

# A Cross-Tool Communication Study on Program Analysis Tool Notifications

Brittany Johnson, Rahul Pandita, Justin Smith, Denae Ford, Sarah Elder,  
Emerson Murphy-Hill, Sarah Heckman, Caitlin Sadowski†  
NC State University; Google†  
Raleigh, North Carolina, USA; Mountain View, CA, USA†  
{bijohnso, rpandit, jssmit11, dford3, seelder}@ncsu.edu, {emerson,  
heckman}@csc.ncsu.edu, supertri@google.com†

## ABSTRACT

Program analysis tools use notifications to communicate with developers, but previous research suggests that developers encounter challenges that impede this communication. This paper describes a qualitative study that identifies 10 kinds of challenges that cause notifications to miscommunicate with developers. Our resulting notification communication theory reveals that many challenges span multiple tools and multiple levels of developer experience. Our results suggest that, for example, future tools that model developer experience could improve communication and help developers build more accurate mental models.

## CCS Concepts

•Human-centered computing → User studies;

## Keywords

program analysis tools, human factors, communication

## 1. INTRODUCTION

Program analysis tools, such as static analysis tools, refactoring tools, and code smell detectors, can ease manual and sometimes tedious software development tasks by automatically analyzing and modifying source code [1, 21]. Output from these tools, such as warnings and errors, come in the form of textual or visual notifications that vary from tool to tool. In our previous interviews, 20 professional developers reported not using static analysis tools, one type of program analysis tool, because notifications can be difficult to interpret [15].

The goal of our research is to understand what makes it challenging for developers to interpret program analysis tool notifications. To motivate this goal, consider Ann, a hypothetical professional developer. While using the FindBugs [4] static analysis tool, she encounters the notification

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.

Copyright is held by the owner/author(s). Publication rights licensed to ACM.

FSE'16, November 13–18, 2016, Seattle, WA, USA  
ACM, 978-1-4503-4218-6/16/11...\$15.00  
<http://dx.doi.org/10.1145/2950290.2950304>

```
95 if (managingFocusForwardTraversalKeys == null) {  
96     managingFocusForwardTraversalKeys = new HashSet<KeyStroke>();  
97     managingFocusForwardTraversalKeys.add(  
98         KeyStroke.getKeyStroke(KeyEvent.VK_TAB, 0));  
99 }
```

Incorrect lazy initialization and update of static field javax...  
managingFocusForwardTraversalKeys in javax...installDefaults()

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 further accesses in the method that set the field serve to initialize the object, then you have a *very serious multi-threading 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.

Figure 1: A notification from FindBugs concerning incorrect lazy initialization (FB2).

in Figure 1. At first glance, the concept of multi-threading is familiar from her experience with the Java compiler. However, she is not familiar with lazy initialization or FindBugs, and realizes that she has to learn the terminology used by FindBugs before she can understand the problem. Because she has limited time and no prior experience with lazy initialization, she enlists the help of outside resources and hopes for a speedy resolution.

While Ann is hypothetical, her challenges are not; this example is based on the challenges encountered by several participants in the study described in this paper. Moreover, the challenges are not unique to FindBugs; we found these challenges occurred when participants used different program analysis tools. In this paper, we describe a think-aloud study where we asked 26 developers with varying backgrounds to interpret notifications from three tools: FindBugs, the Eclipse Java Compiler, and EclEmma. We chose to study multiple program analysis tools to understand cross-tool challenges, not just challenges with individual tools. To identify challenges, we examined tool use through the lens of communication theory [6, 8].

Based on existing research on how computers should talk to people [8], the main contribution of this paper is that it proposes an explanatory theory that describes

why developers encounter difficulties when interpreting tool notifications. Our theory encompasses 10 categories of challenges that emerged from our study, the first cross-tool notification study of which we are aware. We apply our theory by presenting ways that program analysis tools can improve communication with developers, such as by collecting, modeling, and leveraging developer experience.

