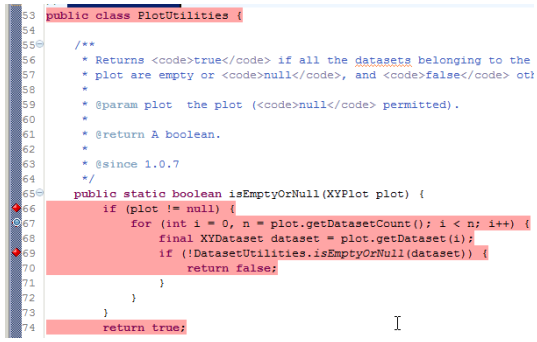
Notification 11 (method not applicable for arguments):





You are trying to call an interrupt method that is not expected for this Interruptible object. You should either call the method you implemented with no parameters or make sure the method in the interface matches this one.

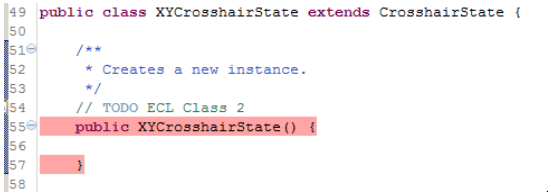
Notification 12 (Red class with red header):



```
53 public class PlotUtilities {
54
55     /**
56      * Returns <code>true</code> if all the datasets belonging to the
57      * plot are empty or <code>null</code>, and <code>false</code> oth
58      *
59      * @param plot  the plot (<code>null</code> permitted).
60      *
61      * @return A boolean.
62      *
63      * @since 1.0.7
64      */
65     public static boolean isEmptyOrNull(XYPlot plot) {
66         if (plot != null) {
67             for (int i = 0, n = plot.getDatasetCount(); i < n; i++) {
68                 final XYDataset dataset = plot.getDataset(i);
69                 if (!DatasetUtilities.isEmptyOrNull(dataset)) {
70                     return false;
71                 }
72             }
73         }
74         return true;
```

You have not instantiated an instance of this class (default constructor) nor have you called any of the methods.

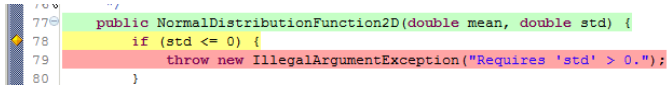
Notification 13 (Red class--constructor only):



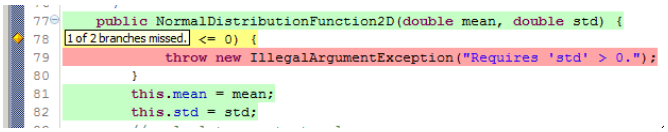
```
49 public class XYCrosshairState extends CrosshairState {
50
51     /**
52      * Creates a new instance.
53      */
54     // TODO ECL Class 2
55     public XYCrosshairState() {
56
57     }
58
```

You have not created/instantiated an instance of this class (implemented constructor)

Notification 14 (Simple if statement 1 of 2 branches):

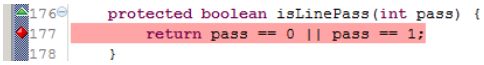


```
77     public NormalDistributionFunction2D(double mean, double std) {
78         if (std <= 0) {
79             throw new IllegalArgumentException("Requires 'std' > 0.");
80         }
```



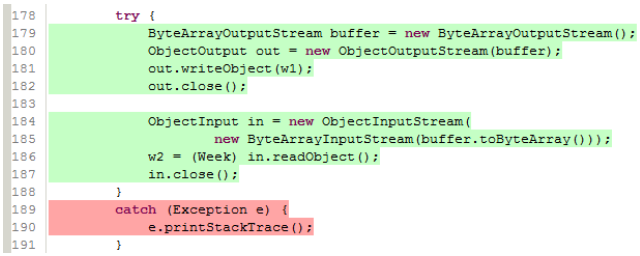
You are only executing one branch of this 2 branch if statement (the false branch). You should run the method with input(s) that will execute the true branch of the if.

Notification 15 (Short circuit return statement):



In the case of the notifications they look at, the methods are not being called. However each return statement mentions branches; 2 branches means you need to test pass = that number and also pass != to that number. 4 branches may mean when each part of the return statement returns true and false.

Notification 16 (try/catch -- no exception caught):



The try block has executed and no exception was caught so the catch block did not execute.

