Improving Error Notification Comprehension in IDEs by Supporting Developer Self-Explanations

Titus Barik*†
*ABB Corporate Research, Raleigh, North Carolina, USA
†North Carolina State University, Raleigh, North Carolina, USA
titus.barik@us.abb.com

***Abstract*—Despite the advanced static analysis techniques available to compilers, error notifications as presented by modern IDEs remain perplexing for developers to resolve. My thesis postulates that tools fail to adequately support *self-explanation*, a core metacognitive process necessary to comprehend notifications. The contribution of my work will bridge the gap between the presentation of tools and interpretation by developers by enabling IDEs to present the information they compute in a way that supports developer self-explanation.**

I. Introduction

Modern software development typically occurs within an integrated development environment (IDE), such as Eclipse or Visual Studio. One task within the IDE is error notification comprehension. During this task, the IDE presents the developer with one or more notifications as text messages in a list box, with visual indicators such as a red wavy underline, that indicate the "primary" location of the error.

Unfortunately, analysis techniques are hampered by the limited visual annotations provided by IDEs [1], [2]. For example, the Roslyn[1] compiler internally computes rich diagnostics, such as static data flow analysis, but this high fidelity information is silently dropped as the information is narrowly funneled textually through the notification presentation engine. Although compilers have significantly more information to share about a diagnostic, much of this information appears to be discarded because visualizations in the IDE are not rich enough to fully express what the diagnostic wants to communicate.

Consequently, notifications remain perplexing for developers to resolve. For example, nearly 30% of Java builds at Google fail due to a compiler error, and the median resolution time for each error is 12 minutes [3]. For novice developers, that is, students using Java in the BlueJ IDE, the situation is worse — through telemetry of over 37 million compilation events, we find that nearly 48% of all compilations fail [4].

The goal of my research is to identify and address the difficulties developers face during these error comprehension tasks. In my thesis, I posit two hypotheses to address these challenges. First, I hypothesize that the current mechanism by which tools present information to the developer is misaligned with how developers need to interpret this information. More precisely, I experimentally provide evidence that tools fail to adequately support the metacognitive process of *self-explanation*. Second, I argue that much of the effort involved during self-explanation is clerical, and that this work is already computed by the compiler in order to generate a diagnostic in the first place. The contribution of my work will bridge this gap, by enabling IDEs to present the information they compute in a way that supports the self-explanation process that developers already perform (Figure 1).

[1]https://github.com/dotnet/roslyn

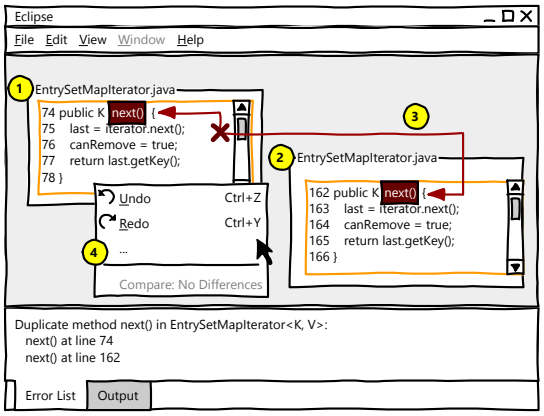

Fig. 1. An explanatory interface mockup for notifications, implemented within Eclipse. This example illustrates a `duplicate method` error, in which the `next` method has been inadvertently defined twice within the same scope. In (1) and (2), the interface uses a fragment-oriented editor to display both methods adjacently. (3) The interface adds visual annotations to make the conflicting relation explicit, using an internal schema that enhances self-explanation [1]. (4) Through the same schema, the interface infers that the developer will likely want to compare these fragments. In this case, the interface speculatively executes a comparison tool and determines that there are no differences.

II. Self-Explanation as an Underlying Theory for Comprehension Failure in Notifications

In one of the more accessible definitions offered by Legare, self-explanation is a process by which we "attempt to understand a causal relation by identifying relevant functional or mechanistic information [5]." In the context of code, a tool that supports self-explanation must make explicit the relations between program elements (that is, the mechanistic information, or *what*), and provide one or more techniques to help the developer reason and make informed decisions about why specification of the source code is a problem (that is, the functional information, or *why*).

I analyzed the notification corpora of several compilers, such as Java, C# and TypeScript, for explicit relations. A message has a relation if it requires two or more program elements to construct the message. Across all compilers, roughly 25% of messages have explicit relations. Triangulating, I

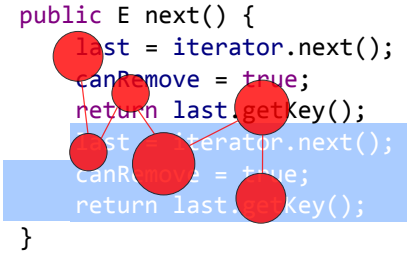

Fig. 2. In Eclipse, a developer attempts to identify whether the logic of two duplicate methods are identical. To do so, the developer manually places the code for the two methods together and highlights the second instance. The red circles represent fixations, and indicates the developer is performing a manual difference of the source code text.

found that the most costly error messages (as defined by their respective authors) are almost always relational [3], [4]. More surprising is that the kind of relational errors, such as types and dependencies, substantially overlap between both novices and experts, suggesting that such errors are difficult, independent of developer experience. Thus, failures in higher-level cognitive processes above knowledge recall, such as self-explanation, are a plausible hypothesis to investigate.

