ABSTRACT

BARIK, TITUS. Error Messages as Rational Reconstructions. (Under the direction of Emerson Murphy-Hill.)

Program analysis tools apply elegant algorithms—such as static analysis, model checking, and type inference—on source code to help developers resolve compiler errors, apply optimizations, identify security vulnerabilities, and reason about the logic of the program. In integrated development environments, program analysis tools provide feedback about their internal diagnostics to developers through error messages, using a variety of textual and visual mechanisms, such as error listings, tooltips, and source code underlined with red squiggles. The design of user-friendly error messages is important because error messages are the primary communication channel through which tools provide feedback to developers.

Despite the intended utility of these tools, the error messages these tools produce are cryptic, frustrating, and generally unhelpful to developers as they attempt to understand and resolve the messages. Existing approaches in programming language research have attempted to surface the internal reasoning process of program analysis tools and present these details to the developer to aid their comprehension process. However, we argue that the tool-centric perspective of simply revealing details in the error message about the tools' internal algorithms is insufficient: the fundamental problem is that computational tools do not reason about the causes of an identified error in the same way as the developer who attempts to understand and reconstruct why the tool produced that particular error.

The goal of this research is to investigate these misalignments through the theoretical framework of rational reconstruction—a model for identifying rationales, or reasons, for arriving a particular conclusion—to the domain of error messages in program analysis tools. Essentially, a rational reconstruction of an error message would present rationales to the developer from a human-centered perspective that aligns with the developers' reasoning process, irrespective of the underlying algorithm or process used by the program analysis tool to identify the error. Through rational reconstructions, we can identify how to design error messages that are most useful for developers, rather than those that are most convenient for the tool.

The thesis of this dissertation is that difficulties interpreting error messages

produced by program analysis tools are a significant predictor of developers' inabilities to resolve defects, and that these difficulties in interpreting error messages can be explained by framing error messages as insufficient rational reconstructions—in both visual and textual output presentations. This dissertation advances knowledge about developer comprehension during error message tasks and defends the claims of this thesis through three studies, evaluated through the theoretical framework of rational reconstruction:

1. To learn how developers use error messages during their own rational reconstructions, we conducted an eye tracking study in which participants resolved common defects within the Eclipse development environment. We found that the difficulty of reading these messages is comparable to the difficulty of reading source code, that difficulty reading error messages significantly predicts participants' task performance, and that participants allocate a substantial portion of their total task to reading error messages (13%-25%).

2. To learn how developers construct and are aided by diagrammatic rational reconstructions, or explanatory visualizations, we conducted a usability design experiment in which developers diagrammatically annotated and explained source code listings for compiler error messages. We found that explanatory visualizations are used intuitively by developers in their own self-explanations of error messages, and that these visualizations are significantly different from baseline visualizations in how they explicate relationships between program elements in the source code.

3. To learn how rational reconstructions aid developer comprehension in text presentations, we conducted a study to analyze confusing error messages on Stack Overflow against human-authored reconstructions of those error messages. We analyzed these answers through a form of rational reconstruction, Toulmin's argument model, and found that developers significantly preferred error messages with proper argument structures over deficient arguments, but will prefer deficient arguments if they provide a *resolution* to the problem. We found that human-authored explanations converge to argument structures that either offer a simple resolution, or employ a proper simple or extended argument structure.

The dissertation concludes with implications and design guidelines for practitioners who wish to improve the usability of error messages for program analysis tools. To assess and operationalize these guidelines, we developed a proof-of-concept compiler called Rational TypeScript—a modified Microsoft TypeScript compiler that presents error messages as rational reconstructions. A focus group discussion conducted with professional software developers suggested that Rational TypeScript messages were more helpful than baseline TypeScript messages, particularly with developers who only sporadically program with TypeScript. Although full-time TypeScript developers generally preferred the brevity of baseline error messages for routine errors, they nevertheless indicated that rational reconstructions would be useful as a presentation option for error messages when working with unfamiliar code.[1]

[1]Draft version: 20180318-02

Error Messages as Rational Reconstructions

by
Titus Barik

A dissertation submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Doctor of Philosophy

Computer Science

Raleigh, North Carolina

2018

APPROVED BY:

Christopher Parnin	James Lester
Jing Feng	Shriram Krishnamurthi

Emerson Murphy-Hill
Chair of Advisory Committee

DEDICATION

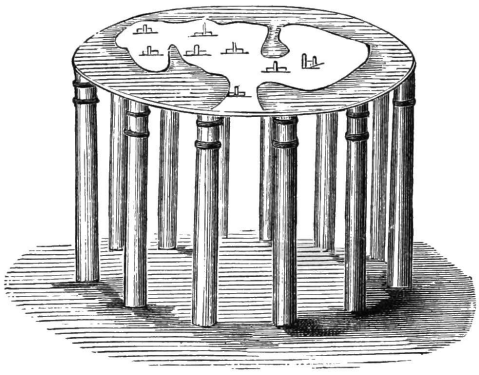
In memory of Mark Lusher (1972–2015).
Then you will shine among them like stars in the sky.

BIOGRAPHY

Titus Barik obtained a Bachelor of Science in Computer Engineering from the Georgia Institute of Technology in 2004. In Atlanta, he spent several years working in industry as a Project Engineer: in factory automation, process control systems, and logistics. While working full-time, Titus obtained a Master of Engineering degree from North Carolina State University in 2010. He obtained his Professional Engineering (P.Eng) license in 2011.

Titus moved to Raleigh, North Carolina and returned to academia at North Carolina State University in 2010 to pursue a PhD in Computer Science. During his academic career, Titus interned at Google (2013 and 2014) and Microsoft Research (2015). He also worked as a Research Scientist at ABB—from 2014 to 2016—to improve developer productivity.

Titus has many research interests, evidenced through publications beyond his core research areas of software engineering and human-computer interaction. He has published in information visualization, cognitive modeling, computer science education, and digital games research. Titus is particularly curious about the role of programming as a form of play, self-discovery, and artistic expression.



The Earth of the Veda Priests
Source: "How the Earth Was Regarded in Old Times,"
Popular Science Monthly Volume 10 (1876)

ACKNOWLEDGEMENTS

I am the architect of my own imprisonment.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF LISTINGS

1 | My Thesis

> People spend all their time making nice things and then other people come along and break them.
>
> The Doctor

Difficulties interpreting error messages produced by program analysis tools are a significant predictor of developers' inabilities to resolve defects: difficulties in interpreting error messages can be explained by framing error messages as insufficient rational reconstructions, in both visual and textual output presentations.[1]

[1] Supporting documentation attached.

2 | Introduction

> Our lives are different to anybody else's. That's the exciting thing. Nobody in the universe can do what we're doing.
>
> The Doctor

2.1 Problem

Program analysis tools are intended to help developers identify problems in their source code: they pinpoint unsafe or undesirable runtime behavior, enforce conformance to programming language specifications, and flag stylistic issues that hinder the readability of the code. These tools could be immensely useful for developers because they identify problems that are subtle to spot through manual inspection. Unfortunately, research has found that the output program of analysis tools—error messages—are confusing, unconstructive, or incomprehensible. As a result, developers spend unnecessary effort in comprehending and resolving the defect identified by the tool or simply abandon otherwise useful tools because they can't understand the error messages.

What makes these error message so confusing for developers?

2.2 Examples

Program analysis tools communicate these problems to developers through error messages, using various text and visual representations. But rather than attempt to

characterize precisely what an error message is (we'll do that in Chapter 3), let's look at concrete examples of difficult to comprehend error messages, through program analysis tools in both text and visual medium. I've organized these examples from the more familiar error messages—such as those produced by program analysis in compilers and bug-pattern tools—to the more elaborate error messages produced by program analysis tools that employ formal methods, such as automated theorem provers. I've selected examples for which I found some third-party confirmation that the error message from the program analysis tool is confusing: from examples in research papers, bug reports, or mailing list and forum discussions.

2.2.1 Portable C Compiler

We'll start the PCC compiler from 1979, in which a developer attempts to writes the venerable "Hello, world!" program. Here's the source code for this program:

```
1 #include <stdio.h>
2
3 int main() {
4   printf("Hello, world!\n")
5 }
```

Despite the simplicitly, this program is useful for sanity checking: seeing the words "Hello, world!" appear on the screen means that the basic libraries are in the right place, and that the source code can compile, execute, and successfully send output to a console.

However, this source listing contains an error: the rules of the C programming specification require that all statements end with a semicolon (;), and the code is missing one at the end of line 4. The program analysis within the compiler identifies this error and outputs an error message:

```
hello.c, line 5: syntax error
hello.c, line 5: cannot recover from earlier errors: goodbye!
error: /usr/libexec/ccom terminated with status 1
```

It turns out this error message isn't technically wrong, but it is misleading. For instance, it claims that the error is on line 5, but it seems to me like the error should actually be reported for line 4. Regardless, the error doesn't provide any reasons for how it came to this conclusion: it leaves it to developer to identify what causes the syntax error. PCC then unhelpfully exits with the message `goodbye!`, followed with a termination status that is only useful to the operating system.

Is it fair to use a compiler written in the 1970s to illustrate confusing error messages? Probably not. But is good baseline for error messages. Despite its historical interest, the error message has the minimal components of modern error messages: A location indicating the problem—`hello.c, line 5`, and a description of the problem. And many of the design decisions within PCC continue to influence modern program analysis tools.

2.2.2 GNU Compiler Collection

Let's now look at `gcc`, a modern C compiler that is part of the GNU Compiler Collection. Here's the error message for the same "Hello, world!" program in `gcc`:

```
hello.c: In function ‘main’:
hello.c:5:1: error: expected ‘;’ before ‘}’ token
 }
 ^
```

The gcc compiler is an improvement over pcc: rather than a vague `syntax error` it does tell us the the semicolon is the actual token that is missing. The program analysis tool also adds a bit of color to the error message, and some context about the source code in which the error appears (lines 3 and 4).

Still, the error message is disorienting from the developers' perspective, because it emits a location that *follows* the problem, rather than the location that immediately *preceeds* it. The easiest way to illustrate this problem is to contrast `gcc` with a compiler that does this correctly, at least in this case.

Here's the output from `clang`, part of LLVM:

```
hello.c:4:28: error: expected ';' after expression
  printf("Hello, world!\n")
                           ^
                           ;
1 error generated.
```

The problem is now apparent, and also matches the way developers would think the problem: "I'm missing a semicolon at the end of line 4", rather than forcing us to mentally realign our thinking process, "I'm missing a semicolon at immediately before the brace on line 5, but it would be strange to add a semi-colon at the start of the line, so it must be even before that. The compiler must actually mean the end of line 4."

Syntax problems like these are nuisances for expert developers—even with the original error from the 1970s pcc compiler—but it's more work than we should

have to do. For novices, however, error messages like these are paralyzing. The reasoning of the compiler, strictly applied, can lead to idiosyncractic but nevertheless conformant fixes like this one:

```
#include <stdio.h>

int main() {
  printf("Hello, world!\n")
;}
```

2.2.3 Oracle Java Compiler (OpenJDK)

The following is a source listing in Java:

```
class Toy {
    Toy() throws Exception { }
}

class Kite extends Toy {
}
```

The source listing results in following error from the OpenJDK compiler:

```
Kite.java:5: error: unreported exception Exception in default constructor
class Kite extends Toy {
^
1 error
```

This error message was reported as a bug in JDK-4071337: “misleading error message when superclass constructor has throws clause.” The report argues that the error message is possible to understand only if you already know the solution to the error.

Let’s construct a rationale for what could cause this error message. First, we note that the error message a default constructor, but no such constructor explicitly appears in `Kite` class. This is probably why the bug marker ^ points to the class itself: default constructors are implicit and therefore there is no constructor we can actually point to directly. Let’s change the source code to add this constructor explicitly, and hope that it compels the program analysis to reveal more of its rationale:

```
class Toy {
    Toy() throws Exception { }
}
```

```
4
5  class Kite extends Toy {
6    Kite() { }
7  }
```

We compile again. Not only does the new error message point to a different location, the message now reveals more information about the problem.

```
Kite2.java:6: error: unreported exception Exception;
must be caught or declared to be thrown
  Kite() { }
         ^
1 error
```

Thus, we add a `throws Exception` in response to this explanation:

```
1  class Toy {
2      Toy() throws Exception { }
3  }
4
5  class Kite extends Toy {
6      Kite() throws Exception { }
7  }
```

This last attempt is accepted by the compiler, and now we can retrospectively find reasons for why. But why did we have to explicitly create a default contructor? Don't these get created automatically if we don't write one ourselves? It turns out that according to the Java Language Specification 9 (8.8.9):

> It is a compile-time error if a default constructor is implicitly declared but the superclass does not have an accessible constructor that takes no arguments and has no throws clause.

Thus, in the case when the default constructor in the superclass has a throws clause, the compiler is forbidden from automatically generating a default constructor on behalf of the developer.

2.2.4 clang Compiler (LLVM)

Let us now look an source listing in C++, named, `ptrcopy.cpp`, in which we'll use the `clang` compiler from LLVM:

```
#include <iostream>
#include <memory>
#include <vector>

int main() {
    std::vector<std::unique_ptr<int>> foo;
    std::vector<std::unique_ptr<int>> bar = foo;
}
```

The intent of this program is presumably to do some sort of copying, from the contents of `bar` to the contents of `foo`. But attempting to do this generates an unwieldily compiler error:

```
In file included from ptrcopy.cpp:2:
In file included from /usr/bin/../lib/gcc/x86_64-linux-gnu/7.2.0/../../../
    ../include/c++/7.2.0/memory:64:
/usr/bin/../lib/gcc/x86_64-linux-gnu/7.2.0/../../../../include/c++/7.2.0/
    bits/stl_construct.h:75:38: error: call to deleted constructor
    of
      'std::unique_ptr<int, std::default_delete<int> >'
    { ::new(static_cast<void*>(__p)) _T1(std::forward<_Args>(__args)...); }
                                       ^   ~~~~~~~~~~~~~~~~~~~~~~~~~~~
/usr/bin/../lib/gcc/x86_64-linux-gnu/7.2.0/../../../../include/c++/7.2.0/
    bits/stl_uninitialized.h:83:8: note: in instantiation of function
    template specialization
      'std::_Construct<std::unique_ptr<int, std::default_delete<int> >,
      const std::unique_ptr<int, std::default_delete<int> > &>'
      requested here
            std::_Construct(std::__addressof(*__cur), *__first);
                 ^
/usr/bin/../lib/gcc/x86_64-linux-gnu/7.2.0/../../../../include/c++/7.2.0/
    bits/stl_uninitialized.h:134:2: note: in instantiation of function
    template specialization
      'std::__uninitialized_copy<false>::__uninit_copy<__gnu_cxx::
      __normal_iterator<const std::unique_ptr<int,
      std::default_delete<int> > *,
      std::vector<std::unique_ptr<int, std::default_delete<int> >,
      std::allocator<std::unique_ptr<int, std::default_delete<int> > > >
      >, std::unique_ptr<int,
      std::default_delete<int> > *>' requested here
        __uninit_copy(__first, __last, __result);
        ^
/usr/bin/../lib/gcc/x86_64-linux-gnu/7.2.0/../../../../include/c++/7.2.0/
    bits/stl_uninitialized.h:289:19: note: in instantiation of function
    template specialization
      'std::uninitialized_copy<__gnu_cxx::__normal_iterator<const
      std::unique_ptr<int, std::default_delete<int> > *,
      std::vector<std::unique_ptr<int,
```

```
19        std::default_delete<int> >, std::allocator<std::unique_ptr<int,
          ↪  std::default_delete<int> > > >, std::unique_ptr<int,
          ↪  std::default_delete<int> > *>' requested here
20      { return std::uninitialized_copy(__first, __last, __result); }
21                    ^
22  /usr/bin/../lib/gcc/x86_64-linux-gnu/7.2.0/../../../../include/c++/7.2.0/
    ↪  bits/stl_vector.h:331:9: note: in instantiation of function template
    ↪  specialization
23        'std::__uninitialized_copy_a<__gnu_cxx::__normal_iterator<const
          ↪  std::unique_ptr<int, std::default_delete<int> > *,
          ↪  std::vector<std::unique_ptr<int,
24        std::default_delete<int> >, std::allocator<std::unique_ptr<int,
          ↪  std::default_delete<int> > > >, std::unique_ptr<int,
          ↪  std::default_delete<int> > *,
25        std::unique_ptr<int, std::default_delete<int> > >' requested here
26            std::__uninitialized_copy_a(__x.begin(), __x.end(),
27                 ^
28  ptrcopy.cpp:7:43: note: in instantiation of member function
    ↪  'std::vector<std::unique_ptr<int, std::default_delete<int> >,
    ↪  std::allocator<std::unique_ptr<int,
29        std::default_delete<int> > > >::vector' requested here
30    std::vector<std::unique_ptr<int>> bar = foo;
31                                            ^
32  /usr/bin/../lib/gcc/x86_64-linux-gnu/7.2.0/../../../../include/c++/7.2.0/
    ↪  bits/unique_ptr.h:388:7: note: 'unique_ptr' has been explicitly marked
    ↪  deleted here
33        unique_ptr(const unique_ptr&) = delete;
34        ^
35  1 error generated.
```

Squinting at this error message, we may be able to derive even clues to help us reconstruct an explanation for what the actual problem might be. For example, we can scan the error message to see that the files of interest are `memory` (Line 2), stl_construct.h (Line 3), `stl_uninitialized.h` (Line 7, Line 11, and Line 17), `stl_vector` (Line 22), and the source listing we wrote, `ptrcopy.cpp` (Line 28). Since the only file the developer has actually touched is `ptrcopy.cpp`, we can infer that the error locations are presented backwards: something happens deep inside `memory`, and this problem bubbles up through the various C++ files until we arrive back at the cause in `ptrcopy`. Although it's true that the problem doesn't occur, strictly speaking, until we reach `memory`, it's not very helpful to have a problem identified within a library that we didn't even write.

What if the problem analysis instead presented an error message from the perspective of the developer, rather than the perspective of the compiler? Here's an example of what such a message might look like:

```
ptrcopy.cpp: cannot construct 'bar' from 'foo':
foo's template type is non-copyable
    std::vector<std::unique_ptr<int>> bar = foo;
                                            ^
```

I think this version of the error message is an interesting contrast for several reasons.[1] First, it pinpoints a location that the developer wrote. Second, it argues for a presentation that is in many ways less precise than the original LLVM error message, yet much easier to read quickly. Third, even with this loss of precision, it's obvious what the problem is: `unique_ptr` is not copyable, and so it can only be moved. Thus, trying to assign a vector of `unique_ptr` to another vector would mean that somewhere in the vector source code the unique pointer would necessarily need to be copied.

2.2.5 eslint for JavaScript

ESLint is a pluggable bug-finding utility for JavaScript. By pluggable, we mean: 1) that developers can add custom bug-findings rules to extend the capability of the tool, and 2) the tool is intended to be bundled and used within order products, such as integrated development environments and build systems. Thus, the tool supports multiple output formats for presentation. Our goal in the section is to illustrate that sometimes the "medium is the message" [242]. That is to say, whether an error message is appropriate depends not only on message content how the error message is situated within the broader environment.

Let's look at an example of this. Consider what happens when we apply ESLint to the following JavaScript file:

```
1 const who = 'Titus';
2 console.log('Hello, ' + who + '!');
```

Depending on the default formatter, this causes ESLint emits the following error message:

```
hello.js: line 2, col 13, Error - Unexpected string concatenation.
↪  (prefer-template)
```

The solution to this error is to use a template literal:

[1] A Stack Overflow refers to this version of the error message as their "ideal error message" to explain the problem: https://stackoverflow.com/questions/8779521/tools-to-generate-higher-quality-error-messages-for-template-based-code

```
1 const who = 'Titus';
2 console.log(`Hello, ${who}!`);
```

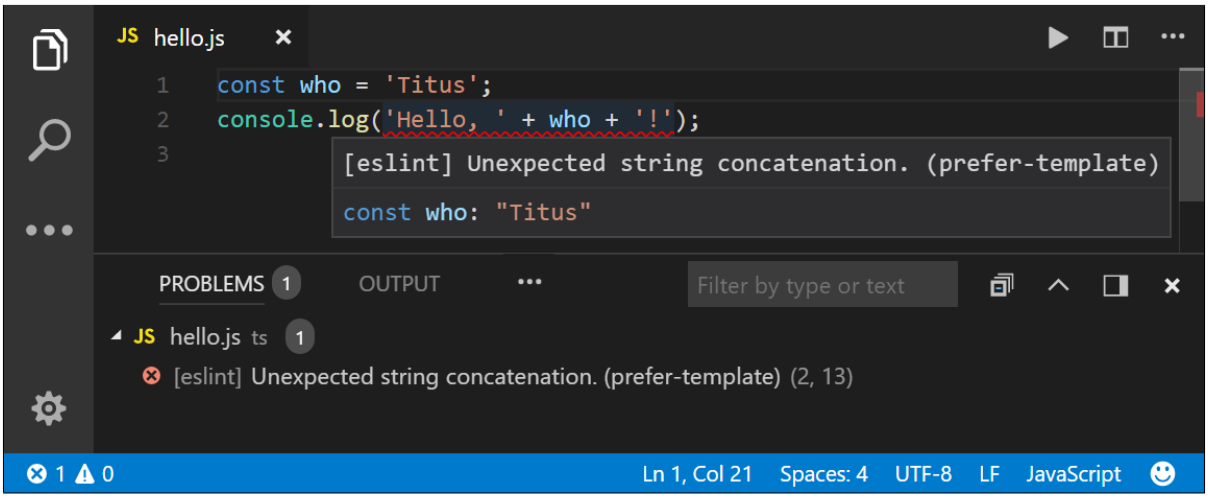
In contrast to the error messages from OpenJDK, GCC, LLVM. the error message from ESLint almost seems like a regression to an earlier era of compilers. Unlike either of these tools (Section 2.2.3), the ESLint error message does not provide a contextual code snippet. Unlike GCC and LLVM, ESLint does not colorize the output. And `prefer-template` seems like an internal error code, which doesn't help the developer.