Notification 17 (try/finally -- exception thrown, partial coverage in finally):

```
296         try {
297             Week w = new Week(new Date(11361098300001),
298                     TimeZone.getTimeZone("GMT"));
299             assertEquals(2005, w.getYearValue());
300             assertEquals(52, w.getWeek());
301             assertFalse(true);
302         }
303         finally {
304             Locale.setDefault(saved);
305         }
306     }
```

The try attempted to execute but failed, which led to the finally being executed and then exiting the method. Because only failure of the try and execution of the finally was tested, the inside of the finally is yellow. If this same test were called twice, once with an exception and once without, presumably at least the inside of the finally would be green. The red bracket at the end of the method suggests that the method exited after executing the finally.

Notification 18 (try/finally -- try executed, partial coverage in finally):

```
314         try {
315             TimeZone zone = TimeZone.getTimeZone("GMT");
316             GregorianCalendar gc = new GregorianCalendar(zone);
317             gc.set(2005, Calendar.JANUARY, 1, 12, 0, 0);
318             Week w = new Week(gc.getTime(), zone);
319             assertEquals(53, w.getWeek());
320             assertEquals(new Year(2004), w.getYear());
321         }
322         finally {
323             Locale.setDefault(saved);
324         }
325     }
```

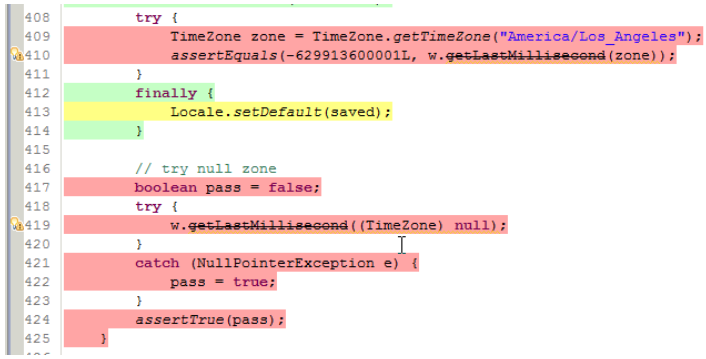
This is the opposite of N17. The try did execute which means the finally does not. Because this code was only called once (with no exception) the inside of the finally is yellow.

Notification 19 (try/catch -- exception caught):

```
378         try {
379             w.getFirstMillisecond((Calendar) null);
380         }
381         catch (NullPointerException e) {
382             pass = true;
383         }
```

The try attempted to executed but failed; an exception was thrown and caught so the catch block was executed.

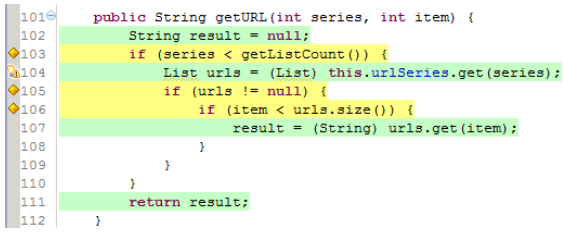
Notification 20 (try/finally -- method exits):



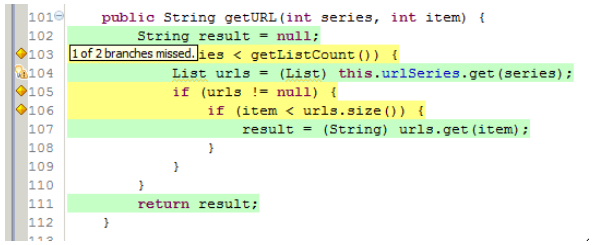
```
408         try {
409             TimeZone zone = TimeZone.getTimeZone("America/Los_Angeles");
410             assertEquals(-629913600001L, w.getLastMillisecond(zone));
411         }
412         finally {
413             Locale.setDefault(saved);
414         }
415
416         // try null zone
417         boolean pass = false;
418         try {
419             w.getLastMillisecond((TimeZone) null);
420         }
421         catch (NullPointerException e) {
422             pass = true;
423         }
424         assertTrue(pass);
425     }
```

This is similar to N17 except here there is more code in the method instead of just the closing bracket so we can see more clearly that the method exited once the finally executed.

