How should static analysis tools explain anomalies to developers?

A Communication Theory of Computationally Supporting Developer Self-Explanations for Static Analysis Anomalies

Titus Barik*

North Carolina State University
http://go.barik.net/proposal
tbarik@ncsu.edu

Abstract Despite the advanced static analysis tools available within modern integrated development environments (IDEs) for detecting anomalies, the error messages these tools produce to describe these anomalies remain perplexing for developers to comprehend. This thesis postulates that tools can computationally expose their internal reasoning processes to generate assistive *error explanations* in a way that *approximates* how developers explain errors to other developers and to themselves. Compared with baseline error messages, these error explanations significantly enhance developers' comprehension of the underlying static analysis anomaly. The contributions of this dissertation are: 1) a *theoretical framework* that formalizes explanation theory in the context of static analysis anomalies, 2) a *set of experiments* that evaluate the extent to which evidence supports the theoretical framework, and 3) a *proof-of-concept IDE extension*, called Radiance, that applies my identified explanation-based design principles and operationalizes these principles into a usable artifact. My work demonstrates that tools stand to significantly benefit if they incorporate explanation principles in their design.

1 Prelude

Dave: *Hello, HAL. Do you read me, HAL?*
HAL: *Affirmative, Dave. I read you.*
Dave: *Open the pod bay doors, HAL.*
HAL: *I'm sorry, Dave. I'm afraid I can't do that.*
Dave: *What's the problem?*
HAL: *I think you know what the problem is just as well as I do.*

— Scene from *2001: A Space Odyssey*

* Thesis proposal submitted to the graduate faculty of North Carolina State University in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

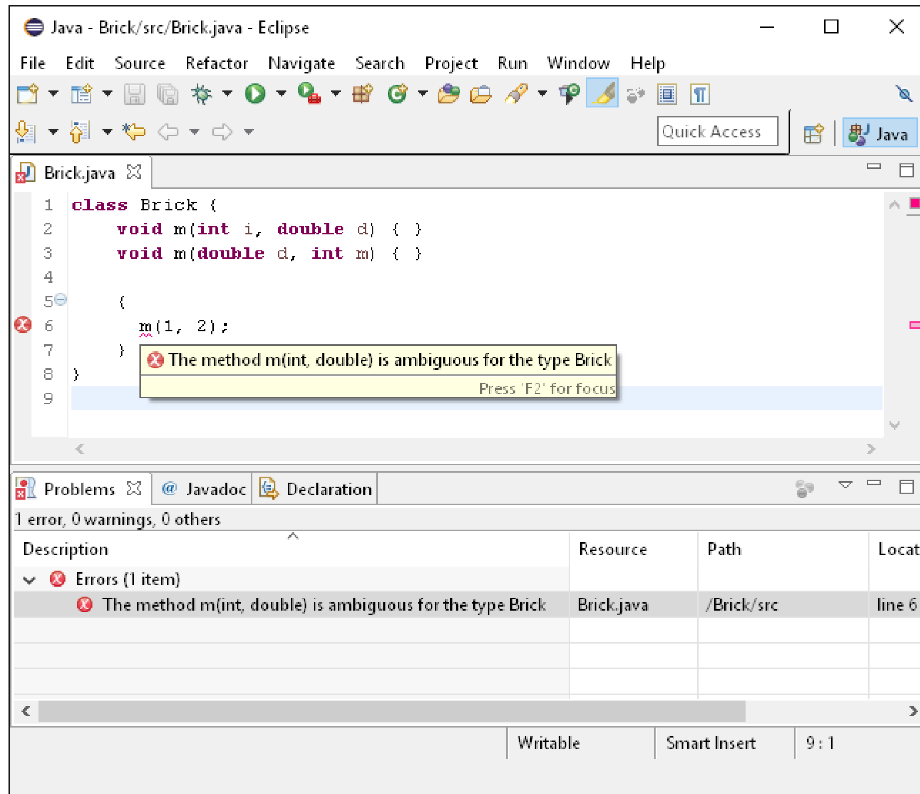

Figure 1. An ambiguous method error within the Eclipse IDE. Understanding this error requires recognizing that Line 2 and Line 3 are also related to the anomaly. The error description text fails to provide a rationale for why the method is ambiguous.

2 Motivating Example

Kevin is a professional software engineer at Google who mostly programs in Java within the Eclipse IDE. While working on his code base (Figure 1), he adds a method `m(double d, int i)` to a source file in the program (Line 3).[1] When he calls the method (Line 6), he is surprised that the compiler's static analysis has sprinkled an error into his IDE. Kevin thinks, "How can the method be ambiguous? And ambiguous to what? I haven't even had a chance to add any functionality to the method."

Fortunately, Kevin has learned through years of experience some clever programmatic tricks that allow him to nudge the otherwise terse IDE to be more forthcoming. One of these tricks is to comment out pieces of code to force the

[1] For presentation purposes, I've condensed the source file so that the essential program elements fit in a single screen capture. In Kevin's actual program, there are many interspersed program elements.

compiler to indirectly spit out its internal reasoning. Thus, he decides to comment out the method declaration that he just added (Line 3). He explains to himself, "if the ambiguous method anomaly disappears, this would mean that the IDE analysis knows about some other method definition in the code that satisfies the call." The error message is indeed removed, and Kevin is satisfied with this explanation. He uses the `Open Declaration` feature in the IDE to quickly find the conflicting method (Line 2), and undoes his code back to the error-generating state.

Kevin now understands what program elements are contributing to the problem, although he's frustrated that the IDE made him jump through many hoops to find the elements. "Why didn't you tell me about these program locations, which are clearly relevant to the problem?" Kevin mutters to his unrelenting IDE.

Unfortunately, Kevin still doesn't have any clarity into why the IDE thinks this is a problem. He renames his method `m` (Line 6) to `m_test`, thinking that the ambiguity might be due to the name of the method.Again, the error message goes away, but Kevin isn't satisfied with his explanation. "This would make sense if it were a language like Go, which doesn't have method overloading, but I'm sure that Java does." He dismisses this explanation as being not all that plausible.

He thinks up yet another explanation just to check his sanity, even though he doesn't find this explanation to be all that plausible either. "Perhaps the problem is because the arguments to the method are constants." He's seen problems like this happen in C++ due to pass-by-reference semantics, so he quickly changes the call to:

```
double d = 1;
int m = 2;
m(d, n);
```

The IDE is now silenced. However, the victory is a hollow one as Kevin feels sheepish about submitting these program additions for a code review without understanding what's actually causing the anomaly in the first place. His past experiences have taught him that acquiescing the IDE doesn't always address the actual anomaly, but instead punts the problem somewhere down the line. As yet another explanation, he wonders if the anomaly could have something to do with the types of the arguments, but can't quite put his finger on it.

Kevin asks himself, "Can someone explain this to me?" and looks around the room inquisitively. But he's the only one working this Saturday morning. Perplexed and frustrated, Kevin thinks it would have been nice if that "someone" could be the IDE.

3 What went wrong?

Automatic type promotion is the missing piece that Kevin would have needed to understand the error, and it is in fact a test the compiler itself has to perform to in order to generate the error.

As explained in the Java Language Specification (JLS), §15.12.2: Compile-Time Step 2: Determining Method Signature[2], the error is essentially the result of automatic type promotion, ultimately yielding a scenario where "it is possible that no method is the most specific, because there are two or more methods that are maximally specific . . . the method invocation is ambiguous, and a compile-time error occurs." A battery of tests, such as identify matching *arity* methods applicable by strict invocation (§15.12.2.2), that is, matching and promoting actual types to formal types, can be used to to determine applicable methods for specificity.

Basically what's happening is that the first argument is an `int`, but it can also be promoted to `double`. The same is true of the second argument. Depending on which of the two arguments we choose to promote, either the method on Line 2 or the method on Line 3 would apply. Since the compiler has no built-in rule indicating which is preferable, it cannot automatically resolve this conflict.

Because the compiler didn't give a complete explanation to Kevin, he was forced to produce multiple plausible, yet incorrect, explanations himself to identify the cause of the anomaly.

It turns out that Kevin's explanation difficulties wouldn't be unique to the Eclipse IDE. For this particular anomaly, I generated conceptually similar anomalies within other compilers, and for multiple languages, both at the console and in the IDE. In C#, the Visual Studio IDE and console output indicate that there are two possible method definitions in the code that cause the ambiguity, but fail to indicate to the developers where those definitions are. The same is true of the Oracle Java compiler, in the console and through the IntelliJ IDE. C++, for GCC and LLVM do a bit better here: they indicate to the developer the method definitions and the actual positions in the code where the ambiguity occurs. However, none of the tools we found explained the reason for the ambiguity, a result of automatic type promotion, to the developer.

4 Objectives and Significance

Modern software development typically occurs within an integrated development environment (IDE), such as Eclipse, Visual Studio, and IntelliJ.[3] One task developers perform within this IDE is understanding anomalies that static analysis tools identify within the environment. These anomalies are presented to the developer through the IDE as *error messages*. In modern IDEs, the error message payload consists of two components. First, it consists of a textual error description of the problem in a list box or console panel as we would typically associate with the concept of message. Second, the payload consists of visual indicators, such as the red wavy underline, that indicate the "primary" location of the error, and other icons or markers in the gutter or margin of the editor.

Despite the sophisticated reasoning processes available to static analysis tools (e.g., abstract interpretation) in identifying an anomaly, developers continue to

[2] `http://docs.oracle.com/javase/specs/jls/se8/html/jls-15.html#jls-15.12.2.2`
[3] `http://pypl.github.io/IDE.html`

have difficulty understanding the messages they produce [Bro83,KK03,APM+07]. Consequently, static analysis anomalies remain perplexing for developers to resolve [Tra10].

In this proposal, I argue that static analysis error messages should be reframed as assistive *error explanations*, in which an explanation is a "set of assumptions which, together with background knowledge, logically entails a set of observations" [NM90]. That is, let us assert that an error message is in fact an explanation, irrespective of whether or not it actually is presented as such in current IDEs. Furthermore, let us assert, using Kass and Leake's taxonomy of explanations, that error explanations are more precisely *material explanations*, where, "if [an] anomalous event contradicts an active expectation of the explainer, it is explained by identifying aspects of the material situation that made the expectation fail ... a [material] explanation must build up from basic laws a causal chain that explains the event" [KL87]. In the case of static analysis tools, the *materials* for the explanation are the explicit program elements visible in the source code, and the *basic laws* are the syntax, semantics, and pragmatics of the programming language — as well as the rules of the reasoning systems — that are causally chained to explain the presence of an anomaly.

