How Developers Visualize Compiler Messages: A Foundational Approach to Notification Construction

Titus Barik, Kevin Lubick, Samuel Christie, and Emerson Murphy-Hill
Computer Science Department
North Carolina State University, USA
tbarik@ncsu.edu, kjlubick@ncsu.edu, schrist@ncsu.edu, emerson@csc.ncsu.edu

***Abstract*—Self-explanation is one cognitive strategy through which developers comprehend error notifications. Self-explanation, when left solely to developers, can result in a significant loss of productivity because humans are imperfect and bounded in their cognitive abilities. We argue that modern IDEs offer limited visual affordances for aiding developers with self-explanation, because compilers do not reveal their reasoning about the causes of errors to the developer.**

The contribution of our paper is a foundational set of visual annotations that aid developers in better comprehending error messages when compilers expose their internal reasoning. We demonstrate through a user study of 28 undergraduate Software Engineering students that our annotations align with the way in which developers self-explain error notifications. We show that these annotations allow developers to give significantly better self-explanations when compared against today's dominant visualization paradigm, and that better self-explanations yield better mental models of notifications.

The results of our work suggest that the diagrammatic techniques developers use to explain problems can serve as an effective foundation for how IDEs should visually communicate to developers.

I. Introduction

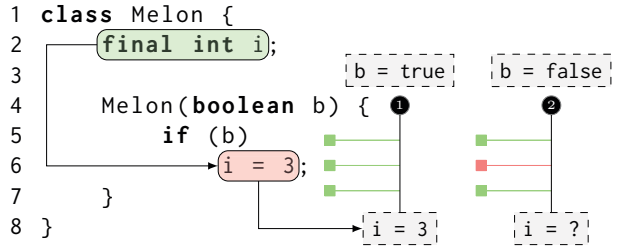
Modern integrated development environments (IDEs), such as Eclipse, IntelliJ, and Visual Studio, offer a number of visualizations to assist developers in more effectively identifying and comprehending compiler error notifications. For example, in addition to the full error message text found in a console output or dedicated error window, such notifications may include an indicator in one or more margins along with a red wavy underline visualization overlaid on the source text to indicate a relevant location of the error.

Many developers consider these error notifications to be cryptic and confusing [1]. We postulate one of the reasons error notifications are confusing is because compilers do not reveal the reasoning used to determine that the error exists.

More explicitly, in order to generate an error notification, the compiler begins with the source code, collects information during its compilation, uses that information to identify that a problem exists, and notifies the developer of the problem through the IDE. Yet, for developers to comprehend the notification, they must mentally duplicate this process through *self-explanation* [2] in essentially reverse order — starting with the error notification, the developer must identify what they think the problem might be from the IDE's presentation, mentally collect all of the program components related to this problem,

```
1    class Melon {
2        final int i;
3
4        Melon(boolean b) {
5            if (b)
6                i = 3;
7        }
8    }
```

(a) Baseline visualization



(b) Explanatory visualization

```
Melon.java:7: error:
   variable i might not have been initialized
    }
    ^
1 error
```

(c) Error message text

Fig. 1: A comparison of a potentially uninitialized variable compiler error through (a) baseline visualizations, the dominant paradigm as found in IDEs today, (b) our explanatory visualizations, and (c) the textual error message.

and finally identify the area or areas of source code necessary to correct the particular defect. This self-explanation process, when left solely to the developer, can result in a significant loss of productivity because humans are imperfect and bounded in knowledge, attention, and expertise [3]. *Much of this self-explanation process may be completely unnecessary since the reasoning process that resulted in the error notification was*

978-0-7695-5305-4/14 $31.00 © 2014 IEEE
DOI 10.1109/VISSOFT.2014.24

already known to the compiler.[1]

Visualizations in IDEs, such as red wavy underlines and margin indicators, take the perspective that compilers are *opaque* black boxes and thus by design are limited in their affordances for helping developers in comprehending error notifications. In this paper, we argue that developers stand to significantly benefit when compilers are made more *transparent* and expose their internal reasoning process to visualization systems. We argue that such systems can leverage these structures to generate expressive, *explanatory visualizations* that align with the way in which developers self-explain error notifications. Our contributions in this paper are:

- A foundational set of composable visual annotations that aid developers in better comprehending error messages.
- An *explanation task* evaluation, using a set of paper mockups, which demonstrates that our explanatory visualizations yield more correct self-explanations than the baseline visualizations used in IDEs today.
- A *recall task* evaluation, in which developers write programs in a minimalistic programming environment to intentionally generate compiler errors, which demonstrates that better self-explanations enable developers to construct better mental models of error notifications.

II. Motivating Example

Yoonki is an experienced C++ developer who has recently transitioned to a project that is being developed in the Java programming language. While programming, he encounters a wavy red underline visualization as shown in Figure 1a, which indicates an error. The problem seems to be related to `final int i`, which Yoonki recognizes as being roughly similar to the concept of a `const` variable in C++. Yoonki investigates further and notices the full text of the error in the bottom pane of his IDE (Figure 1c).

However, Yoonki is now a bit puzzled. The error message indicates the variable might not be initialized at Line 7. He decides this error message is incorrect and ignores it because Line 7 contains only a curly brace, which seems to have nothing to do with his problem. He is comfortable in doing so because in C++, he often received unhelpful notifications.

Yoonki explains to himself that the problem is that `final` variables in Java, like `const` variables in C++, must be assigned at their point of declaration, or in a constructor initializer list. Satisfied with his explanation, he rewrites Line 2 to read `final int i = 3;` but this immediately results in a downstream error, as Line 6 now displays `cannot assign a value to final variable i`. Yoonki realizes that a constant cannot be re-assigned, so he deletes the entire conditional statement. Even though the program now compiles, the fix happens to be an incorrect one.