Notification 21 (Nested if statements):



```
101     public String getURL(int series, int item) {
102         String result = null;
103         if (series < getListCount()) {
104             List urls = (List) this.urlSeries.get(series);
105             if (urls != null) {
106                 if (item < urls.size()) {
107                     result = (String) urls.get(item);
108                 }
109             }
110         }
111         return result;
112     }
```



Only the true branches of each conditional executed (the only path that does anything).

B.2 Pre-Questionnaire and Consent Form

ExpNote: Consent Form and Pre-Questionnaire

This form is used to sign up for a study on the expressiveness and scalability of program analysis tools.

To complete this study, you must:
- Have experience using Eclipse IDE
- Be familiar with FindBugs and/or EclEmma
- Be at least 18 years of age

* Required

By the definition above, I am eligible to participate. *

- ○ Yes
- ○ No

Continue »

ExpNote: Consent Form and Pre-Questionnaire

* Required

Consent Form

North Carolina State University
INFORMED CONSENT FORM for RESEARCH
Title of Study: Expressive and Scalable Notifications for Program Analysis Tools

Principal Investigator: Brittany I. Johnson

Faculty Sponsor: Emerson Murphy-Hill and Sarah Heckman

What are some general things you should know about research studies?
You are being asked to take part in a research study. Your participation in this study is voluntary. You have the right to be a part of this study, to choose not to participate or to stop participating at any time without penalty. The purpose of research studies is to gain a better understanding of a certain topic or issue. You are not guaranteed any personal benefits from being in a study. Research studies also may pose risks to those that participate. In this consent form you will find specific details about the research in which you are being asked to participate. If you do not understand something in this form it is your right to ask the researcher for clarification or more information. A copy of this consent form will be provided to you. If at any time you have questions about your participation, do not hesitate to contact the researcher named above.

What is the purpose of this study?
Research and industry have shown that program analysis tools, such as static analysis tools and refactoring tools, can be have a huge effect on the quality of software. Program analysis tools communicate with software developers through notifications but these notifications suffer from two problems: they are often not expressive enough so developers waste time trying to understand what the tool is telling them, and often not scalable so the developer is forced to fight off numerous notifications before finding the most useful ones. Our research aims to observe and understand the methods programmers use to explain and understand program analysis tool notifications.

What will happen if you take part in the study?
If you agree to participate in this study, you will be asked to explain 18 different program analysis tool notifications from Eclipse's compiler, FindBugs and EclEmma. Notifications can be explained in a variety of ways, including pointing, gesturing, drawing or coding. One session lasts approximately 1 hour and will be recorded Audacity for audio recording and Camtasia for screen recording.

Risks
There are no risks we foresee in conducting this user study.

Benefits
There will be no direct benefits to you for your participation in this study. We will use the information we gather from this study as a basis for the design of tools and activities that would be available to software developers.

Confidentiality
Information we obtain in the study will be kept confidential to the fullest extent allowed by law. Data , including video recordings, will be stored securely in a locked laboratory and on password secure servers. For your protection, we do not plan to include your face or mention your name in any recordings. We will maintain a confidential spreadsheet that links the real names and email addresses of participants with an assigned pseudo-name that will be used in publications. This spreadsheet will be kept entirely separately from any comments you make during the study. You will be given the opportunity to allow the researchers to anonymously publish some or all the data.

Compensation
You will not receive anything for participating.

What if you have questions about this study?
If you have questions at any time about the study or the procedures, you may contact the researchers, Brittany Johnson, at EB II 3222 / [919/817-8371].

What if you have questions about your rights as a research participant?
If you feel you have not been treated according to the descriptions in this form, or your rights as a participant in research have been violated during the course of this project, you may contact Deb Paxton, Regulatory Compliance Administrator, Box 7514, NCSU Campus (919/515-4514).

May we record your session for future analysis (screen and audio)? *
Only the PI will be analyzing the recordings; once analyzed, the recordings will be destroyed any relevant data will be stored on password-protected servers.

- ○ Yes
- ○ No

Informed Consent *

☐ "I have read and understand the above information. I can save a copy of this webpage for my records. I agree to participate in this study with the understanding that I may choose not to participate or to stop participating at any time without penalty or loss of benefits to which I am otherwise entitled."

Email Address *

« Back Continue »

ExpNote: Consent Form and Pre-Questionnaire

* Required

Pre-Questionnaire

Before participating in the study, we would like to know a few things about you.