Positioning error messages as error explanations isn't just an intellectual or philosophical exercise — doing so allows us to leverage existing scientific theories about explanation and adapt and apply those theories to the domain of static analysis anomalies. For example, treating messages as explanations enables us to apply the cognitive theory of *self-explanation*, a cognitive process by which humans self-generate explanations to themselves and to others in order to understand a situation [CBL+89], to the design of tools.

Consider precisely how framing error messages as error explanations provides perspective into static analysis comprehension difficulties:

Problem: Static analysis tools are black boxes.

As described by the authors of the Roslyn compiler, "source code goes in one end, magic happens in the middle, and object files or assemblies come out the other end. As static analysis tools perform their magic, they build up deep understanding of the code they are processing, but that knowledge is unavailable to anyone but the static analysis tool implementation wizards. The information is promptly forgotten after the translated output is produced."[4] In other words, because compilers *do not expose their internal reasoning processes* to the developer, the developer must essentially *self-generate a plausible explanation* for why a particular anomaly has occurred, despite the fact that the explanation was already known to the tool. As Bret Victor points out in 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?"

Problem: IDEs are hampered in their expressiveness.

Modern integrated developments have limited visualizations available to them

[4] https://github.com/dotnet/roslyn/wiki/Roslyn%20Overview

for presenting a static analysis error [BLCMH14,BWJMH14]. For example, the Roslyn[5] 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 static analysis tool needs to explain.*

Problem: No ground truth for representation of error messages. Even if static analysis tools exposed their internal reasoning, and if IDEs were modified to be more expressive, the research community currently lacks an underlying theoretical framework to articulate what information is necessary to communicate [SS86]. Explanation theory allows us to state that the information that is necessary to communicate an anomaly is the information that is necessary to satisfy the requirements of a *material explanation.* Then, to understand how developers form material explanations, we can investigate how *developers spontaneously explain errors messages to themselves and to other developers.*

In short, the *objective* of this work is to understand how static analysis tools should explain anomalies to the developers. The *significance* of this work is that static analysis explanations have the potential to substantially improve developer comprehension of static analysis anomalies. The expected *contributions* of this work are:

1. A theoretical framework that formalizes and guides the research process, making explicit the concepts and connections under investigation (Section 7).
2. A set of experiments that evaluate the extent to which evidence supports the theoretical framework (Section 9). The experiments are derived from theories of explanation and adapted to the domain of static analysis anomaly comprehension. The experiments provide coverage over the concepts and connections of the theoretical framework.
3. A tool called Radiance, implemented as an Eclipse extension, that operationalizes the findings from the experiments into a usable artifact (Section 8). The artifact demonstrates that it is computationally feasible and practical to incorporate findings about static analysis explanations into modern development environments, and that doing so significantly enhances the developers' comprehension of static analysis anomalies.

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

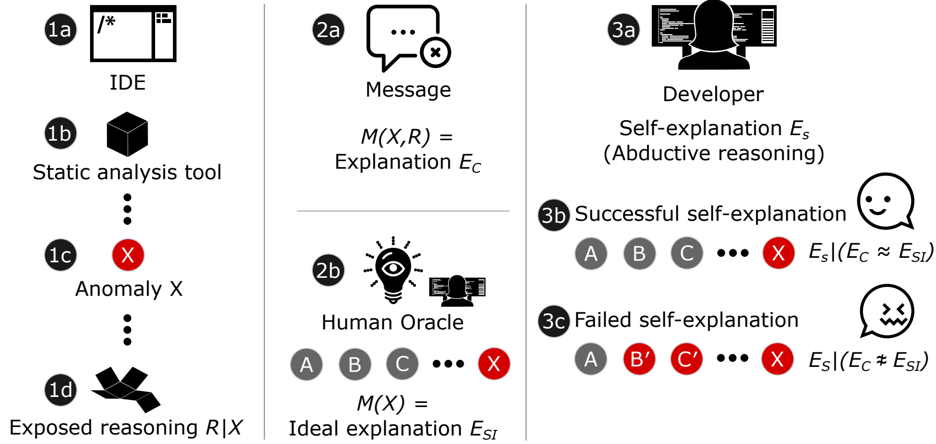

Figure 2. The proposed theoretical framework for self-explanation.

5 My Thesis

> The comprehensibility and utility of error messages for static analysis anomalies can be significantly improved by reframing error messages as *material explanations* that *approximate* how developers explain anomalies to other developers and to themselves.

This thesis statement can be also be stated as a research question: *How should static analysis tools explain anomalies to developers?*

6 Theory: A Static Analysis Explanation Framework

I propose a theoretical framework that models static analysis error messages, and the developers' understanding of such messages, as a material explanation problem (Figure 2). Logically, the framework consists of two primary agents who participate in this communication: 1) at the origin (1a), the IDE acts as the sender of the explanation, and 2) at the termination (3a), the developer uses the IDE's explanation, along with their own self-explanations, to understand a particular anomaly. With these agent end points, we can further characterize this framework in detail by following the flow of concepts that are involved between them.

7 Description of Theoretical Framework

In 1a, the IDE can interface with a variety of static analysis tools, including the IDE's own compiler. For example, in the Eclipse IDE, such analysis tools might

include the Eclipse Compiler for Java, or a third-party extension like FindBugs. Through event handlers or other extension points, the static analysis tool is triggered (1b). During some of these triggers, the source code is a state such that the tool detects an anomaly (1c), which we label X. Note that at this point in time, we have not actually specified how to communicate the anomaly, simply that one has been detected.

Next, given X, the tool introspectively obtains additional information that is relevant to constructing an explanation (1d). For example, in the case of a duplicate variable within the same scope, we may wish to expose the names of the variables, their types, the locations of the variables in the source code, and so on. Alternatively, if this additional information is inexpensive to expose, we can collect it at (1b) and simply discard it when the anomaly isn't detected. In either case, we obtain a data structure R representing the underlying information pertinent to an explanation.

In 2a, the IDE encodes a message, $M(X, R)$, which uses the anomaly and the additional reasoning information for the developer. I assert, within this framework, that such information should be presented in the form of an explanation, E_C. In current IDEs, such an explanation conventionally comes in the form of a textual error description with wavy underline annotations within the source code. But for the moment, we leave $M(X, R)$ as an abstract function that should return an explanation which we have yet to define.

Let us also posit the existence of a "human oracle" (2b). In parallel to 2a, this oracle has the ability to encode, through $M(X)$, a theoretically ideal explanation for a developer. Note that, unlike E_C, the oracle does not depend on R, since the oracle has perfect omniscience. That is to say, E_{SI} is a ground truth. Unfortunately, we are not worthy enough to ever directly observe this oracle. However, by observing experienced developers, we may be able to approximate E_{SI}.

Finally, the developer decodes the explanation received through their IDE (3a). To perform this decoding, the developer uses self-explanation (and other cognitive processes), along with the provided explanation E_C from the IDE, to self-generate explanations that can plausibly account for the anomaly. On the one hand, in 3b, the developer is more likely to be successful in her own self-explanation, E_S, if it is supported by an IDE explanation, E_C, that approximately equals the ideal explanation, E_{SI}. On the other hand, the developer is more likely to produce a failed explanation if E_C does not approximately equal E_{SI}.

Through research questions and experiments, the dissertation will validate the extent to which the obtained evidence supports and justifies this theoretical, static analysis explanation framework.

8 Radiance: An Eclipse Plug-in for Computationally Supporting Developer Self-Explanations

Radiance implements my research findings as an integrated theory of self-explanation for static analysis anomalies, and operationalizes these findings

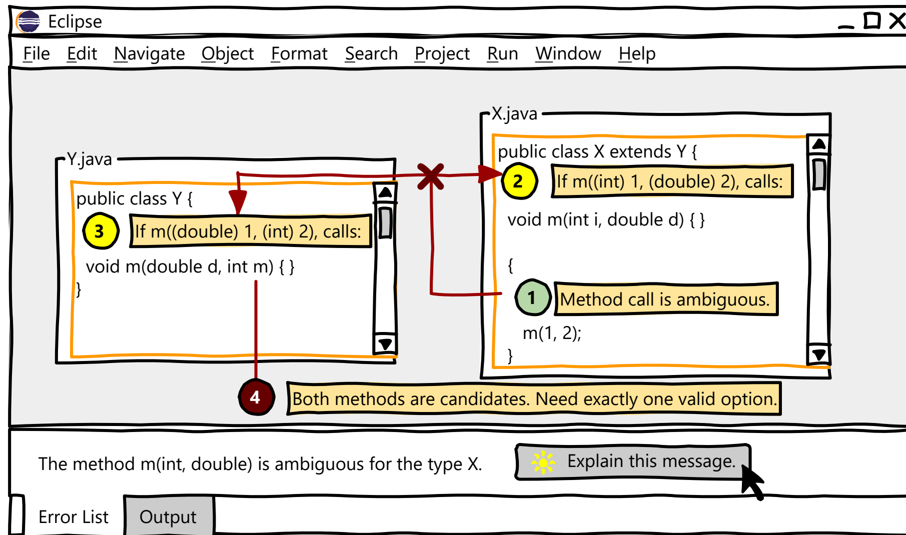

Figure 3. Explanations in the Eclipse IDE.

as a proof-of-concept tool implementation within the Eclipse IDE. The tool will demonstrate that developers benefit from explanatory error messages when such messages are grounded in self-explanation, through support from research studies (Section 9) that cover my theoretical model (Figure 2).

8.1 Design Principles

In this section, we articulate the design principles of the Radiance tool.

Support gap-filling. Research on self-explanation has found that a key failure in successful self-explanation is in *gap-filling*, a process in which the explainer incorrectly believes that he or she has the complete domain knowledge needed to understand a problem, leading to incorrect solutions [VJ93]. I argue that modern IDEs fail to support gap-filling because they treat source code at the level of files and classes, when successful understanding of a static analysis anomaly involves thinking in terms of arbitrary subsets of source code, or *fragments*.

Software engineering researchers recognize the cognitive benefits of tools that support thinking in terms of fragments for a variety of software engineering activities [BZR+10,DR10,HF14,CKM06,BLMH15]; perhaps the most well-known information-fragment tool is Code Bubbles [BZR+10]. I argue that tools that support thinking in terms of information fragments facilitate self-explanation because they make relevant program elements more pronounced while masking unnecessary program elements. Consequently, Radiance implements the "bubbles" metaphor as its mechanism to support gap-filling.