The problem here is that Yoonki has learned a reasonable heuristic for how constant variables work in programming languages, but his heuristic fails in this case. Like C++, Yoonki is correct in that Java `final` variables can only be assigned once. But unlike C++, `final` variables in Java can be assigned at a point other than the declaration. Yoonki has experienced what we could call a *knowledge breakdown* [3]. In this case, Yoonki has a confirmation bias about how the system is supposed to work, and this false hypothesis has worked reasonably well for him until now.

This false hypothesis remains uncorrected by the IDE. In his IDE, the red wavy underline visualization can only indicate a single location related to the error. The IDE is unable to convey that the problem is dependent on several program elements. For example, the error text and the indicated location is accurate in that after this line the variable might be uninitialized, but the IDE does not have an effective way to indicate how that location relates to the `final` variable.

In contrast, consider our approach, shown in Figure 1b. Here, Yoonki may not experience the same knowledge breakdown, because the IDE provides a visual explanation of the problem within his source code. Though Yoonki might once again incorrectly assume `final` variables must be assigned at declaration, the visualization implies that the problem is actually related to control flow. Specifically, the explanatory visualization is showing Yoonki there is a code path in which `i` is assigned a value (when `b = true`), and another code path where it is not (when `b = false`). This time, Yoonki correctly fixes the defect by adding an `else` statement to the condition, initializing it with an appropriate value in the case when `b = false`.

This hypothetical scenario illustrates why the dominant visualization paradigm is not sufficient in supporting the process of self-explanation. As we argue in this paper, this scenario is illustrative of a more general problem with the output of program analysis tools: these tools present only the end-result of the complex reasoning process and therefore do not support the developer in self-explaining.

III. Pilot Study

We conducted a pilot study[2] from undergraduate lab sessions in Software Engineering to address a prerequisite research question:

RQ0 What annotations do developers use when they explain error messages to each other?

We hypothesized that if participants preferred certain types of annotations when explaining error messages to each other, they could also benefit when the same annotations were used to explain error messages to them through their IDE.

Thus, before generating our annotations, we conducted an informal lab activity with third-year Software Engineering students. Each student was given a sheet of paper with a source code listing and the corresponding compiler error message. The source code listings were unadorned and lacked any visual annotations.

[1] As Bret Victor points out his talk "Inventing on Principle" (CUSEC 2012): "If we're writing our code on a computer, why are we simulating what a computer would do in our head? Why doesn't the computer just do it, and show us?"

[2] All experimental study materials are available at http://go.barik.net/errviz.

TABLE I: Frequency of Visual Annotations in Pilot

Annotation	Frequency	Description
Point	49	A particular token or set of tokens has been marked. Examples include underlining or circles the token(s).
Text	45	Natural language text. For example, "assign a value to the variable" or "dead code".
Association	33	An association between two or more program elements, which is accomplished by drawing a connecting line between the elements, with or without arrow heads.
Symbol	20	Symbols include visual annotation such as ? or x, or numbered circles, to name a few.
Code	14	Explanatory code that is written in order to explain the error message, for example, `if (b == false)` or `m(1.0, 2)`. This does not have to be correct Java code, but should be interpretable as pseudocode.
Strikethrough	5	The strikethrough is separated from the point annotation because this annotation is provided by IDEs today, and has pre-established semantics.
Multicolor	-	The use of more than a single color to explain a concept. For example, green may be used to indicate lines that are okay, and red to indicate lines that are problematic. This option was not available to students in the pilot study.

Students were paired for an explainer-listener exercise. This is an exercise in which one student, the explainer, is asked to verbally explain the error message to the other student while visually annotating the source code listing during their explanation. Access to external materials was not allowed. After two minutes of explanation, roles were swapped and the second explainer annotated the second error message.

We randomly assigned one of four source code listings to each student, pulled verbatim from the OpenJDK 7 unit tests for compiler diagnostics framework. These examples, among others used in subsequent studies, are found in Table III. No students within a pair received the same source code listings. In total, we collected 73 samples: 17 from T1 (23%), 12 from T2 (16%), 20 from T3 (27%), and 24 from T6 (33%). Students did not receive tasks T4 or T5, because they had not been created at the time of the pilot study.

From these annotations we performed two passes over the student responses. In the first pass, we created a taxonomy of visual annotations based on our observations. In the second pass, we classified the student responses using this taxonomy. The aggregated results are shown in Table I.

The pilot study informed our explanatory visualizations, which we implement through annotations such as points, associations, symbols and explanatory code. Since students used these types of annotations without any *a priori* prompting, we postulate that they find these types of annotations intuitive to use during explanation.

TABLE II: Visual Annotation Legend

Symbol	Description
code	The starting location of the error.
code	Indicates issues related to the error.
	Arrows can be followed. They indicate the next relevant location to check.
❶	Enumerations are used to number items of potential interest, especially when the information doesn't fit within the source code.
?	The compiler expected an associated item, but cannot find it.
×	Conflict between items.
code	Explanatory code or code generated internally by the compiler. The code is not in the original source.
	Indicates code coverage. Green lines indicate successfully executed code. Red lines indicate failed or skipped lines.

IV. Explanatory Visualizations of Error Messages

We propose a set of eight visual annotations, which are summarized in Table II. We now concretely describe these annotations using the motivational example from Figure 1b. The starting *point* for visual explanation in the source code listing is indicated using code (a green rectangle with rounded corners that surrounds a program element). In our visualization mockups, we choose the starting point to be the same as the source of the error identified by IntelliJ (Figure 1a). In the example, this is `final int i`.