If we were solely targeting a console environment, we might consider a relatively simple improvement to this error message that adds a rationale for the error:

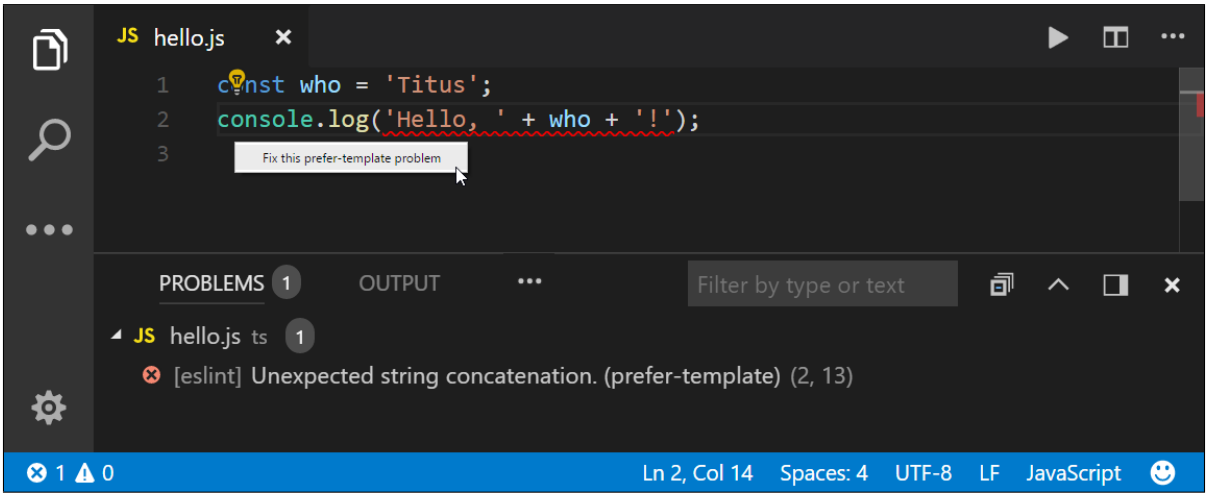
```
× [eslint] Unexpected string concatenation {prefer-template} (2, 13)
           Why?  Template strings give you a readable, concise syntax
                 with proper newlines and string interpolation features.
                 (see: airbnb.io/javascript/#es6-template-literals)
```

Given the rationale for the error, we see that the reason ESLint recommends this change is because the Airbnb JavaScript standard considers template strings to be more readable. An extended explanation is offered through a link, and gives examples of converting string concatenation to template literals.

Although ESLint supports console output as a last resort, ESLint is really intended to be used as program analysis building block to be incorporated within other programming and build environments. To illustrate this, let's consider against the default formatter, but this time rendered within Visual Studio Code (Figure 2.1 on the following page). In Figure 2.1a, we see that ESLint presents its error message as a tooltip when the developer hovers over the red wavy underline. Moreover, the developer can double-click the error message in the problems pane to directly navigate to Line 2 of `hello.js`. With the single-line affordances in the problems pane, it now makes sense to present a terse error messages that the developer can use to quickly navigate to the relevant code context. Similarly, the somewhat cryptic (`prefer-template`) text in the default error message now becomes an interactive affordance that allows the developer to automatically repair the code in their IDE (Figure 2.1b). Finally, it's now clear that providing a contextual code snippet within the error message itself would be redundant, given that we expect the errors to presented in the IDE.



(a) `eslint` identifies problem, in Visual Studio Code.



(b) `eslint` suggested fix for problem, in Visual Studio Code.

Figure 2.1 For this error, the bare eslint error message is appropriate when the messages is presented in the context of the IDE.

2.3 Objectives and Significance

The error message examples I've selected from Section 2.2 illustrate several observations about error messages and how developers comprehend them:

1. **Error messages are routine.** The selected examples are everyday messages that confound developers. They don't require developers to have knowledge of esoteric language concepts, nor are these error messages instances of extraordinary circumstances.

2. **Error messages aren't false positives.** False positives are a known problem with program analysis tools, especially static analysis tools which use approximations in order to determine whether a problem exists. But none of the selected examples were false positives, and all them indicated an actual problem in the code. This suggests that developers have difficulties with error messages even when they are revealing an actual problem in the code.

3. **Developers are intermediate-experts. This isn't just a problem for novices.** The persona we assumed in the examples is that of a developer who is an intermediate-expert in the programming language. They are certainly not novices who are just learning to use languages and program analysis environments.

4. **There isn't an obvious way to automatically repair the program.** Even small programs have a combinatorially-large design space for program transformations that would remove the compiler error message. To illustrate, an alternate way to make the "Hello, world!" syntactically valid for the examples in Section 2.2.1 and Section 2.2.2 would be to simply remove the `printf` statement entirely. Such a fix seems incredulous if the intention of the developer is to print a string to the console, but perfectly logical if the developer intended "Hello, world!" to only act as a starting point for the program they actually intend to implement. Good automatic fixes are useful as accelerators for developers, but they don't remove the need to understand the error message.

5. **Error messages are an insufficient construction of the problem.** The error messages present only the symptom of the problem, and leave it to the developer to come up with the rationale for why the problem occurs. We saw this in the unreported exception example for Java in Section 2.2.3, in which

the developer needed to construct a long chain of reasons to identify the cause of the error.

6. **Error messages are rational, but only from the perspective of the compiler.** Let's turn again to the source listing for the "Hello, world!" example from Section 2.2.2, but this time we will re-format the program listing like so:

```
int main(){printf("Hello, world!")}
```

Some compilers remove insignificant whitespace, or trivia, because they are unnecessary in the program analysis pipeline, resulting a program that appears to the tool like the above. From the perspective of the compiler, the error message `expected ';' before '}' token` is now perfectly rationale if we reorient ourselves to the perspective of how the compiler thinks about the situation. But developers shouldn't have to be compiler authors to benefit from error messages in program analysis tools.

The first three observations eliminate specific factors as being the cause of developer difficulties with error messages: developers experience difficulties even in cases where the error messages are routine, when they aren't false positives, and even in cases where the developers are comfortable with their programming languages and tools. The fourth observation indicates that error messages are useful to developers for verification, even when tools suggest automatic fixes. But it is the fifth and sixth observations that motivates a theory to investigate: these two observations highlight a misalignment between the way program analysis tools present error messages to developers and the way in which developers think as they attempt to comprehend the problem.

Error messages in program analysis tools are problematic for developers to comprehend and resolve (Chapter 3). The *objective* of this research posits and evaluates a novel framework that applies *rational reconstruction* theory to error messages as an explanation for why error messages are difficult for developers to comprehend. We formalize this theoretical framework in Section 2.4 on the next page. The *significance* of this research is that it makes error messages more useful for developers, because these error messages align with the way in which developers strategize and reason about problems in their code. The research advances a systematic, theoretical lens to investigate error messages in program analysis tools.

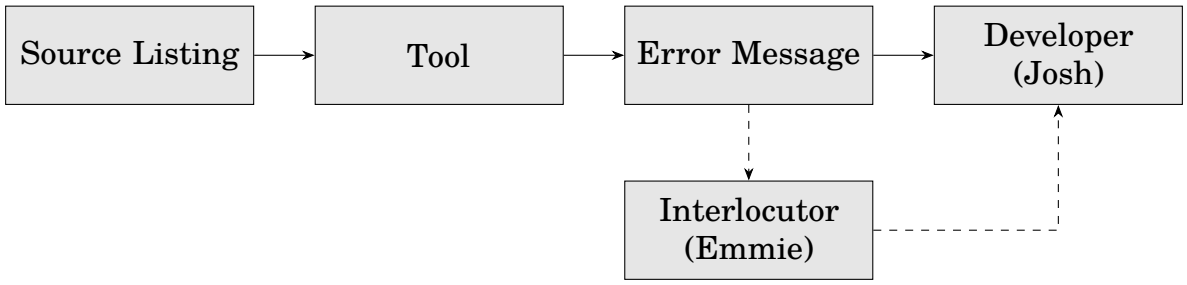


Figure 2.2 The theoretical framework for rational reconstruction.

2.4 Theoretical Framework

Example. Consider a scenario in which a developer, Josh, encounters a confusing error message. They aren't able to comprehend the error message, so they turn to a colleague, Josh, for assistance. Much like we did for the examples in Section 2.2, Emmie explains the error message by offering reasons and justifications to demonstrate the problem is actually a problem. We can illustrate this scenario as the model in Figure 2.2, with Sarah as the interlocutor—a participant in the discourse.

What did Emmie do differently from the error message in the program analysis tool that allowed Josh to understand the problem? Generally, how was the human-human interaction different from the human-computer interaction, and can these differences explain why error messages in program analysis tools are confusing for developers?

Definitions. The theory I apply to investigate developer difficulties with error messages is rational reconstruction [236]. Rationales are the set of reasons for something: here, that something is the error message produced by the program analysis tool. If the error message is deficient and does not provide sufficient rationale, Emmie must reconstruct the rationale for the error message herself and come to the same conclusion as the conclusion of the error message.

The process of identifying these rationale is rational reconstruction. Within this theory, rationales come in two forms [57, 236]: 1) as a justification-explanation (called an argument), in which the rationale functions as evidence that supports the conclusion; and 2) a trace-explanations (sometimes referred without qualifier as simply an explanation), in which the rationale functions as a cause for the conclusion.

Utility. Rational reconstruction is a useful theoretical framework to apply to

error messages, for four reasons:

1. **Rational reconstruction is a process centered around ordinary human dialogue.** Having origins in philosophy and linguistics, rational reconstruction applies a human-centered lens as a means to construct intuitive, yet logical explanations intended for human consumption—such as the exchanges between Emmie and Josh. But are human-human interactions a ground truth for human-computer interactions? The influential theory of computer as social actors suggests that it is: people treat computers and respond to computers as if they were real people [260]. And subsequent research has argued that it is often valuable to for computational agents to mimic how people behave in human-human interactions and approximate them as human-computer interactions [31, 107, 246].

2. **Rational reconstructions are not historical reconstructions.** They admit orthogonal explanations of the problem, and the research implication of this as they we may consider the presentation of an error message as independent from how the tool internally functions. Emmie does not need to know the internals of the program analysis tool in order to provide a sufficient example to Josh for why the error message is emitted. In Chapter 8, we exploit this property to instrument a compiler that produces rational reconstructions as error messages.

3. **Rational reconstruction has been applied with some success to other areas of software engineering.** For example, the design process from Parnas and Clements [277] proses that software design documentation should appear as if it were designed by a precise requirements process, even though we do not actually design products in that way. In other words, the presentation of software design process is a rational reconstruction.

4. **There is formative evidence that rational reconstructions would be useful to developers.** For example, Dean [89], notes that "a message whose meaning has to be explained does not communicate—it fails as a message." Additional related work that explanation is a fruitful direction for error message investigation is found in Section 6.3.

2.5 Research Paradigm

The research paradigm governs the philosophical assumptions and beliefs for how we conduct research and interpret research findings [77]. In this dissertation, I employ a pragmatic research paradigm, which I describe through the four basic questions of the research process [79]:

1. What epistemology informs the theoretical perspective?
2. What theoretical perspective lies behind the methodology in question?
3. What methodology governs our choice and use of methods?
4. What methods do we propose?

2.5.1 Epistemology

The epistemology of the research in this dissertation is pragmatic. It is helpful here to define the epistemology of pragmatism against two epistemological extremes: positivism and constructivism [248]. In positivism, there is one and only one objective truth, and this truth can be uncovered through objective analysis, such as in quantitative methods. In constructivism, theories are social constructs: there is no objective truth and theories are interpretations of subjective inquiries. Pragmatism sidesteps this debate entirely by making no epistemological stance: truth is whatever seems to work for a particular situation [132].

2.5.2 Theoretical Perspective

Pragmatic research places emphasis on identifying the research problem, and then selecting appropriate research tools or instruments to study the problem [105]. Because of the lack of epistemological commitment, the output of pragmatic research are in the form of solution-focused guidelines or interventions to understand or attack the nature of the problem. Within this perspective, theories are problem-solving tools to help understand why a particular problem occurs and what steps we can take to correct the situation. Theories are also self-correcting as new evidence emerges. A central idea of pragmatism is human inquiry: that we can advance understanding through observing how people work in their day-to-day lives, seeing what works, and identifying what doesn't [185].

2.5.3 Methodology

I use mixed-methods in my studies, applying both quantitative and qualitative methodology, both when appropriate and when convenient. Therefore, I reject the incompatibility thesis, which argues that integrating qualitative and quantitative research is incompatible epistemologically [95].

2.5.4 Methods

I used an assortment of research methods in this dissertation. In Chapter 4, I conducted a usability study of error messages within the Eclipse, supported by eye tracking instrumentation. I used established eye tracking measures, such as fixation duration, to understand comprehension difficulties developers have with reading error messages. In Chapter 5, I applied a think-aloud protocol and observed how developers constructed explanatory visualizations for error messages. I used a memorization/recall task technique from Shneiderman [335] to assess developer comprehension. In Chapter 6, I conducted a comparative study between two Java compilers to understand developer preferences for different structures of error messages explanations. I qualitatively investigated Stack Overflow posts related to error messages, and analyzed these posts abductively through the lens of argumentation theory—a form of rational reconstruction.

2.6 How to Read the Dissertation

This dissertation is organized into self-contained chapters (Figure 2.3 on the following page) that combine to tell a coherent story about rational reconstructions.

Chapters 4 to 6 are the principal studies which provide evidence to support the thesis statement, and each of these studies considers different aspects of rational reconstruction. Chapter 4 characterizes difficulties developers confront as they comprehend and resolve error messages within the Eclipse integrated development environment; these findings serve to baseline error messages as rational reconstructions. Chapter 5 studies rational reconstructions as diagrammatic explanations for defects in source code. Chapter 6 studies rational reconstructions as text explanations, using questions and answers from Stack Overflow to understand situations where developers find error messages from program analysis to be insufficient, yet accept a human-authored explanation from a Stack Overflow participant.

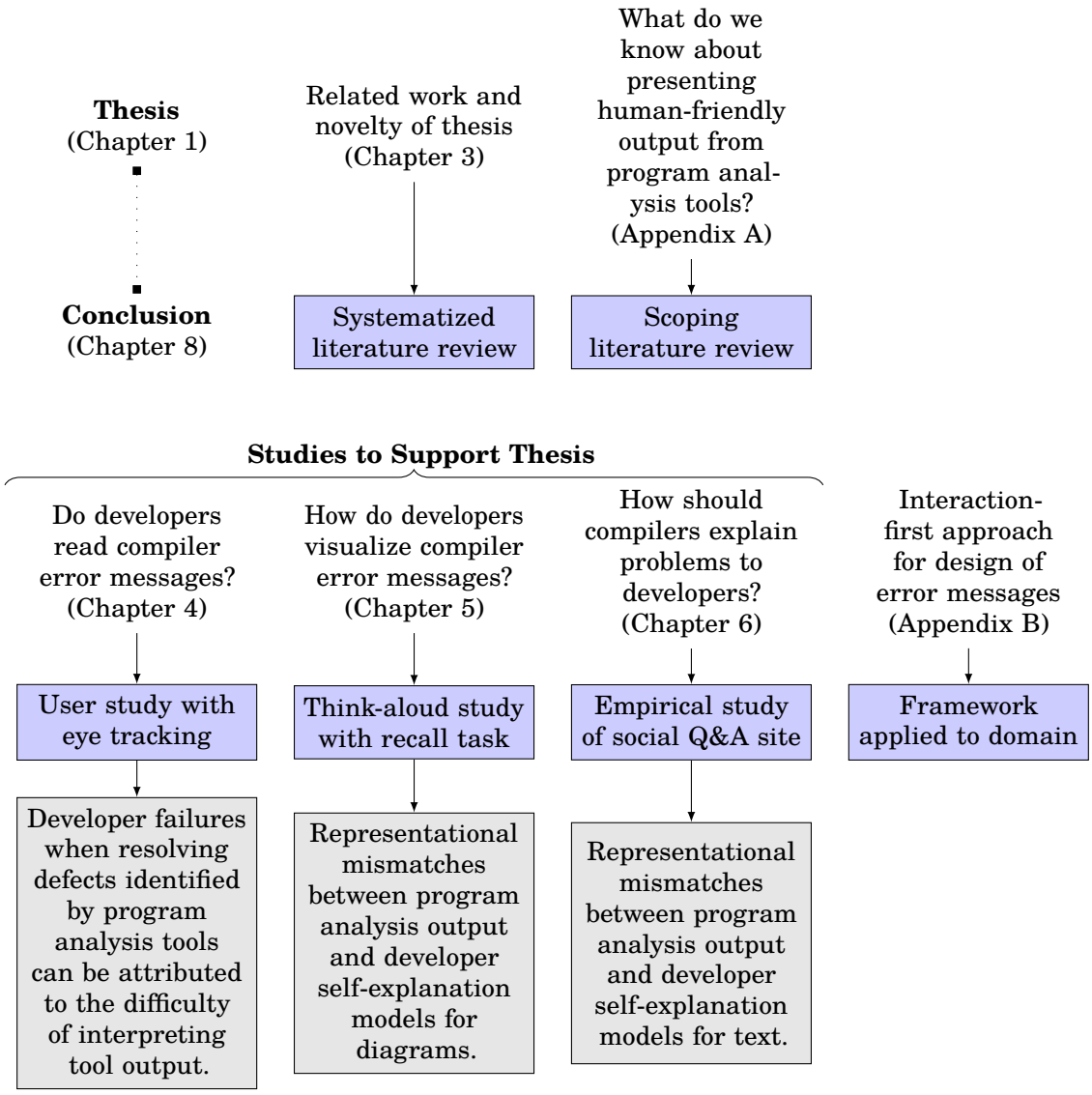


Figure 2.3 A roadmap of the dissertation, organized as self-contained (at least to the extent reasonably possible) chapters.

The remaining chapters in the dissertation scaffold the thesis statement. Chapter 2 (you're reading this now) identifies and provides context for research problem, articulates the objectives and significance of the work, and proposes the theoretical framework through which I investigate error messages. The background presented

in Chapter 3 situates the thesis ontogenically in terms of the other research; the chapter also familiarizes the reader with the research area of program analysis error messages. The related work in Chapter 7 defends the novelty of the thesis; the chapter demonstrates how the work in this dissertation is ontologically distinct from neighboring research disciplines, such as artificial intelligence and human factors. In Chapter 8, we revisit the key contributions of this dissertation. The chapter interprets the findings from the studies and explicates them as operational guidelines. These guidelines are operationalized and evaluated through a prototype compiler implemented in TypeScript.

The appendices provide supplemental material may be of interest to the reader, but this material is not necessary to support the central claims of the thesis. The scoping review in Appendix A identifies rational reconstruction as a problem of alignment, and contributes taxonomy of other design dimensions that are important for the design of human-friendly error messages. This appendix may be of interest to researchers who desire to bridge the research communities of human-computer interaction and programming languages. Appendix B presents several new ideas and emerging results which bootstrapped the work in this dissertation. Appendices C and D compile the materials from the user studies. Appendix E synthesizes existing guidelines for error messages from the research literature. Appendix F catalogues an assortment of error messages from program analysis tool from academia and industry. Appendix G describes the implementation details of the TypeScript prototype. Appendix H contains a delightful Sudoku puzzle. The puzzle is not essential to supporting the thesis.

2.7 Who Did What

I am the lead author of all content presented within this dissertation. As lead author, I made substantial contributions to all aspects of the work, which include designing the experiments, collecting experimental data, performing data analysis, interpreting the data, and drafting and revising the manuscripts for submission to academic venues. Consequently, I take full responsibility for all aspects regarding the accountability and integrity of the work.

With that said, this dissertation would not have been possible without the assistance of several other researchers. In the interest of responsible authorship, I report their contributions to this dissertation:

- Jim Witschey, Brittany Johnson, Emerson Murphy-Hill, and Sarah Heckman

are authors of an early, unpublished, draft: "A Taxonomy for Program Analysis Notifications." The goal of this work was to provide a controlled vocabulary to unify phrasing in error messages. This formative publication not only helped to frame my early thoughts towards this thesis—particularly in terms of presenting error messages to developers (Appendix B.4.2). In this new ideas paper, Witschey provided significant assistance in proofreading the work, and with direction from myself, authored one of the figures (Figure B.3).

The studies within this dissertation contributed to the second iteration of the grant (awarded, NSF 1714538), which I co-wrote with Emerson Murphy-Hill and Sarah Heckman.

- Chris Parnin and Emerson Murphy-Hill are co-authors for my scoping review (Appendix A). Through various exchanges, Parnin and I discussed the motivation and framing for the introduction (Appendix A.2)—to interpret PL papers through an HCI lens. Murphy-Hill and I had several exchanges during the design of the paper on how this work could serve towards a comprehensive literature review.

- Kevin Lubick, Justin Smith, Elisabeth Holmes, Jing Feng, Emerson Murphy-Hill, and Chris Parnin are co-authors on my eye tracking study (Chapter 4). Lubick and I both conducted studies with the participants, with each of us conducting roughly half the studies. I identified the categories of error messages to be used for the study, and the software libraries in which to inject these errors. Lubick injected the error messages into this software library, and verified that the error messages would display properly to the participants. He also ran several early pilot studies on my behalf to identify potential issues with the study design.

  Smith continued to support this work after Lubick transitioned to other projects. Smith helped substantially with the data cleaning pipeline to correct calibration errors in eye tracking data; he also rendered two of the figures in the paper (Figure 4.1 and Figure 4.2). Smith and I pair-wrote the introduction (Section 4.2) and motivating example (Section 4.3) sections of the paper. Holmes spent an entire summer manually tagging the participant videos, with limited success. Her efforts in this area led me to investigate automated and scalable techniques for tagging video data (Section 4.4), which I eventually applied to this study. Feng provided her expertise in working with professional eye tracking equipment. Jess Cherayil, a summer undergraduate research

student, contributed code towards area of interest detection for frames within participant videos.