## 2. RELATED WORK

Existing research has focused on easing the process of understanding and resolving notifications [13, 24, 28, 9] 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. Our work is related in that our findings can be used to improve the design of tools to better support developers. Our work differs in that we investigate different types of tools to identify common 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 [13, 34, 5]. Hartmann and colleagues developed a social recommender system, HELPMEOU, to better assist novices with understanding and resolving compiler notifications [13]. 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 [34]. Based on his findings, Traver developed compiler notification design principles, which includes using consistent messages and including more visual aids.

Muşlu and colleagues developed QUICK FIX SCOUT, an extension to Eclipse Quick Fix, to ease the process of determining an optimal fix [24]. They found programmers could more quickly assess and apply quick fixes when able to easily reason about fix trade-offs. Barik and colleagues studied how developers reason about compiler notifications to improve tool support for understanding and resolving tool notifications [5]. 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 [20]. Wagner and colleagues compared two analysis tools that detect defects to evaluate their efficiency [35]. Other tool evaluations have had the same goal [32, 38].

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 [5, 36, 30]. One is our 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 [5]. In other prior work, we used Diffusion of Innovation theory to explore factors that influence security tool adoption [36]. Similarly,

Rienmenschneider and Hardgrave explored why tools do not get used using the Technology Acceptance Model, based largely on the Theory of Reasoned Action [30]. Lawrance and colleagues used information foraging theory to propose a theory of information foraging for how programmers navigate code when debugging [17]. In contrast, we apply communication theory to understand the challenges developers encounter when interpreting tool notifications.

## 3. METHODOLOGY

We next describe our study design. Our research materials are available on-line to aid other researchers in replication and exploration.<sup>1</sup>

### 3.1 Research Question

In a previous study, we asked developers to recall experiences with static analysis tools and briefly use FindBugs. We found that some developers do not use static analysis tools due to difficulty interpreting the notifications tools use to communicate [15]. To find out how tools could better communicate with developers, our study is designed to answer the question: *Why do developers encounter challenges when interpreting program analysis tool notifications?* Using Hannay and colleagues' guidelines [12], we frame our question as *why* rather than *what* to support our building of a theory that explains the challenges developers encounter.

### 3.2 Participants

We recruited twenty-six participants using mailing lists, classroom recruitment, and personal contacts. Participants include undergraduate students, graduate students, and professional developers, with varying amounts of development and tool usage experience. Figure 2 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 the paper, we 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.

### 3.3 Program Analysis Tools Investigated

Our study focuses on tools that can be used in the Eclipse Integrated Development Environment (IDE) [44]. We chose Eclipse because it is one of the most widely used IDEs [11], making it easier to recruit qualified participants, and because it is compatible with a variety of tools. We 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

<sup>1</sup><http://www4.ncsu.edu/~bijohnso/esnpat.html>

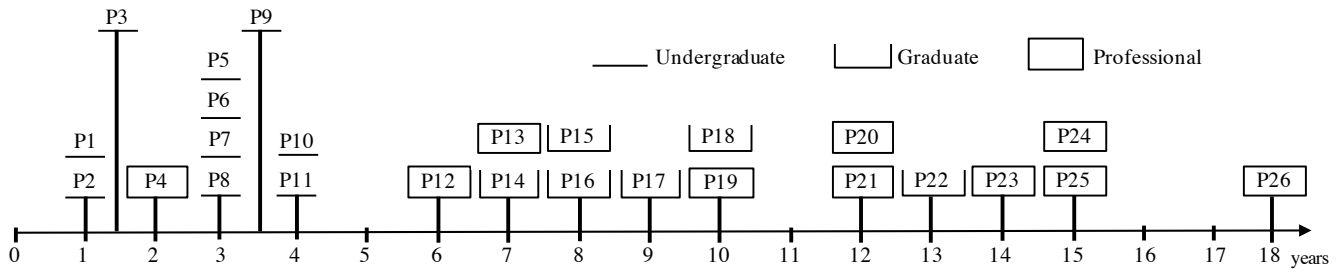


Figure 2: Distribution of participants' years of development experience.

```

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 3: A notification of a previous null check from FindBugs (FB4).

descriptions are available by hovering over or clicking the  icon as seen in Figure 3.

### 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 [40]. 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 at line 159. When the developer mouses over the underlined code or the  icon, the notification displays a text description (Figure 4(b)). Unlike 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 [43], to communicate with the developer about code paths that did and did not get exercised. Although EclEmma communicates about