Continuing our example, the starting point is associated with a second point, `int i`, because this is where the potential assignment to the variable occurs. We indicate the starting point with code (red rectangle with rounded corners), and the association is indicated by (a directional arrow).

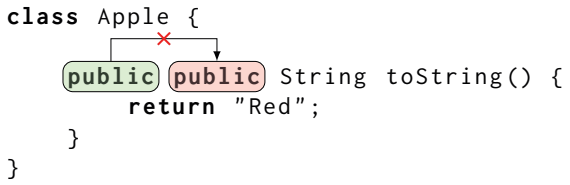
A second association leads the developer to an explanatory code block that contains a copy of the statement. Explanatory code is represented by code (dashed gray rectangle), which indicates the surrounded elements are explanatory and not part of the original source code of the program. This explanatory code block is part of a larger *composite annotation* describing the control flow scenario under which the statement is executed.

This composite annotation demonstrates that several basic annotations can be combined to create a new annotation for expressing a more complex concept. One of these components is the code coverage annotation. This annotation uses (green line) and (red line) to indicate whether or not a line is covered. In addition, the enumerations ❶ and ❷ provide the developer with convenient labels for referring to the branches (for example, "It looks like it works fine in branch 1, but not in branch 2"). The final component is another explanatory

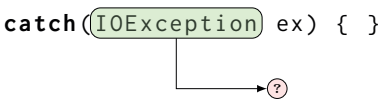
code block indicating one possible condition under which the branch would be executed.

Thus, the composite annotation indicates that `i = 3`, and all statements within branch 1 will be executed when `b = true`. This composite annotation is then used to show the developer a counterexample in which `i` would be uninitialized. A simple text explanation stating that `i is uninitialized when b = false` would have provided the same conclusion, but we hypothesize that the intermediate steps in the explanation are important for developer comprehension.

There are two visual annotations that do not appear in the motivating example that warrant explanation. These are × (red cross), which indicates that a conflict exists between blocks, such as when the developer accidentally specifies repeated modifiers:



Finally, the ? is used to indicate that the program element should be associated with another element, but that the connecting element is not found. This can occur when a `catch` statement is unreachable either because the exception can never be thrown, or because it is always caught by a prior `catch` clause:



V. Methodology

We conducted a second, formal study, which we discuss for the remainder of this paper.

A. Research Questions

We assigned participants randomly to two groups: a control group, having access to the baseline visualization (red wavy underline) in their source code, and a treatment group, having access to our explanatory visualizations. We designed our experiment to elicit answers for four research questions:

- **RQ1** Do explanatory visualizations result in more correct self-explanations by developers?
- **RQ2** Do developers adopt conventions from our visual annotations in their own self-explanations?
- **RQ3** What aspects differentiate explanatory visualizations from baseline visualizations?
- **RQ4** Do better self-explanations enable developers to construct better mental models of error notifications?

Unlike the baseline visualization, explanatory visualizations are intended to expose the reasoning process of the compiler.

For RQ1, we hypothesized that exposing this reasoning process would result in significantly more correct explanations by developers. If this hypothesis was not supported, it would imply that the explanatory visualizations might confuse developers and differ from the way they model error messages.

For RQ2, we hypothesized that both the control group and treatment group would adopt similar annotations when developers explained error messages, because our visualizations are based on conventions that developers would find intuitive for self-explanation.

For RQ3, we wanted to identify the traits of the explanatory visualizations beneficial to developers in comprehending error notifications. Significant differences in traits between the baseline visualization and explanatory visualizations would give us insight into the design of explanatory visualizations in general.

For RQ4, we hypothesized that better explanations result in better mental models, and that developers with explanatory visualizations would tend to have better mental models than the control group.

B. Participants

We recruited 28 participants ($n = 28$) from a third-year undergraduate course in Software Engineering because they were readily available and because we wanted to reserve our more limited industry participants for a full implementation. We offered participants extra credit on their final exam for participating in the study. Participants self-reported demographic data. 23 of the participants were male (82%), and five of the participants were female (18%). The mean age of the participants was 22 ($s = 3.6$). Participants reported a mean of 9 months ($s = 12$) of industry programmer experience.

26 participants reported using the Eclipse IDE as their primary Java programming environment; two participants reported IntelliJ. On a 4-point Likert-type item scale of *Novice—Expert*, 13 participants reported their overall programming ability as Intermediate (46%), 14 as Advanced (50%), and 1 as Expert (4%). No participants ranked themselves as Novice. On a 4-point scale *Not knowledgeable—Very knowledgeable*, 19 participants indicated they were knowledgeable about Java (68%), and the remaining 9 participants indicated they were very knowledgeable about Java (32%).

C. Selection Criteria for Mockups

Because our university requires students to have knowledge of Java, we selected examples in this language to mockup our visualizations.

Pragmatically, we wanted to keep the entire study under an hour, so we could only present six novel visualizations to participants. We admit that the selection of these visualizations was not random, and offer our justification for this decision here.

We selected our compiler error examples from the OpenJDK diagnostics framework.[3] This framework contains a collection of 382 Java code examples, each of which is designed to generate one or more error messages when compiled.

[3]The framework contains a sample source code listing for almost every compiler error within Java. The source files may be downloaded at http://hg.openjdk.java.net/jdk7/tl/langtools/, and then by browsing to `test/tools/javac/diags/examples/`.

