Do Developers Read Compiler Error Messages?

Titus Barik[*], Justin Smith[*], Kevin Lubick[*], Elisabeth Holmes[‡], Jing Feng[†], Emerson Murphy-Hill[*], Chris Parnin[*]
[*]Department of Computer Science, North Carolina State University, Raleigh, North Carolina, USA
[†]Department of Psychology, North Carolina State University, Raleigh, North Carolina, USA
[‡]Department of Psychology, Washington and Lee University, Lexington, Virginia, USA

***Abstract*—In integrated development environments, developers receive compiler error messages through a variety of textual and visual mechanisms, such as popups and wavy red underlines. Although error messages are the primary means of communicating defects to developers, researchers have a limited understanding on how developers actually use these messages to resolve defects. To understand how developers use error messages, we conducted an eye tracking study with 56 participants from undergraduate and graduate software engineering courses at our university. The participants attempted to resolve common, yet problematic defects in a Java code base within the Eclipse development environment. We found that: 1) participants read error messages and the difficulty of reading these messages is comparable to the difficulty of reading source code, 2) difficulty reading error messages significantly predicts participants' task performance, and 3) participants allocate a substantial portion of their total task to reading error messages (13%–25%). The results of our study offer empirical justification for the need to improve compiler error messages for developers.**

***Keywords*-compiler errors, eye tracking, integrated development environments, programmer comprehension, reading, visual attention**

I. Introduction

Compilers are notorious for producing cryptic and uninformative messages [1], [2], [3], [4]. For example, a missing symbol, type mismatch, or incorrect dependency can create situations where error messages can produce misleading or hard to digest information [5]. Unfortunately, compiler errors happen frequently: Seo and colleagues empirically obtained build failures from over 26 million builds at Google [2], and found that over a quarter of builds fail due to compiler errors.

To improve how developers receive notifications about errors in their code, modern integrated development environments (IDEs), such as Eclipse, have incorporated several design elements to support better understandability and expressiveness of error notifications. Wavy red lines, for example, are a popular means for highlighting errors in code and revealing the potential causes associated with an error.

However, there has been limited research on understanding how developers perceive and comprehend error messages through the various ways in which they are presented, for novice and expert developers alike. For example, Denny and colleagues speculated that improvements to error messages were unsuccessful because their students didn't read them [6]. Marceau and colleagues proposed a Read-Understand-Formulate theory, but were unable to confirm whether participants actually read the messages; they suggested the use of eye tracking to provide this missing evidence [7]. In industry, Seo and colleagues provided a distribution of "costly" compiler errors introduced by their expert developers, but their methodology cannot explain *why* these errors are costly [2]. In short, we have limited insights into how developers process, or attend to error messages, during the comprehension and resolution of defects.

To understand if developers read error messages, we conducted an eye tracking study with 56 developers, recruited from undergraduate and graduate software engineering courses at our university, as they resolved common, yet problematic error message defects in an IDE. We collected pixel-level coordinates and times for developers' sustained eye gazes, called *fixations*. We then triangulated these fixations against screen recordings of their interactions in the IDE.

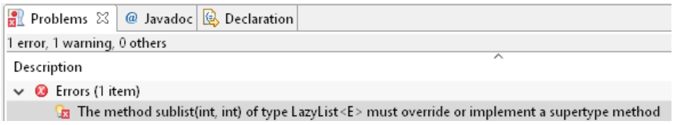
The results of our study provide empirical justification for the need to improve compiler error messages. Specifically, we find that:

- Participants read error messages; unfortunately, the difficulty of reading error messages is comparable to the difficulty of reading source code — a cognitively demanding task.
- Participant difficulty with reading error messages is a significant predictor of task correctness ($p < .0001$), and contributes to the overall difficulty of resolving a compiler error ($R^2 = 0.16$).
- Across different categories of errors, participants allocate 13%–25% of their fixations to error messages in the IDE, a substantial proportion when considering the brevity of most compiler error messages compared to source code.

II. Motivating Example

To illustrate how error messages can become costly for developers to resolve, consider a hypothetical developer, Barry. Barry recently joined a large software company, and needs to implement some missing functionality within a data structures library. Being relatively new to the library, he messages his colleague for some help in getting started. He eventually receives a reply from his colleague with a short code snippet.

Barry copies and pastes this snippet into his `source editor`, and is surprised that the IDE produces an error. He focuses his attention, that is, visually *fixates*, on the error text in the *problems pane* at the bottom of his screen:

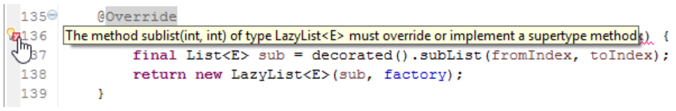


1 error, 1 warning, 0 others

Errors (1 item)

The method sublist(int, int) of type LazyList<E> must override or implement a supertype method

He silently reads the message about the sublist method, and then double-clicks the error in the problems pane. This redirects the IDE to the source editor, and Barry confirms that the error is related to the code that he just added. In the margin of the source editor, he now notices a light bulb icon, which he hovers over to produce an *error popup*:

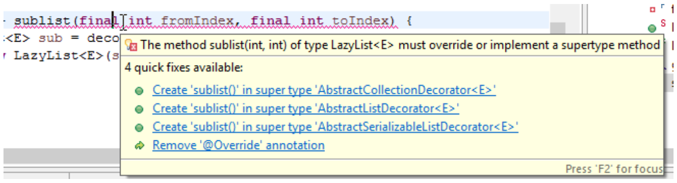


The method sublist(int, int) of type LazyList<E> must override or implement a supertype method

```
    @Override
                                                                                             k)  {
        final List<E> sub = decorated().subList(fromIndex, toIndex);
        return new LazyList<E>(sub, factory);
    }
```

Unfortunately, the popup is less helpful than he expected because it repeats the message he has already seen in the problems pane. Moreover, the error popup text is obscuring the method signature which is where he believes the problem is actually located.