Table 1. Opportunities for Improvement: Explicit Relationality

Static Analysis	All	Relational	Pct (%)
Eclipse JDK (Luna)	598	250	41.8%
Microsoft C# 5.0 (Roslyn)	1107	282	25.5%
Microsoft F# 3.1	1136	198	17.4%
Microsoft TypeScript 1.0	538	116	21.6%
Oracle OpenJDK 7	487	126	25.9%

Explicate relationships. A second design principle in my tool is derived from Legare's definition of self-explanation as an "attempt to understand a casual relation by identifying relevant functional or mechanistic information" [Leg14]. 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). It is therefore not enough to simply identify the program elements; the tool must also explain how these elements relate to each other. Radiance explicates relationships by using a box-and-arrows annotation scheme with enumerated labels [BLCMH14]. At each enumeration, explanatory information about the corresponding program element or elements are presented to the developer.

As a preliminary investigation, I analyzed the error message 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 (Table 1) as a lower-bound. Through self-explanation theory, I argue that developers can benefit if error messages are made more relational.

Prefer diagrammatic representations over sentential representations. If source code is the primary artifact through which developers transform their ideas to computation, it stands to reason that static analysis messages should be contextualized to and presented around the source code. My own research in understanding how developers visualize static analysis anomalies found that developers frequently make annotations directly on top of the source code when explaining errors to themselves. Furthermore, adding diagrams to text (e.g., source code) over text alone has been found to promote the self-explanation effect [AT03].

Moreover, additional research in diagrams by Larkin and Simon compared sentential representations against diagrammatic representations [LS87]. In *sentential representations*, expressions form a a sequence corresponding to sentences in a natural language description of the problem. Sentential representations resemble the predominantly textual error messages that IDEs provide today. In contrast, in *diagrammatic representations*, expressions correspond to components of a diagram

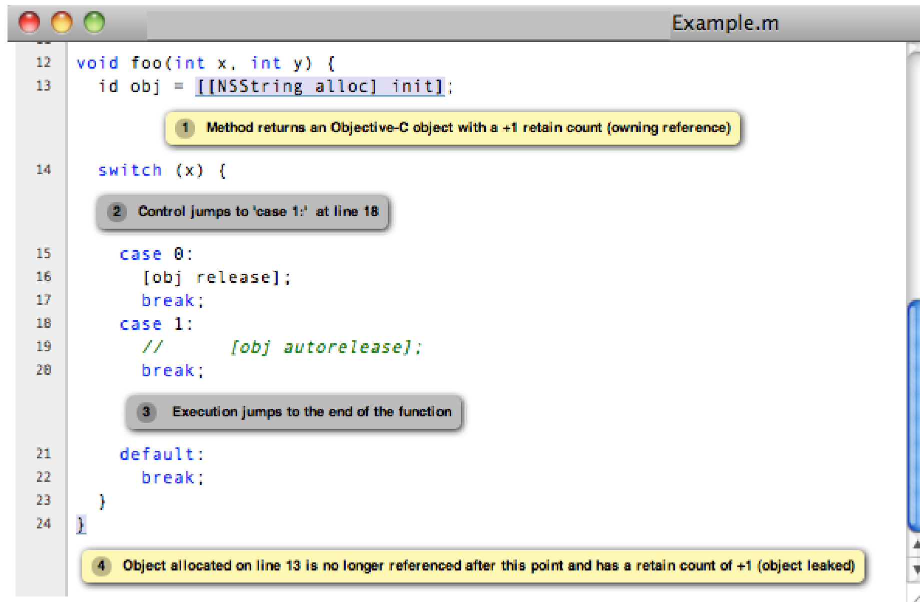

Figure 4. The scan-build analyzer in LLVM supports only a limited number of static analysis anomalies, but displays explanations directly on the source code.

describing the problem whereby each expression contains the information that is stored at one particular *locus* in the diagram, including information about relations with the adjacent loci. Thus, diagrammatic representations are *indexed* by location in a two-dimensional, graphical plane.

Larkin and Simon found that in the diagrammatic representation, as a result of organization by location and location-adjacent information cues, "problem solving can proceed through a smooth traversal of the diagram, and may require very little search or computation of elements that had been implicit [in the sentential representation]" [LS87].

Today, some tools such as Code Lens in Visual Studio and scan-build for LLVM (Figure 4) present information directly overlaid on the source code, and it is thus worth investigating whether such visual overlays are beneficial to developers. Given the findings of Larkin and Simon, it is surprising that existing IDEs do not leverage affordances such as diagrams or location-adjacent explanations on the source code editor to a greater extent. Radiance prefers diagrammatic representations to sentential representations by providing explanations that are directly overlaid on the source code, and positioning information fragments logically by treating the editor pane as a graphical plane.

8.2 Developer Interaction

Let's once again visit our friend Kevin, the frustrated developer we encountered in our motivating example from (Section 2). Recall that Kevin was attempting to understand and resolve an ambiguous method error, but was unable to successfully do so for several reasons. We can now more fully describe these reasons in terms of self-explanation and my design guidelines.

First, Eclipse did not present Kevin with all of the relevant program elements (Figure 1). To gap-fill this incomplete information, Kevin employed some tricks to interrogate the compiler into giving him more information. Second, the error in the IDE failed to provide a causal explanation for why the compiler has generated this particular message. Consequently, Kevin attempted to simulate the internal reasoning of the compiler, but failed to identify type promotion as the underlying cause of the error during self-explanation. Third, even though the source code was his primary artifact of interest, he had to find information about his error through several locations in the IDE and manually relate their corresponding program elements in his source code editor. Although the IDE used some visual annotations, such as gutter markers, to help identify relevant program elements, this representation is not congruent with how developers diagrammatically self-explain errors to themselves (Section 9.1) [BLCMH14].

Kevin's experience would have been significantly improved through the use of Radiance, because the tool incorporates design guidelines that are important for self-explanation (Figure 3). Using Radiance, he would notice that this particular error is supported by Radiance, and would have activated the feature through the "Explain this message" button.

By providing a casual chain of sequential steps against relevant information fragments, Kevin would more effectively gap-fill by simply following the explanatory narrative given by the tool. Unlike Eclipse, in Radiance the relationships are shown explicitly through the use of arrows. Importantly, Radiance contextualizes its explanation in terms of the source code Kevin is already familiar with, and presents this explanation as a visual overlay on top of the source code that he is already examining. Furthermore, this diagrammatic representation aligns with the representation he would use if he had produced such a diagram himself. Finally, Radiance does not return a single, one-shot, line-oriented error message. Instead, Radiance progressively provides explanation through exposing its reasoning process and annotates points in the casual chain with its findings.

Because Radiance better supported Kevin's self-explanation process, Kevin would have been able to come to a correct judgment about the underlying cause of the error.

8.3 Implementation details

Radiance will be a proof-of-concept implementation, to be deployed as an Eclipse plug-in. As a proof-of-concept implementation, the tool is expected to correctly generate explanatory messages, but will do so using naive algorithms. For static analysis errors, Radiance will do nothing in the event of a successful compilation.

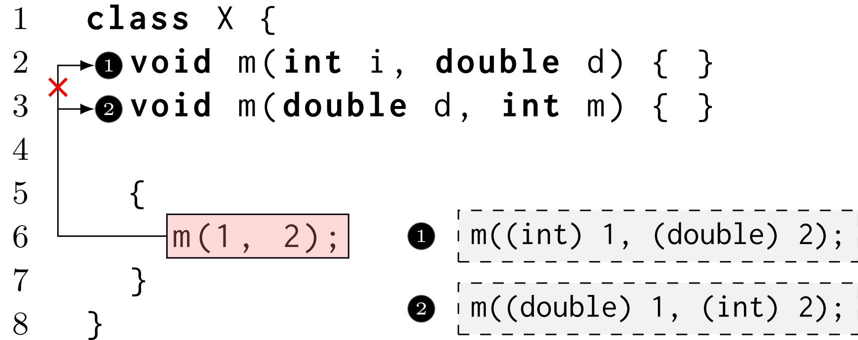

Figure 5. Explanatory visualization in IDEs. Expressive representations are enabled by exposing compiler internals for use by the IDE.

In the event of a detected anomaly, Radiance will *rerun* the compilation, but in this second invocation the tool will pass a flag to the Eclipse Compiler for Java (ECJ) so that it can collect pertinent information about the anomaly. In this case, the ECJ compiler will be instrumented to collect information for a finite set of anomalies that are intended to be explained through Radiance. Upon completion, Radiance will override the editor pane with a canvas that supports a Code Bubble-like presentation.

What type of static analysis tool anomalies will be explained? There exists a diversity of tools, languages, and static analysis techniques within both the industrial and academic communities. My approach for explanations for static analysis anomalies focuses on areas of static analysis in which empirical studies have demonstrated developer difficulties — through modern, practical tools that are currently used in industry [SSE+14]. Using this criteria, a starting point for investigation is the Eclipse IDE, and its associated Eclipse JDT compiler.

9 Experiments and Evaluations

This section describes the experiments and evaluations I will conduct to support my theoretical framework (Figure 2). The current status of each experiment, and the completed or proposed semester of completion, is indicated in parentheses. The brackets in the section header indicate a short-form project code name for the experiment.

9.1 [Diagrams] How do developers visualize compiler error messages? (Completed, Spring 2014)

Study rationale. Modern IDEs, such as Eclipse, IntelliJ, and Visual Studio, offer a number of visualizations to assist developers in more effectively com-

prehending static analysis error messages. 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, to indicate a relevant location of the error. I hypothesize that existing visualizations do not align with the way in which developers self-explain error messages. Specifically, compiler authors have suggested that existing tools do not expose important internal reasoning processes of the compiler that are necessary to self-explain a provided anomaly.[6] Within my theoretical framework (Figure 2), this study covers the Message (2a), Developer (3a), and the resulting quality of the explanations that developers are able to offer (3b and 3c).

Research questions.

- RQ0 (Pilot) What visual annotations do developers use when they explain error messages to each other?
- 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?