TABLE III: Participant Explanation and Recall Tasks

Task Order	Task Name	OpenJDK File	Error Message
T1	Melon	`VarMightNotHaveBeenInitialized.java`	`variable i might not have been initialized`
T2	Kite	`UnreportedExceptionDefaultConstructor.java`	`unreported exception Exception in default constructor`
T3	Brick	`RefAmbiguous.java`	`reference to m is ambiguous, both method m(int,double) in Brick and method m(double,int) in Brick match`

T4	Zebra	`InferredDoNotConformToBounds.java`	`cannot infer type arguments for BlackStripe<>; reason: inferred type does not conform to declared bound(s)`
			
			`inferred: String`
			`bound(s): Number`

T5	Apple	`RepeatedModifier.java`	`repeated modifier`
T6	Trumpet	`UnreachableCatch1.java`	`unreachable catch clause thrown types FileNotFoundException,EOFException have already been caught`

Since some error messages may be more conceptually sophisticated than others (for example, "illegal escape character" is not particularly suited to an explanatory visualization), we hand-selected a set of examples that we believed could benefit most from visual annotations. If no significant results could be identified even from this hand-selected set, then it would suggest that this visualization system is not worth pursuing for a full implementation.

Furthermore, our visualization system is not intended to teach new concepts; rather, it is intended to aid the developer in understanding how a particular instance of an error message applies to a specific source file. Consequently, we selected examples based on concepts that students were expected to already know from their coursework, such as constants and variables, exceptions, and classes.

Ultimately, we selected messages that we believed could effectively demonstrate the rich explanatory potential of visualizations, while considering the capability of the participants. The selected messages are summarized in Table III.

D. Mockup Construction Procedure

Using the six selected error messages, we constructed a total of 12 mockups — six for the control group and six for the treatment group. We designed the paper mockups to resemble how the visualization would appear within the text editor of the IDE, with one mockup per page. Each page contained a listing of the source code with the appropriate visualizations and line numbers. The code listing was followed by the text of the compiler error message.

The control group mockups were designed by directly copying the red wavy underline visualizations provided by the IntelliJ IDE for the Java code examples. IntelliJ also provides interactive tooltips for each error, which are shown when the developer hovers over an annotated substring. However, we did not consider these interactive features since we are specifically interested in the contribution of the explanatory capability of the non-interactive visualizations. We chose IntelliJ over the Eclipse IDE because it uses the same text error messages as the command-line OpenJDK compiler, which is important to our experimental design.

The treatment group mockups were informed by a pilot study through which we elicited an initial taxonomy of visual annotations that appeared to be useful to developers when they explained concepts to other developers (see Section III). We used the annotations from this pilot experiment as a foundation for manually drawing visual annotations for six of the error messages. We used our own experiences with compiler technologies such as Roslyn[4] to render visualizations that we think are plausible for compilers to render if they expose the appropriate data structures to a visualization system.

E. Investigator Training

The first and second authors conducted the experiments. To increase consistency between the authors, the first author conducted a practice session with the second author acting as a participant. The roles were then reversed, and the study was repeated. Through this process, we developed a formal protocol script for conducting the sessions.

F. Experimental Procedure

1) Assignment: We randomly assigned participants to one of two groups — control or treatment, such that each group had an equal number of participants. This resulted in 14 participants per group. The only difference between the treatment and control groups was the type of visualizations that they used during the experiment.

2) Recording: Participants filled out an informed consent form and indicated whether or not they wanted their audio (used in Phase 1 and 2) and screens (used in Phase 2) to be recorded. For participants who agreed to be recorded ($n = 26$),

[4]http://msdn.microsoft.com/en-us/library/roslyn.aspx

we used desktop recorder software to record both the audio of the explanations as well as screen interactions during the experiment.

3) Phase 1: Self-Explanation Phase: The purpose of this phase was to evaluate whether our explanatory visualizations resulted in more correct self-explanations by developers than with baseline visualizations (RQ1), and to identify the extent to which developers adopt conventions from our visual annotations in their own explanations (RQ2).

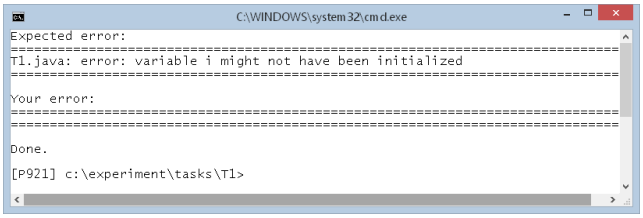
We sequentially provided participants with six error notifications, presented as paper mockups that resembled an IDE. For the mockup, the source code of the OpenJDK file was minimally modified using a random-noun generator to make the class and method names more pronounceable. These tasks are summarized in Table III, and we presented the tasks to the participants alphabetically by Task Name.

In the control group, participants received paper mockups. containing the baseline red wavy underline visualization, such as in Figure 1a. The treatment group received paper mockups containing our explanatory visualization as in Figure 1b. Below the source code listing, all participants received the full error message text (Figure 1c). In the treatment group, we provided participants with a visual annotation legend (Table II), since these participants did not have prior familiarity with our visualizations. Finally, we provided participants with colored pencils and an unadorned mockup of the IDE having the source code and error message text, but no annotations.