Which of the following Integrated Development Environments (IDEs) for Java have you used? *

For more information, please visit
http://en.wikipedia.org/wiki/Comparison_of_integrated_development_environments#Java

- [ ] BlueJ
- [ ] Eclipse
- [ ] Greany
- [ ] Greenfoot
- [ ] IntelliJ IDEA
- [ ] JBuilder
- [ ] JCreator
- [ ] JDeveloper
- [ ] KDevelop
- [ ] NetBeans
- [ ] Rational Application Developer
- [ ] Servoy
- [ ] XCode
- [ ] Other:

Which of the following automated program analysis tools for Java have you used? *

For more information, please visit
http://en.wikipedia.org/wiki/List_of_tools_for_static_code_analysis

- [ ] FindBugs
- [ ] PMD
- [ ] CheckStyle
- [ ] Eclipse built-in refactoring tool
- [ ] EclEmma
- [ ] Cobertura
- [ ] Soot
- [ ] JTest
- [ ] Sonar J
- [ ] Other:

How often do you use... *

	Every day.	More often than not.	Every once in a while.	Rarely.	Never.
Eclipse IDE	○	○	○	○	○
Java Compiler	○	○	○	○	○
FindBugs	○	○	○	○	○
EclEmma	○	○	○	○	○

How many years of software development experience do you have? *
Please enter a number in years. (e.g. 2 or 3.5)

How many years of experience do you have using Eclipse IDE? *
Please enter a number in years. (e.g. 2 or 3.5)

How many years of experience do you have using Java compiler? *
Please enter a number in years. (e.g. 2 or 3.5)

« Back Continue »

ExpNote: Consent Form and Pre-Questionnaire

Submit Form

If you are satisfied with your answers, please use the submit button below.

« Back Submit

B.3 Session Script

Phase 2 Session Script/Talk Points

Hello! My name is _____ and I am currently conducting research in order to understand how expressiveness and scalability can be increased in and across program analysis tools and, if it can, how it affects developers' ability to create software. I would like to thank you in advance for taking the time out of your schedule to help us with this research. We have been having sit down sessions with software developers; during these sessions, we set up screen and audio recording software, as we have here, present you with some code being analyzed by a software tool and ask you to explain what is going on. Is this okay with you?

Before we begin, you can see on the right that there is a list of 18 tasks from 3 different tools. It has taken past participants approximately 1 hour to complete these tasks. I will be keeping track of time if you are pressed for time, however 1 hour is by no means the time limit so do not feel pressured or rushed.

You can use any form of communication you are comfortable with, including pointing, gestures, coding or drawing. If at any time you feel that you would like to go on to the next notification, let us know by saying the keyword 'perfection' and we will continue. We do not want you to feel pressured, as this is not a test of your ability as a developer but an attempt to understand how well program analysis tools are communicating with developers and learn what these tools can be doing to better communicate with you. [*remote sessions only]* We may at times need to intervene to ensure the connection is still maintained however we will attempt to wait until you have come to some sort of stopping point in your conversation or otherwise to do so.

Now, if you're ready, we can get started…

There are a list of Tasks in the Task view. Starting with the FB TODOs, let's go down the list ...these are FindBugs errors that have been found in an open source project. Let's start with the bugs we see here in this class. Can you explain what these errors are trying to tell you? [*Go through all FB TODOs*]

FB Task 1

- How confident are you in your explanation?

FB Task 2

- How confident are you in your explanation?

FB Task 3

- How confident are you in your explanation?

FB Task 4

- How confident are you in your explanation?

FB Task 5

- How confident are you in your explanation?

Let's bring up the first task for Eclipse we're going to look at in the list of COMP TODOs. There are a number of compiler errors found by Eclipse. Can you explain what is going

on/why this class won't properly compile? [*Go through all COMP TODOs*]

COMP Task 1

- How confident are you in your explanation?

COMP Task 2

- How confident are you in your explanation?

COMP Task 3

- How confident are you in your explanation?

COMP Task 4

- How confident are you in your explanation?

COMP Task 5

- How confident are you in your explanation?

COMP Task 6

- How confident are you in your explanation?

Okay, now we're going to take a look at the coverage tool EclEmma using more open source projects. These tasks are labeled ECL TODOs. We have a couple of classes here with various levels of coverage. Let's start with the first one. What's going on there? [*Go through all ECL TODOs*]

ECL Class 1

- How confident are you in your explanation?