- Kevin Lubick, Samuel Christie, and Emerson Murphy-Hill are co-authors on my think-aloud study on how developers visualize error messages (Chapter 5). Lubick ran approximately half of the participants for the study, and helped to annotate the participant data. Christie analyzed the data from the pilot study, and reported the frequency of annotations participants used (Table 5.1). Murphy-Hill provided suggestions on making sure the story about visualization would appeal to the VISSOFT community, recommended some related work, and suggested the title for the paper.
- Denae Ford, Chris Parnin, and Emerson Murphy-Hill are co-authors on my study on using Stack Overflow questions and answers as argument structure for error messages (Chapter 6). Ford and I had many early discussions on how to classify Stack Overflow answers before arriving at argument theory. In the paper, Ford wrote the query to compute and describe Table 6.2. Ford also qualitatively coded half of the data—across multiple programming languages—and labeled Stack Overflow responses in terms of argument structure. Parnin and Murphy-Hill provided feedback on drafts.

2.8 Contributions

This dissertation advances knowledge through several contributions, and defends the claims of the thesis presented in Chapter 1. I have repeated the thesis in full below:

> Difficulties interpreting error messages produced by program analysis tools are a significant predictor of developers' inabilities to resolve defects: difficulties in interpreting error messages can be explained by framing error messages as insufficient rational reconstructions, in both visual and textual output presentations.

To defend the claim that "developer failures when resolving defects identified by program analysis tools can be attributed to the difficulty of interpreting tool output," I offer evidence through the study in Chapter 4. The study finds that:

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

To defend the claim that "difficulties in interpreting tool output are explainable through representational mismatches between program analysis output and developer self-explanation models for both diagrammatic and textual output representations," I offer evidence through two studies. For diagrammatic representations, this evidence is offered through the study in Chapter 5, in which we compare prototype *explanatory* diagrammatic visualizations of error messages overlaid on source code with *baseline* visualizations used in IDEs today. The study finds that:

- Explanatory visualizations yield more *correct* self-explanations than the baseline visualizations used in IDEs today.
- These annotations are used intuitively by developers in their own explanations of error messages, even when explanatory visualizations have not been provided to them.
- Participants identified that explanatory visualizations revealed hidden dependencies, that is, the relationships between different program elements, in a significantly different way than those of baseline visualizations in IDEs.

In addition, the study contributes a foundational set of composable, visual annotations that aid developers in better comprehending error messages.

To defend the claim for textual output representations, defense of the thesis is offered through evidence from the study in Chapter 6. Through the study, we find that:

- Developers prefer error messages with proper argument structures over deficient arguments, but will prefer deficient arguments if they provide a *resolution* to the problem.

- Human-authored explanations converge to argument structures that either offer a simple resolution, or to proper arguments that minimally provide a claim, ground, and warrant.

3 | Background

> A straight line may be the shortest distance between two points, but it is by no means the most interesting.
>
> The Doctor

3.1 Overview of Program Analysis Tools

Program analysis tools refer to a broad class of software intended to help developers, with applications across software architecture, program comprehension, program evolution, testing, software versioning, and verification [34]. This dissertation focuses on a particular class of program analysis tools, source code analysis, that identifies defects within source code [34]:

> Source code analysis is the process of extracting information about a program from its source code or artifacts (e.g., from Java byte code or execution traces) generated from the source code using automatic tools. Source code is any static, textual, human readable, fully executable description of a computer program that can be compiled automatically into an executable form. To support dynamic analysis the description can include documents needed to execute or compile the program, such as program inputs.

We can further elaborate program analysis tools through examining characteristics of these tools, across two implementation dimensions found in existing literature [55, 113, 135, 319].

The first dimension is *when* the program analysis is performed. In static program analysis, tools examine the source code—either directly or as an abstraction of the source code—and identify defects in the code without executing the program (see survey on static analysis tools by Gosain and Sharma [136]). In dynamic analysis, tools instrument or otherwise inspect the runtime and examine the flow of execution to identify defects [55] (see survey on dynamic analysis tools, also by Gosain and Sharma [135]). Static and dynamic analysis can also be applied synergistically to strengthen program analysis [113]. An as example of hybrid analysis, Check 'n' Crash combines static theorem proving with dynamic test execution to eliminate spurious warnings and improve the ease-of-comprehension of error messages through the production of Java counterexamples [80].

The second dimension is *where* the program analysis is performed. For example, FindBugs is a stand-alone static analysis tool for Java that *supplements* the error messages provided by the Java compiler [14]. In contrast, LLVM *embeds* program analysis tools directly within the compilation pipeline; Lattner and Adve [205] demonstrate that this type of "lifelong program analysis" can enable identification of certain defects that would not be possible to find otherwise, such as with interprocedural static leak detection.

Unfortunately, neither of these implementation dimensions is appropriate for understanding developer difficulties from the output of these program analysis tools. That is, whether the information for the same problem is obtained using static, dynamic, or hybrid-analysis is an implementation detail. For example, both LLVM (static) and Valgrind (dynamic) can report uninitialized variables, but the type of analysis should not drive how the error message is presented to the developer. Likewise, where the analysis happens is also an implementation detail.

For these reasons, I've instead opted to organize program analysis through the way in which it *presents* its output to the developer. In Section 3.2, I synthesize the literature for text representations of program analysis output. In Section 3.3, I do the same for emerging visual representations of program analysis output. For a systematic investigation of program analysis output, Appendix A contains a scoping literature review of PLDI papers on program analysis.

The last two sections provide additional background and justification for the thesis. Section 3.4 summarizes the existing literature on empirical distributions of program analysis errors. Section 3.5 reviews existing design guidelines towards presenting human-friendly error messages.

3.2 Text Representations of Program Analysis

3.2.1 Output as Source Location and Template Diagnostic

The familiar error message scheme within program analysis tools consists of a location indicating where the problem occurs, a human-authored description indicting what has gone wrong, and some additional information—such as the severity of the problem, an error code, resolution hints, and code context—to help the developer correct the problem.

In this section, we'll use the following source listings:

1 The Java implementation:

```
class Brick {
    void m(int i, double d) { }
    void m(double d, int m) { }

    {
      m(1, 2);
    }
}
```

2 The C# implementation:

```
namespace Program {
  class Brick {
      void m(int i, double d) { }
      void m(double d, int m) { }

    static int Main(string[] args) {
      var b = new Brick();
      b.m(1, 2);
      return 0;
    }
  }
}
```

3 The C++ implementation:

```
class Brick {
    void m(int i, double d) { }
    void m(double d, int m) { }
};

```

```
int main() {
    Brick b;
    b.m(1, 2);
    return 0;
}
```

In all three implementations—Java, C#, and C++—we have introduced an ambiguous method error: despite the different type signatures, program analysis cannot disambiguate which of two candidate implementations to invoke given a call of method `m(1, 2)`. One possible fix is to explicitly indicate the type of the argument as `m((int)1, (double)2)` (in Java) or `b.m((int)1, (double)2)` (in C# and C+).

For Java error messages, we use the OpenJDK compiler and the Eclipse compiler. For C#, we use Roslyn and Mono. For C++, we use LLVM and GCC. The choice of program analysis tools illustrates the diversity in how toolsmiths choose to present error messages for what is conceptually the same problem. The application of source implementations to their corresponding tools yields the following error messages during compilation:

1 OpenJDK (Java):

```
Brick.java:6: error: reference to m is ambiguous
      m(1, 2);
      ^
  both method m(int,double) in Brick and method m(double,int) in Brick
    ↪ match
1 error
```

2 Eclipse (Java)

```
----------
Brick.java (at line 6)
        m(1, 2);
        ^
The method m(int, double) is ambiguous for the type Brick
----------
1 problem (1 error)
```

3 Roslyn (C#)

```
Brick.cs(8,9): error CS0121: The call is ambiguous between the
  ↪ following methods or properties: 'Brick.m(int, double)' and
  ↪ 'Brick.m(double, int)'
```

4 Mono (C#)

```
Brick.cs(8,9): error CS0121: The call is ambiguous between the
    following methods or properties: `Program.Brick.m(int, double)' and
    `Program.Brick.m(double, int)'
Brick.cs(3,12): (Location of the symbol related to previous error)
Brick.cs(4,12): (Location of the symbol related to previous error)
Compilation failed: 1 error(s), 0 warnings
```

5 GCC (C++)

```
Brick.cpp: In function ‘int main()’:
Brick.cpp:8:13: error: call of overloaded ‘m(int, int)’ is ambiguous
     b.m(1, 2);
             ^
Brick.cpp:2:10: note: candidate: void Brick::m(int, double)
     void m(int i, double d) { }
          ^
Brick.cpp:3:10: note: candidate: void Brick::m(double, int)
     void m(double d, int m) { }
          ^
```

6 LLVM (C++)

```
Brick.cpp:8:7: error: call to member function 'm' is ambiguous
    b.m(1, 2);
    ~~^
Brick.cpp:2:10: note: candidate function
    void m(int i, double d) { }
         ^
Brick.cpp:3:10: note: candidate function
    void m(double d, int m) { }
         ^
1 error generated.
```

There are several differences in the reporting from these tools in location, description of the problem, supporting context, and formatting. OpenJDK, Eclipse, and Roslyn indicate the location of the invocation, but not the locations of the candidate methods. Furthermore, the Eclipse error message also does not indicate to what methods the invocation is ambiguous. This is in contrast to GCC, LLVM, and Mono—which report both the call, candidates, and the locations for each. However, GCC and LLVM report a different column position for the error: GCC indicates the problem at the end of method call (Line 8, Column 13) while LLVM indicates the problem at the name, `b.m`. There are other small variations in how the line and columns are presented to the developer.

OpenJDK, Eclipse, GCC, and LLVM inject snippets directly from the source code into the error report to provide context to the developer; Roslyn and Mono do not. I also found it interesting that Mono seems to indicate that it does not know if `m` is a method of property. Finally, GCC and LLVM colorize the output of the error messages for expressiveness; the LLVM authors reasonably argue that colors make it easier to distinguish the different elements of the error message [223].

The messages I have just described are output from a template diagnostic. Templates are string literals that allow expressions to be embedded, or interpolated, within them. For example, in OpenJDK, the error description string is found in `compiler.properties`:

```
# 0: name, 1: symbol kind, 2: symbol, 3: symbol, 4: symbol kind, 5: symbol,
↪  6: symbol
compiler.err.ref.ambiguous=\
    reference to {0} is ambiguous\n\
    both {1} {2} in {3} and {4} {5} in {6} match
```

The parameters that can be interpolated are indicated as `0`, `1`, `2`, and so on. The Roslyn implementation uses a similar interpolation scheme, through an XML file called `CSharpResources.resx`:

```
<data name="ERR_AmbigCall" xml:space="preserve">
    <value>The call is ambiguous between the following methods or
    ↪  properties: '{0}' and '{1}'</value>
</data>
```

The Roslyn implementation supports only two parameters, which is evidenced in the presented in the error message.

The template string and associated metadata are bundled up as *diagnostic objects* and passed to a formatter within the program analysis tool. The formatter augments the template string with the metadata information, such as the location; the formatter also colorizes the output, and includes pertinent source code snippets—if those capabilities are available in the formatter implementation.

We'll end this section by describing a dastardly form of template messages that routinely frustrates both novice and expert developers alike—the neglected "battle fields of syntax errors" [201]. A syntax error occurs when program analysis encountered an unexpected token, for example, from unintentionally misplaced semicolons, extra or missing braces, or a `case` statement without an enclosing `switch` [4]. In particular, one frustration of syntax error messages is that program analysis tools report the location of the syntax problem far removed from the actual cause of the syntax error [54]. For example, consider the following Racket program, which implements factorial:

```
1 #lang racket
2
3 (define (factorial n)
4   (if (= n 0) 1
5       (* n (factorial (- n 1))))
```

This program implements the correct behavior, but has a syntax error due to a missing closing parenthesis. Racket reports this syntax error to the developer on Line 3:

```
factorial.rkt:3:0: read: expected a `)' to close `('
```

The error is actually correct in that there is no corresponding closing parenthesis for the opening parenthesis beginning with Line 3. However, to actually resolve the defect, the developer must add a closing parenthesis to the end of Line 5:

```
5       (* n (factorial (- n 1))))
```

There have been many research efforts to improve syntax errors —extensively covered in various literature reviews [38, 90, 91]— that historically have occurred in tandem with the construction of even the earliest compilers in the 1950s. Hammond and Rayward-Smith [146] describe some of the early syntactic error recovery and repairs schemes, and characterizes the trade-offs in implementing the different schemes. For instance, in panic mode, one of the earliest and simplest error recovery techniques, the parser deletes incoming tokens until the parser discovers a token that enables it to continue processing the source code. Although this approach is easy to implement, Hammond and Rayward-Smith [146] report that results in cryptic and unhelpful error messages because, and leads to spurious error messages. Subsequent researchers approaches have therefore focused on: 1) *error detection*, or reducing the difference between the location where the error is detected and the point where the error actually occurs, and 2) *error correction*, on providing the developer with one or more candidate repairs that transform the incorrect input into a syntactically correct one [90, 91]. In contrast to techniques attempt to mathematically define notions of minimizing error distance, Campbell, Hindle, and Amaral [54] train a natural language model to improving this error reporting by arguing that humans read code as they read natural language; follow-up work by Santos, Campbell, Patel, and colleagues [323] offers additional evidence that language models can successfully locate and fix syntax errors in human-written code without formal parsing.

Still other approaches have investigated reducing the burden for compiler designers to author more useful error messages. For example, Jeffery [176] describes a tool called Merr—meta error generator—that allows compiler designers to associate hand-authored diagnostic messages with syntax errors *by example*. From this specification of errors and the associated message, Merr identifies the relevant parse states and input token, and inserts an error function into the parse to produce the error message at the appropriate point. [296] improves upon the approach by Jeffery [176] by enabling the parser to automatically build the collection of erroneous statements, rather than having the compiler author provide the examples manually. Though useful, both approaches still require a compiler designer to hand-author an accompanying error message to incorporate this diagnostic information.

3.2.2 Output as Extended Explanations (`--explain`)

Program analysis tools can provide supplemental channels for extended explanations about an error message, as an alternative to the concise, line-oriented error messages presentations from Section 3.2.1. Johnson, Song, Murphy-Hill, and colleagues [183] reported that concise error messages in program analysis tools do not provide enough information. However, the authors reported that developers did not want to sift through volumes output to identify the problem. A supplemental mechanism for providing extended explanations can reconcile these conflicting requirements. Extended explanations can also aid understanding when the developer doesn't know about the concepts in the message [252], and allows them to selectively investigate unfamiliar error messages with a longer, more verbose explanation [253].

To illustrate how extended explanations are useful in practice, let's look at a simplified build pipeline using Bazel [25]—an open source release of the build system used internally at Google. Bazel includes a program analysis tool called Error Prone [114] that identifies defects in Java code. The build pipeline is guided by a `BUILD` file which specifies the targets, or instructions to the build system:

```
java_library(
    name = "shortset",
    srcs = ["ShortSet.java"],
)
```

In this scenario, we have only one build target named `shorted`, and this build target needs to compile only a single Java, `ShortSet.java`. Here's the Java file:

```
1 import java.util.Set;
2 import java.util.HashSet;
```

```
3
4  public class ShortSet {
5    public static void main (String[] args) {
6        Set<Short> s = new HashSet<>();
7        for (short i = 0; i < 100; i++) {
8          s.add(i);
9          s.remove(i - 1);
10       }
11
12       System.out.println(s.size());
13   }
14 }
```

Building the `shortset` target with Blaze results the following output, which includes an error message:

```
1  INFO: Analysed target //:shortset (0 packages loaded).
2  INFO: Found 1 target...
3  ERROR: /BUILD:1:1: Building libshortset.jar (1 source file) failed (Exit 1)
4  ShortSet.java:9: error: [CollectionIncompatibleType] Argument 'i - 1'
5  should not be passed to this method; its type int is not compatible with
6  its collection's type argument Short
7          s.remove(i - 1);
8                   ^
9      (see http://errorprone.info/bugpattern/CollectionIncompatibleType)
10 Target //:shortset failed to build
11 Use --verbose_failures to see the command lines of failed build steps.
12 INFO: Elapsed time: 0.582s, Critical Path: 0.29s
13 FAILED: Build did NOT complete successfully
```

The error message is embedded within other build output, and begins at Line 3 and ends at Line 9. The error message in the build outputs offers a claim for the problem, but does not provide a rationale for why this is a problem. However, the error message points to external documentation that explains extensively the reason for this message, why `remove` is unlikely to actually remove the element, and why the type system in Java alone is not available to detect this problem.[1] Delegating this explanation to an external source reduces the verbosity of the build output, while still allowing the developer to access additional explanation about the message. If the developer is already familiar with the error message, they may not need the extended explanation at all. From this perspective, Error Prone is a modernized implementation of early expert systems, such as the expert system COBOL for debugging programming debugging by Litecky [221]. In Litecky's implementation,

[1] `http://errorprone.info/bugpattern/CollectionIncompatibleType`

the developer could take a cryptic error code such as `3A13` (period missing after `VALUE` clause) and query the error code against a database to obtain a detailed explanation, as well as advice for how to address the problem (for a detailed review of expert systems, see Section 7.3 on page 132.

A limitation of the approach used by Error Prone is that the extended explanation is detached to the context of the developers' code. Here is a snippet of the extended explanation:

> In a generic collection type, query methods such as `Map.get(Object)` and `Collection.remove(Object)` accept a parameter that identifies a potential element to look for in that collection. This check reports cases where this element *cannot* be present because its type and the collection's generic element type are "incompatible." A typical example:
>
> ```
> Set<Long> values = ...
> if (values.contains(1)) { ... }
> ```
>
> This code looks reasonable, but there's a problem: The `Set` contains `Long` instances, but the argument to `contains` is an `Integer`.

In other words, the developer must evaluate an example different from the code they have actually written in order to understand the explanation. Even if the developer understands the extended explanation through the example, they may still not understand how the problem applies to their own code.

To mitigate this challenge, the Dotty compiler provide an `--explain` flag that can provide an explanation that is contextual to the code that the developer has actually written [252]. For instance, consider the following Dotty code snippet (the Dotty language is a superset of Scala):

```
1 try {
2   foo()
3 }
```

By default, this code snippet generates the following error message from the Dotty compiler:

```
-- [E002] Syntax Warning: scala.test ----------------------------------
1 | try {
  | ^
  | A try without catch or finally is equivalent to putting
  | its body in a block; no exceptions are handled.
2 |   foo()
3 | }
```

Like the Rust [253] and Elm [70] programming language communities, usability of error messages and tools are one of the design goals of Dotty. So, this error message is already pretty good: it describes the location of the error in the context of the code, describes what the `try` mechanism would do in this context, and provides a rationale for why this is a problem. Nevertheless, perhaps it is the case that the developer is new to the language and does not understand what the `try` construct actually does. They can then pass the `--explain` flag to Dotty to obtain the extended explanation:

```
1  Explanation
2  ===========
3  A try expression should be followed by some mechanism to handle any exceptions
4  thrown. Typically a catch expression follows the try and pattern matches
5  on any expected exceptions. For example:
6
7  import scala.util.control.NonFatal
8
9  try {
10   foo()
11 } catch {
12   case NonFatal(e) => ???
13 }
14
15 It is also possible to follow a try immediately by a finally - letting the
16 exception propagate - but still allowing for some clean up in finally:
17
18 try {
19   foo()
20 } finally {
21   // perform your cleanup here!
22 }
23
24 It is recommended to use the NonFatal extractor to catch all exceptions as it
25 correctly handles transfer functions like return.
```

Unlike the extended explanation from Error Prone, the explanation from Dotty is situated using code the developer has written. This is easily seen from the use of `foo` (Line 10 and Line 19). The current implementation of this in Dotty is fairly rudimentary, and only slight more elaborate than the template error messages we have discussed:

```
abstract class EmptyCatchOrFinallyBlock(tryBody: untpd.Tree,
    errNo: ErrorMessageID)(implicit ctx: Context)
  extends Message(EmptyCatchOrFinallyBlockID) {
    val explanation = {
```

```
      val tryString = tryBody match {
        case Block(Nil, untpd.EmptyTree) => "{}"
        case _ => tryBody.show
}

...
```

Essentially, Dotty grabs the code block from the source code, stores it in intermediate variables like `tryString`, and then injects these variables throughout the extended explanation. But we could imagine more elaborate implementations of this idea, including the manipulation of the natural language portion of the explanation based on the code context.

Recognizing that developers frequently turn to other information sources such as the web to help with errors or debugging problems, there are several other research implements for generating context-relevant, extended explanations. The implementation of Tutoron by Head, Appachu, Hearst, and colleagues [154] detects explainable code in a web page, parses it, and generates in-situ natural language explanations and demonstrations of code. Their qualitative study found that Tutoron-generated explanations can reduce the need for reference documentation in code modification tasks. The HelpMeOut system for novice students collects examples of code changes that fix errors, and then suggests these examples as solutions to others [152]. The tools Prompter [295], Seahawk [294], and SurfClipse [304]. automatically retrieve pertinent Stack Overflow explanations and present them within the IDE. These availability of these tools also suggest that developers find the human-authored explanations on Stack Overflow to be useful; we investigate these Stack Overflow explanations through an empirical study in Chapter 6.

3.2.3 Output as Type Errors

We consider type systems as a form of program analysis tool for reasoning about programs, with the most obvious benefit of type systems being that they help detect errors. In statically-typed systems, the type of every expression is known at compile time. For example, consider some simple Haskell expressions, prefixed with a `:t` command that tells us the type of an expression:

```
:t True
:t 'a'
```

As expected, `:t True` is `True :: Bool`, or a boolean true or false value. Similarly, the result of `:t 'a'` is `'a' :: Char`, or character.