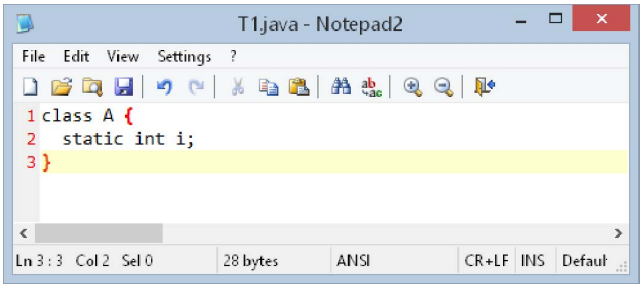
For each task, we provided participants with 30 seconds to individually examine the paper mockup. Then, we instructed participants to think-aloud and verbally explain the cause of the error. During their self-explanation, we encouraged participants to visually annotate the unadorned mockup. We gave participants two minutes for the think-aloud explanation, but allowed them to finish earlier if they were satisfied with their explanation for the task. The investigators were not allowed to correct the participants when they gave incorrect explanations, nor give any hints about the error notification. However, we permitted the investigators to ask clarifying questions (e.g., "Could you explain that in more detail?" or "I didn't hear you. Could you repeat that?"). At the end of each explanation, participants indicated whether or not they had previously encountered this error message, which they categorized as Yes, No, or Unsure.

4) Cognitive Dimensions Survey: To evaluate the aspects of visualizations that developers find useful in self-explanation (RQ3), participants completed a Cognitive Dimensions of Notations questionnaire (CD) [4], which we simplified for error message notifications. We chose this evaluation instrument over other usability instruments because the analysis is usable by non-specialists in HCI (in contrast with Nielson and Molich's heuristic evaluation [5]). The instrument is also quick to apply, and can be used in an early design phase.

The full CD defines 14 dimensions, but not all of these are applicable to our design. Since our visualizations are currently non-interactive, we eliminated all dimensions that assessed interactivity or were otherwise immaterial to our



(a) Command prompt.



(b) Minimal text editor.

Fig. 2: We presented participants with a command prompt in which they had the `compile` command available to them. The limited interaction modality forces participants to rely solely on their own memory to successfully complete the task.

study, among them, viscosity, premature commitment, and progressive evaluation. This left four dimensions:

Consistency
: similar semantics are expressed in similar syntactic forms

Hidden dependencies
: important links between entities are not visible

Hard mental operations
: high demand on cognitive resources

Role expressiveness
: the purpose of a component is readily inferred

A description of each dimension was presented to the participants, along with a 5-point interval scale indicating the degree to which their visualizations satisfied the dimension, which we worded so that higher scores are better. We gave participants 5 minutes to complete the questionnaire.

5) Break: We gave participants a 5 minute break between the first and second phases. We did this partly because of the long duration of the study, but also to minimize short-term memory interference between the two parts of the experiment.

6) Phase 2: Recall Phase: The purpose of this phase was to determine whether better self-explanations enable developers to construct better mental models of error notifications (RQ4). To evaluate this hypothesis, we asked participants to write source code listings on a computer from scratch in order to *generate* a provided compiler error.

Participants did so through the interface shown in Figure 2. We gave the participants a command prompt (Figure 2a) supporting a single command, `compile`. This command printed to the console the expected error for the task, as well as the error

that their source file produced. In addition, participants entered their source code into a minimal text editor (Figure 2b). We chose a minimal text editor to force all participants to recall code entirely from memory, without assistive features like auto-completion. For example, in Figure 2, the participant has been asked to write a source listing that generates the error `variable i might not have been initialized`. However, the source listing as currently written compiles without error.

Participants used this interface to complete a total of six tasks, all of which they had *previously explained* during the Self-Explanation Phase of the experiment. The tasks from this phase are also from Table III, but to avoid serial recall we presented the tasks in Task Order, rather than alphabetically by Task Name. Thus, participants had to successfully *recall* their explanations from the Self-Explanation Phase of the experiment and apply their understanding to this phase of the experiment. We allowed participants an unlimited number of compilation attempts, but restricted the time for each task to 5 minutes. Participants moved on to the next task either when they had successfully replicated the error message, which we term *recall correctness*, or when their time had expired.

The unusual experimental technique in this phase is not without theoretical justification. In 1977, Shneiderman conducted an experiment in which he used memorization/recall tasks as a basis for judging programmer comprehension [6]. Specifically, one component of his experiment involved non-programmers and programmers memorizing a proper `FORTRAN` program printed on paper through a line printer. He also printed a second program with shuffled lines. He found that non-programmers had similar performance in recall with both the proper and shuffled versions of the program, but that programmers had significantly better recall on the proper version of the program. Through the development of his cognitive syntantic/semantic model, he suggests that "performance on a recall task would be a good measure of program comprehension" because such a task cannot be accomplished by rote memorization, and instead requires "recognizing meaningful program structures enabling them to recode the syntax of the program into a higher level internal semantic structure" [6].

Thus, participants had to construct a correct mental model of the error notification through self-explanation in order to successfully complete the task in this phase of the experiment.

VI. Results

A. RQ1: Visualizations Lead to More Correct Explanations

Our hypothesis was that having visual explanations for compiler notifications would result in more correct self-explanations by participants. To validate this hypothesis, we conducted an inter-rater reliability exercise in which the first and second authors independently rated the participants' explanations, without consideration of group. The first author assigned ratings using both the recorded verbal explanations of the participant as well as their paper markings. The second author assigned ratings using only the paper markings. This was a deliberate design decision to ascertain the extent to which visual markings alone can be used to infer the correctness of an explanation.

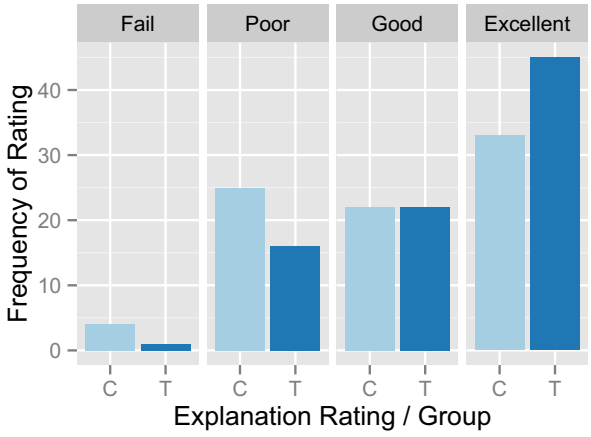


Fig. 3: Explanation rating by group. The treatment group (T) provided significantly higher rated explanations than the control group (C).