Next, he notices the red wavy underline, which in Eclipse indicates the presence of an error. Barry hovers over the underline, releaving a *Quick Fix popup*. Unlike the error popup, the Quick Fix popup provides possible "fixes" that change the source code, in addition to the error message. He spends several seconds attending to both the error message and evaluating it against the proposed fixes. His gaze momentarily leaves the popup as his attention is drawn to the @Override annotation in the source code. He then *revisits* the popup because the fourth option also references this annotation:



```
 sublist(final int fromIndex, final int toIndex) {
<E> sub = deco
 LazyList<E>(s
```

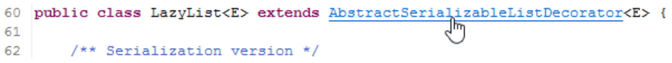
The method sublist(int, int) of type LazyList<E> must override or implement a supertype method

4 quick fixes available:

- Create 'sublist()' in super type 'AbstractCollectionDecorator<E>'
- Create 'sublist()' in super type 'AbstractListDecorator<E>'
- Create 'sublist()' in super type 'AbstractSerializableListDecorator<E>'
- Remove '@Override' annotation

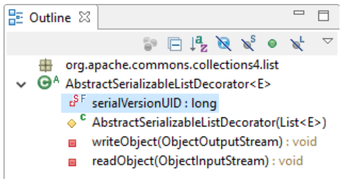
Press 'F2' for focus

Barry knows the @Override annotation is used to inform the compiler that the current method should override a method in a parent class. To see if this is true, he navigates to the class declaration, and control-clicks on the parent class:



```
public class LazyList<E> extends AbstractSerializableListDecorator<E> {

    /** Serialization version */
```

He inspects the *outline pane*, which summarizes all of the methods in the class, and confirms that the parent class contains only unrelated methods like writeObject:

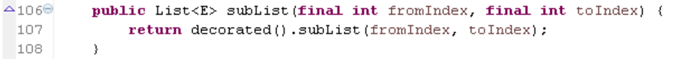


org.apache.commons.collections4.list
AbstractSerializableListDecorator<E>
serialVersionUID : long
AbstractSerializableListDecorator(List<E>)
writeObject(ObjectOutputStream) : void
readObject(ObjectInputStream) : void

He's now convinced that his colleague may have inadvertently included the @Override annotation, which happens to not be applicable to his solution. He returns to the original class one last time, and applies the "Remove '@Override' annotation" fix. Eclipse rebuilds the project and he checks the problems pane one last time to see that the error is no longer present.

If you were Barry, would you have done anything differently? If not, you're not all that different from the participants in our own study, where 53 of the 55 participants adopted a similar comprehension and resolution strategy.

Unfortunately, this fix turns out to be incorrect. The actual problem is that the sublist method declaration is misspelled and should have been called subList, with a capitalized L. Barry might have discovered this misspelling had he navigated one more step up in the class hierarchy, to the grandparent class:



```
    public List<E> subList(final int fromIndex, final int toIndex) {
        return decorated().subList(fromIndex, toIndex);
    }
```

Worse, this scenario is not isolated to Barry. For example, the highest-rated post on StackOverflow for the @Override annotation error suggests that it commonly occurs in situations where method names have been misspelled.[1]

Did Barry simply not pay enough attention to the error message? On close inspection, the error message does in fact mention supertype methods, though not explicitly by name. Or could it also be the case that the error message leads developers to prioritize certain solutions spaces for their code over others?

III. Methodology

A. *Research Questions*

In this study, we investigate the following research questions, and offer our rationale for each:

RQ1. How effective and efficient are developers at resolving error messages for different categories of errors? We ask this research question to assess the representativeness of our experimental tasks with respect to the costly error messages identified by Seo and colleagues [2], where costly is defined as frequency of the error times the median resolution time. Additionally, the results of this research question provide descriptive statistics to identify if some categories of defects are more difficult to resolve than others.

RQ2. Do developers read error messages? Although IDEs present error messages intended to be used by developers, the extent to which developers read these messages in their resolution process is an open question. Without answering this question, toolsmiths who are attempting to improve error messages may be misapplying their efforts. For instance, a developer might use the problems pane not to actually read the error message, but instead because they know that double-clicking an error message in the pane is a convenient way to jump to the offending location in the source code.

RQ3. Are compiler errors difficult to resolve because of the error message? Resolving compiler errors within the IDE requires developers to perform a combination of activities, such as navigating to files and making edits to source code. One hypothesis is that certain compiler errors are difficult to resolve not because the error message itself is cryptic, but because the task requires intricate code modifications in order

[1] http://stackoverflow.com/questions/94361/when-do-you-use-javas-override-annotation-and-why

to address the defect. Alternatively, it may be the case that the resolution requires only a simple code change to correct the defect, but a confusing error message hampers the developer from discovering the required code change. In short, we want to understand the extent to which poor error messages are harmful to the developer.

B. Study Design

Participants. We recruited 56 students from undergraduate and graduate courses in software engineering courses at our university. Through a post-experiment questionnaire, participants reported an average of 1.4 years ($sd = 1.3$) of professional software engineering experience, that is, experience obtained from working as a developer within a company.

Siegmund recommends self-estimation questions as a good indicator for judging programming experience, especially when participants are students [8]. Following this guidance, we asked participants about their familiarity with Eclipse and their knowledge of the Java programming language. Participants self-rated their familiarity with the Eclipse development environment with a median rating of "Familiar with Eclipse (3)," using a 4-point Likert-type item scale ranging from "Not familiar with Eclipse (1)" to "Very familiar with Eclipse (4)." Participants self-rated their knowledge of the Java programming language with a median rating of "Knowledgeable about Java (3)," using a 4-point Likert-type item scale ranging from "Not knowledgeable about Java (1)" to "Very knowledgeable about Java (4)." Participants also self-reported demographic data. Participants reported a mean age of 24 years ($sd = 6$), 46 reported their gender as male, and 10 reported their gender as female.

All participants conducted the experiment in one of two eye tracking labs on campus, and both labs contained identical equipment. Participants received extra credit for participating in the study. The first and third authors of the paper conducted the study.

Tasks. We derived tasks in our eye tracking experiment from prior work conducted by Seo and colleagues, where they empirically obtained build failures from over 26 million builds at Google [2]. From this data, they identified costly error messages that occurred frequently in practice and were time consuming for developers to resolve. To constrain the study to under one hour, we selected the top errors from each category of costly error messages, for a total of ten error messages (Table I). Through an informal pilot study with two other lab members, we found that developers resolved each defect in under five minutes.

However, we did not have access to the actual source code which generated the errors in the Google study. As a substitute, we used the Apache Commons Collections[2] library and manually injected faults into this library to generate error messages.

We chose Apache Commons Collections for several complementary reasons. First, it provides a library of data structures, such as lists, sets, and dictionaries, that are likely to be familiar to even first or second year students. Using such a library also allowed us to isolate the effects of developer difficulties in understanding error messages from that of unfamiliar code. Second, the library is open source, mature, and moderately-sized in terms of lines of code. Third, the library provides unit tests that can be used as a ground truth for the expected behavior of the code.

For each error, we introduced the error message into the Apache code through operations that could reasonably occur in actual development practices. For example, the `@Override` misspelling described in the motivating example (Section II) was applied based on comments on StackOverflow.

Tools and Apparatus. Participants used a Windows 8 machine with a 24-inch monitor, having a resolution of 1920x1080 pixels. The computer was connected to a GazePoint GP3 [9] eye tracking instrument, and this instrument was positioned directly below the monitor. GP3 software and drivers were installed on the computer to collect both the screen recording of the desktop environment and to synchronize the time of the recordings with the eye tracking instrument. Participants used an external keyboard and mouse to interact with the computer. The experimenters also installed custom scripts on the machine so that they could remotely load participant tasks.