Let's consider a function called `map`: this is a function that basically takes a list and applies a function every element in the result. The result of `map` is a new list. Here's a canonical implementation of this function:

```
map _ [] = []
map f (x:xs) = f x : map f xs
```

Haskell can tell us the type of `map`, through `:t map`:

```
map :: (t -> a) -> [t] -> [a]
```

The *type signature* tells us that `map` takes a function that takes a `t` and returns an `a`, and a list of `t`'s. Finally, it and returns a list of `a`. Note that in the above examples we did not specify the type of the expressions explicitly, although we could have easily done so:

```
:t True :: Bool
:t 'a' :: Char

map :: (t -> a) -> [t] -> [a]
map _ [] = []
map f (x:xs) = f x : map f xs
```

Instead, Haskell is able infer that the type was a character or boolean through *type inference* algorithm: a process of reconstructing missing type information through how it is used in the program [33, 82, 104, 287]. During program analysis, *type checking* verifies that the types are sound. If a violation is found, this results in a type error message. Let's induce a type error through the following Haskell snippet:

```
True && 1
```

Since the logical conjunction of a boolean to a number is incompatible in Haskell, this expectedly returns an error message in the form of a type error:

```
<interactive>:25:9: error:
    • No instance for (Num Bool) arising from the literal ‘1’
    • In the second argument of ‘(&&)’, namely ‘1’
      In the expression: True && 1
      In an equation for ‘it’: it = True && 1
```

Even this simple example illustrates a double-edged sword regarding type error messages. For compiler designers, unlike the human-authored messages in Section 3.2.1, the type checker absorbs the bulk of the work as it proceeds through its

standard type inference, and the program analysis can mechanically produce the problem. For the developer, however, this style of output means they must often have a precise understanding of the type inference algorithm in order to comprehend the error message [358]. Worse, relying on type inference as the solely mechanism for inducing error messages can lead to cryptic and counterintuitive error messages, even for simple problems. Considering the following example:

```
print 5
```

This program prints 5. What about this one:

```
print -5
```

Unless you're an experienced Haskell developer, you might be surprised to discover that this results in the following error message:

```
<interactive>:26:1: error:
    • Non type-variable argument in the constraint: Num (a -> IO ())
      (Use FlexibleContexts to permit this)
    • When checking the inferred type
        it :: forall a. (Show a, Num (a -> IO ())) => a -> IO ()
```

The Haskell error is pedantically correct, but framing the error messages in terms of a type system is ultimately uninformative for the developer. The problem becomes self-evident only if when we realize that - itself a function, masquerading as a unary negation:

```
print (- 5)
```

Aha! The fix is to write `print (-5)`.

Although we've thus far only examined Haskell, error messages such as the above are endemic to programming languages that rely on type systems as the vehicle through which to construct error messages to developers. The problems of inscrutable type errors are well known, and the impact of these cryptic type error messages is articulated by Charguéraud [58]. First, cryptic type errors are a major obstacle to learning functional languages, and require the developer to also understand the underlying inference algorithm being applied. Second, he notes that it is tempting to report all error messages through the type inference machinery, but this strategy can quickly lead to verbose and incomprehensible messages. Third, even if the program analysis could produce a short error messages, the type checker must still make a decision on which of many possible locations to actually report

in order to reduce the verbosity of the error. Fourth and finally, type inference algorithms suffer from well-known biases—such as left-right bias [239]—in which the type checker will systematically report type conflicts near the program. Thus, the point at which type inference fails may not be the point at which the developer has made a mistake.

There are several research directions to making type errors more helpful for developers. For example, one possibility might be to generate hand-author ad-hoc error messages for common situations, as with template diagnostics. Helium [157], a user-friendly compiler for learning Haskell, employs this approach: it uses a wide range of heuristics to generate hints that supplement the type inference machinery. Similarly, the approach in Objective Caml, is to "patch" the compiler [58]: the compiler considers the first top-level definition that fails to type-check and attempts to type-check it against a set of carefully-crafted secondary heuristics to identify if the tool can present an alternative error message. If so, this message is presented; otherwise, the default type error is presented to the developer. Their implementation covers the commonly used features of the Objective Caml programming language. In effect, Objective Caml presents a carefully-crafted rational reconstruction of original type error.

To provide user-friendly type error messages, Wu, Campora III, and Chen [402] apply machine learning: they use the type error from the compiler and associated features of the source code to identify a more-precise offending location of the problem, and suggests fixes that would resolve the ill-typed program. Lerner, Flower, Grossman, and colleagues [212] pursue an approach in which the type-checker itself does not produce the final error message. Instead, the type checker is used as an oracle for a search procedure that finds similar programs that do type check. McAdam [238] also suggests a fix-oriented approach to presenting error messages; he presents an algorithm based on *modulo isomorphism* that is applied when type inference fails. The resulting error messages are then in the form of a suggested change. For example, for the source listing:

```
val oneToThreeStrings = map ([1, 2, 3], Int.toString) ;
```

The Moscow ML compiler would normally report:

```
! Toplevel input:
! val oneToThreeStrings = map ([1, 2, 3], Int.toString) ;
!                          ^^^
! Type clash: expression of type
!   ('a -> 'b) -> 'a list -> 'b list
! cannot have type
!   'c * 'd -> 'a list -> 'b list
```

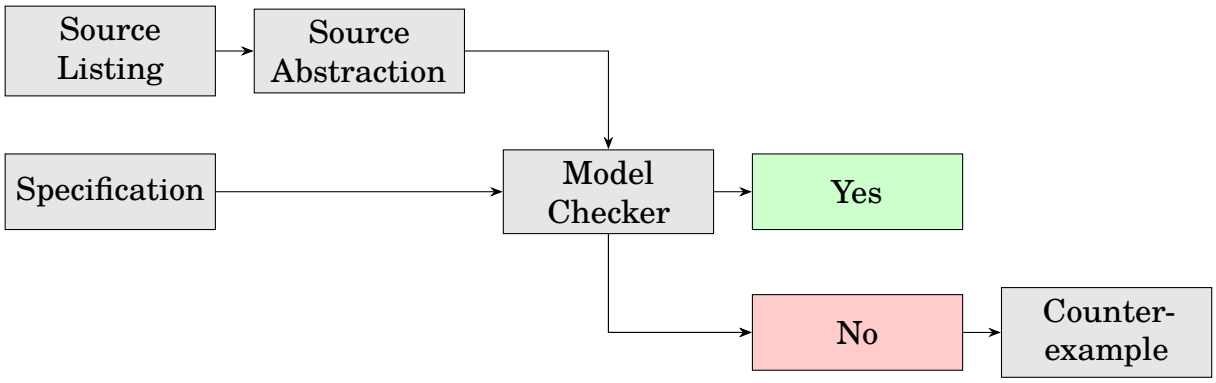


Figure 3.1 A sample model checking pipeline.

Instead, McAdam [238] reports the message as follows:

```
Try changing
  map ([1, 2, 3], Int.toString)
To
  map Int.toString [1, 2, 3]
```

Other approaches to improving type error messages include exposing type constraint information and making this information available to the error message reporter [225], using data flow reasoning to explain type errors [129], various modifications to the underlying type inference algorithm to provide a record of the specific reasoning steps, called *type explanations*, which lead to a program variable having a specific type [26, 104, 111, 361, 385], interactively approaches to querying types [347, 360], pinpointing the actual location of the problem [155, 156, 239, 303, 328, 382, 405].

3.2.4 Output as Examples and Counterexamples

Software model checkers are program analysis tools that formally verify a property of the software [68, 81]. The principle of model checking is that, given a finite state-transition graph of a system (that is, the software model) and a formal specification, model checking systematically checks whether the specification holds in the state-transition graph [67]. Essentially, the model checker *exhaustively* explores this finite state-transition graph. If the model checker does not find a violation of the property, the model checker has proved that the model satisfies the specification. Otherwise,

the model checker reports a diagnostic counterexample or trace that violates the specification. These diagnostic counterexamples can be presented on their own as a component of an error message explanation.

There are a variety of model checking tools for the domain of software engineering, include SLAM [16], SPIN [163], BLAST [159], Bandera [75], and Java PathFinder [378] (for extensive literature reviews on model checkers, see [250] and [177]). Concretely, let us illustrate model checking with CBMC [69]—a model checker for C and C++ programs—applied to the diagram in Figure 3.1.[2] For this example, we'll use the following source listing, written in C:

```
1 void f(int a, int b, int c) {
2   int temp;
3   if (a > b) {temp = a; a = b; b = temp;}
4   if (b > c) {temp = b; b = c; c = temp;}
5   if (a < b) {temp = a; a = b; b = temp;}
6
7   assert (a <= b && b <= c);
8 }
```

The intention of this function is that after executing the `if` statements in Lines 3-5, the values of the variables will be swapped such that `a <= b` and `b <= c` at Line 7. This specification is embedded within the source listing using an `assert` statement (Line 7), and the `assert` does not otherwise affect the behavior of the source listing.

Does our program satisfy this specification? Because this program is artificially trivial, we can manually identify a counterexample and test that it violates the specification. For example, when `a = 1`, `b = 1`, and `c = 0`, the resulting values will be `a = 1` and `b = 0` at Line 7. Of course, the CBMC program analysis tool also detects that the source listing violates the specification, and emits the following snippet (the full result is found in Appendix F.5.1 on page 275):

```
State 17 file file.c line 1 thread 0
------------------------------------------------
  INPUT a: 124955 (00000000000000011110100000011011)

State 19 file file.c line 1 thread 0
------------------------------------------------
  INPUT b: 256027 (00000000000000111110100000011011)
```

[2]The term *model* is ambiguous and has multiple meaning depending on the literature: it can refer to the source code, the source abstraction, the source abstraction and specification, or examples produced by the model checker. To avoid ambiguities with the term model, we will instead use the terms in Figure 3.1.

```
State 21 file file.c line 1 thread 0
------------------------------------------------
  INPUT c: 124954 (00000000000000011110100000011010)

Violated property:
  file file.c line 7 function f
  assertion a <= b && b <= c
  a <= b && b <= c


** 1 of 1 failed (1 iteration)
VERIFICATION FAILED
```

Model checkers are useful because they produce concrete instances as evidence of a problem. However, the counterexamples they produce, as we saw in the case of CBMC, are arbitrary. As with the type errors in Section 3.2.3, the presented counterexample reflects what was found by through the execution of the internal algorithm, and not what is necessarily the best example to present to the developer.

This has led to principles approaches to presenting output to users of model checking tools. For example, tools like Aluminum [264] and Razor [320] provide a *minimal* example to the user: an example that contain no more information than is necessary. Danas, Nelson, Harrison, and colleagues [83] investigate principled techniques for evaluating model checking output through user studies; for example, they evaluate whether counterexample *minimization* helps users.

Beer, Ben-David, Chockler, and colleagues [29] apply the idea of *causality* to formally define a set of causes for the failure of the specification on the given counterexample trace; these causes are marked as red dots and presented to the user as a visual explanation of the failure. Amalgam [263] allow the developer to interrogate the model checker and ask—through "why?" and "why not?" questions—about the counterexamples they provided.

Several approaches use multiple counterexamples to facilitate understanding, or supplement the counterexample with additional information. Groce and Visser [141] argues that although model checking is effective at finding subtle errors, these errors can be difficult to understand from only a single counterexample. They propose an automated technique for finding multiple counterexamples for an error, as well as traces that do not produce an error. Next, they analyze these executions to produce a more *succinct* description of the key elements of the error. Similarly, Ball, Naik, Rajamani, and colleagues [15] present an algorithm that exploits the existence of correct traces in order to localize the error cause in an error trace, report a single error trace per error cause, and generate multiple error traces having

independent causes. Wang, Yang, Ivančić, and colleagues [386] propose a technique in developers are able to zoom in to potential software defects by analyzing a single concrete counterexample. Gurfinkel and Chechik [143] annotate counterexamples with additional proof steps: they argue that this approach does not sacrifice any of the advantages of traditional counterexamples, yet allows the user to understand the counterexample better.

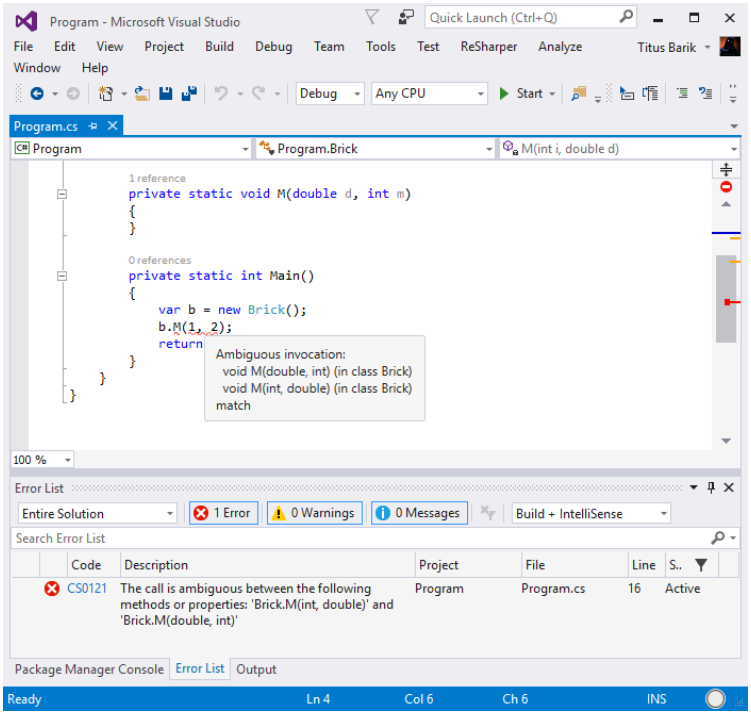
Finally, Hoskote, Kam, Ho, and colleagues [167] introduce the idea of model checker coverage, similar to that of code coverage; they propose a coverage metric to estimate the *completeness* of a set of properties verified by model checking.

3.3 Visual Representations of Program Analysis

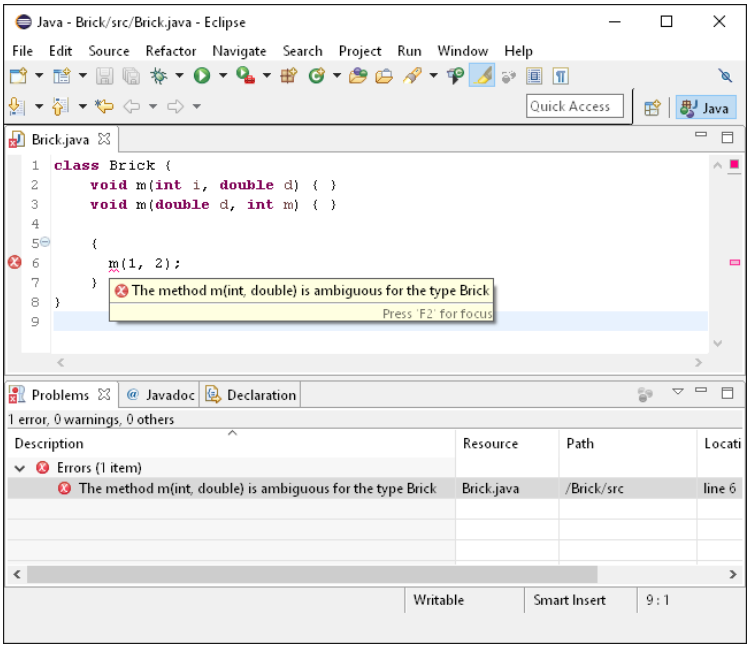
Thus far, we have presented error messages predominantly through terminal, text-only interfaces. In addition to terminals, many software developers utilize a modern integrated development environments (IDE) such as Eclipse [106], Visual Studio [379], and IntelliJ [170], as a core part of their development process [356]. These environments are intended to increase developer productivity by bringing together multiple tools and making these tools accessible within a unified experience [84].

Unsurprisingly, error messages from program analysis tools are also made accessible within these environments: the diagnostic objects (as discussed in Section 3.2.1 on page 26) are unbundled and rendered them through appropriate interface elements in the IDE. Consider the Visual Studio IDE in Figure 3.2a on the following page, with a project file containing an call to an ambiguous method. The IDE presents the error messages to the developer through several interface elements. The Error List window at the bottom of the IDE displays the current error messages in the project. Within this window, errors are searched, sorted, and filtered. The diagnostic object is unbundled—and its components, such as the description, file, and line number, are presented as individual columns in the table. Within the source code editor, a red wavy underline is present under the corresponding source code that generates an error message. The developer can hover over the wavy underline to reveal a tooltip containing the error message description. In addition, the margin contains visual indicators that provide an overview of where problems are in the current file; unit testing tools extensions such as NCrunch [262] and dotCover [102], leverage the margin indicators, as well as source code highlighting, to indicate unit test coverage (Figure 3.3).

The Eclipse IDE in Figure 3.2b on the next page uses similar affordances as

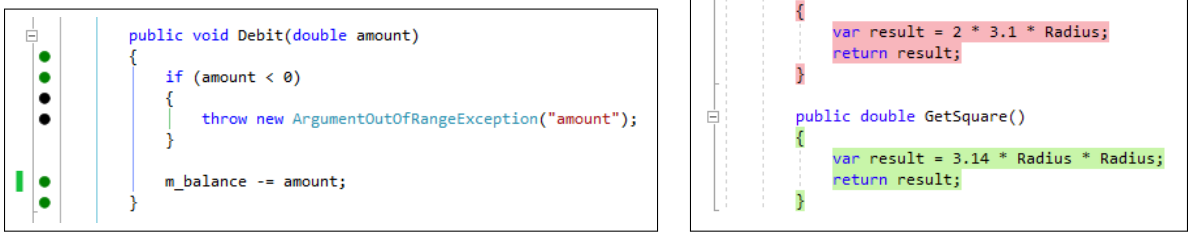


(a) Visual Studio



(b) Eclipse

Figure 3.2 Modern IDEs have converged on affordances for presenting error messages to developers.



(a) Highlighting markers **(b)** Colored backgrounds

Figure 3.3 NCrunch and JetBrains dotCover are concurrent unit testing and code coverage tools that integrates with Visual Studio. Shown here are two methods for displaying code coverage information: (a) in NCrunch, as highlighting markers in the margin, or (b) in JetBrains dotCover, as colored backgrounds on the source. Green means that tests pass, red indicates that at least one test that covers the statement fails, and black or gray shows uncovered code.

Visual Studio, and the popular IDEs such as Atom, Xcode, and Sublime Text, appear to have converged on how they present errors to developers.

Several research tools explore diagrammatic, box-and-arrow representations of error messages. For example, the Refactoring Annotations tool by Murphy-Hill and Black [255] displays diagrammatic control flow and data flow information and overlay this information on the source code; through these annotations, developers understand the causes of refactoring errors significantly faster and more accurately than standard Eclipse error messages. The study on visual compiler error messages in Chapter 5 is influenced by refactoring annotations. MrSpidey is a user-friendly interactive static debugger for scheme [121]; the tool assists developers in pinpointing run-time errors, and uses arrows overlaid on the source code to explicate portions of the value flow graph to the developer. A relatively recent implementation of MrSpidey can found in its spiritual successor, DrRacket [302]. Whyline renders arrows between related elements in source code files, and fades the rest of the code; the tool also automatically arranges windows when arrows span multiple files [198].

The Rust Enhanced [318] extension for Sublime Text inlines error message information through the *phantoms*. Phantoms are like tooltips, but the supplemental information is directly embedded within the text view and remains persistent. Lieber,

Brandt, and Miller [218] found that developers adopt debugging strategies that are unique to this form of always-on information, such as navigation through the code.

Representations of error messages also present information alongside the source editor. Risley and Smedley [310] implement a visualization for compiler errors in Java; they contribute a visual syntax that renders diagrammatic representations for incorrect assignments, type checking, and exceptions. The Stench Blossom tool is an ambient visualization composed of semi-circles on the right-hand side of the editor pane; each sector, calls a petal, corresponds to a category of errors [254]. The tool is tailored for situations where there may be multiple, simultaneous issues within the code—and where identified issues would require the experience of the developer to judge whether the issue actually needs to be addressed. The study finds that ambient visualizations help developers make more informed judgements about the code they have written.

Other bells and whistles for communicating errors to developers, such as dashboards and e-mail notifications, are described in the dissertation by Johnson [180].

3.4 Errors Developers Make

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 [329]. 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 [329]. 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[3], the situation is even worse—through telemetry of over 37 million compilation events, Altdmri and Brown identified that nearly 48% of all compilations fail [6]. 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.

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

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 [298]. 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 [329].

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 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.

3.5 Design Guidelines for Error Messages

Design guidelines are rules that a designer can follow to increase the usability of the tool [100]. Researchers have contributed design guidelines to the domain of errors messages, here we present these contributions as a brief chronology:

1967 Moulton and Muller [251] construct guidelines and apply them to the design of DITRAN, a diagnostic compiler for FORTRAN. The DITRAN compiler is a compile-and-go batch processing systems, but the authors suggest that the design guidelines could be applied to interactive environments. Many of these early design guidelines—such as describing errors in terms of the source language, and ideally suggesting corrections—are found in modern program analysis tools.

1974 Horning [165] investigates error messages from the perspective of communication: as conversations with the compiler that are initiated by the user,

with feedback from the compiler. Horning proposes characteristics which good error messages will exhibit.

1982 Dean [89] argues for design guidelines that emphasize humans goals, such as helping people correct errors as easily as they make them, and giving people control over the messages they receive.

Shneiderman [337] conducted controlled experiments with novice developers and had them repair COBOL Programs. The developers repaired programs under different error messages presentations, including 1) a ? for the message, 2) a brief error message, 3) standard system messages, and 4) improved messages. These initial experiments led Shneiderman to recommend design guidelines for error messages, such as having a positive tone, being specific and addressing the problem in the user's terms, and placing the user in control of the situation.

1983 Brown [48] proposes design guidelines which take advantage of high-resolution displays and windowing—such as print several lines of source code alongside the error message, and using color to identify the offending program elements.

1986 Kantorowitz and Laor [188] observe that current compilers produce in some situations wrong error messages that mislead the developer and harm their confidence in the system. They propose error messages guidelines that follow from a prime requirement to avoid inaccurate messages.

1989 Shaw [334] criticizes the formal, intimidating, and vague error messages in the APL. Shaw [334] suggests some overall directions to reduce the feeling of frustration and alienation with error messages. The contributed design guidelines are expected to produce error messages that are more revealing, and more user-friendly.

1990 The National Cryptologic School [261] contends that designers and programmers of error messages do not take into account that error messages are designed for a person, and that human beings are not perfect. They propose eight guidelines designed to increase productivity and developer satisfaction.