```

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

```

(a) Source Code

The type `new AbstractInterruptibleChannelInterruptible()` must implement the inherited abstract method `new AbstractInterruptibleChannelInterruptible.interrupt()`

(b) Text Description

Figure 4: An Eclipse compiler notification about unimplemented methods (CMP5).

```

133     if (this.std != that.std) {
134         return false;
135     }


```

(a) Source Code with Highlighting

1 of 2 branches missed

(b) Text Description

Figure 5: An EclEmma notification about partial branch coverage (ECL3).

one particular execution, as with the other tools it provides information to the developer regarding code (during runtime rather than compile-time). EclEmma uses highlighting to indicate code execution; code highlighted in green was executed, red was not executed, and yellow was partially executed. Figure 5 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 5(b)).

These tools may seem quite different, but we 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.

Table 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

### 3.4 Study Protocol

Each session with a participant lasted approximately one hour. Prior to each session, we asked participants to fill out a consent form and pre-questionnaire.<sup>2</sup> Each session consisted of seventeen tasks.

Source code for the tasks came from OpenJDK [45] and JFreeChart [42]. We chose Open JDK because it has a large code base from which we could easily find bugs using their publicly available FindBugs cloud report [39]. We chose JFreeChart because it is a large code base with working JUnit test cases that exhibit less-than-perfect code coverage.

For each task, we presented participants with and asked them to interpret one or more notifications from a given tool. We 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. Allowing use of the browser would have added data that does not help answer our current research question. We also wanted to see if developers could interpret tool notifications without the aid of web resources. During many tasks, and at least once for every participant, participants discussed or completed notification resolution. We did not require them to do so, since it would be unfair to ask them to resolve a notification if they did not understand it. As participants explained the notifications, the first author asked follow-up questions as necessary.

Table 1 shows a list of the notification tasks participants encountered during each session. For each task, we chose notifications to represent the types of notifications developers may encounter when programming. For FindBugs and the Eclipse compiler, we chose notifications that appeared frequently in the OpenJDK project. We chose EclEmma notifications from JFreeChart to exercise a range of its coverage scenarios. Because EclEmma’s documentation does not specify the range of notifications it uses, the first author manually went through JFreeChart’s codebase after running the tool and took note of each new coverage scenario encountered. We then included an example of every coverage scenario in the EclEmma tasks.

For FindBugs, each task during the session corresponded

<sup>2</sup>This work was approved under IRB No. 2787.

```

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 6: A notification from the compiler about generics (CMP2).

to a single notification. All but one compiler task corresponded to a single notification; because the two notifications on CMP2 (Figure 6) contribute to the same problem on the same line, we presented them as one task. Each EclEmma task consisted of participants explaining coverage notifications for the entire class.

### 3.5 Data Collection

We recorded audio and the screen in each session for analysis. We 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, we added a description of that navigation to the transcript.

### 3.6 Data Analysis

We analyzed each session using open and selective coding [7] to discover participant challenges. To identify a challenge, we needed concrete criteria. We propose 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 [8]. We used this logic to determine when a challenge occurred, using three criteria for inclusion: 1) the participant explicitly states a challenge, 2) is unable to explain the notification, or 3) has 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.

The first and second authors 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, we used a card sorting methodology similar to that of Muşlu and colleagues [23]. The goal of our card sort was to identify themes based on our codes. We used five of the eight authors on this paper 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. 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 our criteria is participant inability to explain a notification, any actions or statements made surrounding that occurrence was included in our 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 we do not discuss them in this paper. We likewise do not discuss the emergent theme of tool feature requests. These excluded themes are available with our other on-line research materials.