We choose the Eclipse IDE [10] for this study because its presentation of errors, for example, through the problems pane and Quick Fix popups, are characteristic of the way errors are presented in other modern IDEs such as IntelliJ [11] and Visual Studio [12]. A default Eclipse installation was deployed on the machine, with minimal customizations. Specifically, we disabled Eclipse themes and turned off rounded edges on windows to facilitate subsequent detection during the data cleaning phase of the research.

C. Procedure

Onboarding. All participants signed a consent form before participating in the study.[3] Using a script, the experimenter verbally instructed participants with the details of the study. Participants were informed that they would be identifying and resolving ten source code defects, to be presented as compiler error messages in their IDE. Participants were given five minutes per task. If the participants finished early, they were asked to alert the experimenter and proceed to the next task. After two minutes, participants were also provided the option to discontinue the task.

We asked participants to provide a reasonable solution for the defect that they felt best captured the intention of the code. For example, although deleting all the files in the project might remove the compiler defect, it would be highly unlikely that this is an intended resolution. We told participants they were not expected to successfully fix all the defects, and that some defects might be more difficult than others.

[2] http://commons.apache.org/proper/commons-collections/

[3] North Carolina State University IRB 5372, "Evaluating text and visual notifications in integrated development environments during debugging tasks."

TABLE I
PARTICIPANT COMPILER ERROR TASKS

Task	Error Message[1]	Package	Category	Defect Introduced
T1	The method sublist(int, int) of type LazyList<E> must override or implement a supertype method	List	Semantic	Renamed sublist to subList, breaking existing @Override annotation.
T2	The type CursorableLinkedList<E> must implement the inherited abstract method List<E>.isEmpty() The type NodeCachingLinkedList<E> must implement the inherited abstract method List<E>.isEmpty()	List	Semantic	Deleted isEmpty method from abstract parent class.
T3	The import org.apache.commons.collections3 cannot be resolved (... repeated 50 times)	Map	Dependency	Changed version of collections4 to non-existent collections3 library in import statements.
T4	The method get() is undefined for the type Queue<E>	Queue	Dependency	Renamed method invocation from element() to non-existent get().
T5	The method add(E) in the type Collection<E> is not applicable for the arguments (int, capture#8-of ? extends E)	Set	Type mismatch	Added additional argument of 0 to add method call.
T6	Type mismatch: cannot convert from Set<Map.Entry<K,V>> to Set<Map.Entry<V,K>> Type mismatch: cannot convert from Set<Map.Entry<V,K>> to Set<Map.Entry<K,V>>	Map	Type mismatch	Swapped key and value in dictionary from Entry<K,V> to Entry<V,K>.
T7	Unhandled exception type InstantiationException	Map	Other	Changed less specific exception Exception to IllegalAccessException, which is not thrown by the code.
T8	Duplicate method next() in type EntrySetMapIterator<K,V> Duplicate method next() in type EntrySetMapIterator<K,V>	Iterators	Other	Copied and pasted next method to create duplicate method.
T9	Cannot make a static reference to the non-static type E	Queue	Semantic	Added static modifier to readObject method.
T10	Syntax error on token "default", : expected after this token	Map	Syntax	Removed : from default: in switch statement.

[1] For each error message, we compile the defective version of the code under Open JDK to replicate the compiler-internal error message key from the Seo and colleagues study at Google [2]. The Eclipse version of this message is shown to the participants.

Because of limitations with the eye tracking equipment, we asked participants to leave the Eclipse window full-screen. We also asked them to not use any resources (such as a web browser) outside of the Eclipse, because doing so would confound external information with error messages in the IDE. However, we permitted participants to use any of the features available within Eclipse, as long as these features did not change any of the Eclipse preferences or install any new Eclipse packages.

Finally, we provided the participants with a notifications sheet, which detailed all of the locations where error message information could appear in the IDE.

Calibration. The eye tracker must be calibrated for each participant. To avoid repeating the calibration, we requested participants to adjust their seating to a position that would feel comfortable for the duration of the study. We conducted a 9-point calibration using the software provided by the eye tracker, in which participants must fixate on circles that appear at different locations on the screen. To confirm that the calibration had successful applied, we conducted a stimuli task in the Eclipse environment. For this task, we asked the participant to navigate to the About dialog box within the Help menu, and read the version number of Eclipse. We also asked them to read a provided warning message in the problems pane of the IDE. Together, these tasks established a baseline for calibration.

Experiment. To control for learning effects, participants sequentially received one of ten tasks in randomized order. Participant were not allowed to revisit previous tasks, nor were they allowed to ask questions to the experimenter. Participants received no feedback on the correctness of their solution. On average our participants took approximately 45 minutes to complete the study. Following the experiment, participants completed a post-questionnaire about basic demographic information and experience.

D. Data Collection and Cleaning

Data collection. For each participant and task, we collected screen recordings in video format (at 10 frames per second) and a time-indexed data file containing all eye movements recorded by the eye tracking instrument.

Data cleaning of titles. We used the OpenCV computer vision library [13] to process videos on a frame-by-frame basis. To obtain the currently opened source file, we used the Tesseract

OCR engine to identify the titlebar for each frame [14]. Due to errors in OCR translation, we performed two data cleaning steps on the title. First, we cropped each frame to only the title bar and scaled it by a factor of three to artificially increase the font size. Second, we modified Tesseract to recognize only alphanumeric characters, dash (`-`), period (`.`), and forward slash (`/`).

After Tesseract processing, several OCR errors remained. Thus, we applied a Gestalt pattern matching algorithm [15] to match the OCR'd title against the known set of all Java files in the Apache Commons Collection library.[4] We manually added the strings, `Java - Eclipse`, which appears in the title when no file is open, as well as several classes from the `java.util` package to this processing step. The output of this step a list of the title associated with each frame.

Data cleaning of areas of interest. Areas of interest (AOIs) are labeled, two-dimensional rectangular regions of the screen that represent a logical component within the interface. In our experiment, we first characterized four areas of interest that are typically always present on the screen: 1) the explorer pane, which appears on the left-side of the screen and allows the developer to navigate the project files, 2) the outline pane, which appears on the right-hand side of the screen and contains a list of methods for the current class, 3) the problems pane, at the bottom of the screen and contains a list of the identified errors and warnings in the project, and 4) the source editor, which appears in the center of the screen and contains the source code.

We characterized two additional areas of interest that transiently displayed error messages: 1) the error popup, which appears when the developer hovers over the icon in the margin of the source editor, and 2) the Quick Fix popup, which appears when the developer hovers over the red wavy underline on program elements in the source code, or when they activate the Quick Fix feature explicitly.