2010 Traver [369] offers a human-computer interaction perspective on compiler error messages. Through a systematic literature review, Traver contributes

a set of desirable characteristics of messages in compilers. The guidelines are derived from examples of compiler errors and the author's experience as a developer, and are congruent to those identified by Horning [165].

2012 Murphy-Hill and Black [255] propose guidelines for presenting error messages related to refactoring. They implement an alternative to textual error messages through a Refactoring Annotations plugin for the Eclipse environment. Refactoring annotations are diagrammatically overlaid on program text. The authors contribute guidelines derived from comparing textual error messages against refactoring annotations.

2013 Murphy-Hill, Barik, and Black [254] develop an ambient visualization tool for a category of error messages called *code smells*. Code smells are a form of recommendations that are intended to help developers create high-quality artifacts, but may turn out to be bad advice. The authors explicate the usability characteristics of the tool through design guidelines. In comparison to refactoring annotations [255], guidelines such as partiality, availability, and unobtrusiveness are tailored for ambient presentations of error messages.

2015 Sadowski, Gogh, Jaspan, and colleagues [319] present Tricoder, a program analysis platform used at Google for building data-driven ecosystems around program analysis. The authors apply their experiences and philosophies regarding program analysis to four design guidelines that inform when and how to incorporate new error messages into Tricoder. Like Kantorowitz and Laor [188], they suggest that program analysis tools minimize errors messages that are false positives.

The full set of guidelines can be found in Appendix E. In Section 8.2 on page 142, we contribute design guidelines derived from the studies on rational reconstruction in this dissertation.

4 | Do Developers Read Compiler Error Messages?

Don't blink. Blink and you're dead.

The Doctor

4.1 Abstract

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

[1]This chapter was previously published in: T. Barik, J. Smith, K. Lubick, and colleagues, "Do developers read compiler error messages?" In *Proceedings of the 39th International Conference on*

4.2 Introduction

Compilers are notorious for producing cryptic and uninformative messages [64, 183, 329, 369]. For example, a missing symbol, type mismatch, or incorrect dependency can create situations where error messages can produce misleading or hard to digest information [298]. Unfortunately, compiler errors happen frequently: Seo and colleagues empirically obtained build failures from over 26 million builds at Google [329], and found that over a quarter of builds fail due to compiler errors.

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

However, there has been limited research on understanding how developers perceive and comprehend error messages through the various ways in which they are presented, for novice and expert developers alike. For example, Denny and colleagues speculated that improvements to error messages were unsuccessful because their students didn't read them [93]. Marceau and colleagues proposed a Read-Understand-Formulate theory, but were unable to confirm whether participants actually read the messages; they suggested the use of eye tracking to provide this missing evidence [230]. In industry, Seo and colleagues provided a distribution of "costly" compiler errors introduced by their expert developers, but their methodology cannot explain *why* these errors are costly [329]. In short, we have limited insights into how developers process, or attend to error messages, during the comprehension and resolution of defects.

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

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

- Participants read error messages; unfortunately, the difficulty of reading error messages is comparable to the difficulty of reading source code—a cognitively

Software Engineering, ser. ICSE '17, Buenos Aires, Argentina: IEEE Press, 2017, pp. 575–585.

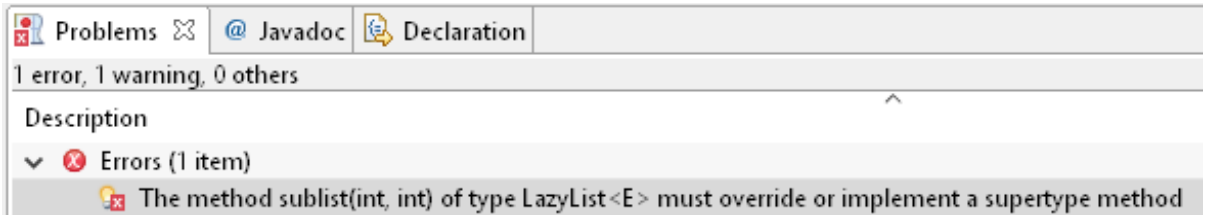
demanding task.

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

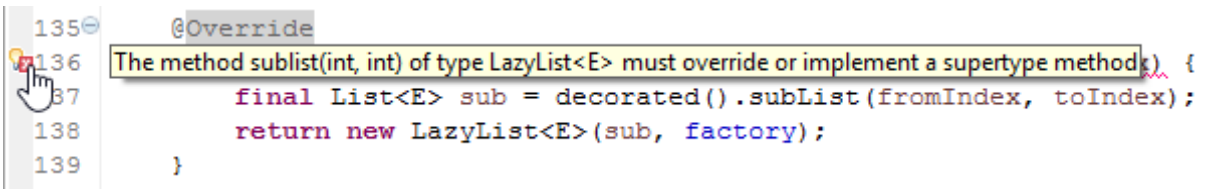
4.3 Motivating Example

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

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

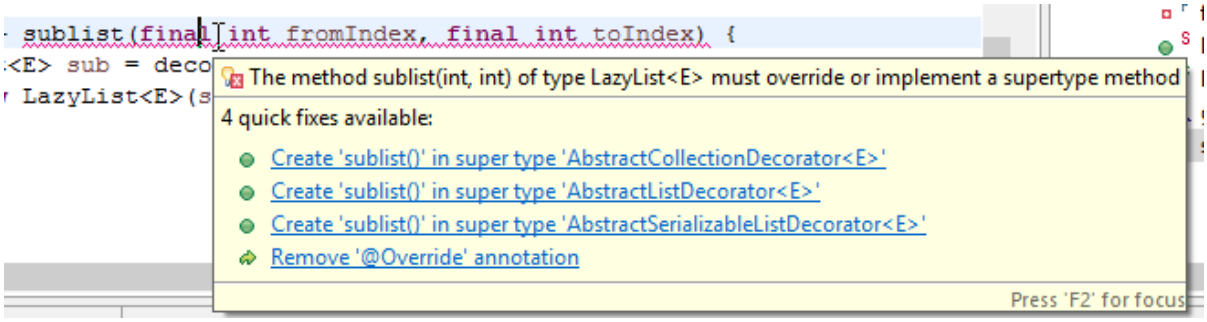


He silently reads the message about the `sublist` method, and then double-clicks the error in the problems pane. This redirects the IDE to the source editor, and Barry confirms that the error is related to the code that he just added. In the margin of the source editor, he now notices a light bulb icon,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

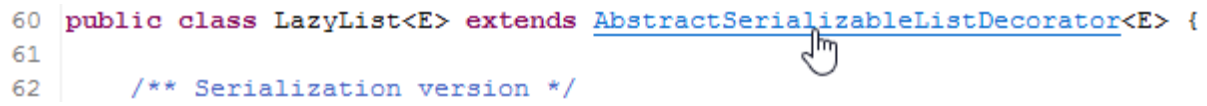


Unfortunately, the popup is less helpful than he expected because it repeats the message he has already seen in the problems pane. Moreover, the error popup text is obscuring the method signature which is where he believes the problem is actually located.

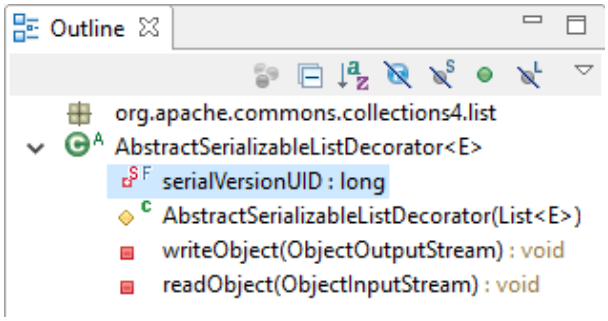
Next, he notices the red wavy underline, which in Eclipse indicates the presence of an error. Barry hovers over the underline, releaving a *Quick Fix popup*. Unlike the error popup, the Quick Fix popup provides possible "fixes" that change the source code, in addition to the error message. He spends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



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



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



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

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

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

```
106    public List<E> subList(final int fromIndex, final int toIndex) {
107        return decorated().subList(fromIndex, toIndex);
108    }
```

Worse, this scenario is not isolated to Barry. For example, the highest-rated post on StackOverflow for the `@Override` annotation error suggests that it commonly occurs in situations where method names have been misspelled.[2]

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

4.4 Methodology

4.4.1 Research Questions

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

RQ1. How effective and efficient are developers at resolving error messages for different categories of errors? We ask this research question to assess the representativeness of our experimental tasks with respect to the costly error messages identified by Seo and colleagues [329], where costly is defined as frequency

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

of the error times the median resolution time. Additionally, the results of this research question provide descriptive statistics to identify if some categories of defects are more difficult to resolve than others.

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

RQ3. Are compiler errors difficult to resolve because of the error message? Resolving compiler errors within the IDE requires developers to perform a combination of activities, such as navigating to files and making edits to source code. One hypothesis is that certain compiler errors are difficult to resolve not because the error message itself is cryptic, but because the task requires intricate code modifications in order to address the defect. Alternatively, it may be the case that the resolution requires only a simple code change to correct the defect, but a confusing error message hampers the developer from discovering the required code change. In short, we want to understand the extent to which poor error messages are harmful to the developer.

Table 4.1 Participant Compiler Error Tasks

Task	Error Message[1]	Package	Category	Defect Introduced
T1	`The method sublist(int, int) of type LazyList<E> must override or implement a supertype method`	List	Semantic	Renamed `sublist` to `subList`, breaking existing `@Override` annotation.

T2	`The type CursorableLinkedList<E> must implement the inherited abstract method List<E>.isEmpty()`
	
	`The type NodeCachingLinkedList<E> must implement the inherited abstract method List<E>.isEmpty()`
		List	Semantic	Deleted `isEmpty` method from abstract parent class.

T3	`The import org.apache.commons.collections3 cannot be resolved ( ... repeated 50 times)`	Map	Dependency	Changed version of `collections4` to non-existent `collections3` library in `import` statements.
T4	`The method get() is undefined for the type Queue<E>`	Queue	Dependency	Renamed method invocation from `element()` to non-existent `get()`.
T5	`The method add(E) in the type Collection<E> is not applicable for the arguments (int, capture#8-of ? extends E)`	Set	Type mismatch	Added additional argument of `0` to `add` method call.

T6	`Type mismatch: cannot convert from Set<Map.Entry<K,V>> to Set<Map.Entry<V,K>>`
	
	`Type mismatch: cannot convert from Set<Map.Entry<V,K>> to Set<Map.Entry<K,V>>`
		Map	Type mismatch	Swapped key and value in dictionary from `Entry<K,V>` to `Entry<V,K>`.

T7	`Unhandled exception type InstantiationException`	Map	Other	Changed less specific exception `Exception` to `IllegalAccessException`, which is not thrown by the code.
T8	`Duplicate method next() in type EntrySetMapIterator<K,V>` `Duplicate method next() in type EntrySetMapIterator<K,V>`	Iterators	Other	Copied and pasted `next` method to create duplicate method.
T9	`Cannot make a static reference to the non-static type E`	Queue	Semantic	Added `static` modifier to `readObject` method.
T10	`Syntax error on token "default", : expected after this token`	Map	Syntax	Removed `:` from `default:` in `switch` statement.

[1] For each error message, we compile the defective version of the code under Open JDK to replicate the compiler-internal error message key from the Seo and colleagues study at Google [329]. The Eclipse version of this message is shown to the participants.

4.4.2 Study Design

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

Siegmund recommends self-estimation questions as a good indicator for judging programming experience, especially when participants are students [342]. Following this guidance, we asked participants about their familiarity with Eclipse and their knowledge of the Java programming language. Participants self-rated their familiarity with the Eclipse development environment with a median rating of "Familiar with Eclipse (3)," using a 4-point Likert-type item scale ranging from "Not familiar with Eclipse (1)" to "Very familiar with Eclipse (4)." Participants self-rated their knowledge of the Java programming language with a median rating of "Knowledgeable about Java (3)," using a 4-point Likert-type item scale ranging from "Not knowledgeable about Java (1)" to "Very knowledgeable about Java (4)." Participants also self-reported demographic data. Participants reported a mean age of 24 years ($sd = 6$), 46 reported their gender as male, and 10 reported their gender as female.

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

Tasks. We derived tasks in our eye tracking experiment from prior work conducted by Seo and colleagues, where they empirically obtained build failures from over 26 million builds at Google [329]. From this data, they identified costly error messages that occurred frequently in practice and were time consuming for developers to resolve. To constrain the study to under one hour, we selected the top errors from each category of costly error messages, for a total of ten error messages (Table 4.1). Through an informal pilot study with two other lab members, we found that developers resolved each defect in under five minutes.

However, we did not have access to the actual source code which generated the errors in the Google study. As a substitute, we used the Apache Commons Collections[3] library and manually injected faults into this library to generate error messages.

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

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

For each error, we introduced the error message into the Apache code through operations that could reasonably occur in actual development practices. For example, the `@Override` misspelling described in the motivating example (Section 4.3) was applied based on comments on StackOverflow.

Tools and Apparatus. Participants used a Windows 8 machine with a 24-inch monitor, having a resolution of 1920x1080 pixels. The computer was connected to a GazePoint GP3 [130] eye tracking instrument, and this instrument was positioned directly below the monitor. GP3 software and drivers were installed on the computer to collect both the screen recording of the desktop environment and to synchronize the time of the recordings with the eye tracking instrument. Participants used an external keyboard and mouse to interact with the computer. The experimenters also installed custom scripts on the machine so that they could remotely load participant tasks.

We choose the Eclipse IDE [106] for this study because its presentation of errors, for example, through the problems pane and Quick Fix popups, are characteristic of the way errors are presented in other modern IDEs such as IntelliJ [170] and Visual Studio [379]. A default Eclipse installation was deployed on the machine, with minimal customizations. Specifically, we disabled Eclipse themes and turned off rounded edges on windows to facilitate subsequent detection during the data cleaning phase of the research.

4.4.3 Procedure

Onboarding. All participants signed a consent form before participating in the study.[4] Using a script, the experimenter verbally instructed participants with the details of the study. Participants were informed that they would be identifying and resolving ten source code defects, to be presented as compiler error messages in their IDE. Participants were given five minutes per task. If the participants finished

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

early, they were asked to alert the experimenter and proceed to the next task. After two minutes, participants were also provided the option to discontinue the task.

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

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

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

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

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

4.4.4 Data Collection and Cleaning

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

Data cleaning of titles. We used the OpenCV computer vision library [272] to process videos on a frame-by-frame basis. To obtain the currently opened source file, we used the Tesseract OCR engine to identify the titlebar for each frame [350]. Due to errors in OCR translation, we performed two data cleaning steps on the title. First, we cropped each frame to only the title bar and scaled it by a factor of three to artificially increase the font size. Second, we modified Tesseract to recognize only alphanumeric characters, dash (`-`), period (`.`), and forward slash (`/`).

After Tesseract processing, several OCR errors remained. Thus, we applied a Gestalt pattern matching algorithm [306] to match the OCR'd title against the known set of all Java files in the Apache Commons Collection library.[5] We manually added the strings, `Java - Eclipse`, which appears in the title when no file is open, as well as several classes from the `java.util` package to this processing step. The output of this step a list of the title associated with each frame.

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

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

To support automatic detection of each of the six areas of interest, we implemented several techniques. For fixed-sized AOIs, such as the popups, we extracted isolated screen captures of each popup and saved them as *templates*. For dynamically-

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

sized AOIs, we extracted the boundaries of the essential features of the elements, and then performed a calculation to dynamically compute its bounding rectangle. For example, to identify the source editor, we internally match against three *sub-templates*: the top-left tab, the top-right minimize button, and a thin horizontal line that delimits the source code from any panes below it.

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

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

After performing offset adjustment, we used the GazePoint software to extract a list of fixations. For each record in the list, the record contains the time it began, its duration, and its coordinates.[6]

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

4.5 Analysis

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

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

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

For participant effort, we calculated a metric called response time effort [397]. Intuitively, if incorrect solutions are achieved in significantly less time than correct solutions, then it would suggest that participants are not expending sufficient effort to reasonably resolve a compiler error message. That is, the participant may simply be careless, irrespective of the quality of the compiler error message or the difficulty of the task. We performed a two-tailed t-test between task times, excluding timeouts, under correct and incorrect solution conditions to gauge response time effort.

4.5.2 RQ2: Do developers read error messages?

Determining if developers are reading error messages through overt experimental designs is surprisingly tricky. For example, asking participants to think aloud as they resolve error message tasks adds a cognitive burden that has been found to negatively impact task performance [145]. Directly questioning the participant at the end of each task can introduce social-desirability and prime the participants' behavior for subsequent tasks [392], thus causing them to read error messages

more carefully than they otherwise would have. Moreover, visual attention is a largely subconscious process; participants in visual attention tasks, such as reading, only have a rudimentary awareness of how they allocate their attention [151]. The use of eye tracking to answer this research question mitigates these issues, but introduces one of its own: eye tracking data is noisy. For example, routine, involuntary movements such as rubbing eyes and adjusting hair can block the eye tracking camera, introducing spurious data points. Our analysis techniques are constrained to those that are robust in the presence of noise.

Previous eye tracking work by Rayner modeled "reading" as distribution times of fixations under a variety of visual stimuli [307]. Rayner characterized the distribution times of fixations under different reading conditions, such as silent and oral reading. Through a meta-analysis, Rayner found that the mean fixation time of a distribution increases with the difficulty of the text.

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

We then characterized the distribution of source code, error messages, and silent reading (provided by Rayner [307]) through a symmetric Kullback-Leibler (KL) divergence, for each of pair of distributions. Essentially, KL divergence is an information-based measure of disparity: the larger the value of the divergence between two distributions, the more information is "lost" when one distribution is used to model the second distribution [187].

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

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

Although Seo and colleagues identified a distribution of costly error messages [329], this does not automatically imply that the reason the error message is costly to resolve is due to the error message itself. For example, an error message about a mismatched brace may be easy to comprehend for the developer, yet costly to resolve because the developer must spend most of their effort in the source code editor to identify where to add or remove a brace. In answering this research question, our goal is specifically to understand the extent to which difficulties with reading error messages can be attributed to task performance difficulties.

To understand if compiler errors are difficult to resolve because of the error message, we used the eye tracking measure of *revisits*, that is, leaving an area of interest and then subsequently returning to it, as a measure of reading difficulty. In prior work, Jacobson and Dodwell identified the relationship that increasing fixation revisits to an area of text is a measure of increased reading difficulty for that area [173].

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

To evaluate the model, we computed the Nagelkerke's coefficient of determination, R^2 [257]. Of course, there are many variables that influence whether or not a developer will be successful at resolving a compiler error, such as previous knowledge, experience, and familiarity with the code [182]. Consequently, we expect that the coefficient of determination will be low, because reading difficulty is only a second-order variable for these other factors. Fortunately, reading difficulty latently encodes many of these variables: if a developer has little experience with a particular error message, this lack of experience should manifest itself through how they read the error message.

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

4.6 Verifiability

To support replication of our findings, we have provided several materials on our website.[7] The website contains the videos of each of the participants' tasks. Additionally, we provide a data file containing all gazes for the tasks, and a cleaned data file containing only the fixations for the tasks. To support verification, we provide annotated diagnostic videos of participants' tasks that display rendered rectangles on identified areas of interest as the video plays. Finally, we provide data files from the output of our data cleaning process.

[7] `http://go.barik.net/gazerbeams`

Table 4.2 Overview of Task Performance

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

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

4.7 Results

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

Table 4.2 summarizes the solution the developer makes based on our correctness criteria. For every task, at least one participant made a code change congruent with the correct solution, which indicates that it is at least possible to make a correct fix for every task given the information in the error messages. The distribution of correct and incorrect solutions are clearly skewed for many of the tasks. For example, only two participants generated correct solutions for T1, and only one participant generated a correct solution for T2. And for tasks T5, T6, T8, and T10, nearly all or all participants arrived at the correct solution.

The n_{bins} columns in Table 4.2 indicates, for every incorrect solution, the types of solutions provided by the participants. For example, consider T1, the problem Barry faced in our motivating example (Section 4.3). Recall that the correct solution to

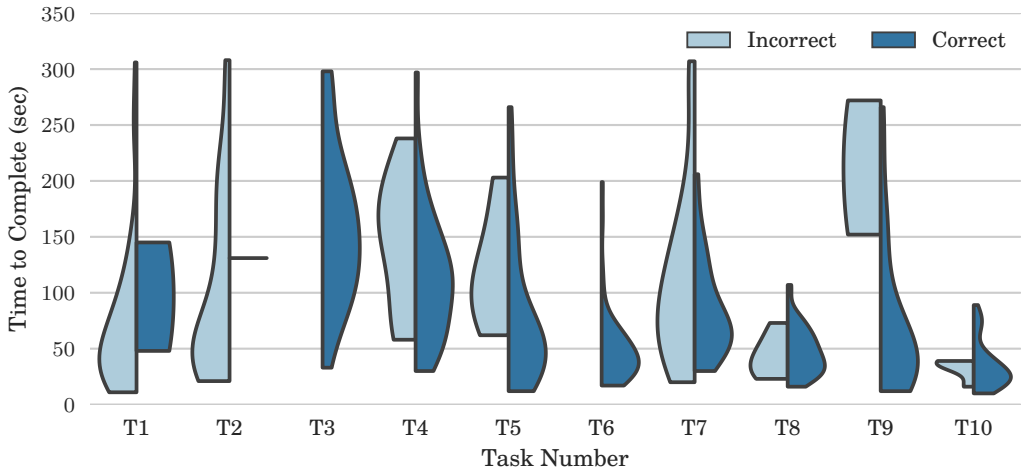


Figure 4.1 The time required for developer to commit to a solution that is correct or incorrect. Nearly all tasks (exceptions, T8 and T10) have high variance in resolution time to arrive, irrespective of correctness.

this problem is to rename the `sublist` method to `subList`. Participants provided two incorrect solutions to this problem. They either removed the `@Override` annotation, which suppresses the error but does not resolve the defect, or they created a stub `sublist` method in the parent class, without recognizing the existing similarly-named method. Across all tasks, participants converged to a small set of incorrect solutions.