### 3.7 Study Credibility & Findings Validation

There are inherent threats to the validity of empirical research [27]. Despite these inherent threats, prior research suggests there are ways we can increase confidence in the credibility and validity of our findings [10, 18]. Following the safeguards for conducting empirical research proposed by Li [18], we ensured the following in the collection, interpretation, and reporting of our data:

- *Voluntary participation and anonymity.* To receive truthful responses from participants, we 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, we recruited participants from academia and industry with varying levels of programming experience.
- *Triangulation.* To increase reliability, we 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 of our sessions 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, we set up our environment and recruited participants familiar with that environment and programming language. We 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. We also included multiple researchers throughout the multi-step analysis and reporting process.
- *Member checks.* To ensure validity of the data and our interpretation, we reached out to all participants, providing them with a summary of our findings, a copy of the written report, and a form for providing feedback on our findings.
- *Thick description.* To enable judgment of how our research fits with other contexts, we describe in detail the methods used to collect our data and the setting in which it was collected.

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

## 4. RESULTS

### 4.1 The Theory

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” [7]. While explicitly putting forward theories is rare in software engineering [12], one example is Lawrance and colleagues’ theory of how programmers navigate code during debugging [17]. In the same way that Lawrance and colleagues’ build on information foraging theory [29], our theory builds on communication theory [6]. We summarize our notification communication theory as:

The challenges developers encounter when interpreting tool notifications are caused by gaps and mismatches between developers’ programming knowledge, based on their individual experiences, and methods used by notifications to communicate information about developers’ source code.

We define software development knowledge as any knowledge relevant to understanding, writing, or maintaining software, such as defect resolution. The challenges that comprise our theory are shown in Figure 7. Vertical lines represent the tasks and the horizontal bars indicate