To support automatic detection of each of the six areas of interest, we implemented several techniques. For fixed-sized AOIs, such as the popups, we extracted isolated screen captures of each popup and saved them as *templates*. For dynamically-sized AOIs, we extracted the boundaries of the essential features of the elements, and then performed a calculation to dynamically compute its bounding rectangle. For example, to identify the source editor, we internally match against three *sub-templates*: the top-left tab, the top-right minimize button, and a thin horizontal line that delimits the source code from any panes below it.

At this point, we have computationally usable templates that represent meaningful areas of interest, and we need to identify where these templates occur within video frames. To do this, we used a template matching algorithm provided by OpenCV. This algorithm essentially takes a given template, and slides it over the entire frame. For every overlap, the algorithm computes a normalized score between 0.0 and 1.0, where 0.0 represents the least likely match, and 1.0 represents a perfect match. Through trial-and-error, we found that a threshold of 0.95 yields accurate detection of AOIs. Because this is a computationally demanding operation, we down-sampled both the templates and the video frames to 50% of their original size to reduce the number of pixels needing to be processed at each frame. We then up-scaled the identified pixel locations to map them to the original video locations. The output of this procedure is a data file containing the identified AOIs for each video frame and the bounding box of that AOI.

Data cleaning of fixations. The eye tracking instrument internally has a proprietary algorithm for differentiating fixations, or sustained eye gazes, from other types of rapid eye movement that naturally occurs as people process information. However, the instrument has systematic measurement error in that the fixations locations are misaligned by a constant factor. Thus, for each of the participants' tasks, we used the stimuli task as a baseline to determine the initial horizontal and vertical offset. For the remaining tasks, we adjusted the baseline as necessary.

After performing offset adjustment, we used the GazePoint software to extract a list of fixations. For each record in the list, the record contains the time it began, its duration, and its coordinates.[5]

IV. Analysis

A. *RQ1: How effective and efficient are developers at resolving error messages for different categories of errors?*

We calculated the effectiveness for each task by using correctness as a proxy for effectiveness. For us to consider a solution to be correct, the solution must pass all of the unit tests in the Apache Commons Collections library after removal of the compiler error. Otherwise, the solution is considered incorrect. Next, we cataloged and binned the incorrect solutions observed for each task, with the intuition that if unsuccessful participants make the same types of mistakes, there is some common underlying cause.

We calculated efficiency from two time-derived metrics: time to complete task, and participant effort. For time to complete task, we extracted the start and end times from the videos using the icon in the problems pane label as a trigger, using transitions of the icon from errors to no errors to indicate task boundaries. Tasks for which no end transition was found were marked as a timeout. Participants who declined to continue the task and did not resolve the defect were also assumed to timeout.

For participant effort, we calculated a metric called response time effort [16]. Intuitively, if incorrect solutions are achieved in significantly less time than correct solutions, then it would suggest that participants are not expending sufficient effort to reasonably resolve a compiler error message. That is, the participant may simply be careless, irrespective of the quality of the compiler error message or the difficulty of the task. We performed a two-tailed t-test between task times, excluding timeouts, under correct and incorrect solution conditions to gauge response time effort.

[4] In Python, this algorithm is available as `get_closest_match`.

[5] In the GazePoint software, this corresponds to the `FPOGS`, `FPOD`, `FPOGX` and `FPOGY` columns.

B. RQ2: Do developers read error messages?

Determining if developers are reading error messages through overt experimental designs is surprisingly tricky. For example, asking participants to think aloud as they resolve error message tasks adds a cognitive burden that has been found to negatively impact task performance [17]. Directly questioning the participant at the end of each task can introduce social-desirability and prime the participants' behavior for subsequent tasks [18], thus causing them to read error messages more carefully than they otherwise would have. Moreover, visual attention is a largely subconscious process; participants in visual attention tasks, such as reading, only have a rudimentary awareness of how they allocate their attention [19]. The use of eye tracking to answer this research question mitigates these issues, but introduces one of its own: eye tracking data is noisy. For example, routine, involuntary movements such as rubbing eyes and adjusting hair can block the eye tracking camera, introducing spurious data points. Our analysis techniques are constrained to those that are robust in the presence of noise.

Previous eye tracking work by Rayner modeled "reading" as distribution times of fixations under a variety of visual stimuli [20]. Rayner characterized the distribution times of fixations under different reading conditions, such as silent and oral reading. Through a meta-analysis, Rayner found that the mean fixation time of a distribution increases with the difficulty of the text.

For fixations within the source code and error message AOIs, we replicated this analysis, postulating that developers must read at least the source code in order to resolve a compiler error message as a baseline.

We then characterized the distribution of source code, error messages, and silent reading (provided by Rayner [20]) through a symmetric Kullback-Leibler (KL) divergence, for each of pair of distributions. Essentially, KL divergence is an information-based measure of disparity: the larger the value of the divergence between two distributions, the more information is "lost" when one distribution is used to model the second distribution [21].

Finally, to understand how developers allocate their attention to error messages against other areas of interest, for each task, we computed across participants the percentage of fixations for the areas of interest in the task.

C. RQ3: Are compiler errors difficult to resolve because of the error message?

Although Seo and colleagues identified a distribution of costly error messages [2], this does not automatically imply that the reason the error message is costly to resolve is due to the error message itself. For example, an error message about a mismatched brace may be easy to comprehend for the developer, yet costly to resolve because the developer must spend most of their effort in the source code editor to identify where to add or remove a brace. In answering this research question, our goal is specifically to understand the extent to which difficulties with reading error messages can be attributed to task performance difficulties.

To understand if compiler errors are difficult to resolve because of the error message, we used the eye tracking measure of *revisits*, that is, leaving an area of interest and then subsequently returning to it, as a measure of reading difficulty. In prior work, Jacobson and Dodwell identified the relationship that increasing fixation revisits to an area of text is a measure of increased reading difficulty for that area [22].

To answer this question, we computed a nominal logistic model between correctness and revisits to error messages. The output of the model is a probability of correctness against the number of visits, over a distribution of tasks.

To evaluate the model, we computed the Nagelkerke's coefficient of determination, R^2 [23]. Of course, there are many variables that influence whether or not a developer will be successful at resolving a compiler error, such as previous knowledge, experience, and familiarity with the code [24]. Consequently, we expect that the coefficient of determination will be low, because reading difficulty is only a second-order variable for these other factors. Fortunately, reading difficulty latently encodes many of these variables: if a developer has little experience with a particular error message, this lack of experience should manifest itself through how they read the error message.

We also evaluated the model using a likelihood-ratio Chi-square test (G^2) computed between a *full* model, using the number of revisits as a predictor variable, against a *reduced* or intercept-only model, fit without any predictor variables. If the addition of a predictor variable is identified by the test as significant ($\alpha < 0.05$), then the predictor variable significantly improves the fit of the model.

V. Verifiability

To support replication of our findings, we have provided several materials on our website.[6] The website contains the videos of each of the participants' tasks. Additionally, we provide a data file containing all gazes for the tasks, and a cleaned data file containing only the fixations for the tasks. To support verification, we provide annotated diagnostic videos of participants' tasks that display rendered rectangles on identified areas of interest as the video plays. Finally, we provide data files from the output of our data cleaning process.