Methodology. *Data collection.* In a bootstrap pilot study, we asked third-year Software Engineering students to pair with another student for an *explainer-listener* activity. During this activity, one student (designated explainer) was asked to verbally explain the error message to the other student while visually annotating a source code listing during their explanation. For this process, each student was given a sheet of paper with a source code listing and the corresponding static analysis error message. The source code listings were unadorned and lacked any visual annotations. We collected their marked sheets, and created a taxonomy of visual annotations based on our observations. Source code was typically annotated as box-and-arrow diagrams, similar to those identified by Cherubini and colleagues in their own explanation experiment on how software developers draw diagrams for code on the whiteboard [CVDK07] (RQ0). With this taxonomy, we then conducted a controlled lab study, recruiting 28 participants (23 male, 5 female) from another third-year Software Engineering course. *Approach.* We created six paper-and-pencil mockups of *explanatory visualizations* from error-related unit tests in the OpenJDK diagnostics frameworks[7]. An example of such a visualization is shown in Figure 5. As a proof-of-concept, we intentionally

[6] `https://github.com/dotnet/roslyn/wiki/Roslyn%20Overview`

[7] 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/`

selected the mockups that we believed could benefit most from visual annotations. We constructed 12 mockups in total: six for the control group (baseline visualizations) and six for the treatment group (explanatory visualizations). Participants were randomly assigned to the baseline or explanatory group. We conducted the experiment in two phases. In the first phase, for each group, we sequentially provided participants with six error messages. We gave participants 30 seconds to individually examine the paper mockup, and then instructed participants to think-aloud and verbally explain, that is, self-explain, the cause of the error. We encouraged participants to visually annotate the unadorned mockup during their explanation. In the second phase, we used a recall-correctness experimental design inspired by Shneiderman, which acts as proxy for measuring programming comprehension (for details, see [Shn77]). In this experimental design, we asked participants to write source code listings on a computer from scratch in order to *generate* a provided compiler error; all six provided errors came from an error that they had *previously explained* during the first phase of the experiment. Between the first and second phase of the experiments, participants were also given Cognitive Dimensions of Notations questionnaire [GP96], which we simplified and adapted for error messages. *Analysis.* For RQ1, we conducted an inter-rater reliability exercise in which the first and second authors independently rated the participants' explanations, without consideration of group. For each of the 168 tasks, we assigned a 4-point Likert-style score using a rubric. For RQ2, we coded the annotation types used for each task, partitioned into control and treatment groups. For RQ3, we used the Cognitive Dimensions questionnaire to statistically identify differences between the baseline visualizations and explanatory visualizations. For RQ4, we algorithmically analyzed the participant source code listings and tagged them as correct or incorrect, based on whether the listing generated the provided compiler error.

Results. The results of this study are published in VISSOFT 14 [BLCMH14]. For RQ1, we confirmed that participants gave significantly better explanations in the treatment group ($n_1 = n_2 = 84, Z = 2.23, p = .026$). For RQ2, we found that the treatment group used significantly more visual annotation types in their explanations than the control group ($n_1 = n_2 = 84$, $Z = 2.15$, $p = .032$), which indicates that participants adopted our explanatory visualizations into their own explanations. Furthermore, participants in both groups used and applied 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. For RQ3, we found that the distribution of responses to our Cognitive Dimensions instrument were significantly different for hidden dependencies ($n_1 = n_2 = 14$, $Z = -2.64$, $p = .008$). Thus, explanatory visualizations reveal more of the hidden dependencies, that is, the internal reasoning process of the compiler, than the baseline visualizations. For RQ2, we were unable to replicate the findings of the recall-correctness task from Shneiderman, which suggested that better comprehension would result in better recall correctness [Shn77]. We speculate that our failure to replicate this

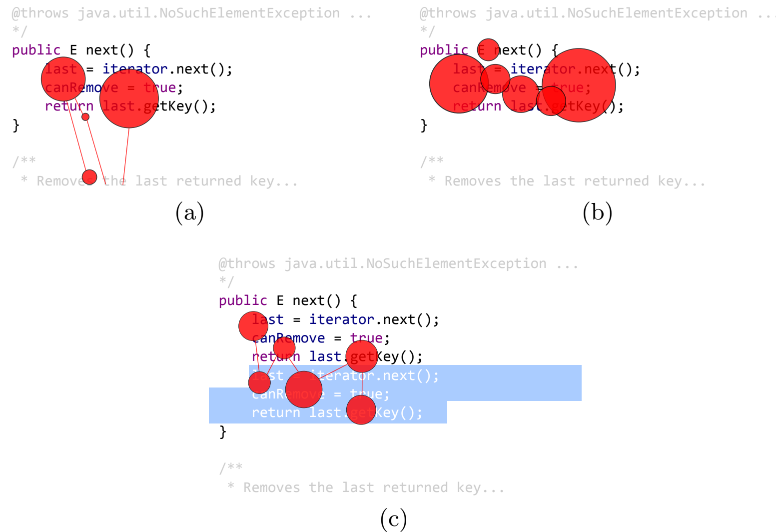

Figure 6. A red circle indicates a fixation point, whose size is proportional to the duration of the fixation. In this figure, the developer is attempting to comprehend a notification about a duplicate method error. (a) Developer rapidly and unconsciously changes their eye fixation, or *saccades*, between the onscreen method and the duplicate method they expect to see below it. (b) Developer spends significant time on relevant portion of the code. (c) Developer engages in a stable but suboptimal copy and paste approach, bringing the method contents in closer proximity to examine the differences between the two method more easily.

experiment was the result of think-aloud, because the action of performing a think-aloud can artificially promote the self-explanation effect [CC05].

9.2 [Gazerbeams] What can eye gaze tell us about failures in self-explanation during compiler error comprehension? (In-Progress, Summer 2016)

Study rationale. Seo and colleagues conducted an empirical study at Google to characterize the reasons for build errors that occur during the development process [SSE+14]. I capitalize on this prior work for two reasons.

First, unlike prior investigations that focus on helping novices through teaching environments [AB15,MFK11,HMRM03,NPM08], Seo and colleagues study relatively expert developers using Eclipse. Alarmingly, the compiler errors that experts frequently encounter are not all that different from the errors novices get, although experts do make fewer trivial syntax errors [AB15,SSE+14]. Although some learning does co-occur with self-explanation [FHdJ90], given this expert population, approaches that attempt to address a lack of conceptual knowledge on the part of the developer appear insufficient to moderate the self-explanation effect [JPMHH15].

Second, the authors rank the errors through a conventional probability-impact risk function ($R = PI$), computed as the frequency (probability) multiplied by the median resolution time (impact). This provides me with a target compiler error distribution to investigate in my theoretical framework (Figure 2).

A limitation of the data-driven Google study is that although it pinpoints problematic compiler errors for developers, the study methodology does not provide explanatory power for understanding *why* these particular compile errors are perplexing for developers. This study seeks to address this limitation, and investigates how and why $M(X, R)$ in modern IDEs (Figure 2) fail to support developers.

Research questions. The study attempts to understand why some developers are able to correctly resolve a defect ("successful developers") while others are not ("unsuccessful developers").

RQ1 How does the sequence of information that developers use when understanding a static analysis anomaly differ between successful and unsuccessful developers?

RQ2 What features of the static error messages do developers actually use, and to what extent?

RQ3 VanLehn and Jones [VJ93] found that the gap-filling explanations ("when [students] seem not to discover their own ignorance") accounted for most of the self-explanation effect. Does gap-filling explain the difference between successful and unsuccessful developers?

RQ4 Using validated eye tracking measures for cognitive processes, such as for split-attention, cueing or signaling, cognitive load, confidence, and self-explanation [AA13], which measures best explain the differences between successful and unsuccessful developers?

Proposed methodology. *Data collection.* We collected data from 60 participants from undergraduate and graduate Software Engineering courses at our University. During the user study, the investigator calibrated an eye tracker which recorded eye fixations during their session. *Approach.* Through ten randomly assigned tasks, participants were asked to identify and make program modifications to remove a compiler anomaly within the Eclipse IDE. We selected the compiler errors to cover the space of difficult errors as identified by Seo and colleagues [SSE+14]. For the tasks, we manually injected these compiler errors by permuting Apache Common collections library[8]. We injected compiler errors into collections that were likely to be known to the participants, such as lists, hash tables, and queues, in order to minimize the time needed to understand the functionality of the code itself. *Analysis.* First, I will code the eye tracking data into an event-sequence form that is suitable for subsequent analysis. To do so, I will use a combination of manual and automated techniques. For example, I

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

will use algorithms in OpenCV to identify the underlying region of the IDE that the participant is fixating on at a given time. I will use manual inspection to determine, for a given error, whether the participant successfully resolved the error. To address RQ1, I will then conduct an analysis that interprets events as a string-based sequence analysis problem, which will allow me to apply sequence mining techniques to analyze the data. I will also perform a link analysis [Din95], as commonly used in human factors research. For RQ2, I will use the sequence data to identify which affordances developers use during comprehension. For RQ3, I will identify the relevant program elements necessary to successfully solve the problem, and determine whether or not unsuccessful participants see that information. For RQ4, I will apply existing eye tracking algorithms for different cognitive processes and determine the extent to which each process can explain the underlying behavior of the participants.

9.3 [`Stack Overflow`] How do developers explain static analysis anomalies to other developers through computer-mediated communications? (Proposed, Summer 2016)

Study rationale. The purpose of this experiment is to provide support for the Human Oracle (2b) in my theoretical framework (Figure 2). I propose that Stack Overflow provides a suitable approximation of an ideal explanation, E_{SI}.

For example, consider a typical communication exchange in Figure 7. A user asks a question about an ambiguous method compiler anomaly (Figure 7a). The user asks the question using terms such as "trying to understand" and "getting an error", which are indicative of a question requiring an explanation. The question has also been tagged with `java` and `method-overloading` and contains the quoted compiler error (`The method sum(int, long) is ambiguous for type Overloading_OverRiding`).

The answer in Figure 7b is a good explanation. Perhaps this is because the answer is given in the context of the original code, that is, `sum(int i, long j)`, and because it enumerates the space of possible ambiguities and describes why each interpretation is ambiguous. The explanation also concludes by giving a generalization of the specific-instance of the problem: "the compiler always needs exactly one valid option."

In contrast, the answer in Figure 7c is a poor explanation. Perhaps this is because it refers to a program element (`5`), but the reader is unable to determine which `5` from the question it is referring to. It may also be a poor explanation because it simply restates, rather than clarifies, the error description of the original question ("For this reason, method call is ambiguous").

In addition to the illustrated example, prior work supports the position that Stack Overflow contains pertinent and high-quality explanations of static analysis anomalies. For example, Parnin and colleagues found that developers may be getting as much as half of their documentation from Stack Overflow, and that developers continuously reference Stack Overflow questions during development

The method is ambiguous

1

I am trying to understand method overloading and I'm not able to understand the reason for the following code error in the following example

```
public class OverLoading_OverRiding {
     public static void main(String[] args) {
        OverLoading_OverRiding o = new OverLoading_OverRiding();
        o.sum(5, 5);
    }

    void sum(int i, long j) {   }
    void sum(long i, int j) {   }
}
```

I am getting an error:

> The method sum(int, long) is ambiguous for the type OverLoading_OverRiding.