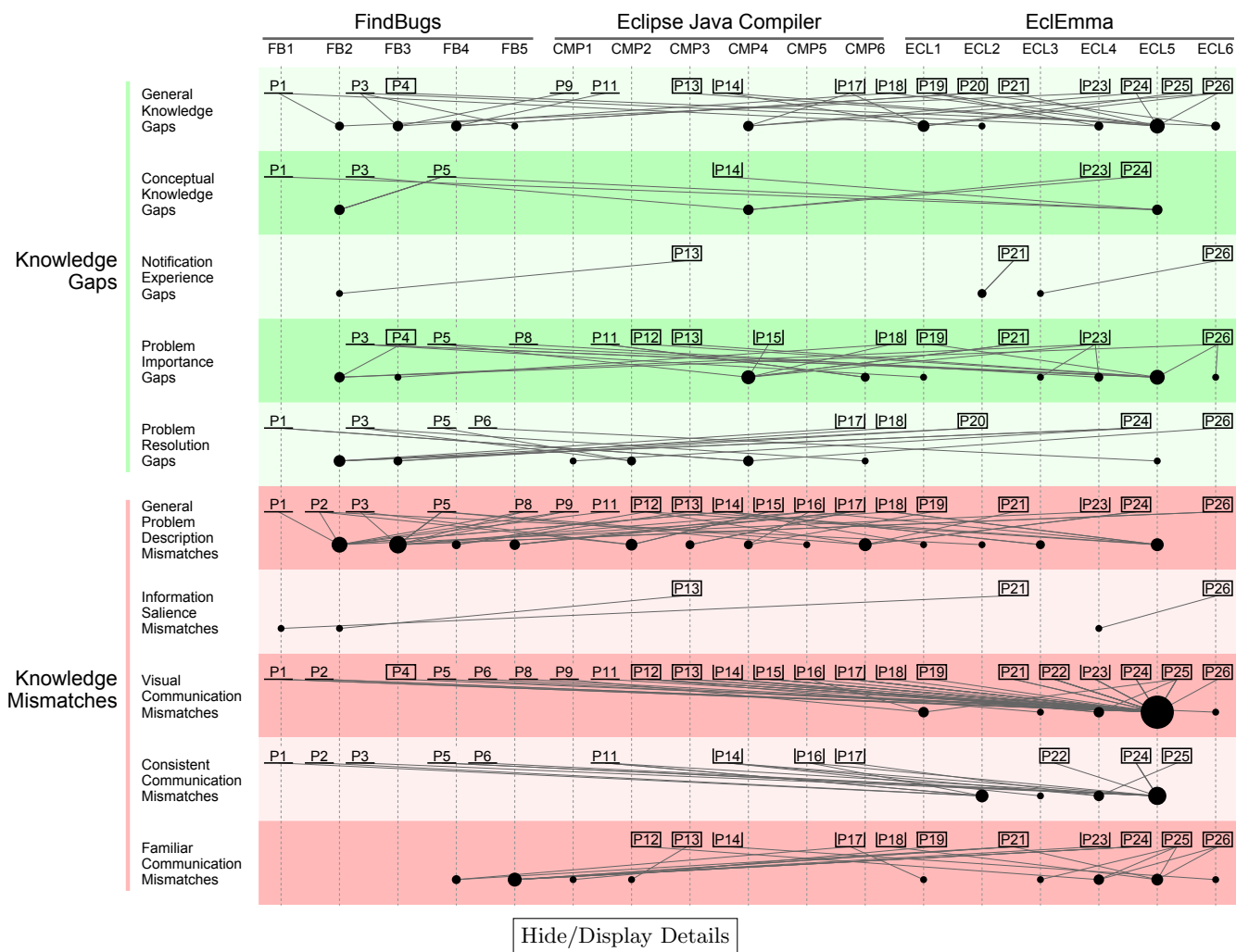


Figure 7: Distribution of challenges encountered and notifications that caused them.

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. We describe each challenge type, and our validation of our findings, in detail in the remainder of this section.

## 4.2 Knowledge Gaps

Knowledge gaps occurred when there was a gap between what participants know and the information provided by the notification. We speak about knowledge here and throughout as the culmination of experiences [16, 3]. 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 can occur regardless of programming or industry experience. In this subsection, we describe general knowledge gap challenges, followed by four specific kinds of knowledge gaps we identified from our study.

### 4.2.1 General Knowledge Gaps

General knowledge gap challenges occurred when there was a gap between the general software development knowledge participants have relevant to the notification and the information provided by the notification. When participants did not provide enough information for us to map a challenge to a more specific kind of knowledge gap, we placed that challenge in this theme. Participants experienced knowledge gap challenges across all three tools (Figure 7).

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 understand 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.

Our findings confirm that more is not always better [25] for closing knowledge gaps and that these gaps exist with visual communication as well. Along with general knowledge gap challenges, we discuss four specific types of knowledge gaps that emerged and led to challenges for participants.

#### 4.2.2 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. Therefore, 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 definitely 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<sup>3</sup>. This, coupled with being his first

<sup>3</sup>The reader may also find this confusing, but this was

experience with EclEmma notifications, led to his inability to interpret the notifications in ECL5.

#### 4.2.3 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. Their lack of experience with the 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 is, 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. We discuss those next.

#### 4.2.4 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 does tell him that he is synchronizing on a mutable field, it does 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 provides quick fixes, deciding which fix is best for P5 was dependent on what the ID is used for, which the notification does not specify.

#### 4.2.5 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 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 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.

of information foraging that involves leaving their working context [2], 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 do 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` is or how it should be used.

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 he would resolve the notification.

### 4.3 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, we describe General Knowledge Mismatch challenges, followed by discussion of four specific kinds of knowledge mismatches we identified from our study.

#### 4.3.1 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 [34]. Similarly, O’Neil discussed the importance of language considerations in data breach notifications [26].

Representative of textual communication mismatch challenges is [P16]’s experience interpreting FB5. After reading the text provided by the notification, [P16] could not 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 had to move away from the text of the notification to come to any sort of conclusion about the problem being communicated.

For some participants, the language 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 means. 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 is ambiguous. He found the use of the word “applicable” to be odd in this context and not clearly indicative of the message he assumes the tool is trying to communicate.