VI. Results

A. RQ1: How effective and efficient are developers at resolving error messages for different categories of errors?

Table II summarizes the solution the developer makes based on our correctness criteria. For every task, at least one participant made a code change congruent with the correct solution, which indicates that it is at least possible to make a correct fix for every task given the information in the error messages. The distribution of correct and incorrect solutions are clearly skewed for many of the tasks. For example, only two participants generated correct solutions for T1, and only one participant generated a correct solution for T2. And for

[6] http://go.barik.net/gazerbeams

TABLE II
OVERVIEW OF TASK PERFORMANCE

	Correct		**Incorrect**			**Timeout**	
Task	n	%	n	%	n_{bins}[1]	n	%
T1	2	4%	47	85%	2	6	11%
T2	1	2%	49	91%	1	4	7%
T3	30	55%	0	0%	0	25	45%
T4	36	65%	10	18%	3	9	16%
T5	49	89%	5	9%	2	1	2%
T6	55	100%	0	0%	0	0	0%
T7	22	40%	23	42%	1	10	18%
T8	48	87%	5	9%	1	2	4%
T9	28	51%	2	4%	2	25	45%
T10	50	91%	5	9%	3	0	0%

[1] n_{bins} indicates the number of observed incorrect solution types for each task, that is, the cardinality of the incorrect solution set.

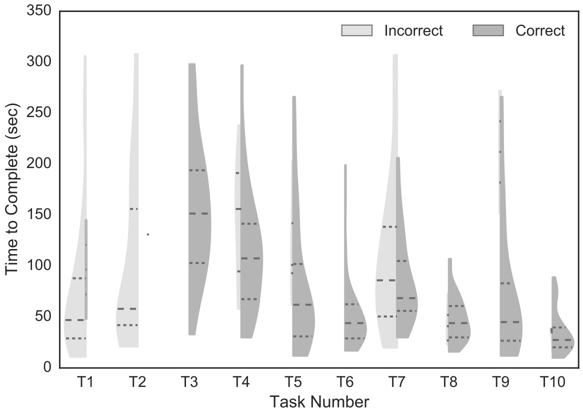


Fig. 1. The time required for developer to commit to a solution that is correct or incorrect. Nearly all tasks (exceptions, T8 and T10) have high variance in resolution time to arrive, irrespective of correctness.

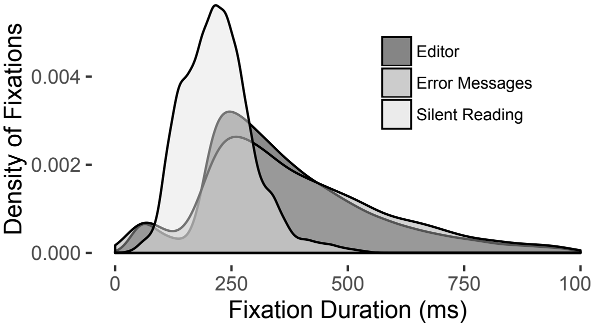


Fig. 2. Comparison of fixation time distributions for silent reading of English passages, reading source in the editor, and reading of error messages.

tasks T5, T6, T8, and T10, nearly all or all participants arrived at the correct solution.

The n_{bins} columns in Table II indicates, for every incorrect solution, the types of solutions provided by the participants. For example, consider T1, the problem Barry faced in our motivating example (Section II). Recall that the correct solution to this problem is to rename the `sublist` method to `subList`. Participants provided two incorrect solutions to this problem. They either removed the `@Override` annotation, which suppresses the error but does not resolve the defect, or they created a stub `sublist` method in the parent class, without recognizing the existing similarly-named method. Across all tasks, participants converged to a small set of incorrect solutions.

Figure 1 illustrates a violin plot of the time required for developers to apply a resolution for both correct and incorrect solutions. The dashed lines indicate quartile boundaries for each task. For incorrect solutions, timeouts are excluded from the plot. Like Seo and colleagues, we also found large variation in the time required to arrive at a solution for nearly all tasks [2]. For some tasks, however, such as T8 and T10, most participants were able to correctly resolve the defect, and with relatively tight variation in time to resolution. As Seo and colleagues defined costly in terms of both frequency of occurrence and median time to resolution [2], it is likely these errors are costly because they are arise frequently as a nuisance for developers, not because they are particularly difficult to resolve.

For response time effort, a t-test identified a significant difference in resolution time between correct and incorrect solutions ($t(20.24) = 2.86$, $p = 0.0045$), with incorrect solutions requiring an additional mean time of 20 seconds ($sd = 7$) over the correct solution. The results of this test provide support that participants provided sufficient effort in attempting to solve the task, and rejects the hypothesis that participants chose an incorrect solution because it most quickly resolved the defect.

B. RQ2: Do developers read error messages?

Figure 2 illustrates the distribution of known silent reading durations against the distribution of fixation times for error message areas of interest in our tasks. For visualization purposes, the distributions are normalized as a probability density function. That is, the probability of a random fixation to fall within a particular region is the area under the curve for that region.

The mean fixation time for reading in the source code editor is 394ms ($sd = 240$, $n_{\mathrm{fix}} = 81098$). In comparison, previous work found that silent reading of English passages of text yield mean fixation times of 275ms ($sd = 75$), and that for reading and typing English passages yield a mean fixation time of 400ms (sd not provided in original study by Rayner) [20]. Thus, reading source code is much more difficult than reading a general English passage, and marginally less difficult than the activity of typing while reading.

We compute the KL divergence between the source editor distribution and error message distribution ($D_{KL} = 0.059$), source editor distribution and silent reading distribution ($D_{KL} = 3.38$), and error message distribution and silent reading distribution ($D_{KL} = 2.37$). From the relatively small KL divergence between the source editor distribution and the error message distribution, we conclude that error message reading is comparable to source code reading ($u = 419$ms,

TABLE III
PARTICIPANT FIXATIONS TO AREAS OF INTEREST

	Task									
Area of Interest	T1	T2	T3	T4	T5	T6	T7	T8	T9	T10
Source editor	■ 66%	■ 66%	■ 74%	■ 79%	■ 68%	■ 65%	■ 78%	■ 68%	■ 80%	■ 65%
Error areas	■ 23%	■ 23%	■ 15%	■ 14%	■ 23%	■ 25%	■ 15%	■ 17%	■ 13%	■ 20%
Navigation areas	■ 10%	■ 11%	■ 11%	■ 6%	■ 9%	■ 11%	■ 7%	■ 15%	■ 7%	■ 15%
Error Areas Breakdown[1]										
Error popup	■ 1%	■ 0.5%	—	■ 0.4%	■ 0.7%	■ 2%	■ 0.5%	■ 1%	■ 1%	■ 2%
Problems pane	■ 12%	■ 19%	■ 15%	■ 11%	■ 16%	■ 17%	■ 9%	■ 14%	■ 11%	■ 16%
Quick Fix popup	■ 10%	■ 3%	—	■ 3%	■ 6%	■ 5%	■ 6%	■ 1%	■ 1%	■ 2%
Navigation Breakdown										
Explorer pane	■ 10%	■ 8%	■ 10%	■ 5%	■ 8%	■ 11%	■ 6%	■ 12%	■ 5%	■ 14%
Outline pane	■ 1%	■ 3%	■ 1%	■ 1%	■ 1%	■ 1%	■ 1%	■ 3%	■ 1%	■ 1%

[1] To understand reading, the error areas breakdown aggregates areas of interest to those that provide the text of the error message.