We assigned ratings to each of the 168 tasks on a Likert-type scale from 1–4, labeled Fail, Poor, Good, and Excellent, respectively. For each task, we developed a rubric for what constituted a correct explanation and noted common misconceptions. Cohen's Kappa (squared weights), found moderate agreement between the raters ($n = 168$, $\kappa = 0.58$, 95% CI: $[0.46, 0.68]$). Furthermore, a paired Wilcoxon Signed-Rank Test did not identify the differences between the two raters as being significant ($n_1 = n_2 = 168, S = 200, p = .21$), Thus, the data suggest that visual annotations capture the correctness of the full explanation adequately. No attempts were made to reconcile disagreement. In subsequent analysis, we use the explanation ratings from the rater using both verbal and written explanations. Because this rater had access to more information from which to assign a rating, these ratings are likely to be more accurate than ratings assigned from written markings alone.

The distribution between the two groups, binned by rating, is shown in Figure 3. Between the control and treatment groups, a Wilcoxon Rank-Sum Test confirms that participants gave significantly better explanations in the treatment group ($n_1 = n_2 = 84, Z = 2.23, p = .026$). A potential confound is that participants are simply providing better explanations in the treatment group because more of them had previously encountered the error messages, but a Pearson Chi-squared Test did not identify a significant difference between the groups ($n = 168$, $df = 2$, $\chi^2 = 3.37$, $p = .19$).

B. RQ2: Availability of Explanatory Visual Annotations Promotes More Frequent Use of Annotations During Self-Explanation

Our hypothesis was that both the control group and treatment group would adopt similar annotations when developers

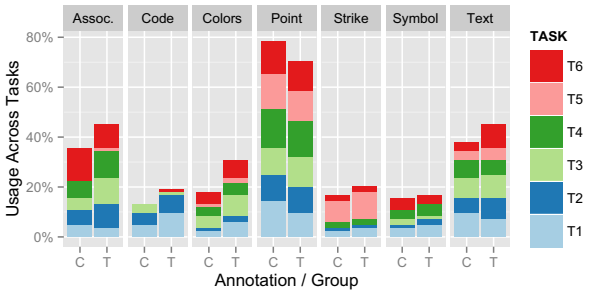


Fig. 4: Annotations by group, filled with usage across tasks. The distribution of annotations used by the control (C) and treatment groups (T) were not identified as being significantly different, but the treatment group used annotations significantly more often.

TABLE IV: Number of Features by Task and Group

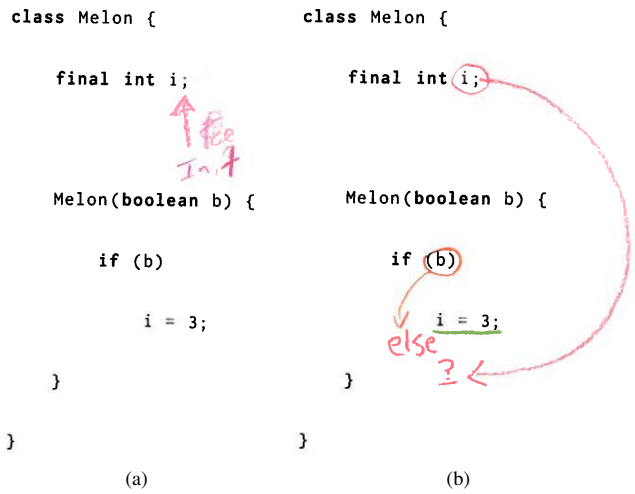
	Number of Features			
	Control		Treatment	
Task	Median	Dist	Median	Dist
T1	2		3	
T2	2		2	
T3	2		2	
T4	2		3	
T5	1		2	
T6	3		3	

explained error messages, if these annotations were grounded in conventions that developers found intuitive.

Consider for a moment the visualizations drawn by two participants in our study, shown in Figure 5. In Figure 5a, the control group participant receives a score of Fail, because he incorrectly self-explains that the problem must be due to not initializing the variable at its point of declaration. He then either ignores the conditional statement in which the constant value is re-assigned, or fails to notice that the variable is declared as `final`. In Figure 5b, the participant, aided by the explanatory visualization, correctly self-explains that the problem is actually in the conditional statement, and provides explanatory code to demonstrate a case in which the variable remains uninitialized. In addition, the treatment participant uses more annotations, such as colors, points, and associations, in his explanation than the control group participant.

Table IV summarizes the number of annotation types used for each task, partitioned into control and treatment groups. Using a Wilcoxon Rank-Sum Test, we find that the treatment group used significantly more visual annotation types in their explanations than the control group ($n_1 = n_2 = 84$, $Z = 2.15$, $p = .032$).

One concern is that participants in the treatment group used these annotations simply because they were readily *available*, not because they were *useful* to their explanations. Figure 4 shows the distribution of the annotations by group. The bars are filled with the usage of that annotation by task to



(a) (b)

Fig. 5: A contrast between visual explanations offered by (a) control group participant with explanation rating of Fail, and (b) treatment group participant with explanation rating of Excellent.