Although these textual mismatch challenges encountered are general, specific types of mismatches with the text portion of the notification emerged; we discuss those below.

#### 4.3.2 Information Salience Mismatches

Information salience mismatches occurred when there was a mismatch between the information a participant thinks is most relevant and the information the notification makes salient. This aligns with McCrickard and Chewar’s suggestion that general computer users are dissatisfied with notification systems because of mismatched information prioritization [19].

Representative of these challenges is [P13]’s attempt to interpret FB2 (Figure 1). [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 elaborates on what is wrong with where and how the field of interest is set when working with threads. This was what he was looking for, as it helped him understand why it is a “*very serious* multi-threading bug,” as the notification states. 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 is in the code and why it is a problem. The critical pieces of information, such as that it is a multi-threading problem concerning where synchronization is placed, got buried.

We only observed this phenomena with more experienced developers; this suggests that more experienced developers may have more concrete expectations of what information the tool 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.

#### 4.3.3 Visual Communication Mismatches

Visual communication mismatches occurred when there was a mismatch between how the participant would com-



```

302     finally {
303         Locale.setDefault(saved);
304     }

```

**Figure 8: A notification from EclEmma regarding finally coverage (ECL5).**

municate 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 participants 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 8) 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 can 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 does not understand the way the tool communicates.

Seven participants had expectations regarding how `try` blocks work that did not match how EclEmma reports `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 is 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, were clicked; in fact, markers and notifications from other tools within EclEmma’s interface was sometimes a distraction for 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 providing 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.

These expectation mismatch challenges are general and focus on visual communication. Participants also encountered related, but more specific knowledge expectation mismatch challenges. We discuss those next.

#### 4.3.4 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 [34]. Our results 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 8. 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.

#### 4.3.5 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.

## 4.4 Member Check

To assess the validity of our interpretation of the data, and the experiences developers have when interpreting tool notifications, we conducted a member check. Of the seven responses, two developers agreed with our findings and five strongly agreed. Many found our 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 Saliency Mismatches, followed by the third most common responses (3) of Conceptual Knowledge Gaps, Visual Communication Mismatches, and Familiar Communication Mismatches.

## 5. IMPLICATIONS

Current tools do not support developer knowledge gaps (Section 4.2) and conflict with developer knowledge (Section 4.3). We discuss several implications in this section.

**Filling Developer Knowledge Gaps.** Despite the experience of some participants, every participant encountered at least one notification they could 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.

Our findings suggest tools can fill developer knowledge gaps by consistently providing information about the options for fixing a notification (Section 4.2.5) and reasoning for resolution (Section 4.2.4). 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, which we discussed in Section 2 [24]. 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.

**Matching Developer Expectations.** Developer expectations can have an effect on their ability to interpret notification messages (Section 4.3). We propose that tools can improve how they communicate to developers if they are able to ascertain developers' knowledge and experiences, which inform their expectations [8]. For each notification in our 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.

What if we could determine the best links to external resources for a developer based on the concepts relevant to the notification the developer knows the least about? Or display information based on what is most needed or used by the developer? One solution, modeled after intelligent tutoring systems (ITS) [22], would be 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 [14].

Imagine two developers, D1 and D2; D1 frequently has developed 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 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 [19, 33, 37] 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 [31]. 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.

## 6. CONCLUSION AND FUTURE WORK

We propose a notification communication theory that program analysis tools and developers miscommunicate because of knowledge gaps and knowledge mismatches. This theory serves as a foundation for two major pieces of future work. First, the theory is only proposed here; future work is required to strengthen and refine the theory, for example through field studies, case studies, and controlled experiments. And second, the theory serves as a foundation to create more effective tools and notifications. In exploring and building these tools, we imagine a future where tools communicate fluidly and seamlessly with developers.

## Acknowledgments

Many thanks to the participants in our study for their time. Special thanks to Robert Bowdidge, Michael Ernst, Kevin Lubick, Allaire Welk, Olga Zielinska, and the Developer Liberation Front [41]. This material is based upon work supported by the National Science Foundation under Grant No. 1217700 and 1318323, a Google Faculty Award, and a National Science Foundation Graduate Research Fellowship under Grant No. DGE-0946818.