Figure 4.1 illustrates a violin plot of the time required for developers to apply a resolution for both correct and incorrect solutions. The dashed lines indicate quartile boundaries for each task. For incorrect solutions, timeouts are excluded from the plot. Like Seo and colleagues, we also found large variation in the time required to arrive at a solution for nearly all tasks [329]. For some tasks, however, such as T8 and T10, most participants were able to correctly resolve the defect, and with relatively tight variation in time to resolution. As Seo and colleagues defined costly in terms of both frequency of occurrence and median time to resolution [329], it is likely these errors are costly because they are arise frequently as a nuisance for developers, not because they are particularly difficult to resolve.

For response time effort, a t-test identified a significant difference in resolution time between correct and incorrect solutions ($t(20.24) = 2.86$, $p = 0.0045$), with incorrect solutions requiring an additional mean time of 20 seconds ($sd = 7$) over the

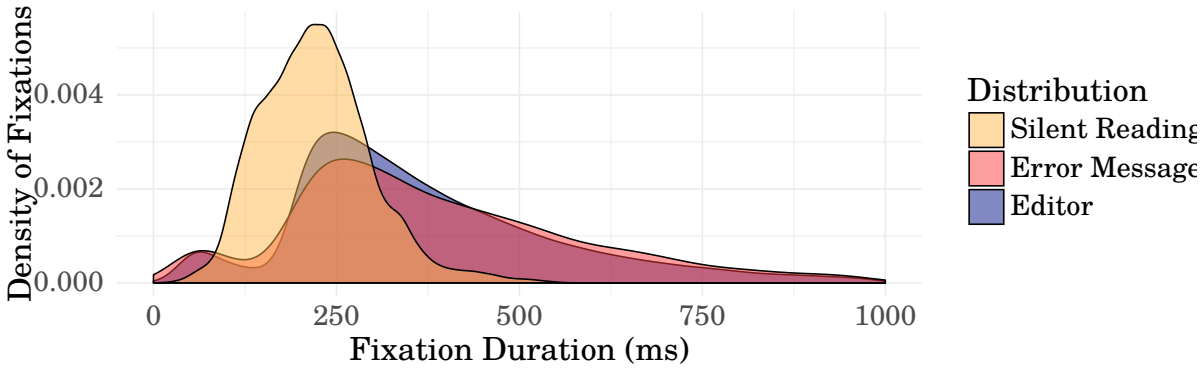


Figure 4.2 Comparison of fixation time distributions for silent reading of English passages, reading source in the editor, and reading of error messages.

correct solution. The results of this test provide support that participants provided sufficient effort in attempting to solve the task, and rejects the hypothesis that participants chose an incorrect solution because it most quickly resolved the defect.

4.7.2 RQ2: Do developers read error messages?

Figure 4.2 illustrates the distribution of known silent reading durations against the distribution of fixation times for error message areas of interest in our tasks. For visualization purposes, the distributions are normalized as a probability density function. That is, the probability of a random fixation to fall within a particular region is the area under the curve for that region.

The mean fixation time for reading in the source code editor is 394ms ($sd = 240$, $n_{\text{fix}} = 81098$). In comparison, previous work found that silent reading of English passages of text yield mean fixation times of 275ms ($sd = 75$), and that for reading and typing English passages yield a mean fixation time of 400ms (sd not provided in original study by Rayner) [307]. Thus, reading source code is much more difficult than reading a general English passage, and marginally less difficult than the activity of typing while reading.

We compute the KL divergence between the source editor distribution and error message distribution ($D_{KL} = 0.059$), source editor distribution and silent reading distribution ($D_{KL} = 3.38$), and error message distribution and silent reading distribution ($D_{KL} = 2.37$). From the relatively small KL divergence between the

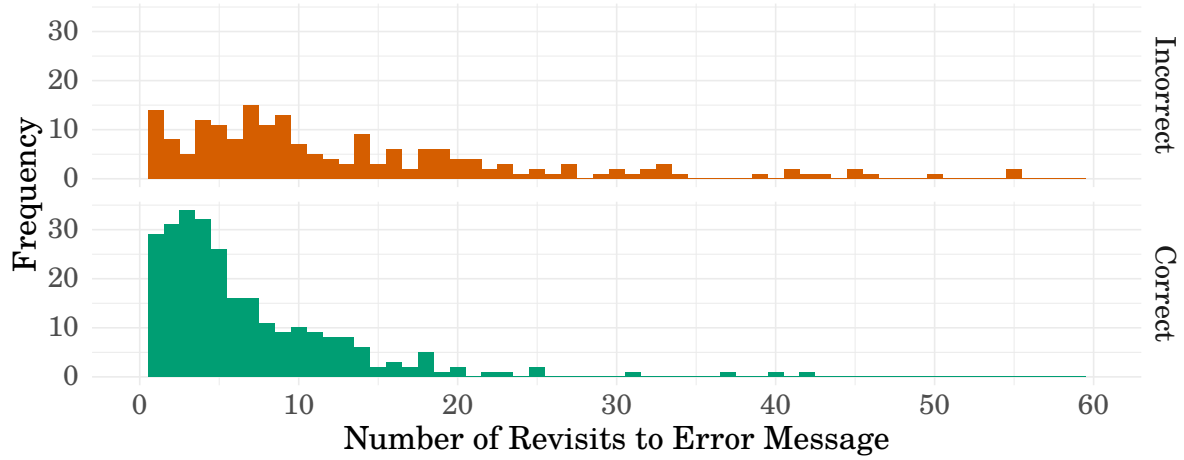
Table 4.3 Participant Fixations to Areas of Interest

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

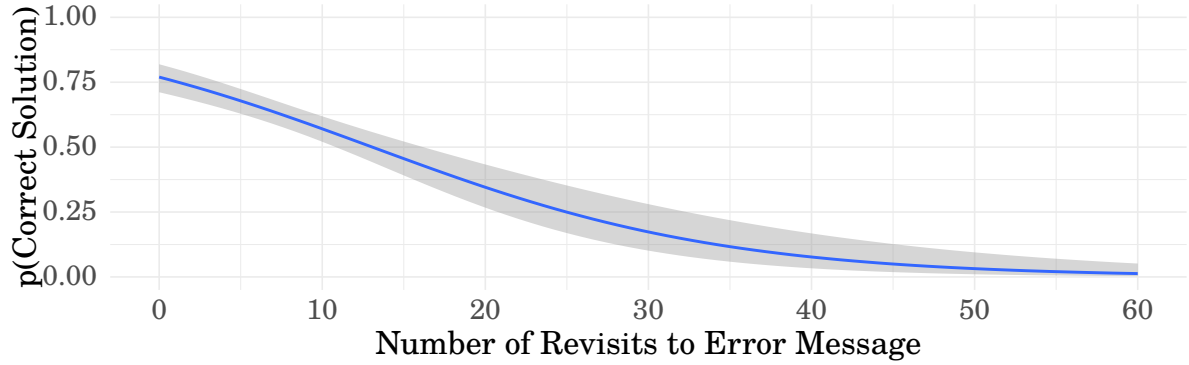
[1] To understand reading, the error areas breakdown aggregates areas of interest to those that provide the text of the error message.

source editor distribution and the error message distribution, we conclude that error message reading is comparable to source code reading ($u = 419$ms, $sd = 270$, $n_{\text{fix}} = 18573$), and unlikely to be a different activity than reading.

Another perspective on understanding whether developers read error messages is to examine how they allocate their fixations across different areas of interest during the task (Table 4.3). Across tasks, we found that participants spent 65%–80% of their fixations in the source editor, with 13%–25% of their fixations on error message areas of interest. Most participants use the error message information in either the problems pane and Quick Fix popup; the error popup is rarely used.



(a)



(b)

Figure 4.3 In (a), histogram of correct and incorrect task solutions by number of revisits on error message areas of interest. In (b), nominal logistic model of the probability of applying a correct solution number by revisits on error message areas of interest. As revisits to error messages increase, the probability of successfully resolving a compiler error decreases.

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

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

A Chi-squared test reveals a statistically significant difference between the number of revisits and the outcome of correctness over a distribution of tasks ($df = 1, G^2 = 60.9, p < .0001$). This suggests that the number of revisits, a proxy measure for reading difficulty [173], is a significant predictor variable for the task difficulty. However, Nagelkerke's coefficient of determination for the logistic fit is low ($R^2 = 0.16$), which implies that reading difficulty is only one of many factors that contribute to the overall difficulty of a task.

Figure 4.3 presents this statistical result more intuitively. From Figure 4.3a, we can see that as the number of revisits increases, the distribution of correct solutions rapidly diminishes. We also see a long-tail of incorrect solutions. Figure 4.3b plots a nominal logistic model for the number of revisits to error messages against the probability of the developer applying a correct solution for the task. The probability of a correct solution also diminishes as revisits increase. Thus, task difficulty is attributable to the reading difficulty of error messages.

4.8 Discussion

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

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

This causes the compiler to emit two error messages:

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

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

For this message, it is not surprising that developers would be led to believe that they should implement the `isEmpty()` method in both classes. Indeed, all participants except for one came to this incorrect conclusion (Table 4.1, $n_{\text{bin}} = 1$).

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

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

Given the fact that our participants took cue from the error messages in the first error message set, it is plausible that participants would have arrived at the correct solution, adding the missing method to the `AbstractLinkedList<E>` class, if they had instead been presented with the second error message. Unfortunately, it's difficult for the compiler to know which of these messages would be more appropriate to present to developer without having situational information, such as recent edit history.

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

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

To postulate why this might be, let's return to Table 4.1. Consider an error message as in T4, which reads:

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

Even for relatively short error messages like this one, participants spent 14% of their time in the total task with fixations across essentially nine "words." Prior research in language cognition for bilingual sentence reading has found that switching languages is associated with a cognitive processing cost [247]. Similarly, one explanation for the apparent difficulty of reading error messages is that error messages consist of both natural language ("`is undefined for`") and code ("`Queue<E>`"), but are not entirely either. Consequently, developers must context-switch between two different modalities of reading to fully capture the information presented in an error message, leading to increased reading difficulty.

A second explanation for why error messages are comparable in difficulty to reading source code is because reading error messages requires the developer to

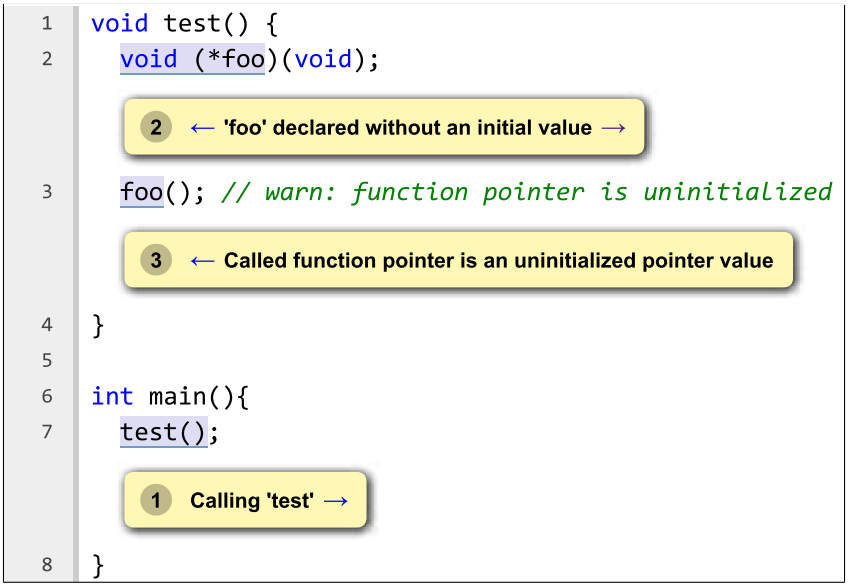


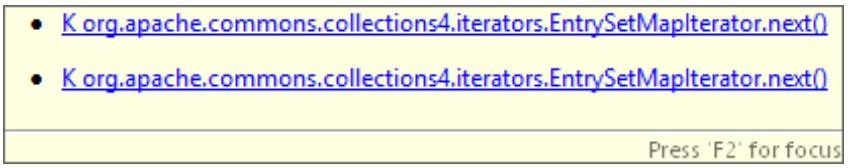
Figure 4.4 Emerging error reporting systems like LLVM scan-build provide stark contrast to those of conventional IDEs. Here, scan-build presents error messages for a potential memory leak as a sequence of steps alongside the source code to which the error applies.

also switch between the error message and the source code in order to understand the full context of the task. For example, a developer might read "`The method get()`" and then suspend their reading of the error message to determine in the source editor the calling context of this method and figure out why and whether it should be called. In this case, error presentation approaches such as those found in LLVM scan-build [66] may prove beneficial to developers (Figure 4.4). Unlike conventional error messages, which decouple the error message from the code context, scan-build presents the error as a sequence of steps that the developer can follow alongside in the context of the code to which the error message applies.

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

In observing the participant videos for these tasks, we found several instances where participants attempted to use tools that fail in the presence of a compiler error message during navigation-related tasks. As one example, a participant in T8

attempted to navigate to the appropriate method through a usage of that method. Although the Eclipse IDE would reveal both locations, it makes no special effort to distinguish the two methods, leading to potential visual disorientation within their IDE [8]:



In T8, yet another participant was aware of a tool within the IDE to facilitate comparison between two methods, but they could not recall how to invoke it. This example illustrates how tools that support understanding of a defect may be just as important as tools that provide a resolution. But unlike Quick Fix popups, which appear automatically, comprehension tools such as `Compare With` must be invoked manually by the developer. Analogous to Quick Fix popups, perhaps toolsmiths should offer "Quick Understanding" popups, which recommend appropriate tools, such as `Compare With`, that are known to be helpful in understanding a particular compiler error. Our own work in defect resolutions provides a starting point for automatically bringing relevant tools to the developer to help with comprehension [17].

4.9 Limitations

Although we derived our errors based on frequency and difficulty of resolution from a prior Google study [329], we could not ensure that we used the similar phrasing in our replication of error messages. Google uses a proprietary version of OpenJDK with custom messages not available to the general public. We also do not have access to the source code that generated these errors. As a result, the messages we use in Eclipse are not identical to those previously studied. Instead, in our study design, we seeded errors that approximate the scenario described by the original message.

We only investigated ten error messages with our participants. However, research by Seo and colleagues found that improving even the 25 of the top errors would cover over 90% of all the errors ever encountered at Google. A similar result has since been replicated for novice developers in Java [298]. Furthermore, we sampled error messages across multiple categories of defects to increase generalizability.

One construct validity issue is the accuracy of the eye tracker. We used a commodity eye tracking instrument which has a lower sample rate than professional

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

A threat to external validity is that we used student developers in place of full-time professional developers. Although many of our participants had professional experience, these participants may not fully represent industry developers in all situations [322]. For example, it is possible that senior developers with extensive experience of their own code base would arrive at a correct solution for some tasks irrespective of the information in the message (RQ1). Although our participants rarely used error popups in their IDE (RQ2), we might expect that senior developers, again having familiarity with their code base, would utilize these on-demand information sources more frequently than the always-available problems pane. Developer effectiveness for a task and its relationship to error message revisit frequency may be less pronounced when participants have a broader range of developer experience than those in our own study (RQ3). Thus, we should be careful in generalizing our findings to all developers.

Lastly, our analysis is also limited to insights that can be obtained through eye tracking measures. A future study in which participants perform a gaze-cued, retrospective think-aloud on their own recordings could yield additional insights on participant behavior [112].

4.10 Related Work

To our knowledge, this study is the first to use eye tracking to explain how developers make use of error message information to resolve defects within the IDE. In this section, we discuss related work from eye tracking studies in debugging and defect understanding. Rodeghero and colleagues used eye tracking to understand how developers summarize code; the results of their experiment were used to improve algorithms that automatically summarize code [314]. Romero and colleagues used eye tracking to understand how developers found defects in source code under different representations of the source code, such as diagrams [316]. Uwano and

colleagues asked developers to perform code review tasks, during which participants had to locate defects in the code [372]. The authors identified common scan patterns in subjects' eye movements. In a partial replication of Uwano's study, Sharif and colleagues found differences between experts' and novices' scan patterns while locating defects [332]. Bednarik and Tukiainen found that repetitive patterns of visual attention were associated with lower performance [28]. In our study, we also found that revisits to error message information is statistically significant with the probability of correctness. Like other research attempting to find patterns in debugging activities, Hejmady and Narayanan found visual pattern differences based on programming experience and familiarity with the IDE [158]. Busjahn and colleagues were interested in how novices read source code; from eye tracking, they found that experts read code less linearly than novices [51]. In our own work, we are interested, for example, in whether developers read error messages at all.

Outside of eye tracking, other studies are related to our investigation of comprehension and resolution of error messages. Researchers de Alwis and Murphy proposed a theory of visual momentum, identifying factors that may lead developers to become disoriented when exploring programs in the IDE [8]. Lawrence and colleagues present an information foraging theory on how developers debug code. They examined programmers' verbalizations and found that their debugging approaches more often concerned scent-following than hypotheses [208]. However, the use of verbalization has been found to affect the performance and decisions participants make on tasks [73].

4.11 Conclusion

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

The results of this study reveal several problematic areas in the way development

environments today present compiler error messages to developers, and identify opportunities for toolsmiths to address these problems. The contribution of our work is that it offers an empirical justification for improving compiler error messages for developers. Stated simply: error messages matter.

4.12 Acknowledgments

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

5 | How Do Developers Visualize Compiler Error Messages?

> Never ignore coincidence. Unless, of course, you're busy. In which case, always ignore coincidence.
>
> The Doctor

5.1 Abstract

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

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

The results of our work suggest that the diagrammatic techniques developers use to explain problems can serve as an effective foundation for how IDEs should visually communicate to developers.[1]

5.2 Introduction

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

Many developers consider these error notifications to be cryptic and confusing [369]. We postulate one of the reasons error notifications are confusing is because compilers do not reveal the reasoning used to determine that the error exists.

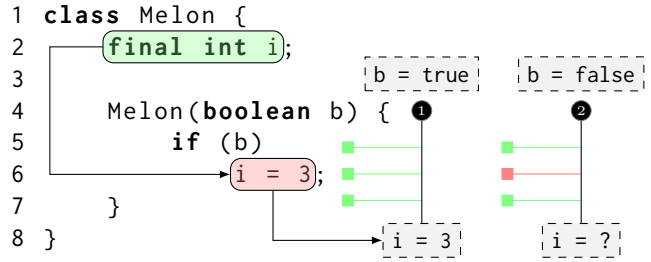
More explicitly, in order to generate an error notification, the compiler begins with the source code, collects information during its compilation, uses that information to identify that a problem exists, and notifies the developer of the problem through the IDE. Yet, for developers to comprehend the notification, they must mentally duplicate this process through *self-explanation* [278] in essentially reverse order—starting with the error notification, the developer must identify what they think the problem might be from the IDE's presentation, mentally collect all of the program components related to this problem, and finally identify the area or areas of source code necessary to correct the particular defect. This self-explanation process, when left solely to the developer, can result in a significant loss of productivity because humans are imperfect and bounded in knowledge, attention, and expertise [197]. *Much of this self-explanation process may be completely unnecessary since the reasoning process that resulted in the error notification was already known to the compiler.*[2]

[1] This chapter was previously published in: T. Barik, K. Lubick, S. Christie, and colleagues, "How developers visualize compiler messages: A foundational approach to notification construction," in *2014 Second IEEE Working Conference on Software Visualization*, Sep. 2014, pp. 87–96.

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

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

(a) Baseline visualization



(b) Explanatory visualization

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

(c) Error message text

Figure 5.1 A comparison of a potentially uninitialized variable compiler error through (a) baseline visualizations, the dominant paradigm as found in IDEs today, (b) our explanatory visualizations, and (c) the textual error message.

Visualizations in IDEs, such as red wavy underlines and margin indicators, take the perspective that compilers are *opaque* black boxes and thus by design are limited in their affordances for helping developers in comprehending error notifications. In this paper, we argue that developers stand to significantly benefit when compilers are made more *transparent* and expose their internal reasoning process to visualization systems. We argue that such systems can leverage these

structures to generate expressive, *explanatory visualizations* that align with the way in which developers self-explain error notifications. Our contributions in this paper are:

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

5.3 Motivating Example

Yoonki is an experienced C++ developer who has recently transitioned to a project that is being developed in the Java programming language. While programming, he encounters a wavy red underline visualization as shown in Figure 5.1a, which indicates an error. The problem seems to be related to `final int i`, which Yoonki recognizes as being roughly similar to the concept of a `const` variable in C++. Yoonki investigates further and notices the full text of the error in the bottom pane of his IDE (Figure 5.1c).

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

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

The problem here is that Yoonki has learned a reasonable heuristic for how constant variables work in programming languages, but his heuristic fails in this case. Like C++, Yoonki is correct in that Java `final` variables can only be assigned once. But unlike C++, `final` variables in Java can be assigned at a point other than the declaration. Yoonki has experienced what we could call a *knowledge breakdown* [197]. In this case, Yoonki has a confirmation bias about how the system is supposed to work, and this false hypothesis has worked reasonably well for him until now.

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

In contrast, consider our approach, shown in Figure 5.1b. Here, Yoonki may not experience the same knowledge breakdown, because the IDE provides a visual explanation of the problem within his source code. Though Yoonki might once again incorrectly assume `final` variables must be assigned at declaration, the visualization implies that the problem is actually related to control flow. Specifically, the explanatory visualization is showing Yoonki there is a code path in which `i` is assigned a value (when `b = true`), and another code path where it is not (when `b = false`). This time, Yoonki correctly fixes the defect by adding an `else` statement to the condition, initializing it with an appropriate value in the case when `b = false`.

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

5.4 Pilot Study

We conducted a pilot study[3] from undergraduate lab sessions in Software Engineering to address a prerequisite research question:

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

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

Table 5.1 Frequency of Visual Annotations in Pilot

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

each other?

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

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

source code listings were unadorned and lacked any visual annotations.