When i am performing explicit casting on the same example then it works:

```
    o.sum(5, (long)5);
```

java method-overloading

share edited Mar 31 '15 at 8:03 giorashc 10.4k ●3 ●18 ●50 asked Mar 31 '15 at 7:29 Dhruv Raval 1,080 ●6 ●17

(a) Question submitted by user about an ambiguous method compiler error.

3

The problem is that your compiler dont know which method to use if you call `o.sum(5, 5);`

1. he could use `void sum(int i, long j) {   }` where he takes the first 5 as int and the second 5 as long
2. he could use `void sum(long i, int j) {   }` where he takes the first 5 as long and the second 5 as int

since both methods would be valid to use in this example and your compiler always needs **exactly one** valid option, you get the error msg that your method is ambiguous.

if you call `o.sum(5, (long)5);` it matches only the method `void sum(int i, long j) {   }`

share edit flag edited Mar 31 '15 at 10:03 answered Mar 31 '15 at 7:33 TobiasR. 380 ●10

(b) Highest rated answer, from user with 1080 reputation, and further clarified by user with 10.4k reputation.

0

This is because 5 could be implicitly long or integer .. For this reason, method call is ambiguous.

share edit flag answered Mar 31 '15 at 7:37 wntun 26 ●3

add a comment

(c) Lowest rated answer, from user with 26 reputation.

Figure 7. A Stack Overflow communication exchange for a question about an ambiguous method compiler error, and two selected answers.

via search.[9] One of the reasons they may do so, as identified by Mamykina and colleagues, is that the site provides high-quality answers as a result of its carefully-crafted reputation system [MMM+11]. Finally, Nasehi and colleagues identified *debugging and corrective question types* as one of the main concerns of posters, and found that *explanations* accompanying source code example were an important characteristic of high-quality Stack Overflow answers [NSMB12]. Together, these findings give us confidence that Stack Overflow contains high-quality explanations for static analysis anomalies that approximate E_{SI}.

Research questions.

RQ1 What questions do developers ask when they want to understand a static analysis error?
RQ2 What plausible self-explanations have they already generated when they frame their question to other developers (that is, when in the explanation process did they get "stuck")?
RQ3 What features of a response make for a good explanation?
RQ4 What features of a response make for a poor explanation?

Proposed methodology. *Data collection.* I use a labeled data set from Stack Overflow, where the ranking of the response, as well as the reputation of the user who posted the response, is used as a proxy to identify high-quality and low-quality responses. To find questions in Stack Overflow, I will use existing tags (`compiler-errors`), the text of the error message ("ambiguous method"), and common phases ("can someone explain"). *Approach.* I will use a qualitative attribute coding scheme that categorizes the features present in both styles of questions and answers [Sal09]. *Analysis.* For RQ1, I will catalog the types of questions that developers ask within their posts to StackOverflow. For RQ2, I will qualitatively classify the types of self-explanations developers generate before getting stuck. For RQ3 and RQ4, I will a identify a set of features. For example, one feature may be whether or not the answer refers to any source code in the original question as part of its explanation. Given the ratings of each question and answer, I will then construct a multiple linear regression model that weights the importance of these features to identify the important features in responses to questions.

9.4 [`Rust`] How do compiler authors design and instrument static analysis messages in their tools? (Out of Scope)

Study rationale. It's easy to criticize the state of static analysis error messages. Certainly, the research community today isn't plagued by a dearth of complaints about perplexing messages [Tra10,JSMHB13]. But such criticism puts the lens on the end-user and the tool itself, while marginalizing the actual developers

[9] `http://blog.ninlabs.com/2013/03/api-documentation/`

Improve error about close delimiter #10636

Closed **evilpie** opened this issue on Nov 24, 2013 · 0 comments

evilpie commented on Nov 24, 2013

I wrote code like this for servo:

```
pub fn trace_option(tracer: *mut JSTracer, description: &str, option: Option<Reflector>) {
    option.map(|some| trace_reflector(tracer, description, some.reflector());
}
```

which was missing the closing) for the map. The error however was about something different: `error: incorrect close delimiter: `}``

(a) Rust community member improves error message about close delimiter.

rust-lang / **rust** Watch 957 Star 15,227 Fork 2,970

Code Issues 2,223 Pull requests 86 Pulse Graphs

Improve error message when module resolve fails #28081

Closed **bombless** wants to merge 1 commit into `rust-lang:master` from `bombless:resolve-error-message`

Conversation 6 Commits 1 Files changed 2 +45 −2

Showing 2 changed files with 45 additions and 2 deletions. Unified Split

28 src/librustc_resolve/lib.rs Show notes View

```
@@ -1325,6 +1325,21 @@ impl<'a, 'tcx> Resolver<'a, 'tcx> {
1325             }                           1325             }
1326         }                               1326         }
1327                                         1327
                                             1328 +        fn search_child_modules(needle: Name, module:
                                                  &Rc<Module>)
                                             1329 +                              -> Option<Rc<Module>> {
                                             1330 +            for (name, name_bindings) in
                                                  &*module.children.borrow() {
                                             1331 +                if let Some(x) =
                                                  name_bindings.get_module_if_available() {
                                             1332 +                    if needle.as_str() == name.as_str() {
                                             1333 +                        return Some(x)
```

(b) Rust community member checks in code to improve error message on module resolve failure.

Figure 8. The Rust community is actively working on improving error messages generated by the `rustc` compiler. Such a community is ideal for action research.

who build them. These static analysis authors have spent significant time and energy, commercially or in open source communities, to make such tools available. Surprisingly, Traver has observed that “there seems to not be any formal study on why commercial compilers have neglected the area of diagnostics” and speculates that “higher priority has been paid to other product features such as compilation speed or the speed or the resulting executable program” [Tra10].

Traver's speculation sounds plausible, but it's a "just-so story." I propose to conduct an empirical study to learn how static analysis authors: a) decide when to indicate an anomaly, b) consider how to present the anomaly to the end-user, c) invest or allocate effort in implementing the anomaly detection, and collaborate with other authors during this process throughout the static analysis tool development lifecycle.

Understanding static analysis errors from the perspective of the static analysis author will illuminate the static analysis internals (1a-1d), as well as the message explanation (2a) in my theoretical framework (Figure 2).

Research questions.

RQ1 At what points in the software development lifecycle do static analysis authors work with static analysis anomalies?
RQ2 How do static analysis authors make decisions about which anomalies to implement?
RQ3 How do static analysis authors evaluate their error messages?
RQ4 What types of discussions happen around error messages?
RQ5 What are the challenges that static analysis authors identify that hinder the generation of good error messages?

Proposed methodology. *Data collection.* Semi-structured interviews with static analysis authors as identified by version history commits and communications (RQ 3.2, RQ 3.3, RQ 3.5), and qualitative analysis of mailing lists, version control issues, and IRC logs to form a grounded theory (RQ 3.1, RQ 3.4). Investigator will interact with the community through these channels. *Approach.* In contrast with purely observational research methods, I propose to apply a method of practical *action research* [RH08] in which the principal investigator engages in the community of practice to influence, improve, and implement the recommendations. The Rust community is one possible avenue. First, the team is open source, which means that contributions to the project are available to the community-at-large. Second, the community has an open development process, with extensive communications through mailing lists and IRC channels. Third, and most importantly, Rust is currently in the process of establishing user guidelines for error messages.[10] Thus, the community is likely to be receptive to academics interested in the comprehension of static analysis anomalies. *Analysis.* Empirical methods, such as descriptive and axial coding, for a qualitative case study on Rust, with a focus on static analysis anomalies.

9.5 [`Radiance`] Can instrumenting our findings as a practical tool improve developer static analysis error comprehension? (Out of Scope)

Study rationale. In this experiment, I propose to conduct an integrative tool evaluation of Radiance, whose details are described in Section 6. This tool

[10] `https://internals.rust-lang.org/t/pre-rfc-rustc-ux-guidelines/2419`

evaluation serves as capstone for my theoretical framework and applies my theoretical results to a practical tool. Strictly speaking, this study is unnecessary for purposes of a successful dissertation, as the other proposed studies provide adequate coverage and evidence for the theoretical framework. However, the implementation of a practical tool is career aspirational: like Parnas, I believe that the output of successful software engineering research should have relevance to industry [Par98,LNZ15]. Moreover, having such a tool provides me with greater career flexibility in the future.

Research questions.

RQ1 How do developers assess the tool under traditional HCI measures, such as effectiveness and efficiency, when compared with my prior Eclipse experiment (Section 9.2)?
RQ2 Do developers adopt different strategies with Radiance than with Eclipse?
RQ3 Replicating the eye tracking research questions from the Eclipse study, in what ways do they differ?
RQ4 What do developers say about why Radiance helps them perform their task?
RQ5 In what ways to developers feel that Radiance could be improved?

Proposed methodology. *Data collection.* I will recruit a mix of novices and developers from industry to obtain participants with a spectrum of experience. Participants will conduct tasks identical to the setup in Eclipse (Section 9.2), except with the Radiance extension. *Approach.* The approach is replication of the Eclipse experimental study design. As an addition to the original experimental design, I will conduct a follow-up participatory design exercise with the participants to to collect opinions and judgments about the Radiance tool [WM91]. The evaluation will consist of a short, cooperative, verbal evaluation of the interface. The purposes of this qualitative instruments is to understand how developers perceive the self-explanation design principles as used in the tool, against their own experiences with IDEs. *Analysis.* Data are analyzed using the same techniques as in the Eclipse study (Section 9.2). In addition, a qualitative report of Radiance will be provided using the findings from the heuristic evaluation and semi-structured interviews.

10 Broader Impact and Generalization

I expect my work to generalize in a variety of ways. First, in generating a theory of explanation, I hope that the discovered principles of explanation through computer-mediated communications can be applied to other static analysis domains. For example, if developers find that being able to identify relationships is important to resolving bug defects, it is plausible that similar tool affordances will help developers when they are attempting to understand explanations about, say, compiler optimizations [vDD11]. Second, if the principles are cognitively justified, then these principles should also generalized across languages. For

example, if having example suggested fixes is beneficial to the developer for self-explanation, that should be true whether the language is a functional, imperative, or something else.

11 Related Work

In this section, I report on existing theories and evidence for self-explanation that justify applying self-explanation principles towards the improvement of static analysis anomalies. In order to focus potential tool improvement efforts, I then characterize the state space of error messages that developers encounter in practice. Next, I examine techniques in the programming languages community that I can leverage in explanations for static analysis tools. Finally, I describe prior tools and systems for software engineering tasks that embody the "spirit" of self-explanation to establish how incorporating self-explanation principles into tools benefit developers.