$sd = 270$, $n_{\text{fix}} = 18573$), and unlikely to be a different activity than reading.

Another perspective on understanding whether developers read error messages is to examine how they allocate their fixations across different areas of interest during the task (Table III). Across tasks, we found that participants spent 65%–80% of their fixations in the source editor, with 13%–25% of their fixations on error message areas of interest. Most participants use the error message information in either the problems pane and Quick Fix popup; the error popup is rarely used.

Both the KL divergence analysis and the allocation of fixations to the task support that developers are reading messages.

C. RQ3: Are compiler errors difficult to resolve because of the error message?

A Chi-squared test reveals a statistically significant difference between the number of revisits and the outcome of correctness over a distribution of tasks ($df = 1, G^2 = 60.9, p < .0001$). This suggests that the number of revisits, a proxy measure for reading difficulty [22], is a significant predictor variable for the task difficulty. However, Nagelkerke's coefficient of determination for the logistic fit is low ($R^2 = 0.16$), which implies that reading difficulty is only one of many factors that contribute to the overall difficulty of a task.

Figure 3 presents this statistical result more intuitively as a plot of the nominal logistic model for the number of revisits to error messages against the probability of the developer applying a correct solution for the task. The dotted line indicates the regression fit, separating correct and incorrect task solutions. Tasks are plotted against the number of revisits, and randomly jittered vertically above and below the regression fit to help facilitate visualization. From the figure, we can see that as the number of revisits increases, more incorrect tasks fall above the regression fit. Thus, task difficulty is attributable to the reading difficulty of error messages.

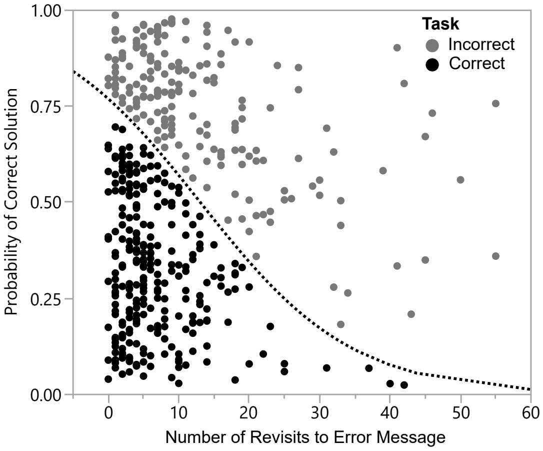


Fig. 3. Nominal logistic model of number of revisits on error message areas of interest against probability of applying a correct solution. As revisits to error messages increase, the probability of successfully resolving a compiler error decreases.

VII. DISCUSSION

From the results of our research questions, we identify three problem areas in current development environments, and offer suggestions for toolsmiths towards addressing these problems.

Error messages are not situationally-aware (RQ1). Although our tasks covered the distribution of costly errors, the way in which the defects themselves can manifest is situationally-dependent. For example, consider T2, in which through a merge the method `isEmpty()` has inadvertently been deleted from the parent class, `AbstractLinkedList<E>`, whose children are `CursorableLinkedList<E>` and `NodeCachingLinkedList<E>`.

This causes the compiler to emit two error messages:

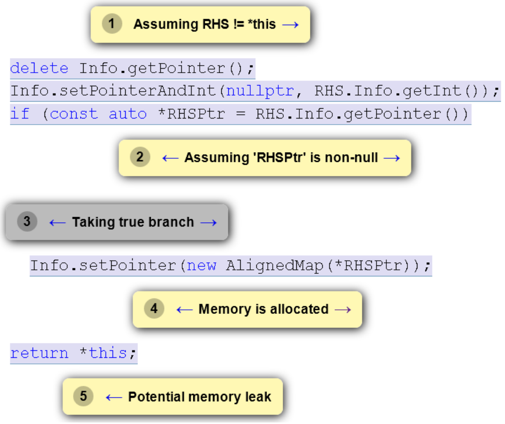
```
The type CursorableLinkedList<E> must implement the
inherited abstract method List<E>.isEmpty()

The type NodeCachingLinkedList<E> must implement the
inherited abstract method List<E>.isEmpty()
```

For this message, it is not surprising that developers would be led to believe that they should implement the `isEmpty()` method in both classes. Indeed, all participants except for one came to this incorrect conclusion (Table I, $n_{\text{bin}} = 1$).

Instead, consider if the compiler had presented the following alternative message:

```
The type AbstractLinkedList<E> must implement the
inherited abstract method List<E>.isEmpty()
```

Given the fact that our participants took cue from the error messages in the first error message set, it is plausible that participants would have arrived at the correct solution, adding the missing method to the `AbstractLinkedList<E>` class, if



1 Assuming RHS != *this →

```
delete Info.getPointer();
Info.setPointerAndInt(nullptr, RHS.Info.getInt());
if (const auto *RHSPtr = RHS.Info.getPointer())
```

2 ← Assuming 'RHSPtr' is non-null →

3 ← Taking true branch →

```
  Info.setPointer(new AlignedMap(*RHSPtr));
```

4 ← Memory is allocated →

```
return *this;
```

5 ← Potential memory leak

Fig. 4. Emerging error reporting systems like LLVM scan-build provide stark contrast to those of conventional IDEs. Here, scan-build presents error messages for a potential memory leak as a sequence of steps alongside the source code to which the error applies.

they had instead been presented with the second error message. Unfortunately, it's difficult for the compiler to know which of these messages would be more appropriate to present to developer without having situational information, such as recent edit history.

Compiler designers may want to consider incorporating situational awareness into their choice of presentation to aid developers in more accurately identifying and resolving the root cause of a defect.

Error messages appear to take the form of natural language, yet are as difficult to read as source code (RQ2). Although we expected error message mean fixations to be somewhat higher than silent reading due to more technical language, we were surprised to find that error messages were not only significantly more difficult to read than the silent reading of natural language, they were also slightly more difficult to read than even the source code.