## 7. REFERENCES

- [1] S. Adolph, W. Hall, and P. Kruchten. Using grounded theory to study the experience of software

- development. *Empirical Software Engineering*, 16(4):487–513, 2011.
- [2] E. M. Altmann and J. G. Trafton. Task interruption: Resumption lag and the role of cues. Technical report, Defense Technical Information Center, 2004.
- [3] L. Argote and E. Miron-Spektor. Organizational learning: From experience to knowledge. *Organization science*, 22(5):1123–1137, 2011.
- [4] N. Ayewah, D. Hovemeyer, J. D. Morgenthaler, J. Penix, and W. Pugh. Using static analysis to find bugs. *IEEE Software*, 25(5):22–29, 2008.
- [5] T. Barik, K. Lubick, S. Christie, and E. Murphy-Hill. How developers visualize compiler messages: A foundational approach to notification construction. In *2nd IEEE Working Conference on Software Visualization*, 2014.
- [6] J. P. Bowman and A. S. Targowski. Modeling the communication process: The map is not the territory. *Journal of Business Communication*, 24(4):21–34, 1987.
- [7] J. Corbin and A. Strauss. *Basics of qualitative research: Techniques and procedures for developing grounded theory*. Sage publications, 2014.
- [8] M. Dean. How a computer should talk to people. *IBM Systems Journal*, 21(4):424–453, 1982.
- [9] T. Fritz, D. C. Shepherd, K. Kevic, W. Snipes, and C. Bräunlich. Developers’ code context models for change tasks. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 7–18. ACM, 2014.
- [10] S. Gasson. Rigor in grounded theory research: An interpretive perspective on generating theory from qualitative field studies. *The handbook of information systems research*, pages 79–102, 2004.
- [11] G. Goth. Beware of the March of this IDE: Eclipse is overshadowing other tool technologies. *IEEE Software*, 22(4):108–111, 2005.
- [12] J. E. Hannay, D. I. Sjöberg, and T. Dybå. A systematic review of theory use in software engineering experiments. *Software Engineering, IEEE Transactions on*, 33(2):87–107, 2007.
- [13] B. Hartmann, D. MacDougall, J. Brandt, and S. R. Klemmer. What would other programmers do: suggesting solutions to error messages. In *Proceedings of the SIGCHI Conference on Human Factors in Computing*, pages 1019–1028, 2010.
- [14] B. Johnson, R. Pandita, E. Murphy-Hill, and S. Heckman. Bespoke tools: adapted to the concepts developers know. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 878–881. ACM, 2015.
- [15] B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge. Why don’t software developers use static analysis tools to find bugs? In *Software Engineering (ICSE), 2013 35th International Conference on*, pages 672–681. IEEE, 2013.
- [16] P. N. Johnson-Laird. Mental models. 1989.
- [17] J. Lawrance, C. Bogart, M. Burnett, R. Bellamy, K. Rector, and S. D. Fleming. How programmers debug, revisited: An information foraging theory perspective. *Software Engineering, IEEE Transactions on*, 39(2):197–215, 2013.
- [18] D. Li. Trustworthiness of think-aloud protocols in the study of translation processes. *International Journal of Applied Linguistics*, 14(3):301–313, 2004.
- [19] D. S. McCrickard and C. M. Chewar. Attuning notification design to user goals and attention costs. *Communications of the ACM*, 46(3):67–72, 2003.
- [20] W. Mettrey. A comparative evaluation of expert system tools. *Computer*, 24(2):19–31, 1991.
- [21] E. Murphy-Hill and A. Black. An interactive ambient visualization for code smells. In *Proceedings of International Symposium on Software Visualization*, pages 5–14, 2010.
- [22] T. Murray. Authoring intelligent tutoring systems: An analysis of the state of the art. *International Journal of Artificial Intelligence in Education (IJAIED)*, 10:98–129, 1999.
- [23] K. Muşlu, C. Bird, N. Nagappan, and J. Czerwonka. Transition from Centralized to Distributed Version Control Systems: A Case Study on Reasons, Barriers, and Outcomes. In *Proceedings of the International Conference on Software Engineering*, 2014.
- [24] K. Muşlu, Y. Brun, R. Holmes, M. D. Ernst, and D. Notkin. Speculative analysis of integrated development environment recommendations. *ACM SIGPLAN Notices*, 47(10):669–682, 2012.
- [25] M. Nienaltowski, M. Pedroni, and B. Meyer. Compiler Error Messages: What Can Help Novices? In *Proceedings of SIGCSE Technical Symposium on Computer Science Education*, pages 168–172, 2008.
- [26] F. O’Neil. Target data breach: applying user-centered design principles to data breach notifications. In *Proceedings of the 33rd Annual International Conference on the Design of Communication*, page 47. ACM, 2015.
- [27] A. J. Onwuegbuzie and N. L. Leech. Validity and qualitative research: An oxymoron? *Quality & Quantity*, 41(2):233–249, 2007.
- [28] R. Pham, Y. Stoliar, and K. Schneider. Automatically recommending test code examples to inexperienced developers. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 890–893. ACM, 2015.
- [29] P. Pirolli and S. Card. Information foraging. *Psychological review*, 106(4):643, 1999.
- [30] C. K. Riemenschneider and B. C. Hardgrave. Explaining software development tool use with the technology acceptance model. *The Journal of Computer Information Systems*, 41(4):1, 2001.
- [31] M. P. Robillard, W. Maalej, R. J. Walker, and T. Zimmermann. *Recommendation systems in software engineering*. Springer, 2014.
- [32] C. K. Roy, J. R. Cordy, and R. Koschke. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Science of Computer Programming*, 74(7):470–495, 2009.
- [33] D. Sow, M. Ebling, R.-P. Lehmann, J. Davis, and L. Bergman. Scout contextually organizes user tasks. In *e-Business Engineering, 2005. ICEBE 2005. IEEE International Conference on*, pages 94–101, oct. 2005.
- [34] V. J. Traver. On Compiler Error Messages: What