11.1 Theory of self-explanation

Chi and colleagues, in their seminal work, coined the term *self-explanation*, or self-generated explanations, that "good" and "poor" students produced through think-aloud while studying worked-out examples of mechanics problems [CBL+89]. The authors hypothesized that students learn and understand an example via the explanations they give while studying it, and found that "good" students generate significantly more explanations than "poor" students. Their results demonstrate that self-explanation is an essential cognitive processes through which one identifies the conditions and consequences of actions, forms *relationships* between actions to goals, and forms *relationships* of goals and actions to principles. Essentially, self-explanation theory provides the "cognitive bridge" through which humans translate declarative facts to *understanding*.

Since the original finding, self-explanation has been replicated in a variety of domains [Ale02,CE07,RN08,WG05], including computer programming tasks [VJ93,BPB95]; the cognitive process of self-explanation appears to be not only essential, but ubiquitous [AA13]. Thus, if developers find error messages to be perplexing and difficult to understand, it is plausible that tools do not adequately support the self-explanation processes developers use to diagnose static analysis anomalies.

Subsequent work by Chi and colleagues have found that self-explanation can be elicited through explicit prompts [CDCL94], whether by humans or by computers [Ale02]. Furthermore, Ainsworth and Th Loizou [AT03] found that self-explanation effects are significantly enhanced when diagrams are used over text-alone because of: a) *computational offloading*, or reducing the amount of effort needed to solve the problem, b) *re-representation*, through alternative external representations that utilize perceptual processes rather than cognitive operations, and c) *graphical constraining*, which *limits* the range of inferences that

can be made about the represented concept.[11] Both of these findings suggest that tools can effectively support self-explanation of static analysis anomalies through a combination of self-explanation prompts and diagrammatic representations.

11.2 Error message distributions

A single programming language implementation, such as Java or C#, contains several thousand possible static analysis anomalies. Given finite resources, it's therefore prudent to characterize the space of error messages that developers actually receive in order to effectively target tool improvements.

To understand this space, Seo and colleagues conducted an empirical case study at Google of 26.6 million builds produced over nine months by thousands of developers [SSE+14]. The authors found that nearly 30% of builds at Google fail due to a static analysis error, and that the median resolution time for each error is 12 minutes [SSE+14]. Surprisingly, the costly errors that developers make are rather mundane, relating to basic issues such as dependencies, type mismatches, syntax, and semantic errors.

For novice developers, that is, students using Java in the BlueJ IDE[12], the situation is even worse — through telemetry of over 37 million compilation events, Altdmri and Brown identified that nearly 48% of all compilations fail [AB15]. Similar to the errors made by experts made by developers at Google, novices also had primarily syntax errors, type errors, and other semantics errors. For some reason, it appears that experience alone isn't making these errors go away.

Using a Python corpus of 1.6 million code submissions, of which 640,000 resulted in an error (approximately 40%), and re-examining the BlueJ dataset, Pritchard model-fit the distribution of these error messages and found that they empirically resemble a Zipf-Mandelbrot distribution [Pri15]. Such power law distributions have a small set of values that dominate the distribution, followed by a long tail that rapidly diminishes. Although Seo and colleagues did not model-fit the distributions (their paper, Figure 7), a visual inspection of Java suggests that a similar power-law effect is present [SSE+14].

The triangulation of these multiple data sources indicates several consistent features about anomalies across programming languages. First, the dominant errors, both in terms of cost and frequency, are relatively consistent irrespective of developer experience. This is interesting in that a single error explanation representation is likely to benefit a spectrum of developer experiences. Second, the power-law distribution suggests that addressing even a small number of

[11] Graphical constraining merits some further discussion, as it seems counterintuitive. How can the lack of expressiveness make a diagram more effective? As argued by Stenning and Oberlander [SO95], "text permits the expression of ambiguity in the way that graphics cannot easily accommodate." Concretely, consider the Java error message, `name clash: m(Param<Integer>) in B and m(Param<String) in A have the same erasure, yet neither hides the other`. In a diagrammatic representation, the notation would necessitate explicitly pointing to the program elements. In this sentential representation, terms like "neither" and "other" introduce ambiguity.

[12] `http://bluej.org/`

dominant errors could substantially benefit developer experiences with static analysis anomalies. Third and finally, the categories of errors messages are rather mundane: as a tool implementation, this means that improvements to such errors (for example, displaying all relevant program elements) can be feasibly tackled using conventional AST parsing and analysis techniques.

11.3 Programming languages techniques for explanation

Static analysis tools comprise an extensive set of analysis techniques, ranging from hand-crafted, informal methods such as *bug patterns*, as found in `lint`-style tools (for example, FindBugs [HP04]), to formal method approaches which provide stricter guarantees such as abstract interpretation (for example, Frama-C [CKK+12] and Jakstab [KV08]), model checking (for example, SLAM [BR01]), and program querying (for example, PQL [MLL05] and NDepend[13]).

Researchers have been particularly interested in a class of abstract interpretation, type inference, and applying those techniques to modern dynamic languages [FAFH09,AGD05] to detect anomalies before runtime. Moreover, type errors are an area in which researches have focused on improving and simplifying their explanations. For example, Lerner and colleagues propose a system where the type-checker itself does not produce the error message; instead, the system searches for similar program that do type-check [LFGC07]. Boustani and Hage, for generics errors in Java, commented that standard Java compilers do not "explain why generic method invocations fails to type check;" their approach proposed a fix-oriented, heuristic technique to add suggestion how the defect might be resolved [EH10].

To support these techniques, my work applies an empirical HCI lens to these existing techniques, to understand why or why not certain types of explanations benefit developers.

11.4 Self-explanation systems

Lim and colleagues conducted a study in which participants were shown an intelligent system's operation along with various automatically generated explanations [LDA09]. The authors found that explanations describing *why* the system behaved a certain way resulted in better understanding and stronger feelings of trust. Lim argued that most of these types of systems employ complex rules or models, and that lack of intelligibility can lead users to "mistrust the system, misuse it, or abandon it altogether [LDA09]." Even when the developer already understands the static analysis anomaly, having a second opinion can give the developer confidence that their own self-generated explanation is a plausible one. And when there is a mismatch between the developer's expectation and the static analysis explanation, the developer can more readily identify if it is the tool or themselves that is at fault.

[13] `http://www.ndepend.com/`

Kulesza and colleagues also investigated explanations for intelligent systems, focusing on how properties of soundness and completeness affect the final fidelity of an end users' mental models [KSB+13]. Their findings suggest that completeness is more useful than soundness. This result is useful in that most static analysis anomalies are computationally undecidable[14], and therefore tools must make heuristic guesses about a particular anomaly [MS15]. For an incorrect anomaly, a developer is likely to make a more accurate judgment about the anomaly when accompanied with an explanation.

The Whyline system is a prototype interrogative debugging interface in which developers can ask the system "Why" or "Why not" questions about runtime events [KM04]. Ko and Myers found that 50% of all logic errors in the Alice programming language were due to a developers' false assumptions in the hypotheses they formed while debugging existing errors. In terms of self-explanation, we can say that developers failed to generate plausible explanations for a given runtime anomaly.

Theseus is an IDE extension that visualizes runtime behavior within a JavaScript code editor [LBM14]. Theseus "proactively addresses misconceptions by drawing attention to similarities and differences between the programmer's idea of what code does and what it actually does" [LBM14]. In terms of my theoretical framework, we would say that Theseus challenges developers to reconcile their own self-generated explanations for a particular problem against explicit, potentially conflicting evidence generated by the tool. Results indicated that users quickly adopted these visualizations and incorporated the information in their own self-explanations.

The developers of DrRacket, an IDE for novice developers, have invested considerable effort into the design of error messages [MFK11]. Their investigations found that students commonly interpreted the semantics of a highlight to mean "edit here", despite the fact that the authors identified five possible semantic interpretations of the highlight visualization depending on context. They concluded that students needed to understand or infer the *internal parsing strategy of the compiler* to successfully comprehend the error. Among several findings, their work identified that, for students, resolutions for error messages should be distinct from explanations of errors messages, and that IDEs should help students match message terms to code fragments. For interactive static debugging, Flanagan and colleagues developed a system called MrSpidey that augments DrRacket (formerly DrScheme) with visual highlighting and on-demand arrows to explain the flow of values [FFK+96]. Their results suggest that the visual affordances provide easier access to the results of the analysis than the text interface. These findings are also congruent with those of self-explanation, and further imply that compilers can and should expose their internal reasoning processes to the developer.

[14] "Rice's theorem is a general result from 1953 that informally can be paraphrased as stating that all interesting questions about the behavior of programs are *undecidable*" [MS15]. Bummer.

Table 2. Framework Constraints

Criteria	In-Scope	Out-of-Scope
Communication model	Linear (1-way)	Transactional (2-way)
Independent invocations	Use only current snapshot to generate explanations	Use prior compilation history to generate explanations
Closed-world reasoning	Use information derivable from project source code	Add external knowledge sources, such as code search
Explanation soundness	Assume that explanations are sound	Assess utility of incorrect explanations
Co-intervention bias	Self-explanation improvements occur alongside other cognitive processes	Isolates improvements in self-explanation processes and other cognitive processes through meta-analysis of multiple tool designs

Murphy-Hill and Black, describe a set of graphical annotations, called *refactoring annotations*, which are overlaid directly on source code text as an alternative to textual, refactoring-related error messages [MHB12]. Their experiments show that developers can use refactoring annotations to quickly and accurately understand the cause of refactoring errors. Though Murphy-Hill and Black do not situate their tool as a self-explanation tool, understanding the cause of an error is, by definition, a self-explanation process. My own work in visualizing error messages further support Murphy-Hill and Black and demonstrate that annotations directly on source code are an effective technique to support self-explanation [BLCMH14].

HelpMeOut is a *social* recommender system instrumented within the IDE to helps novice programmers understand compiler errors and exception messages [HMBK10]. The system provides suggestions to novice developers using a database of examples that their peers have applied in the past. Rather than using compiler documentation for explanation, a community of expert users author specific explanations for frequently requested fixes. This approach suggests that understanding how humans explain errors to other humans is an appropriate avenue for investigating how static analysis tools *should* explain errors to developers.