TABLE V: Cognitive Dimensions Questionnaire Responses

	Control		Treatment		
Dimension	Median	Dist	Median	Dist	p
Hidden Dependencies*	3		4		.008
Consistency	4		4		.979
Hard Mental Operations	3		2.5		.821
Role Expressiveness	4		4		.130

indicate how a particular annotation is distributed among the tasks. A Pearson Chi-squared Test was unable to identify any significant differences in the *distribution* of these annotation types between groups ($n = 389$, $\chi^2 = 4.20$, $df = 5$, $p = .65$), suggesting that these annotations are intuitive even without priming the participants. Although Figure 4 shows that the control group used the point annotation more than the treatment group, this difference was not found to be significant ($n = 168$, $df = 1$, $\chi^2 = 1.53$, $p = .22$).

In addition, none of the participants in the treatment group used our invented code coverage annotation, nor did this annotation appear directly in our pilot study. This suggests that participants are using these annotations only when they find them to be useful in self-explanation.

Thus, participants in both groups used and applied the annotations found in our explanatory visualizations, despite the fact that we did not expose the control group to our visualizations. This indicates that these annotations are intuitive and useful for participants. Moreover, the presence of explanatory visualizations promotes their usage during self-explanation by participants.

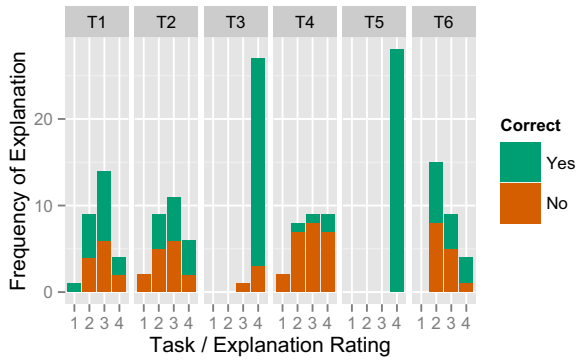


Fig. 6: Task by explanation rating. Each of the six tasks are broken by explanation rating (1 = Fail, 2 = Poor, 3 = Good, 4 = Excellent) from the first phase of the experiment. For each explanation rating, the frequency of correct and incorrect recall tasks from the second phase of the experiment is indicated by filling in the bars. Higher rated explanations lead to significantly better recall correctness.

C. RQ3: Explanatory Visualizations Reveal Hidden Dependencies

We wanted to know which factors participants considered to be significant improvements over the baseline visualization. We had no explicit hypothesis for this research question.

Table V summarizes the results from our Cognitive Dimensions questionnaire. Median results for the hidden dimensions for control and treatment groups were 3 and 4, respectively. The distribution of responses in the two groups were significantly different ($n_1 = n_2 = 14$, $Z = -2.64$, $p = .008$). The result suggests that our explanatory visualizations reveal more of the hidden dependencies, that is, the internal reasoning process of the compiler, than the baseline visualizations.

We were unable to identify any statistically significant differences from the remaining dimensions in the questionnaire.

D. RQ4: Higher Rated Explanations Lead to Better Mental Models, and Better Recall Correctness

Figure 6 illustrates the explanation rating for each task, the frequency of the explanation for each rating within the task, and the recall correctness. Remember from Section VI-A that the treatment group had higher explanation ratings than the control group. Our expectation was that these higher rated explanations would translate to better correctness scores during the recall phase of the experiment.

A Kruskal-Wallis Test revealed a significant difference between performance on explanation correctness and performance on recall correctness ($\chi^2 = 29.39$, $df = 3$, $p < .001$), and the mean ranks indicate that recall correctness generally increases with explanation correctness ($u_1 = 51.8$, $u_2 = 69.8$, $u_3 = 69.3$, $u_4 = 102.8$). This confirms that explanation is valuable for improving correctness in the recall task, but two potentially problematic issues arise.

In Figure 6, we observe that task T5 (repeated modifier) has both perfect recall correctness and uniformly excellent explanation rating, which we postulate is attributable to this being trivial problem. Our first concern is that this task is artificially inflating the influence of the explanation correctness to recall correctness. As a contradictory example, we visually identify that T4 (cannot infer type arguments) has some participants who have poor performance during recall despite excellent explanation correctness. We found that even without T5, the difference is still significant ($\chi^2 = 12.33$, $df = 3$, $p = 0.006$), and the general trend remains ($u_1 = 49.0$, $u_2 = 64.0$, $u_3 = 63.6$, $u_4 = 84.0$).

However, a second issue remains — if the treatment group gives higher rated explanations, then we would expect that they have greater correctness in recall. Unfortunately, we were unable to identify this as being significant ($n_1 = n_2 = 84$, $Z = 1.09$, $p = 0.27$).

We conclude that better explanations yield improved recall correctness, though with some reservations.

VII. Threats to Validity

In real code bases, developers have to explain error messages in functional code intertwined with erroneous code, and across multiple source files. Our tasks contained only the code directly pertinent to generating the error, and within a single source file. We don't yet know if explanatory visualizations will be equally beneficial or scale to more realistic contexts.

We applied a set of visualizations to only six hand-selected tasks that could fit on a single screen. As such, it remains to be seen whether visual annotations can be effectively applied to the broader set of error messages, including those in languages other than Java. Thus, we cannot and do not claim that these annotations are comprehensive.

We think there exists a construct validity problem in that explanation ratings were significantly better in the treatment group, but this performance did not translate to better recall correctness. We postulate that this situation occurred because it was possible for developers to successfully explain the task, yet still have gaps in their mental model that prevent them from successfully completing the task. In addition, we observed that some participants had significant difficulties with syntax, and in some cases even introduced secondary compiler errors not related to the recall task during the process.