To postulate why this might be, let's return to Table I. Consider an error message as in T4, which reads:

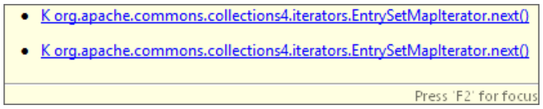
```
The method get() is undefined for the type Queue<E>
```

Even for relatively short error messages like this one, participants spent 14% of their time in the total task with fixations across essentially nine "words." Prior research in language cognition for bilingual sentence reading has found that switching languages is associated with a cognitive processing cost [25]. Similarly, one explanation for the apparent difficulty of reading error messages is that error messages consist of both natural language ("`is undefined for`") and code ("Queue<E>"), but are not entirely either. Consequently, developers must context-switch between two different modalities of reading to fully capture the information presented in an error message, leading to increased reading difficulty.

A second explanation for why error messages are comparable in difficulty to reading source code is because reading error messages requires the developer to also switch between the error message and the source code in order to understand the full context of the task. For example, a developer might read "`The method get()`" and then suspend their reading of the error message to determine in the source editor the calling context of this method and figure out why and whether it should be called. In this case, error presentation approaches such as those found in LLVM scan-build [26] may prove beneficial to developers (Figure 4). Unlike conventional error messages, which decouple the error message from the code context, scan-build presents the error as a sequence of steps that the developer can follow alongside in the context of the code to which the error message applies.

Tools fail developers in the presence of compiler errors (RQ3). While difficult error messages are a significant predictor of correctness, the low R^2 from RQ3 suggests that other factors exist which affect the difficulty of a resolution task. For example, in addition to reading error messages, participants in our study also had fixations within navigation areas of interest for 6%–15% of the total task.

In observing the participant videos for these tasks, we found several instances where participants attempted to use tools that fail in the presence of a compiler error message during navigation-related tasks. As one example, a participant in T8 attempted to navigate to the appropriate method through a usage of that method. Although the Eclipse IDE would reveal both locations, it makes no special effort to distinguish the two methods, leading to potential visual disorientation within their IDE [27]:



- K org.apache.commons.collections4.iterators.EntrySetMapIterator.next()
- K org.apache.commons.collections4.iterators.EntrySetMapIterator.next()

Press 'F2' for focus

In T8, yet another participant was aware of a tool within the IDE to facilitate comparison between two methods, but they could not recall how to invoke it. This example illustrates how tools that support understanding of a defect may be just as important as tools that provide a resolution. But unlike Quick Fix popups, which appear automatically, comprehension tools such as `Compare With` must be invoked manually by the developer. Analogous to Quick Fix popups, perhaps toolsmiths should offer "Quick Understanding" popups, which recommend appropriate tools, such as `Compare With`, that are known to be helpful in understanding a particular compiler error. Our own work in defect resolutions provides a starting point for automatically bringing relevant tools to the developer to help with comprehension [28].

VIII. Limitations

Although we derived our errors based on frequency and difficulty of resolution from a prior Google study [2], we could not ensure that we used the similar phrasing in our replication of error messages. Google uses a proprietary version of OpenJDK with custom messages not available to the general public. We also do not have access to the source code that generated these errors. As a result, the messages we use in Eclipse are not identical to those previously studied. Instead, in our study design, we seeded errors that approximate the scenario described by the original message.

We only investigated ten error messages with our participants. However, research by Seo and colleagues found that improving even the 25 of the top errors would cover over 90% of all the errors ever encountered at Google. A similar result has since been replicated for novice developers in Java [5]. Furthermore, we sampled error messages across multiple categories of defects to increase generalizability.

One construct validity issue is the accuracy of the eye tracker. We used a commodity eye tracking instrument which has a lower sample rate than professional eye tracking equipment. For example, our eye tracker did not perform well with participants with glasses, and was also sensitive to lighting conditions. The commodity eye tracker also limited our analysis to larger areas of interest; we were unable to perform line or word level analysis, which would be useful for further understanding parts of the text developers actually read. We had to discard 51 tasks due to equipment malfunction during participant sessions. An additional 79 tasks were dropped because reasonable eye tracking data was not obtainable from the participant data, even after manual offset correction.

A threat to external validity is that we used student developers in place of full-time professional developers. Although many of our participants had professional experience, these participants may not fully represent industry developers in all situations [29]. For example, it is possible that senior developers with extensive experience of their own code base would arrive at a correct solution for some tasks irrespective of the information in the message (RQ1). Although our participants rarely used error popups in their IDE (RQ2), we might expect that senior developers, again having familiarity with their code base, would utilize these on-demand information sources more frequently than the always-available problems pane. Lastly, developer effectiveness for a task and its relationship to error message revisit frequency may be less pronounced when participants have a broader range of developer experience than those in our own study (RQ3). Thus, we should be careful in generalizing our findings to all developers.

IX. Related Work

To our knowledge, this study is the first to use eye tracking to explain how developers make use of error message information to resolve defects within the IDE. In this section, we discuss related work from eye tracking studies in debugging and defect understanding. Rodeghero and colleagues used eye tracking to understand how developers summarize code; the results of their experiment were used to improve algorithms that automatically summarize code [30]. Romero and colleagues used eye tracking to understand how developers found defects in source code under different representations of the source code, such as diagrams [31]. Uwano and colleagues asked developers to perform code review tasks, during which participants had to locate defects in the code [32]. The authors identified common scan patterns in subjects' eye movements. In a partial replication of Uwano's study, Sharif and colleagues found differences between experts' and novices' scan patterns while locating defects [33]. Bednarik and Tukiainen found that repetitive patterns of visual attention were associated with lower performance [34]. In our study, we also found that revisits to error message information is statistically significant with the probability of correctness. Like other research attempting to find patterns in debugging activities, Hejmady and Narayanan found visual pattern differences based on programming experience and familiarity with the IDE [35]. Busjahn and colleagues were interested in how novices read source code; from eye tracking, they found that experts read code less linearly than novices [36]. In our own work, we are interested, for example, in whether developers read error messages at all.

Outside of eye tracking, other studies are related to our investigation of comprehension and resolution of error messages. Researchers de Alwis and Murphy proposed a theory of visual momentum, identifying factors that may lead developers to become disoriented when exploring programs in the IDE [27]. Lawrence and colleagues present an information foraging theory on how developers debug code. They examined programmers' verbalizations and found that their debugging approaches more often concerned scent-following than hypotheses [37]. However, the use of verbalization has been found to affect the performance and decisions participants make on tasks [38].

X. Conclusion

In this work, we conducted a study using eye tracking as a means to investigate if developers read error messages within the Eclipse IDE. Through distribution comparisons between source code, error messages, and prior work on silent reading, we found support that developers are reading error messages, but also found that the difficulty of reading error messages is comparable to reading source code — a cognitively demanding task. By examining developer fixations, we found that developers spend a substantial amount of time on error message areas of interest, despite the fact that most tasks had only a single error message present. An analysis of revisit times as a proxy for reading difficulty suggests that difficulty in solving a task can be attributed to difficulties in reading the error message.