- They Say and What They Mean. *Advances in Human-Computer Interaction*, 2010.
- [35] S. Wagner, F. Deissenboeck, M. Aichner, J. Wimmer, and M. Schwalb. An evaluation of two bug pattern tools for java. In *Software Testing, Verification, and Validation, 2008 1st International Conference on*, pages 248–257. IEEE, 2008.
- [36] S. Xiao, J. Witschey, and E. Murphy-Hill. Social influences on secure development tool adoption: why security tools spread. In *Proceedings of the 17th ACM conference on Computer supported cooperative work & social computing*, pages 1095–1106. ACM, 2014.
- [37] L. Zhang, N. Tu, and D. Vronay. Info-lotus: a peripheral visualization for email notification. In *CHI '05 extended abstracts on Human factors in computing systems*, CHI EA '05, pages 1901–1904, New York, NY, USA, 2005. ACM.
- [38] J. Zheng, L. Williams, N. Nagappan, W. Snipes, J. P. Hudepohl, and M. A. Vouk. On the value of static analysis for fault detection in software. *Software Engineering, IEEE Transactions on*, 32(4):240–253, 2006.
- [39] Findbugs cloud tutorial, 2011. <https://code.google.com/p/findbugs/wiki/FindBugsCloudTutorial>.
- [40] JDT core component, 2013. <http://www.eclipse.org/jdt/core/index.php>.
- [41] Developer Liberation Front. <http://research.csc.ncsu.edu/dlf/>.
- [42] JFreeChart, 2013. <http://www.jfree.org/jfreechart/>.
- [43] JUnit 4. <http://www.junit.org>.
- [44] Eclipse. <http://www.eclipse.org>.
- [45] OpenJDK source releases. <http://download.java.net/openjdk/jdk8/>.