Furthermore, the act of performing a think-aloud can enhance self-explanation, and in turn, the construction of mental models for notifications. This process was necessary in order to evaluate participant explanations, but in doing so, we may have unintentionally enhanced the performance of the control group in their recall tasks. Another issue is that participants were already familiar with the baseline visualizations, but had no prior experience or any training with our explanatory visualizations. This may explain why we found no statistical difference in hard mental operations: the potential cognitive benefit of our visual annotations was counterbalanced by the difficulty of understanding an unfamiliar visualization.

VIII. Related Work

Self-explanation. Lim and colleagues demonstrate that explaining why a system behaves a certain way results in better understanding and stronger feelings of trust [7]. We were also inspired by the work of Ainsworth and Th Loizou, who showed that the use of diagrams promote the self-explanation effect significantly more than text [8].

Improving error notification comprehension. Jeffrey created a tool called *Merr* that overrides the error handler of the LR parser generator of a compiler to automatically provide more useful syntax error messages [9], and Kantorowitz and Laor likewise propose modifications to the parser generator [10]. While these tools apply to text error messages, they illustrate that tools can improve error messages when they can interact with compiler internals. However, even detailed messages do not necessarily improve understanding, which suggests that alternative representations of error notifications may sometimes be more appropriate [11], [12].

Hartmann and colleagues introduce a social recommendation system that presents examples of how other developers understand and correct errors [13]. In contrast, our approach argues that the compiler can offer its own reasoning process to aid developer comprehension. Other approaches attempt to provide better diagnostics or reduce false positives in compiler errors [14], [15], [16]. We expect that our visualizations can leverage such improvements in compiler technology.

IX. Future Work

We suggest several potential research directions. One direction is the feasibility challenge of developing algorithms and techniques for recording compiler analysis traces so they can be exposed to visualization systems. We know compilers generate a significant amount of information during the compilation process, but it remains an open question as to what information is pertinent to aiding developer comprehension, and how to represent this information in a way usable by visualization systems. One approach to demonstrate this feasibility may be to modify an implementation such as MiniJava, a useful but restricted subset of the Java language [17].

An empirical direction is to determine the extent to which visualizations can be applied to notifications, given that some annotations appear to be more suitable than others. A systematic investigation into categorizing these notifications, such as through taxonomy construction, may offer researchers insights into this design space.

X. Conclusion

Our work in this paper demonstrates the potential for facilitating developer self-explanations when opaque compiler reasoning processes are made available for visualization. Through error notifications, we demonstrated that when such visualizations align with developer expectations, developers better comprehend error notifications, use these visualizations more often in their own self-explanations, and construct better mental models of error notifications. We think the diagrammatic techniques developers use to explain problems to other developers and to themselves can serve as an effective foundation for how IDEs should visually communicate to developers.

Acknowledgments

This material is based upon work supported by the National Science Foundation under Grant No. 1217700. We thank the Software Engineering group at ABB Corporate Research for their funding and support.

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] C. Parnin, "Subvocalization - Toward hearing the inner thoughts of developers," in *ICPC '11*, Jun. 2011, pp. 197–200.

[3] A. J. Ko and B. A. Myers, "A framework and methodology for studying the causes of software errors in programming systems," *Journal of Visual Languages & Computing*, vol. 16, no. 1, pp. 41–84, 2005.

[4] 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.

[5] J. Nielsen and R. Molich, "Heuristic evaluation of user interfaces," ser. CHI '90, 1990, pp. 249–256.

[6] B. Shneiderman, "Measuring computer program quality and comprehension," *International Journal of Man-Machine Studies*, vol. 9, no. 4, pp. 465–478, 1977.

[7] B. Y. Lim, A. K. Dey, and D. Avrahami, "Why and why not explanations improve the intelligibility of context-aware intelligent systems," in *CHI '09*, Apr. 2009, pp. 2119–2129.

[8] S. Ainsworth and A. T. Loizou, "The effects of self-explaining when learning with text or diagrams," *Cognitive Science*, vol. 27, no. 4, pp. 669–681, Aug. 2003.

[9] C. L. Jeffery, "Generating LR syntax error messages from examples," *ACM Transactions on Programming Languages and Systems*, vol. 25, no. 5, pp. 631–640, Sep. 2003.

[10] E. Kantorowitz and H. Laor, "Automatic generation of useful syntax error messages," *Software: Practice and Experience*, vol. 16, no. 7, pp. 627–640, Jul. 1986.

[11] M.-H. Nienaltowski, M. Pedroni, and B. Meyer, "Compiler error messages: What can help novices?" in *SIGCSE '08*, Feb. 2008, pp. 168–172.

[12] P. Denny, A. Luxton-Reilly, and D. Carpenter, "Enhancing syntax error messages appears ineffectual," in *ITiCSE '14*, Jun. 2014, pp. 273–278.

[13] B. Hartmann, D. MacDougall, J. Brandt, and S. R. Klemmer, "What would other programmers do," in *CHI '10*, Apr. 2010, pp. 1019–1028.

[14] N. Boustani and J. Hage, "Improving type error messages for generic Java," *Higher-Order and Symbolic Computation*, vol. 24, no. 1-2, pp. 3–39, Jun. 2011.

[15] J. C. Campbell, A. Hindle, and J. N. Amaral, "Syntax errors just aren't natural: Improving error reporting with language models," in *MSR '14*, May 2014, pp. 252–261.

[16] S. Chen, M. Erwig, and K. Smeltzer, "Let's hear both sides: On combining type-error reporting tools," in *VL/HCC '14*, Jul. 2014, pp. 145–152.

[17] E. Roberts, "An overview of MiniJava," in *SIGCSE '01*, vol. 33, no. 1, Mar. 2001, pp. 1–5.