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

We randomly assigned one of four source code listings to each student, pulled verbatim from the OpenJDK 7 unit tests for compiler diagnostics framework. These examples, among others used in subsequent studies, are found in Table 5.3. No students within a pair received the same source code listings. In total, we collected 73 samples: 17 from T1 (23%), 12 from T2 (16%), 20 from T3 (27%), and 24 from T6 (33%). Students did not receive tasks T4 or T5, because they had not been created at the time of the pilot study.

From these annotations we performed two passes over the student responses. In the first pass, we created a taxonomy of visual annotations based on our observations. In the second pass, we classified the student responses using this taxonomy. The aggregated results are shown in Table 5.1.

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

5.5 Explanatory Visualizations of Error Messages

We propose a set of eight visual annotations, which are summarized in Table 5.2. We now concretely describe these annotations using the motivational example from Figure 5.1b. The starting *point* for visual explanation in the source code listing is indicated using `code` (a green rectangle with rounded corners that surrounds a program element). In our visualization mockups, we choose the starting point to be the same as the source of the error identified by IntelliJ (Figure 5.1a). In the example, this is `final int i`.

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

Table 5.2 Visual Annotation Legend

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

the association is indicated by  (a directional arrow).

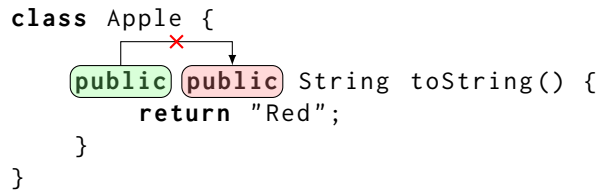
A second association leads the developer to an explanatory code block that contains a copy of the statement. Explanatory code is represented by code (dashed gray rectangle), which indicates the surrounded elements are explanatory and not part of the original source code of the program. This explanatory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

This composite annotation demonstrates that several basic annotations can be combined to create a new annotation for expressing a more complex concept. One of these components is the code coverage annotation. This annotation uses  (green line) and  (red line) to indicate whether or not a line is covered. In addition, the

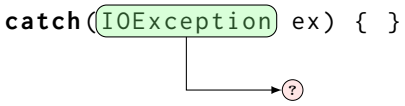
enumerations ❶ and ❷ provide the developer with convenient labels for referring to the branches (for example, “It looks like it works fine in branch 1, but not in branch 2”). The final component is another explanatory code block indicating one possible condition under which the branch would be executed.

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

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



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



5.6 Methodology

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

5.6.1 Research Questions

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

RQ1 Do explanatory visualizations result in more correct self-explanations by developers?

RQ2 Do developers adopt conventions from our visual annotations in their own self-explanations?

RQ3 What aspects differentiate explanatory visualizations from baseline visualizations?

RQ4 Do better self-explanations enable developers to construct better mental models of error notifications?

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

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

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

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

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

5.6.2 Participants

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

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

5.6.3 Selection Criteria for Mockups

Table 5.3 Participant Explanation and Recall Tasks

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

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

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

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

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

We selected our compiler error examples from the OpenJDK diagnostics framework.[4] This framework contains a collection of 382 Java code examples, each of which is designed to generate one or more error messages when compiled.

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

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

Ultimately, we selected messages that we believed could effectively demonstrate the rich explanatory potential of visualizations, while considering the capability of the participants. The selected messages are summarized in Table 5.3.

5.6.4 Mockup Construction Procedure

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

The control group mockups were designed by directly copying the red wavy underline visualizations provided by the IntelliJ IDE for the Java code examples.

[4]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/`.

IntelliJ also provides interactive tooltips for each error, which are shown when the developer hovers over an annotated substring. However, we did not consider these interactive features since we are specifically interested in the contribution of the explanatory capability of the non-interactive visualizations. We chose IntelliJ over the Eclipse IDE because it uses the same text error messages as the command-line OpenJDK compiler, which is important to our experimental design.

The treatment group mockups were informed by a pilot study through which we elicited an initial taxonomy of visual annotations that appeared to be useful to developers when they explained concepts to other developers (see Section 5.4). We used the annotations from this pilot experiment as a foundation for manually drawing visual annotations for six of the error messages. We used our own experiences with compiler technologies such as Roslyn[5] to render visualizations that we think are plausible for compilers to render if they expose the appropriate data structures to a visualization system.

5.6.5 Investigator Training

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

5.6.6 Experimental Procedure

5.6.6.1 Assignment

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

5.6.6.2 Recording

Participants filled out an informed consent form and indicated whether or not they wanted their audio (used in Phase 1 and 2) and screens (used in Phase 2) to be

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

recorded. For participants who agreed to be recorded ($n = 26$), we used desktop recorder software to record both the audio of the explanations as well as screen interactions during the experiment.

5.6.6.3 Phase 1: Self-Explanation Phase

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

We sequentially provided participants with six error notifications, presented as paper mockups that resembled an IDE. For the mockup, the source code of the OpenJDK file was minimally modified using a random-noun generator to make the class and method names more pronounceable. These tasks are summarized in Table 5.3, and we presented the tasks to the participants alphabetically by Task Name.

In the control group, participants received paper mockups. containing the baseline red wavy underline visualization, such as in Figure 5.1a. The treatment group received paper mockups containing our explanatory visualization as in Figure 5.1b. Below the source code listing, all participants received the full error message text (Figure 5.1c). In the treatment group, we provided participants with a visual annotation legend (Table 5.2), since these participants did not have prior familiarity with our visualizations. Finally, we provided participants with colored pencils and an unadorned mockup of the IDE having the source code and error message text, but no annotations.

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

5.6.6.4 Cognitive Dimensions Survey

To evaluate the aspects of visualizations that developers find useful in self-explanation (RQ3), participants completed a Cognitive Dimensions of Notations questionnaire (CD) [138], which we simplified for error message notifications. We chose this evaluation instrument over other usability instruments because the analysis is usable by non-specialists in HCI (in contrast with Nielson and Molich's heuristic evaluation [267]). The instrument is also quick to apply, and can be used in an early design phase.

The full CD defines 14 dimensions, but not all of these are applicable to our design. Since our visualizations are currently non-interactive, we eliminated all dimensions that assessed interactivity or were otherwise immaterial to our study, among them, viscosity, premature commitment, and progressive evaluation. This left four dimensions:

Consistency similar semantics are expressed in similar syntactic forms

Hidden dependencies important links between entities are not visible

Hard mental operations high demand on cognitive resources

Role expressiveness the purpose of a component is readily inferred

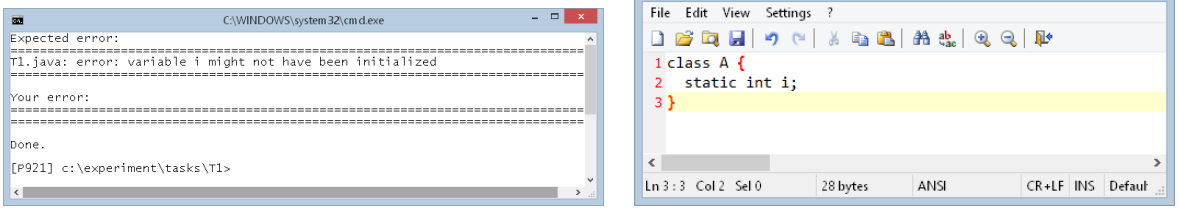
A description of each dimension was presented to the participants, along with a 5-point interval scale indicating the degree to which their visualizations satisfied the dimension, which we worded so that higher scores are better. We gave participants 5 minutes to complete the questionnaire.

5.6.6.5 Break

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

5.6.6.6 Phase 2: Recall Phase

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



(a) Command prompt.

(b) Minimal text editor.

Figure 5.2 We presented participants with a command prompt in which they had the `compile` command available to them. The limited interaction modality forces participants to rely solely on their own memory to successfully complete the task.

Participants did so through the interface shown in Figure 5.2. We gave the participants a command prompt (Figure 5.2a) supporting a single command, `compile`. This command printed to the console the expected error for the task, as well as the error that their source file produced. In addition, participants entered their source code into a minimal text editor (Figure 5.2b). We chose a minimal text editor to force all participants to recall code entirely from memory, without assistive features like auto-completion. For example, in Figure 5.2, the participant has been asked to write a source listing that generates the error `variable i might not have been initialized`. However, the source listing as currently written compiles without error.

Participants used this interface to complete a total of six tasks, all of which they had *previously explained* during the Self-Explanation Phase of the experiment. The tasks from this phase are also from Table 5.3, but to avoid serial recall we presented the tasks in Task Order, rather than alphabetically by Task Name. Thus, participants had to successfully *recall* their explanations from the Self-Explanation Phase of the experiment and apply their understanding to this phase of the experiment. We allowed participants an unlimited number of compilation attempts, but restricted the time for each task to 5 minutes. Participants moved on to the next task either when they had successfully replicated the error message, which we term *recall correctness*, or when their time had expired.

The unusual experimental technique in this phase is not without theoretical justification. In 1977, Shneiderman conducted an experiment in which he used memorization/recall tasks as a basis for judging programmer comprehension [335]. Specifically, one component of his experiment involved non-programmers and programmers memorizing a proper `FORTRAN` program printed on paper through a line

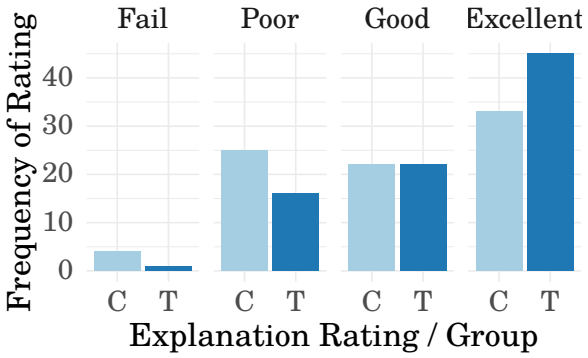


Figure 5.3 Explanation rating by group. The treatment group (T) provided significantly higher rated explanations than the control group (C).

printer. He also printed a second program with shuffled lines. He found that non-programmers had similar performance in recall with both the proper and shuffled versions of the program, but that programmers had significantly better recall on the proper version of the program. Through the development of his cognitive syntantic/semantic model, he suggests that "performance on a recall task would be a good measure of program comprehension" because such a task cannot be accomplished by rote memorization, and instead requires "recognizing meaningful program structures enabling them to recode the syntax of the program into a higher level internal semantic structure" [335].

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

5.7 Results

5.7.1 RQ1: Visualizations Lead to More Correct Explanations

Our hypothesis was that having visual explanations for compiler notifications would result in more correct self-explanations by participants. To validate this hypothesis, we conducted an inter-rater reliability exercise in which the first and second authors independently rated the participants' explanations, without consideration of group. The first author assigned ratings using both the recorded verbal explanations of the

participant as well as their paper markings. The second author assigned ratings using only the paper markings. This was a deliberate design decision to ascertain the extent to which visual markings alone can be used to infer the correctness of an explanation.

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

The distribution between the two groups, binned by rating, is shown in Figure 5.3. Between the control and treatment groups, a Wilcoxon Rank-Sum Test confirms that participants gave significantly better explanations in the treatment group ($n_1 = n_2 = 84, Z = 2.23, p = .026$). A potential confound is that participants are simply providing better explanations in the treatment group because more of them had previously encountered the error messages, but a Pearson Chi-squared Test did not identify a significant difference between the groups ($n = 168$, $df = 2$, $\chi^2 = 3.37$, $p = .19$).

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

Our hypothesis was that both the control group and treatment group would adopt similar annotations when developers explained error messages, if these annotations were grounded in conventions that developers found intuitive.

Consider for a moment the visualizations drawn by two participants in our study, shown in Figure 5.5. In Figure 5.5a, the control group participant receives a score of Fail, because he incorrectly self-explains that the problem must be due to not initializing the variable at its point of declaration. He then either ignores the

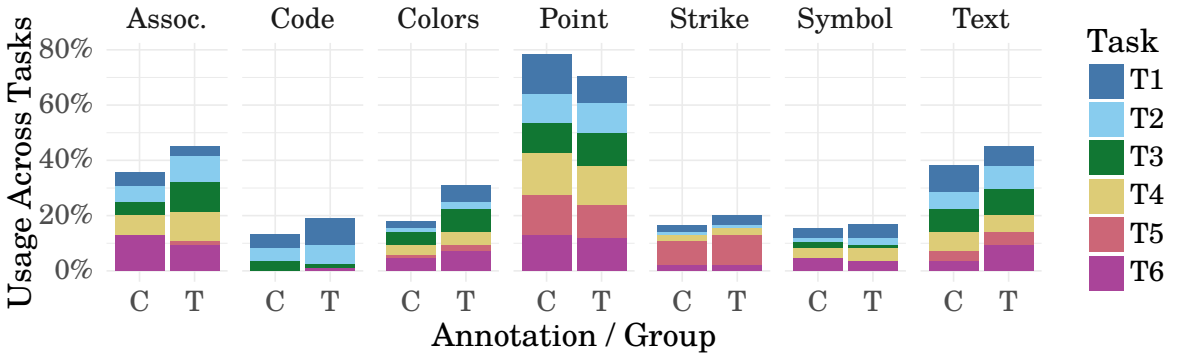


Figure 5.4 Annotations by group, filled with usage across tasks. The distribution of annotations used by the control (C) and treatment groups (T) were not identified as being significantly different, but the treatment group used annotations significantly more often.

Table 5.4 Number of Features by Task and Group

	Number of Features			
	Control		Treatment	
Task	Median	Dist	Median	Dist
T1	2	▂▇▅▂	3	▃▅▇▃
T2	2	▃▇▂▁	2	▃▇▃▂
T3	2	▃▇▂▂	2	▃▇▃▃
T4	2	▂▇▅▃	3	▃▅▇▂
T5	1	▇▅▂▂	2	▃▇▃▂
T6	3	▃▅▇▂	3	▂▇▂▂

conditional statement in which the constant value is re-assigned, or fails to notice that the variable is declared as `final`. In Figure 5.5b, the participant, aided by the explanatory visualization, correctly self-explains that the problem is actually in the conditional statement, and provides explanatory code to demonstrate a case in which the variable remains uninitialized. In addition, the treatment participant uses more annotations, such as colors, points, and associations, in his explanation than the control group participant.

Table 5.4 summarizes the number of annotation types used for each task, partitioned into control and treatment groups. Using a Wilcoxon Rank-Sum Test, we find that the treatment group used significantly more visual annotation types in

Table 5.5 Cognitive Dimensions Questionnaire Responses

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

their explanations than the control group ($n_1 = n_2 = 84$, $Z = 2.15$, $p = .032$).

One concern is that participants in the treatment group used these annotations simply because they were readily *available*, not because they were *useful* to their explanations. Figure 5.4 shows the distribution of the annotations by group. The bars are filled with the usage of that annotation by task to indicate how a particular annotation is distributed among the tasks. A Pearson Chi-squared Test was unable to identify any significant differences in the *distribution* of these annotation types between groups ($n = 389$, $\chi^2 = 4.20$, $df = 5$, $p = .65$), suggesting that these annotations are intuitive even without priming the participants. Although Figure 5.4 shows that the control group used the point annotation more than the treatment group, this difference was not found to be significant ($n = 168$, $df = 1$, $\chi^2 = 1.53$, $p = .22$).

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

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

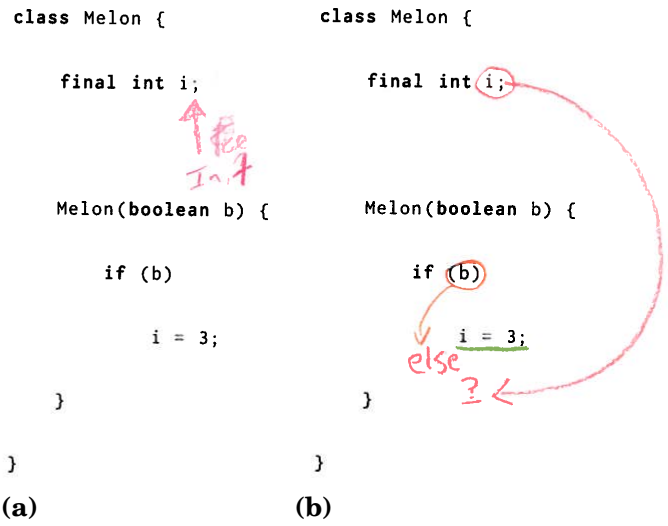


Figure 5.5 A contrast between visual explanations offered by (a) control group participant with explanation rating of Fail, and (b) treatment group participant with explanation rating of Excellent.

5.7.3 RQ3: Explanatory Visualizations Reveal Hidden Dependencies

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

Table 5.5 summarizes the results from our Cognitive Dimensions questionnaire. Median results for the hidden dimensions for control and treatment groups were 3 and 4, respectively. The distribution of responses in the two groups were significantly different ($n_1 = n_2 = 14$, $Z = -2.64$, $p = .008$). The result suggests that our explanatory visualizations reveal more of the hidden dependencies, that is, the internal reasoning process of the compiler, than the baseline visualizations.

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

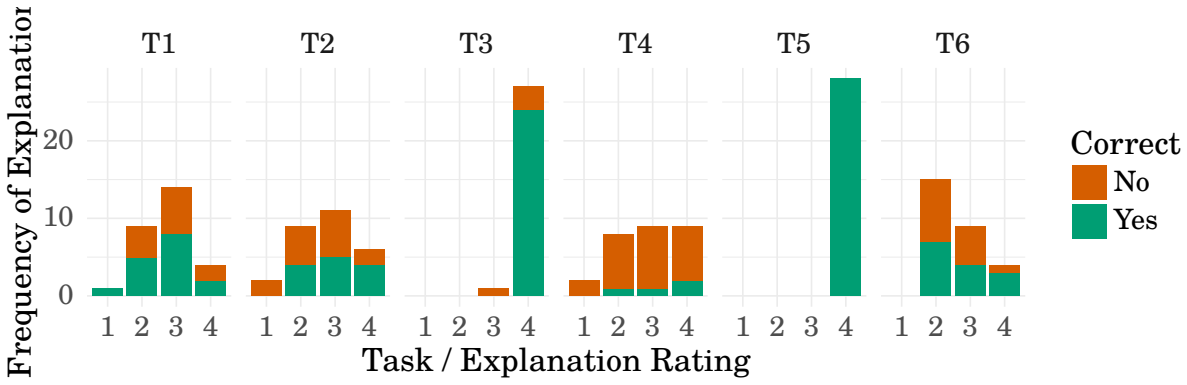


Figure 5.6 Task by explanation rating. Each of the six tasks are broken by explanation rating (1 = Fail, 2 = Poor, 3 = Good, 4 = Excellent) from the first phase of the experiment. For each explanation rating, the frequency of correct and incorrect recall tasks from the second phase of the experiment is indicated by filling in the bars. Higher rated explanations lead to significantly better recall correctness.

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

Figure 5.6 illustrates the explanation rating for each task, the frequency of the explanation for each rating within the task, and the recall correctness. Remember from Section 5.7.1 that the treatment group had higher explanation ratings than the control group. Our expectation was that these higher rated explanations would translate to better correctness scores during the recall phase of the experiment.

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

In Figure 5.6, we observe that task T5 (repeated modifier) has both perfect recall correctness and uniformly excellent explanation rating, which we postulate is attributable to this being trivial problem. Our first concern is that this task is artificially inflating the influence of the explanation correctness to recall correctness. As a contradictory example, we visually identify that T4 (cannot infer type

arguments) has some participants who have poor performance during recall despite excellent explanation correctness. We found that even without T5, the difference is still significant ($\chi^2 = 12.33$, $df = 3$, $p = 0.006$), and the general trend remains ($u_1 = 49.0$, $u_2 = 64.0$, $u_3 = 63.6$, $u_4 = 84.0$).

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

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

5.8 Threats to Validity

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

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

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

Furthermore, the act of performing a think-aloud can enhance self-explanation, and in turn, the construction of mental models for notifications. This process was necessary in order to evaluate participant explanations, but in doing so, we may have unintentionally enhanced the performance of the control group in their recall tasks. Another issue is that participants were already familiar with the baseline

visualizations, but had no prior experience or any training with our explanatory visualizations. This may explain why we found no statistical difference in hard mental operations: the potential cognitive benefit of our visual annotations was counterbalanced by the difficulty of understanding an unfamiliar visualization.

5.9 Related Work

Self-explanation. Lim and colleagues demonstrate that explaining why a system behaves a certain way results in better understanding and stronger feelings of trust [220]. We were also inspired by the work of Ainsworth and Th Loizou, who showed that the use of diagrams promote the self-explanation effect significantly more than text [5].

Improving error notification comprehension. Jeffrey created a tool called *Merr* that overrides the error handler of the LR parser generator of a compiler to automatically provide more useful syntax error messages [176], and Kantorowitz and Laor likewise propose modifications to the parser generator [188]. While these tools apply to text error messages, they illustrate that tools can improve error messages when they can interact with compiler internals. However, even detailed messages do not necessarily improve understanding, which suggests that alternative representations of error notifications may sometimes be more appropriate [93, 268].

Hartmann and colleagues introduce a social recommendation system that presents examples of how other developers understand and correct errors [152]. In contrast, our approach argues that the compiler can offer its own reasoning process to aid developer comprehension. Other approaches attempt to provide better diagnostics or reduce false positives in compiler errors [40, 54, 60]. We expect that our visualizations can leverage such improvements in compiler technology.

5.10 Future Work

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

One approach to demonstrate this feasibility may be to modify an implementation such as MiniJava, a useful but restricted subset of the Java language [312].

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

5.11 Conclusion

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

Acknowledgments

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

6 | How Should Compilers Explain Problems to Developers?

> Go forward in all of your beliefs, and prove to me that I am not mistaken in mine.
>
> The Doctor

6.1 Abstract

Compilers primarily give feedback about problems to the developer through the use of error messages. Unfortunately, developers routinely find these messages to be confusing and unhelpful. In this paper, we postulate that because error messages present poor explanations, theories of explanation—such as Toulmin's model of argument—can be applied to improve their quality. To understand how compilers should present explanations to developers, we conducted a comparative evaluation with 68 professional software developers and an empirical study of compiler error messages found in Stack Overflow questions across seven different programming languages.