ECL Class 2

- How confident are you in your explanation?

ECL Class 3

- How confident are you in your explanation?

ECL Class 4

- How confident are you in your explanation?

ECL Class 5

- How confident are you in your explanation?

ECL Class 6

- How confident are you in your explanation?

ECL Class 7

- How confident are you in your explanation?

<u>Talk Points</u>

- I'm not sure if I understand what you mean by that…can you explain a bit more?

- Did the notification/the way the tool explained the problem confirm or change what you understood about the problem?

- Yeah, this one is a bit confusing...would you like to try fixing it to see if you what you think is the problem actually is the problem or would you like to continue to the next piece of code?

- I don't know much about this subject…how would you explain this to me? Is there a better way to convey this problem than how it is explained here?

- I can tell you know what the problem is, but you seem to be having a hard time explaining it in words…what do you think is the best way to explain a problem like this?

- We've spent a lot of time on this notification and I want to make sure we stay on schedule so we should probably move on to the next one and then we can come back to this one if we have time.

And that was the last code segment. [*If any notifications were skipped and there's time, go back to them now*]

Do you have anything else you would like add or any questions you have for us?

[*If not*]
All right, then that's it. Again thank you for you time and participation!

<u>Debriefing</u> (keep recording)

So, now that we have complete the study, are there any questions you have for us?

Just to re-iterate, we were not observing your ability as a programmer; we were observing how you understand and react to tool notifications. That being said, I recall you saying that you wanted to know whether your explanation was correct [*give explanation of ones they were interested in*]

Questions/Talking Points for debfriefing:
- Were there any notifications that you felt were the least helpful/useful? Were there any that were not clear?

- Is there a better way to explain this in your opinion?

- Is there information that you would have expected to have been available that was not?

- What made this notification easy to understand / this notification difficult to understand?

- Did you find it distracting having notifications from other tools present when attempting to work with one tool? (*make note of if this actually occurs during the session*)

I will be creating a transcription of our session and using the data from screen capture and my transcriptions for my research. The data will be anonymized and only viewed by myself or my research advisors. If you have any concerns about confidentiality please let me know so we can address it accordingly.

APPENDIX

C

CHAPTER 6 ARTIFACTS

C.1 Example Concept Inventory (Generics)

1. Consider the following code (Oracle):

```
public class Bucket{
    private Object object;

    public void set(Object object) { this.object = object; }
    public Object get() { return object; }
}
```

Which of the following is an implementation of a generic version of this class that doesn't have compiler warnings or errors?

a.
```
public class Bucket<> {

    private Object t;

    public void set(Object t) { this.t = t; }
    public Object get() { return t; }
}
```

b.
```
public class Bucket<T> {

    private T t;

    public void set(T t) { this.t = t; }
    public T get() { return t; }
}
```

c.
```
public class Bucket<Object> {

    private Bucket<Object> t;

    public void set(Bucket<Object> t) { this.t = t; }
    public Bucket<Object> get() { return t; }
}
```

d.
```
public class Bucket<T> {

    private Object t;

    public void set(Object t) { this.t = t; }
    public Object get() { return t; }
}
```

e.
```
public class Bucket <T> {

    private <T> t;
```

```
    public void set(<T> t) { this.t = t; }
    public <T> get() { return t; }
}
```

2. Which of the following is an instantiation of the `Bucket` class from number 1 for a bucket of integers that will not lead to compiler warnings or errors?
   a. `Bucket<T> integerBucket = new Bucket<Integer>();`
   b. `Bucket<Integer> integerBucket = new Bucket();`
   c. `Bucket<Object> integerBucket = new Bucket<Integer>();`
   d. `Bucket<Integer> integerBucket = new Bucket<Integer>();`
   e. `Bucket<> integerBucket = new Bucket<Integer>();`

3. Which of the following is **not allowed** with Java generics?
   a.
   ```
   List<String> l1 = new List<String>();
   ArrayList<String> l2 = (ArrayList<String>)l1;
   ```
   b.
   ```
   public static void rtti(List<?> list) {
       if (list instanceof ArrayList<?>) {         // ...   }
   }
   ```
   c.
   ```
   List<Integer>[] arrayOfLists = new List<Integer>[2];
   arrayOfLists.add(1);
   ```
   d.
   ```
   public class Parser<T extends Exception> {
       public void parse(File file) throws T {     // ...   }
   }
   ```
   e.
   ```
   public static <E> void append(List<E> list, Class<E> cls) throws
   Exception {
       E elem = cls.newInstance();
       list.add(elem);
   }
   ```