III. Evaluation Plan

Phase I (Completed): How should IDEs visually communicate with developers? To understand how current IDEs fail to support developer self-explanations, I conducted a user study with 28 undergraduate Software Engineering students [1]. I asked participants in this paper-and-pencil study to think-aloud and explicitly self-explain compiler errors, while making diagrammatic markings on top of the corresponding code. Through a modified Cognitive Dimensions Framework survey [6], participants identified that current visualizations fail to show important relations between program elements. Importantly, the diagrams that participants drew to explain their own understanding of error notifications, which I intend to adopt in Phase III, differed significantly from those of existing IDEs.

Phase II: What difficulties do developers encounter in modern IDEs? (In Progress) To further understand developer difficulties with notifications, I conducted an eye tracking study in which participants from undergraduate and graduate Software Engineering courses used the Eclipse IDE to resolve the most frequent Java compiler errors as reported by Seo and colleagues [3]. An initial analysis suggests that on average, developers spend less than 5% of their time for a task on the error message content; instead, they spend a significant amount of effort on collecting relevant information, arranging that information to make sense of it, and attempting to assess the impact of their intended change. Our preliminary results suggest that IDEs are not adequately providing developers the appropriate mechanistic and functional information needed to efficiently comprehend an error notification.

For example, consider the participant in Figure 2, who needs to determine which of the two `next` methods in a `duplicate method` error he wants to keep. To easily compare the two methods, he manually copies and pastes the logic of one method and places it adjacent to the other, highlighting one of them to differentiate it from the other. Then, he visually scans the code to conclude that the methods are identical, and realizes that either can be removed to resolve the defect. This is a clerical and time consuming process.

Phase III: How should tools incorporate self-explanation principles? (Not Started)

Consider the `duplicate method` error once again, but through an alternative explanatory interface (Figure 1). Unlike before, the interface brings together the two methods, through the use of a fragment-oriented editor [7]. Visual annotations, such as arrows, make the relationship explicit between methods. The compiler, having appropriate semantic information, can speculatively determine that there is no difference between the methods. Thus, we expect that this interface better supports the developer during self-explanation.

I intend to build a software artifact to evaluate the effectiveness of such an interface. An open challenge is to develop a computational approach that can automatically aid the developer. One possible avenue is to adapt techniques by Erwig and Walkingshaw, who describe a textual notation for specifying the semantics of an explanation-producing program, coupled with a visual notation for presenting the explanation [8].

IV. Conclusion

My thesis postulates that failures in self-explanation result in failures in error notification comprehension. The results of this work will advance our understanding of how developers comprehend notifications and provide specific guidelines to tool designers for notification tasks.

Acknowledgment

This material is based upon work supported by the National Science Foundation under Grant No. 1217700. I thank Dr. Margaret Burnett and Dr. Scott Fleming for their extensive feedback. I also thank my advisor, Dr. Emerson Murphy-Hill, for his insights and support.

References

[1] T. Barik, K. Lubick, S. Christie, and E. Murphy-Hill, "How developers visualize compiler messages: A foundational approach to notification construction," in *VISSOFT '14*, Sep. 2014, pp. 87–96.

[2] T. Barik, "Improving error notification comprehension through visual overlays in IDEs," in *VL/HCC GC '14*, Jul. 2014, pp. 177–178.

[3] H. Seo, C. Sadowski, S. Elbaum, E. Aftandilian, and R. Bowdidge, "Programmers' build errors: A case study (at Google)," in *ICSE '14*, May 2014, pp. 724–734.

[4] A. Altadmri and N. C. Brown, "37 million compilations: Investigating novice programming mistakes in large-scale student data," in *SIGCSE '15*, Feb. 2015, pp. 522–527.

[5] C. H. Legare, "The contributions of explanation and exploration to children's scientific reasoning," *Child Development Perspectives*, vol. 8, no. 2, pp. 101–106, Jun. 2014.

[6] T. Green and M. Petre, "Usability analysis of visual programming environments: A Cognitive Dimensions framework," *Journal of Visual Languages & Computing*, vol. 7, no. 2, pp. 131–174, 1996.

[7] A. Bragdon, R. Zeleznik, S. P. Reiss, S. Karumuri, W. Cheung, J. Kaplan, C. Coleman, F. Adeputra, and J. J. LaViola, "Code Bubbles: A working set-based interface for code understanding and maintenance," in *CHI '10*, Apr. 2010, pp. 2503–2512.

[8] M. Erwig and E. Walkingshaw, "A visual language for explaining probabilistic reasoning," *Journal of Visual Languages & Computing*, vol. 24, no. 2, pp. 88–109, Apr. 2013.