12 Scope of Work

In this section, I explore the theoretical framework described in Section 7 as a map that scopes and makes explicit the constraints that guide the research process. I summarize these constraints in Table 2, and provide an avenue for

how future research might address these constraints (the label in parentheses indicates the constraint location in the theoretical framework):

Communication model (2a). The framework assumes a linear communication model. In this model, the IDE is a sender that encodes information into a computer-mediated message to the developer. The developer is then the receiver who decodes the message, perhaps through a noisy channel [AR05]. In this proposal, I am interested in psychological noise, that is, forces that interfere with the ability of the developer to successfully self-explanation an anomaly. Unlike transaction communication models, a limitation of this model is that the communication flows one-way: from the IDE to the developer. Because the current framework does not support feedback from the developer, this bounds our ability to approximate E_{SI} to communications that resemble a linear model, such as answers to questions on Stack Overflow. In other words, the explanations in a linear communication model are like lectures, but future work will investigate how to make explanations more like conversations.

Independent invocations (1b). As a simplification to the theory, the framework treats each static analysis invocation as an independent event. It does not use prior invocations of the static analysis tool in constructing $M(X, R)$, and limits the exposed reasoning in $R|X$. Using prior history is likely to increase the quality of explanations.

Closed-world reasoning (1d). With the closed-world assumption, explanations are only generated for anomalies that can be explained using only the source code. For example, an anomaly that occurs because of file access restrictions to a library would fall outside the scope of this proposal. External libraries are still within the scope of this proposal, as long as source code for those libraries are available within the project.

Explanation soundness (1c). I assume that static analysis tools detect anomalies with perfect precision, but imperfect recall. Put another way, if the tool detects an anomaly, that anomaly will always be a true positive. However, Kulesza and colleagues demonstrate that when systems provide explanations for their recommendations, completeness is more important than soundness [KSB+13]. Therefore, explanations may be useful for developers even when the detected anomaly is a false positive, because the explanation can give the developer a more accurate belief about the trustworthiness of the explanation [LDA09].

Co-intervention bias (3b). Human intelligence occurs through an interplay of simultaneous and complex cognitive processes, such as perception, attention, decision making, and comprehension. For example, if we designed a tool that used diagrams intended to improve comprehension, it could be the case that the diagram is primarily beneficial because it instead makes relevant program elements easier to identify, aiding visual perception. Although my experiments are guided by existing theories of self-explanation, I do not attempt to isolate the self-explanation process from other cognitive mechanisms. Rather, I

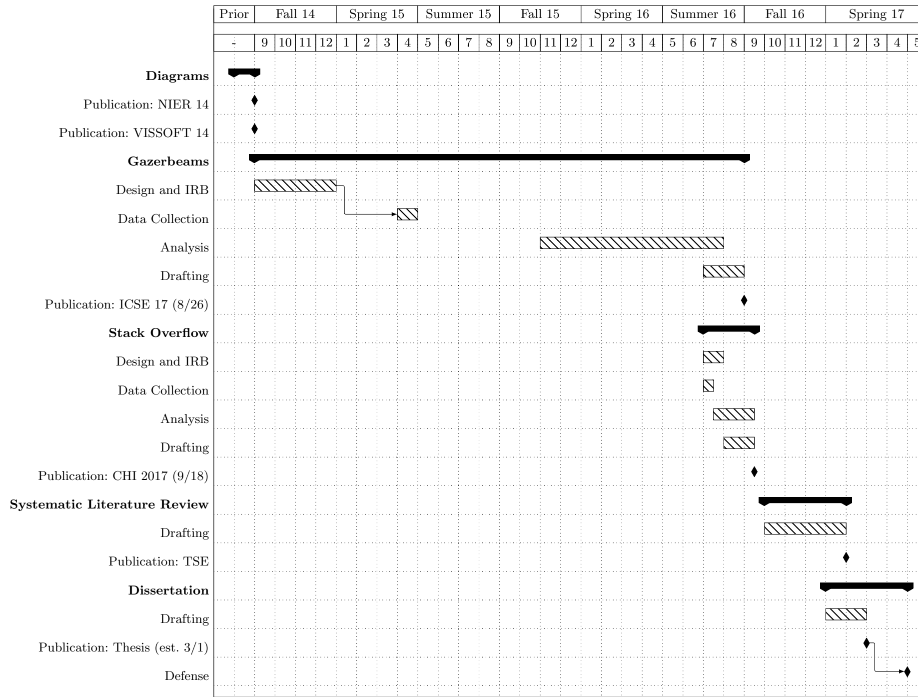

Figure 9. Proposed project plan with publication milestones.

assume implicitly that the self-explanation processes are accompanied with and facilitated by other cognitive processes.

13 Project Plan

Figure 9 is a Gantt chart that lists the *projects* and their associated *tasks* for each semester, partitioned by month. Each project is indicated with a bold label, with an aggregate chart grouping to indicate the estimated overall time of the project. Each project is further divided into high-level tasks, with horizontal bars indicating the estimated duration of the task. Arrows indicate required finish-to-start dependencies between tasks. For example, the Gazerbeams project requires that the Design and IRB be completed and approved before conducting data collection because the experiment involves human subjects.

The deliverable for each project is a *publication*, indicated on the chart as a diamond-shaped milestone. The Diagrams project is considered to be complete. Thus, only the deliverables for this project are shown in the chart. The dissertation is listed as its own project, with the majority of the drafting to be done in Spring 2017.

14 Project Risks and Mitigation

Schedule slips. Particularly in the Spring 2016 and Summer 2016 semesters, projects are back-to-back and must be completed under a compressed schedule. In the Summer of 2016, two projects must be completed essentially in parallel. The mitigation strategy for this slippage is to add an additional semester to the project plan beyond Spring 2017.

Paper rejections. It is also possible that the results of the experiments are insufficient for a top-tier publication venue, that such a venue is missed due to schedule slips, and that the papers are rejected for any number of other opaque reasons. To mitigate this risk, I propose secondary venues for paper submissions. For ICSE and FSE, these venues are ICPC, ICSME, and VL/HCC. For CSCW, alternative venues are SPLASH and ICSE Software Engineering in Practice (SEIP).

15 Addendum: Thesis Contract

To address the committee feedback from the thesis proposal presentation, I will provide the committee with the following revised deliverables upon completion of the dissertation:

- ☐ Dissertation chapter on `Diagrams` study.
- ☐ Dissertation chapter on `Gazerbeams` study.
- ☐ Dissertation chapter on `Stack Overflow` study.
- ☐ Dissertation chapter containing a comprehensive, systematic literature review on program analysis tool user interfaces.

These revised deliverables supersede any commitments made prior to committee deliberations.

References

AA13. Roger Azevedo and Vincent Aleven, editors. *International Handbook of Metacognition and Learning Technologies*, volume 28 of *Springer International Handbooks of Education*. Springer New York, New York, NY, 2013.

AB15. Amjad Altadmri and Neil C.C. Brown. 37 Million Compilations: Investigating Novice Programming Mistakes in Large-Scale Student Data. In *Proceedings of the 46th ACM Technical Symposium on Computer Science Education - SIGCSE '15*, pages 522–527, New York, New York, USA, February 2015. ACM Press.

AGD05. Christopher Anderson, Paola Giannini, and Sophia Drossopoulou. Towards type inference for javascript. In *ECOOP 2005-Object-Oriented Programming*, pages 428–452. Springer, 2005.

Ale02. V Aleven. An effective metacognitive strategy: learning by doing and explaining with a computer-based Cognitive Tutor. *Cognitive Science*, 26(2):147–179, April 2002.

APM+07. Nathaniel Ayewah, William Pugh, J. David Morgenthaler, John Penix, and YuQian Zhou. Evaluating static analysis defect warnings on production software. In *Proceedings of the 7th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering - PASTE '07*, pages 1–8, New York, New York, USA, June 2007. ACM Press.

AR05. Ronald B. Adler and George Rodman. *Understanding Human Communication*. Oxford University Press, 9th edition, 2005.

AT03. Shaaron Ainsworth and Andrea Th Loizou. The effects of self-explaining when learning with text or diagrams. *Cognitive Science*, 27(4):669–681, August 2003.

BLCMH14. Titus Barik, Kevin Lubick, Samuel Christie, and Emerson Murphy-Hill. How developers visualize compiler messages: A foundational approach to notification construction. In *2014 Second IEEE Working Conference on Software Visualization*, pages 87–96, September 2014.

BLMH15. Titus Barik, Kevin Lubick, and Emerson Murphy-Hill. Commit bubbles. pages 631–634, May 2015.

BPB95. Katerine Bielaczyc, Peter L Pirolli, and Ann L Brown. Training in Self-Explanation and Self-Regulation Strategies: Investigating the Effects of Knowledge Acquisition Activities on Problem Solving. *Cognition and Instruction*, 13(2):221–252, January 1995.

BR01. Thomas Ball and Sriram K Rajamani. The slam toolkit. In *Computer aided verification*, pages 260–264. Springer, 2001.

Bro83. P. J. Brown. Error messages: the neglected area of the man/machine interface. *Communications of the ACM*, 26(4):246–249, April 1983.

BWJMH14. Titus Barik, Jim Witschey, Brittany Johnson, and Emerson Murphy-Hill. Compiler error notifications revisited: an interaction-first approach for helping developers more effectively comprehend and resolve error notifications. In *Companion Proceedings of the 36th International Conference on Software Engineering - ICSE Companion 2014*, pages 536–539, May 2014.

BZR+10. Andrew Bragdon, Robert Zeleznik, Steven P. Reiss, Suman Karumuri, William Cheung, Joshua Kaplan, Christopher Coleman, Ferdi Adeputra, and Joseph J. LaViola. Code bubbles: A Working Set-based Interface for Code Understanding and Maintenance. In *Proceedings of the 28th international conference on Human factors in computing systems - CHI '10*, pages 2503–2512, New York, New York, USA, April 2010. ACM Press.

CBL+89. Michelene T.H. Chi, Miriam Bassok, Matthew W. Lewis, Peter Reimann, and Robert Glaser. Self-Explanations: How Students Study and Use Examples in Learning to Solve Problems. *Cognitive Science*, 13(2):145–182, April 1989.

CC05. L. Cooke and E. Cuddihy. Using eye tracking to address limitations in think-aloud protocol. In *IPCC 2005. Proceedings. International Professional Communication Conference, 2005.*, pages 653–658. IEEE, 2005.

CDCL94. Michelene T.H. Chi, Nicholas De Leeuw, Mei-Hung Chiu, and Christian Lavancher. Eliciting self-explanations improves understanding. *Cognitive Science*, 18(3):439–477, July 1994.

CE07. Kent J. Crippen and Boyd L. Earl. The impact of web-based worked examples and self-explanation on performance, problem solving, and self-efficacy. *Computers & Education*, 49(3):809–821, November 2007.

CKK+12. Pascal Cuoq, Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. Frama-c. In *Software Engineering and Formal Methods*, pages 233–247. Springer, 2012.