The results of this study reveal several problematic areas in the way development environments today present compiler error messages to developers, and identify opportunities for toolsmiths to address these problems. The contribution of our work is that it offers an empirical justification for improving compiler error messages for developers. Stated simply: error messages matter.

XI. Acknowledgments

This material is based upon work supported by the National Science Foundation under grant number 1217700 and 1318323. Computational resources for this study were provided through an AWS in Education Research grant from Amazon.

References

[1] V. J. Traver, "On compiler error messages: What they say and what they mean," *Advances in Human-Computer Interaction*, vol. 2010, pp. 1–26, 2010.

[2] H. Seo, C. Sadowski, S. Elbaum, E. Aftandilian, and R. Bowdidge, "Programmers' build errors: A case study (at Google)," in *ICSE*, May 2014, pp. 724–734.

[3] B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge, "Why don't software developers use static analysis tools to find bugs?" in *ICSE*, May 2013, pp. 672–681.

[4] M. Christakis and C. Bird, "What developers want and need from program analysis: An empirical study," in *ASE*, 2016, pp. 332–343.

[5] D. Pritchard, "Frequency distribution of error messages," in *PLATEAU*, Oct. 2015, pp. 1–8.

[6] P. Denny, A. Luxton-Reilly, and D. Carpenter, "Enhancing syntax error messages appears ineffectual," in *ITiCSE*, Jun. 2014, pp. 273–278.

[7] G. Marceau, K. Fisler, and S. Krishnamurthi, "Measuring the effectiveness of error messages designed for novice programmers," in *SIGCSE*, Mar. 2011, pp. 499–504.

[8] J. Siegmund, "Framework for measuring program comprehension," Ph.D. dissertation, University of Magdeburg, Nov. 2012.

[9] "GazePoint," http://www.gazept.com/.

[10] "Eclipse," http://www.eclipse.org/luna/.

[11] "IntelliJ," https://www.jetbrains.com.

[12] "Visual Studio," https://www.visualstudio.com/.

[13] "OpenCV," http://opencv.org/.

[14] R. Smith, "An overview of the Tesseract OCR engine," in *ICDAR*, vol. 2, Sep. 2007, pp. 629–633.

[15] J. W. Ratcliff and D. E. Metzener, "Pattern matching: The Gestalt approach," *Dr Dobbs Journal*, vol. 13, no. 7, p. 46, 1988.

[16] S. L. Wise and X. Kong, "Response time effort: A new measure of examinee motivation in computer-based tests," *Applied Measurement in Education*, vol. 18, no. 2, pp. 163–183, Apr. 2005.

[17] M. van den Haak, M. De Jong, and P. Jan Schellens, "Retrospective vs. concurrent think-aloud protocols: Testing the usability of an online library catalogue," *Behaviour & Information Technology*, vol. 22, no. 5, pp. 339–351, Sep. 2003.

[18] D. T. Welsh and L. D. Ordonez, "Conscience without cognition: The effects of subconscious priming on ethical behavior," *Academy of Management Journal*, vol. 57, no. 3, pp. 723–742, Jun. 2014.

[19] L. R. Harris and M. Jenkin, "Vision and Attention," in *Vision and Attention*. Springer New York, 2001, pp. 1–17.

[20] K. Rayner, "Eye movements in reading and information processing: 20 years of research." *Psychological Bulletin*, vol. 124, no. 3, pp. 372–422, 1998.

[21] J. M. Joyce, *Kullback-Leibler Divergence*. Springer Berlin Heidelberg, 2011, pp. 720–722.

[22] J. Z. Jacobson and P. Dodwell, "Saccadic eye movements during reading," *Brain and Language*, vol. 8, no. 3, pp. 303–314, 1979.

[23] N. J. D. Nagelkerke, "A note on a general definition of the coefficient of determination," *Biometrika*, vol. 78, no. 3, pp. 691–692, 1991.

[24] B. Johnson, R. Pandita, J. Smith, D. Ford, S. Elder, E. Murphy-Hill, S. Heckman, and C. Sadowski, "A cross-tool communication study on program analysis tool notifications," in *FSE*, 2016, pp. 73–84.

[25] E. M. Moreno, K. D. Federmeier, and M. Kutas, "Switching languages, Switching palabras (words): An electrophysiological study of code switching," *Brain and Language*, vol. 80, no. 2, pp. 188–207, 2002.

[26] "Clang Static Analyzer," http://clang-analyzer.llvm.org/.

[27] B. de Alwis and G. Murphy, "Using visual momentum to explain disorientation in the Eclipse IDE," in *VL/HCC*, 2006, pp. 51–54.

[28] T. Barik, Y. Song, B. Johnson, and E. Murphy-Hill, "From Quick Fixes to Slow Fixes: Reimagining Static Analysis Resolutions to Enable Design Space Exploration," in *ICSME*, 2016.

[29] I. Salman, A. T. Misirli, and N. Juristo, "Are students representatives of professionals in software engineering experiments?" in *ICSE*, 2015, pp. 666–676.

[30] P. Rodeghero, C. McMillan, P. W. McBurney, N. Bosch, and S. D'Mello, "Improving automated source code summarization via an eye-tracking study of programmers," in *ICSE*, May 2014, pp. 390–401.

[31] P. Romero, R. Cox, B. du Boulay, and R. Lutz, *Visual Attention and Representation Switching During Java Program Debugging: A Study Using the Restricted Focus Viewer*, 2002, pp. 221–235.

[32] H. Uwano, M. Nakamura, A. Monden, and K.-i. Matsumoto, "Analyzing individual performance of source code review using reviewers' eye movement," in *ETRA*, 2006, pp. 133–140.

[33] B. Sharif, M. Falcone, and J. I. Maletic, "An eye-tracking study on the role of scan time in finding source code defects," in *ETRA*, 2012, p. 381.

[34] R. Bednarik and M. Tukiainen, "Temporal eye-tracking data: Evolution of debugging strategies with multiple representations," in *ETRA*, 2008, pp. 99–102.

[35] P. Hejmady and N. H. Narayanan, "Visual attention patterns during program debugging with an IDE," in *ETRA*, 2012, pp. 197–200.

[36] T. Busjahn, R. Bednarik, A. Begel, M. Crosby, J. H. Paterson, C. Schulte, B. Sharif, and S. Tamm, "Eye movements in code reading: Relaxing the linear order," in *ICPC*, 2015, pp. 255–265.

[37] J. Lawrance, C. Bogart, M. Burnett, R. Bellamy, K. Rector, and S. D. Fleming, "How programmers debug, revisited: An information foraging theory perspective," *IEEE Transactions on Software Engineering*, vol. 39, no. 2, pp. 197–215, Feb. 2013.

[38] L. Cooke and E. Cuddihy, "Using eye tracking to address limitations in think-aloud protocol," in *International Professional Communication Conference*, 2005, pp. 653–658.