4. Which of the following is a proper **explicit** invocation of the generic method `compare(Pair<K, V> p1, Pair<K, V> p2)`?
   a. `boolean same = Util.compare<Integer, String>(p1, p2);`
   b. `boolean same = Util.compare(p1, p2);`
   c. `boolean same = Util.<Integer, String>compare(p1, p2);`
   d. `boolean same = Util.<T, T>compare(p1, p2);`
   e. `boolean same = Util.compare((Pair<Integer,String>)p1, (Pair<Integer,String>)p2);`
5. When using type inference, which of the following will compile without warning or errors?
   a. `Map<String, List<String>> myMap = new HashMap();`

b. `Map<String, List<String>> myMap = new HashMap<>();`

c. `Map myMap = new HashMap<String, List<String>>();`

d. `Map<T, V> myMap = new HashMap<String, List<String>>();`

e. `Map<> myMap = new HashMap<String, List<String>>();`

6. What change(s) needs to be made to properly bind the generic type parameter `U` to `String` in the following code:

```
public <U> void inspect(U u){ ... }
```

a. `public <U> void inspect((String)U u){ ... }`
b. `public <U implements String> void inspect(U u){ ... }`
c. `public <U> void inspect(String u){ ... }`
d. `public <String> void inspect(U u){ ... }`
e. `public <U extends String> void inspect(U u){ ... }`

7. Consider the following code:

```
public class Node<T> {

    private T data;
    private Node<T> next;

    public Node setData(T data, Node<T> next) }
        this.data = data;
        this.next = next;
    }

    public T getData() { return data; }

    public static <T extends Bucket> void draw(T bucket) { ... }
}
```

Which of the following is how the code will look, to the Java compiler, after it is compiled?

a.
```
public class Node {

        private Object data;
        private Node next;

        public Node setData(Object data, Node next) { ... }

        public Object getData() { return data; }

        public static void draw(Bucket bucket) { ... }
    }
```

b.
```
public class Node {

    private Object data;
    private Node next;

    public Node setData(Object data, Node next) { ... }

    public Object getData() { return data; }

    public static void draw(Object bucket) { ... }
}
```
c.
```
public class Node<> {

    private Object data;
    private Node<> next;

    public Node setData(Object data, Node<> next) { ... }

    public Object getData() { return data; }

    public static <Object extends Bucket> void draw(Object
bucket) { ... }
}
```
d.
```
public class Node {

    private ? data;
    private Node<?> next;

    public Node setData(? data, Node<?> next) { ... }

    public ? getData() { return data; }

    public static <? extends Bucket> void draw(? bucket) { ... }
}
```
e. It looks the same. The code doesn't change until runtime.

8. "Is a" relationships in Java mean that because `Integer` *is a* `Object` you can assign an `Integer` to an `Object` (`someObject = someInteger;`). The same can be said for the relationship between `Integer` and `Number`.

   Now, consider the following code:

   ```
   public void shapeTest(Shape<Number> n) { /* ... */ }
   ```

Based on the signature of this method, could you pass in `Shape<Integer>` and/or `Shape<Object>`?

a. Yes, you could pass in both.
b. No, you can't pass in either.
c. You can pass in Shape<Integer> but not Shape<Object>.
d. You can pass in Shape<Object> but not Shape<Integer>.
e. You can pass in either if you cast it to Shape<Number>.

9. Consider the following code:

```
List list = new ArrayList<>();
Collections.sort(list);
```

Upon compilation, you get the following warning:
Type safety: Unchecked invocation max(List) of the generic method max(Collection<? extends T>) of type Collections.