CKM06. Michael J. Coblenz, Andrew J. Ko, and Brad A. Myers. JASPER: An Eclipse plug-in to facilitate software maintenance tasks. In *Proceedings of the 2006 OOPSLA workshop on eclipse technology eXchange - eclipse '06*, pages 65–69, New York, New York, USA, October 2006. ACM Press.

CVDK07. Mauro Cherubini, Gina Venolia, Rob DeLine, and Andrew J. Ko. Let's go to the whiteboard. In *Proceedings of the SIGCHI conference on Human factors in computing systems - CHI '07*, page 557, New York, New York, USA, April 2007. ACM Press.

Din95. Thomas A Dingus. A Meta-Analysis of Driver Eye-Scanning Behavior While Navigating. *Proceedings of the Human Factors and Ergonomics Society Annual Meeting*, 39(17):1127–1131, October 1995.

DR10. Robert DeLine and Kael Rowan. Code canvas: zooming towards better development environments. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - ICSE '10*, volume 2, pages 207–210, New York, New York, USA, May 2010. ACM Press.

EH10. Nabil El Boustani and Jurriaan Hage. Corrective hints for type incorrect generic Java programs. In *Proceedings of the ACM SIGPLAN 2010 workshop on Partial evaluation and program manipulation - PEPM '10*, page 5, New York, New York, USA, January 2010. ACM Press.

FAFH09. Michael Furr, Jong-hoon David An, Jeffrey S Foster, and Michael Hicks. Static type inference for ruby. In *Proceedings of the 2009 ACM symposium on Applied Computing*, pages 1859–1866. ACM, 2009.

FFK+96. Cormac Flanagan, Matthew Flatt, Shriram Krishnamurthi, Stephanie Weirich, and Matthias Felleisen. Catching bugs in the web of program invariants. In *Proceedings of the ACM SIGPLAN 1996 conference on Programming language design and implementation - PLDI '96*, volume 31, pages 23–32, New York, New York, USA, May 1996. ACM Press.

FHdJ90. Monica G.M. Ferguson-Hessler and Ton de Jong. Studying Physics Texts: Differences in Study Processes Between Good and Poor Performers. *Cognition and Instruction*, 7(1):41–54, March 1990.

GP96. T.R.G. Green and M. Petre. Usability Analysis of Visual Programming Environments: A Cognitive Dimensions Framework. *Journal of Visual Languages & Computing*, 7(2):131–174, 1996.

HF14. Austin Z. Henley and Scott D. Fleming. The patchworks code editor. In *Proceedings of the 32nd annual ACM conference on Human factors in computing systems - CHI '14*, pages 2511–2520, New York, New York, USA, April 2014. ACM Press.

HMBK10. Björn Hartmann, Daniel MacDougall, Joel Brandt, and Scott R. Klemmer. What would other programmers do. In *Proceedings of the 28th International Conference on Human Factors in Computing Systems - CHI '10*, pages 1019–1028, New York, New York, USA, April 2010. ACM Press.

HMRM03. Maria Hristova, Ananya Misra, Megan Rutter, and Rebecca Mercuri. Identifying and correcting Java programming errors for introductory computer science students. *ACM SIGCSE Bulletin*, 35(1):153, January 2003.

HP04. David Hovemeyer and William Pugh. Finding bugs is easy. *ACM SIGPLAN Notices*, 39(12):92, December 2004.

JPMHH15. Brittany Johnson, Rahul Pandita, Emerson Murphy-Hill, and Sarah Heckman. Bespoke tools: adapted to the concepts developers know. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering - ESEC/FSE 2015*, pages 878–881, New York, New York, USA, August 2015. ACM Press.

JSMHB13. Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. Why don't software developers use static analysis tools to find bugs? In *2013 35th International Conference on Software Engineering (ICSE)*, pages 672–681. IEEE, May 2013.

KK03. Sarah K. Kummerfeld and Judy Kay. The neglected battle fields of syntax errors. pages 105–111, January 2003.

KL87. Alex Kass and David Leake. Types of explanations. Technical report, DTIC Document, 1987.

KM04. Andrew J. Ko and Brad A. Myers. Designing the whyline. In *Proceedings of the 2004 conference on Human factors in computing systems - CHI '04*, pages 151–158, New York, New York, USA, April 2004. ACM Press.

KSB+13. Todd Kulesza, Simone Stumpf, Margaret Burnett, Sherry Yang, Irwin Kwan, and Weng-Keen Wong. Too much, too little, or just right? Ways explanations impact end users' mental models. In *2013 IEEE Symposium on Visual Languages and Human Centric Computing*, pages 3–10. IEEE, September 2013.

KV08. Johannes Kinder and Helmut Veith. Jakstab: A static analysis platform for binaries. In *Computer Aided Verification*, pages 423–427. Springer, 2008.

LBM14. Tom Lieber, Joel R. Brandt, and Rob C. Miller. Addressing misconceptions about code with always-on programming visualizations. In *CHI '14*, pages 2481–2490, April 2014.

LDA09. Brian Y. Lim, Anind K. Dey, and Daniel Avrahami. Why and why not explanations improve the intelligibility of context-aware intelligent systems. In *Proceedings of the 27th international conference on Human factors in computing systems - CHI 09*, pages 2119–2129, New York, New York, USA, April 2009. ACM Press.

Leg14. Cristine H. Legare. The Contributions of Explanation and Exploration to Children's Scientific Reasoning. *Child Development Perspectives*, 8(2):101–106, June 2014.

LFGC07. Benjamin S Lerner, Matthew Flower, Dan Grossman, and Craig Chambers. Searching for type-error messages. In *ACM SIGPLAN Notices*, volume 42, pages 425–434. ACM, 2007.

LNZ15. David Lo, Nachiappan Nagappan, and Thomas Zimmermann. How practitioners perceive the relevance of software engineering research. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering - ESEC/FSE 2015*, pages 415–425, New York, New York, USA, August 2015. ACM Press.

LS87. Jill Larkin and Herbert Simon. Why a Diagram is (Sometimes) Worth Ten Thousand Words. *Cognitive Science*, 11(1):65–100, January 1987.

MFK11. Guillaume Marceau, Kathi Fisler, and Shriram Krishnamurthi. Mind your language: On novices' interactions with error messages. In *Proceedings of the 10th SIGPLAN Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software - ONWARD '11*, pages 3–17, New York, New York, USA, October 2011. ACM Press.

MHB12. E Murphy-Hill and A P Black. Programmer-Friendly Refactoring Errors. *Software Engineering, IEEE Transactions on*, 38(6):1417–1431, 2012.

MLL05. Michael Martin, Benjamin Livshits, and Monica S. Lam. Finding application errors and security flaws using PQL. *ACM SIGPLAN Notices*, 40(10):365, October 2005.

MMM+11. Lena Mamykina, Bella Manoim, Manas Mittal, George Hripcsak, and Björn Hartmann. Design lessons from the fastest Q&A site in the west. In *Proceedings of the 2011 annual conference on Human factors in computing systems - CHI '11*, page 2857, New York, New York, USA, May 2011. ACM Press.

MS15. Anders Møller and Michael I. Schwartzbach. *Lecture Notes on Static Program Analysis*. 2015.

NM90. Hwee Tou Ng and Raymond J Mooney. On the Role of Coherence in Abductive Explanation. *Proceedings of the Eight National Conference on Artificial Intelligence (AAAI-90)*, pages 337–342, 1990.

NPM08. Marie-Hélène Nienaltowski, Michela Pedroni, and Bertrand Meyer. Compiler error messages: What can help novices? In *ACM SIGCSE Bulletin*, pages 168–172. ACM, February 2008.

NSMB12. Seyed Mehdi Nasehi, Jonathan Sillito, Frank Maurer, and Chris Burns. What makes a good code example?: A study of programming Q&A in StackOverflow. In *2012 28th IEEE International Conference on Software Maintenance (ICSM)*, pages 25–34. IEEE, September 2012.

Par98. David Lorge Parnas. Successful software engineering research. *ACM SIGSOFT Software Engineering Notes*, 23(3):64–68, May 1998.

Pri15. David Pritchard. Frequency distribution of error messages. In *Proceedings of the 6th Workshop on Evaluation and Usability of Programming Languages and Tools - PLATEAU 2015*, pages 1–8, New York, New York, USA, October 2015. ACM Press.

RH08. Per Runeson and Martin Höst. Guidelines for conducting and reporting case study research in software engineering. *Empirical Software Engineering*, 14(2):131–164, December 2008.

RN08. P. Reimann and C. Neubert. The role of self-explanation in learning to use a spreadsheet through examples. *Journal of Computer Assisted Learning*, 16(4):316–325, October 2008.

Sal09. Johnny Saldaña. *The Coding Manual for Qualitative Researchers*. SAGE Publications, 2009.

Shn77. B. Shneiderman. Measuring computer program quality and comprehension. *International Journal of Man-Machine Studies*, 9(4):465–478, 1977.

SO95. Keith Stenning and Jon Oberlander. A Cognitive Theory of Graphical and Linguistic Reasoning: Logic and Implementation. *Cognitive Science*, 19(1):97–140, January 1995.

SS86. James C. Spohrer and Elliot Soloway. Novice mistakes: are the folk wisdoms correct? *Communications of the ACM*, 29(7):624–632, July 1986.

SSE+14. Hyunmin Seo, Caitlin Sadowski, Sebastian Elbaum, Edward Aftandilian, and Robert Bowdidge. Programmers' build errors: a case study (at google). In *Proceedings of the 36th International Conference on Software Engineering - ICSE 2014*, pages 724–734, New York, New York, USA, May 2014. ACM Press.

Tra10. V. Javier Traver. On compiler error messages: What they say and what they mean. *Advances in Human-Computer Interaction*, 2010:1–26, 2010.

vDD11. Daniel von Dincklage and Amer Diwan. Integrating program analyses with programmer productivity tools. *Software: Practice and Experience*, 41(7):817–840, June 2011.

VJ93. Kurt VanLehn and M. Randolph Johnes. What mediates the self-explanation effect? Knowledge gaps, schemas or analogies? In *Proceedings of the Fifteenth Annual Conference of the Cognitive Science Society*, pages 1034–1039, 1993.

WG05. Michael B. W. Wolfe and Susan R. Goldman. Relations Between Adolescents' Text Processing and Reasoning. *Cognition and Instruction*, 23(4):467–502, December 2005.

WM91. Peter C. Wright and Andrew F. Monk. A cost-effective evaluation method for use by designers. *International Journal of Man-Machine Studies*, 35(6):891–912, December 1991.