What change needs to be made to resolve this warning and prevent others?

a.
```
List list = new ArrayList<String>();
Collections.sort(list);
```
b.
```
List list = new ArrayList();
Collections.sort(list);
```
c.
```
List<> list = new ArrayList<>();
Collections.sort(list);
```
d.
```
List<String> list = new ArrayList();
Collections.sort(list);
```
e.
```
List<String> list = new ArrayList<String>();
Collections.sort(list);
```

10. Consider the following code:

```
StringBuilder myText = new StringBuilder();
List<String> myList = new ArrayList<String>();
boolean containsMyText = myList.contains(myText);
```

After running static analysis, you get the following notification:

Bug: StringBuilder is incompatible with expected argument type String in new util.Configuration(String, String)

This call to a generic collection method contains an argument with an incompatible class from that of the collection's parameter (i.e., the type of the argument is neither a supertype nor a subtype of the corresponding generic type argument). Therefore, it is unlikely that the collection contains any objects that are equal to the method argument used here. Most likely, the wrong value is being passed to the method.

In general, instances of two unrelated classes are not equal. For example, if the Foo and Bar classes are not related by subtyping, then an instance of Foo should not be equal to an instance of Bar. Among other issues, doing so will likely result in an equals method that is not symmetrical. For example, if you define the Foo class so that a Foo can be equal to a String, your equals method isn't symmetrical since a String can only be equal to a String.

In rare cases, people do define nonsymmetrical equals methods and still manage to make their code work. Although none of the APIs document or guarantee it, it is typically the case that if you check if a Collection<String> contains a Foo, the equals method of argument (e.g., the equals method of the Foo class) used to perform the equality checks.

Which of the following will **not** resolve the notification?

a.
```
String myText = “my text”;
...
List<String> myList = new ArrayList<String>();
boolean containsMyText = myList.contains(myText);
```
b.
```
StringBuilder myText = new StringBuilder();
...
List<String> myList = new ArrayList<String>();
boolean containsMyText = myList.contains(myText.substring(0));
```
c.
```
StringBuilder myText = new StringBuilder();
...
List<String> myList = new ArrayList<String>();
boolean containsMyText = myList.contains(myText.toString());
```
d.
```
StringBuilder myText = new StringBuilder();
...
List<StringBuilder> myList = new ArrayList<StringBuilder>();
boolean containsMyText = myList.contains(myText);
```
e.
```
StringBuilder myText = new StringBuilder();
...
List<String> myList = new ArrayList<String>();
boolean containsMyText = myList.contains((String)myText);
```

C.2 Example Feature Hierarchy (Generics)

To determine what generics may be more advanced than others, we explored is how generics usage might relate to levels of experience. Under the assumption that higher frequency of usage corresponds to familiarity, we explored frequency of generics usage types to hypothesize the progression developers may take through generics usage types. If usage counts are generally low, we assume generally developers are less familiar with those types of generics. We also assume that the concepts developers are less likely to be familiar with are more advanced concepts.

To create our hierarchy, we first ordered the different generic usage types in order of most to least frequently used for each developer with generics in their repository. We then compared the position of each type of generics usage *across developers* in relation to other types of generics usage. If two generics usage types appeared next to one another in more than half of the repository orderings, we noted a potential ancestor relationship between the two (if A is an ancestor of B, in the context of our work , A and B are learned in succession).

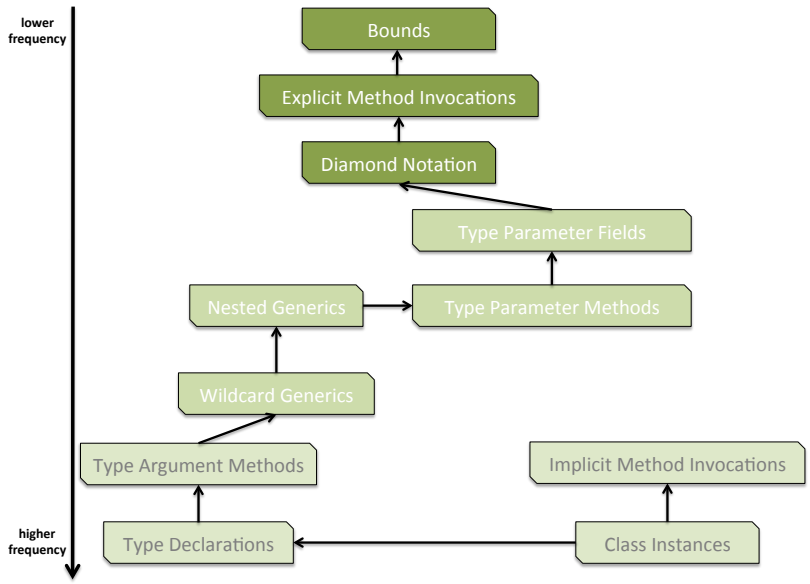


Figure 1. Proposed generics usage type progression

Our proposed progression through generics usage types is shown in the Figure 1. The colors indicate the level of generics usage we associate with that type of generics usage (beginner, intermediate, expert). The arrow on the left hand side

indicates the direction of code contribution counts. The name for each type of generics usage was determined based on what they are called in ASTParser, the library we used to analyze code, and map to code as follows:

- **Type Declarations →** `public class <T> Box{}`
- **Class Instances →** `List<String> a = new ArrayList<String>();`
- **Implicit Method Invocations →** `HashMap<String, String> a = b.getMap(); getMap()` returns a HashMap
- **Type Argument Methods →** `public List<String> method()`
- **Wildcard Generics →** Any generics that uses the wildcard (`?`) generic type
- **Nested Generics →** generics in the form of `Observable<Event<T>>` or `Observable<Event<String>>`
- **Type Parameter Methods →** `public T get();`
- **Type Parameter Fields →** `public T t;`
- **Diamond Notation →** `List<String> a = new ArrayList<>();`
- **Explicit Method Invocations →** `HashMap<String, String> a = b.<HashMap<String, String>>getMap();`
- **Bounds →** `public <T extends Comparable<T>> T max(){}`

To determine the cutoffs for the levels in our hierarchy, we looked at the increase in repositories without that usage type. For example, only one repository contained generic bounds, the usage type at the top of our hierarchy.

To validate our groupings, we compared our hierarchy to the questions missed by developers that scored highly on the concept inventory. The concepts at the top half of our hierarchy map to the questions on the inventory missed by top scorers. For example, all top scorers missed the question about explicit method invocations, one of usage types at the top of our hierarchy.

In future work, we will further explore the validity of our hierarchy by creating another hierarchy with a larger set of developers and comparing the two.

C.3 Example Concept Map and Bloom's Taxonomy Assessment Mapping (Generics)

Concepts	Ancestor Concepts					Resources	Remember	Understand	Apply	Evaluate	Analyze	Create
Generic Types	10					https://docs.oracle.com/javase/tutorial/java/generics/types.html		1		2		
Generic Methods						https://docs.oracle.com/javase/tutorial/java/generics/methods.html			9	4		
Bounded Type Parameters	11					https://docs.oracle.com/javase/tutorial/java/generics/bounded.html						6
Generics, Inheritance, and Subtypes						https://docs.oracle.com/javase/tutorial/java/generics/inheritance.html		8				
Type Inference						https://docs.oracle.com/javase/tutorial/java/generics/genTypeInference.html					5	
Wildcards	16	17	18	19	20	https://docs.oracle.com/javase/tutorial/java/generics/wildcards.html			10			
Type Erasure	9	10	11	12		https://docs.oracle.com/javase/tutorial/java/generics/erasure.html			7			
Erasure of Generic Types						https://docs.oracle.com/javase/tutorial/java/generics/genTypes.html						
Erasure of Generic Methods						https://docs.oracle.com/javase/tutorial/java/generics/genMethods.html						
Effects of Type Erasure and Bridge Methods						https://docs.oracle.com/javase/tutorial/java/generics/bridgeMethods.html						
Non-Reifiable Types						https://docs.oracle.com/javase/tutorial/java/generics/nonReifiableVarargsType.html						
Restrictions on Generics						https://docs.oracle.com/javase/tutorial/java/generics/restrictions.html	3					
Raw Types						https://docs.oracle.com/javase/tutorial/java/generics/rawTypes.html						
Generic Methods and Bounded Type Parameters						https://docs.oracle.com/javase/tutorial/java/generics/boundedTypeParams.html						
Upper Bounded Wildcards						https://docs.oracle.com/javase/tutorial/java/generics/upperBounded.html						
Unbounded Wildcards						https://docs.oracle.com/javase/tutorial/java/generics/unboundedWildcards.html						
Lower Bounded Wildcards						https://docs.oracle.com/javase/tutorial/java/generics/lowerBounded.html						
Wildcards and Subtyping						https://docs.oracle.com/javase/tutorial/java/generics/subtyping.html						
Wildcard Capture and Helper Methods						https://docs.oracle.com/javase/tutorial/java/generics/capture.html						

APPENDIX

D

CHAPTER 7 ARTIFACTS