Our findings indicate that developers significantly prefer error messages that employ proper argument structure over deficient argument structure when neither offers a resolution, but will accept deficient argument structure if they provide

a resolution to the problem. Human-authored explanations on Stack Overflow converge to one of the three argument structures: those that provide a resolution to the error, simple arguments, and extended arguments that provide additional evidence for the problem. Finally, we contribute three practical design principles to inform the design and evaluation of compiler error messages.

6.2 Introduction

Compilers primarily give feedback about problems to developers through the use of error messages. Despite the intended utility of error messages, researchers and practitioners alike have described their output as "cryptic" and "difficult to resolve" [369], "not very helpful" [385], "appalling" [48], "unnatural" [54], and "basically impenetrable" [346].

While poor error messages are paralyzing for novices, even experienced developers have substantial difficulties when comprehending and resolving them. A study conducted at Google found that nearly 30% of builds fail due to a compiler error, and that the median resolution time for each error is 12 minutes [329]. Surprisingly, the costly errors that developers make are rather mundane, relating to basic issues such as dependencies, type mismatches, syntax, and semantic errors. Barik, Smith, Lubick, and colleagues [20] conducted an eye-tracking study with developers and found almost 25% of their task time on reading error messages. In addition, developers in a study by Johnson, Song, Murphy-Hill, and colleagues [183] reported that error messages were often not useful because they did not adequately *explain* the problem.

It isn't difficult to come up with instances of poor error message explanations, even for routine problems. Consider the following Java code snippet:

```
2  void m() {
3    final int x;
4    while (true) {
5      x = read();
6    }
7  }
```

and the resulting error message from the OpenJDK compiler:

```
F.java:5: error: variable x might be assigned in loop
            x = read();
            ^
1 error
```

Although the location of the message is reasonable, intuitively, this is a poor explanation. The problem isn't just that the variable `x` is being assigned a loop—it is that this particular variables happens to be marked `final` (Line 3). A `final` variable can only be assigned once. What if we had received the following error message instead?

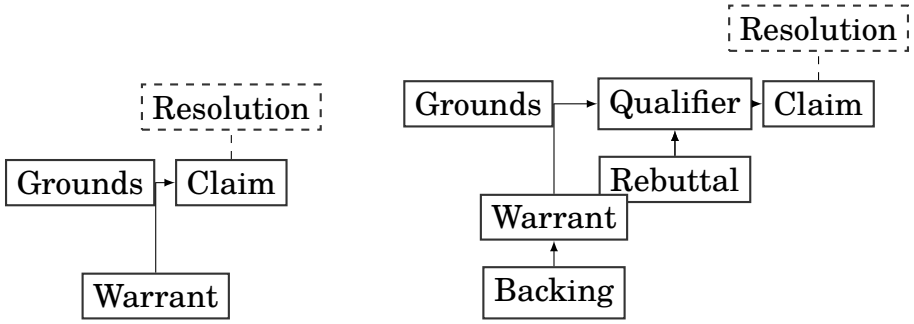
```
F.java:5: error: The blank final variable "x" cannot
be assigned within the body of a loop that may execute
more than once.
            x = read();
            ^
```

This second message gives a better explanation, and developers in our study preferred it significantly over the first (Section 6.6.1). Specifically, the second message not only indicates that there is not only a problem (`the blank variable "x" cannot be assigned`"), the message also supports this claim by offering evidence, or grounds, that clarify why this is a problem—because `"x" cannot be assigned within the body of a loop that may execute more than once`. That is to say, the second message has a better explanatory *structure* than the first. This message also delivers more specific *content*. In contrast to the relatively vague `variable x` in the first message, it is immediately apparent in the second message that `x` is a `blank final`, without being too verbose.

If compiler error messages are framed as explanations, then it follows that we can *apply theories of explanation to understand why some error messages are more effective than others*. To that end, this paper applies Toulmin's model of argumentation (Section 6.3)—a theory for the structure and content of messages in everyday discourse—to the design and evaluation of compiler error messages.

To understand if developers find explanatory error messages helpful, we conducted a comparative study between two compilers for the same programming language, and had experienced developers within a large software company indicate which message they would prefer in their compiler. Then, to understand why some error messages produced by compilers are less helpful than others, we conducted an empirical study through a popular question-and-answer site, Stack Overflow.[1] From Stack Overflow, we extracted 210 question-answer pairs posted by developers about compiler error messages, across seven different programming languages. For every question-answer pair, we qualitatively coded the compiler error message found within the question, and the human-authored answer, through the theoretical model of argumentation. We characterized these question-answer pairs both in terms of

[1] `https://www.stackoverflow.com`



(a) Simple argument layout **(b)** Extended argument layout

Figure 6.1 A prototypical Toulmin's model of argumentation for (a) simple argumentation layout, and (b) extended argument layout. The possible need for auxiliary steps to convince the other party yields the extended argument layout.

the *structure* and *content* of their explanation. From this analysis, we can better understand the structure and content that compilers should use in explanations to developers.

The results of our studies provide support for presenting compiler error messages to developers as explanations, and inform the design of compiler error messages. We find that: *1)* developers prefer error messages with proper argument structures over deficient arguments, but will prefer deficient arguments if they provide a *resolution* to the problem (Section 6.6.1). *2)* human-authored explanations converge to argument structures that offer a simple resolution, or to structures with proper arguments (Section 6.6.2). They do so using a catalog of content in within the structure (Section 6.6.3). We contribute three design principles for compiler authors to inform the design and evaluation of error messages (Section 6.9).

6.3 Background on Explanations

Arguments are a form of justification-explanations, in which reasons are used as evidence to support a conclusion [384]. Argumentation theory provides a lens through which we can evaluate the effectiveness of arguments [110, 289]. Within argumentation theory, the Toulmin model of argument posits one such model precisely for this style of argument: an informal reasoning model which characterizes everyday arguments, or how arguments occur in practice [367]. Specifically, the Toulmin

Table 6.1 OpenJDK and Jikes Error Message Descriptions

Tag	Compiler	Error Message
E1	OpenJDK	`Variable x might be assigned in loop.`
	Jikes	`The blank final variable "x" cannot be assigned within the body of a loop that may execute more than once.`

E2	OpenJDK	`cannot find symbol`
		`symbol: variable varnam`
		`location: class Foo`

	Jikes	`No field named "varnam" was found in type "Foo". However, there is an accessible field "varname" whose name closely matches the name "varnam".`
E3	OpenJDK	`static method should be qualified by type name, Foo, instead of by an expression.`
	Jikes	`Invoking the class method "f" via an instance is discouraged because the method invoked will be the one in the variable's declared type, not the instance's dynamic type.`

E4	OpenJDK	`method remove in class A.B cannot be applied to given types`
		`required: no arguments`
		`found: int`
		`reason: actual and formal argument lists differ in length`

	Jikes	`The method "void remove(int x);" contained in the enclosing type "A" is a perfect match for this method call. However, it is not visible in this nested class because a method with the same name in an intervening class is hiding it.`
E5	OpenJDK	`Illegal static declaration in inner class A.B.` `Modifier 'static' is only allowed in constant variable declarations.`
	Jikes	`This static variable declaration is invalid, because it is not final, but is enclosed in an inner class, "B".`

model of argument is a *macrostructure* model. Macrostructure examines how components combine to support the larger argument rather; in contrast, *microstructure* examines the phrasing and composition of the "sentence-level" statements. For clarity, we will refer to macrostructure simply as *structure* and microstructure as *content*.

In the simple argument layout (Figure 6.1a), the components consist first of a *claim*—the assertion, view or judgment to be justified. Resolutions are also a form of *claim*, The second component is *ground*, or data that provides evidence for this claim. The third component is the justification or *warrant*, which acts as a bridge between the grounds and the claim (for example, "[claim] because [ground]"). Together, the claim, the grounds, and the warrant provide a simple argument layout. The simple argument layout is the minimal *proper* argument structure. Arguments that do not have at least three components are considered to be *deficient*. Specific to error messages are claim-resolution: the first claim states the problem, and the second claim states the resolution or fix. Although these are not proper argument structures, they are nevertheless useful.

Toulmin also devised an extended model of argument, to acknowledge the possibility of needing to infuse additional components to the simple argument layout (Figure 6.1b). In addition to the simple argument layout components, the extended argument layout offers a *rebuttal* when an exception has to be inserted into the argument. The claim may also not be absolute: in this case, a *qualifier* component can temper the claim. Finally, the warrant may also not be immediately accepted by the other party, in which case additional *backing* is needed to support the warrant. If *any* of these additional components are used in the argument, the argument is an extended argument structure. An example for a compiler error message can be mapped to an argument structure is described in Figure 6.2; in this example, the error message is an extended argument because it contains a backing.

6.4 Methodology

6.4.1 Research Questions

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

`Error:(31, 58) java:` incompatible types(C):
bad return type in lambda expression(bc W, G)
java.lang.String cannot be converted to void(B)

Figure 6.2 A compiler error message from Java, annotated with argumentation theory constructs. This particular message contains all of the basic argumentation constructs to satisfy Toulmin's argument: `(C)` = Claim, `(bc W)` = implied "because" Warrant, `(G)` = Grounds. It also includes an extended construct, `(B)` = Backing.

RQ1: Are compiler errors presented as explanations helpful to developers? If explanatory compiler error messages are useful to developers, then they should prefer them over error messages that are less explanatory in their presentation. If this is not confirmed, then developers prefer error message presentation based on other factors, such as the verbosity of the error message.

RQ2: How is structure of explanations in Stack Overflow different from compilers error messages? If compiler error messages and Stack Overflow accepted answers use significantly different argument layout components, this would suggest that *structure* differences in argumentation play an important role in confusion developers face with compiler errors. While some approaches to improving compiler error messages focus on the content (for example, "confusing wording" in the message [183, 268]), the structure differences emphasize how components combine to support the larger argument rather than the statements themselves. Content improvements may be ineffectual without a supporting structure layout.

Further, the answer to this question helps us to understand the types of argument layouts that are used in accepted answers. In other words, toolsmiths can use the design space of argument layout to model and structure automated compiler error messages for developers. Importantly, the argument layout space can also be used as a means to evaluate existing error messages, and to identify potential gaps in argument components for these messages.

RQ3: How is the content of explanations in Stack Overflow different from compiler error messages? Once the structure argument layouts are identified, learning how the components within these layouts are instantiated provide content details for what information developers find useful within each component. For example, one way to instantiate backing for a warrant might be to provide a link to external documentation—and if we find that accepted answers do so, toolsmiths may also consider incorporating such information in the presentation of

their compiler error messages.

6.4.2 Phase I: Study Design for Comparative Evaluation

To answer RQ1, we asked professional software developers to indicate their preference between corresponding compile error messages that explained the same problem, but produced by different compilers.

Compiler selection rationale. We needed to compare two compilers which produced different error messages for the same problem in code, preferably where one compiler produced error messages with better explanatory structure than the other. We selected the Jikes and OpenJDK compiler for this purpose. Jikes is a Java compiler created by IBM for professional use, with one primary design goal focusing on high-quality explanations produced by the compiler [59]. Though now discontinued, Jikes has been lauded by the developer community for giving "better error messages than the JDK compiler" [85].

Task selection. To select candidate error messages, we examined error messages produced by Jikes and identified error messages which contained argument structures. To determine if an error message contained any elements of an argument structure, we tagged each message using labels from the Toulmin model of argument: claim, grounds, warrant, qualifier, rebuttal, and backing. From this analysis, we found 30 error messages which used a simple argument in Jikes. We then examined the corresponding OpenJDK messages and found only 7 error messages used simple arguments.

To keep the study brief, we selected 5 OpenJDK and Jikes compiler error messages (Table 6.1) that address the same problem, but differ in argument structure. For each pair of error messages, we formed a hypothesis on how differences in argument structure would influence our expected results prior to the study.

E1 *Deficient argument vs. simple argument.* Both OpenJDK and Jikes make a *claim* that the variable might be assigned to in a loop. But Jikes completes a simple argument by presenting a *ground* for why this problem is actually a problem: if the loop executes more than once.

E2 *Deficient argument vs. extended argument.* Again, OpenJDK only presents a *claim*. Jikes presents a *ground* (`there is an accessible field "varname"`), which is qualified through a rebuttal (`However`).

E3 *Claim-resolution vs. extended argument* The `should` in the OpenJDK message would suggest that this an extended argument, but the error message has

no ground. Thus, it is a `claim-resolution` structure, which is not formally considered an argument. The Jikes message is an extended argument because of `discouraged`, but Jikes does not offer a resolution for how to address the problem.

E4 *Different claim, same extended argument.* Both messages provide an extended argument, but for different claims. OpenJDK assumes that the developer is trying to recursively call the current method, `remove()`. Jikes assumes that the developer wants to a call a class method, `remove(int x)`, from the method `remove()`. Since the developer does not know which fix is actually intended, their judgment about which message is correct is determined by the content, not the argument structure.

E5 *Same claim, same simple argument.* Both OpenJDK and Jikes present the same argument (`but is enclosed in an inner class, "B"` is simply long-form of `A.B` in the OpenJDK version). The content of both messages are essentially the same, with minor variations in wording: `final` versus `constant`.

Participants. We recruited developers at a large software company to participate in this study. As we were primarily interested with professional software development, we selected our population from full-time software developers from a large software company—excluding interns or roles such as testers or project managers. We distributed our study to 300 developers and received 68 respondents. The average reported experience of our participants was 6.3 years.

Procedure. We designed a questionnaire which could be distributed and answered electronically. In the questionnaire, we asked demographic questions, including years of programming experience and proficiency in programming languages.

To measure preference for compiler messages, we presented participants with a required binary response for either the Jikes or OpenJDK version of the error message. We randomized error message order. On average, participants took seven minutes to complete our study.

6.4.3 Phase II: Study Design for Stack Overflow

Research context. Previous research by Treude, Barzilay, and Storey [370] identified questions regarding error messages as being one of the top categories, and other research supports that Stack Overflow today is a primary resource for software engineering problems [228]. Additionally, Stack Overflow provides an open-access

Table 6.2 Compiler Errors and Warnings Count by Tag

	Question Count[2]			
Tag[1]	Errors[3]	Warnings[4]	Total	% Accepted[5]
C++	3508	421	3929	63%
Java	2078	170	2248	55%
C	1179	286	1465	61%
C#	783	122	905	69%
Objective-C	270	109	379	65%
Swift	246	17	263	56%
Python	211	4	215	53%
Totals	11736	1553	13289	58%

[1] Programming languages are indicated in bold.
[2] Questions may be counted more than once if they have multiple tags, for example, `C` and `C++`.
[3] Questions tagged as `compiler-errors`.
[4] Questions tagged with `compiler-warnings`, but not `compiler-errors`.
[5] Percentage of questions tagged as `compiler-errors` and `compiler-warnings` that have accepted answers.

API, through Stack Exchange Data Explorer,[2] that allows researchers to mine their database. An initial query against this dataset confirmed that questions about compiler error messages exist in Stack Overflow across a diversity of programming languages and platforms.

Data collection. We extracted all posts of type question or answer, tagged as "compiler-errors" or "compiler-warnings." Some systems allow the developer to flag warnings as errors, and thus we included these warnings in our set. We extracted a total of 13289 questions; 7741 of which have accepted answers. Including co-occurring tags, we identified 1690 "compiler-warnings" and 11736 "compiler errors".

A subset of these questions link to an associated *accepted answer*, which in this paper we term *question-answer pairs*. An accepted answer is an answer marked by the original questioner as being satisfactory in resolving or addressing their original question. Although a question may have multiple answers, only one may

[2] `http://data.stackexchange.com/`

be marked as accepted. We used accepted answers as a proxy to identify helpful answers.

For each question, we extracted the compiler error message from the compiler used in the question. If the question did not contain a compiler error message, the question-answer pair was dropped from analysis.

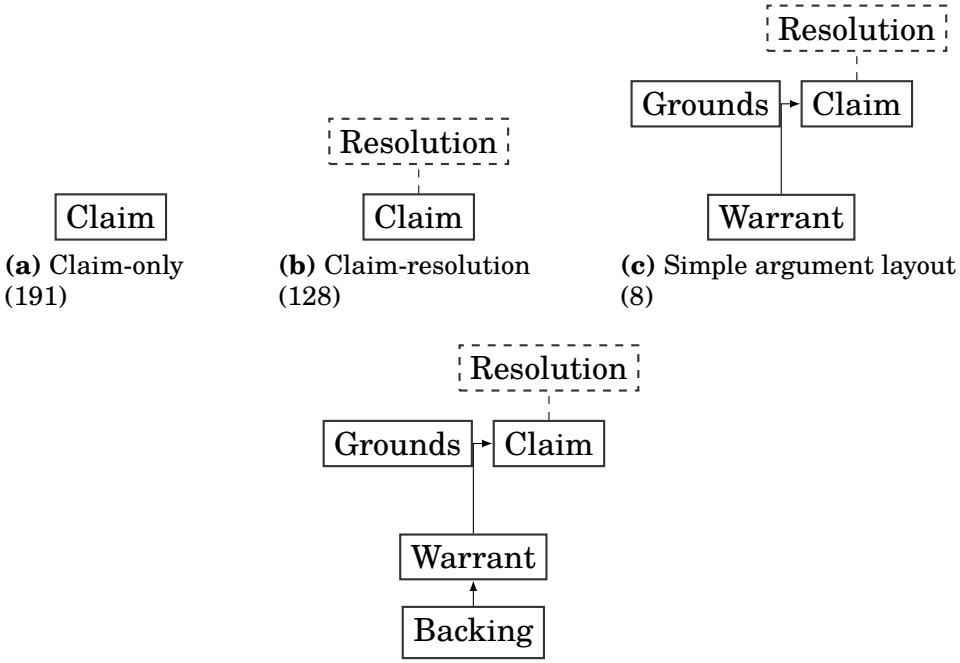
Sampling strategy. To obtain diversity across programming languages, we use stratified sampling across the top languages on Stack Overflow for compiler errors, until we covered over 95% of all of the messages. This threshold was exceeded at Python (Table 6.2). Within each stratum, we used simple random sampling for selecting question-answer pairs to analyze, in which each question-answer pair has an equal probability of being selected. As we sampled, we discarded questions that did not refer to or display a specific error message, were incorrectly tagged (for example, not relating to an error message), were related to issues in not being able to invoke the compiler in the first place (for example, "g++ not found"), or question-answer pairs that are unambiguously "trolling," [149] such as through deliberately bogus questions.[3] The time required to manually categorizing question-answer pairs has high variance, from 5-15 minutes, depending on the complexity of the pair. Thus, to balance breath of languages and depth of error messages in each language—while still keep categorization tractable—we continued this process until we obtained 30 question-answer pairs for each of the top seven languages, for a total of 210 question-answer pairs.

Qualitative closed coding. The first and second authors performed closed coding, that is, coding over pre-defined labels, for compiler error messages extracted from the Stack Overflow question and over the complete Stack Overflow accepted answer for that question. We tagged each using labels from the Toulmin model of argument: claim (and resolutions as claim), grounds, warrant, qualifier, rebuttal, and backing. Thus, we had a total of seven labels, and a compiler error message or Stack Overflow accepted answer may be assigned more than one label.

During the coding process, we employed the technique of *negotiated agreement* as a means to address the reliability of coding [53]. Using this technique, the first and second authors collaboratively code to achieve agreement and to clarify the definitions of the codes; thus, measures such as inter-rater agreement are not applicable.

Supporting verifiability. If using a supported PDF reader, quotations from

[3]For example, the post "Why is this program erroneously rejected by three C++ compilers?" attempts to compile a hand-written C++ program scanned as an image, through three different compilers. The offered answers are equally sardonic. (`http://stackoverflow.com/questions/5508110/`)



(a) Claim-only (191)

(b) Claim-resolution (128)

(c) Simple argument layout (8)

(d) With backing (1)

Figure 6.3 Identified argument layouts for compiler error messages (as found in Stack Overflow questions). Counts are indicated in parentheses.

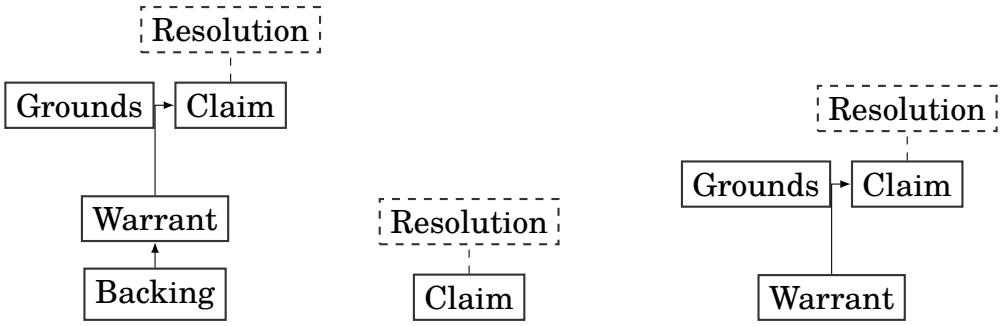
Stack Overflow are hyperlinked and can be clicked to take the reader to the corresponding Stack Overflow page.[4]

6.5 Analysis

6.5.1 RQ1: Are compiler errors presented as explanations helpful to developers?

For the analysis of RQ1, we performed a Chi-squared test on each of the five error messages (E1-E5, Table 6.1), using the developer responses as the observed values for OpenJDK and Jikes. If we use a null hypothesis where both messages are equally

[4]These references are indicated as Q:*id* or A:*id*, and can be directly accessed through `https://www.stackoverflow.com/questions/:id`



(a) With backing (102) **(b)** Claim-resolution (59) **(c)** Simple argument layout (49)

Figure 6.4 Identified argument layouts for Stack Overflow accepted answers. Counts are indicated in parentheses.

acceptable, then the expected values would be split such that OpenJDK and Jikes receive exactly half of the counts. In effect, this situation is essentially analogous to a coin toss problem, where heads is, say, OpenJDK, and tails is Jikes. The null hypothesis is rejected ($\alpha = 0.05$) if the observed values are significantly different from the expected values.

For the responses on how developers use Stack Overflow, we report the data descriptively.

6.5.2 RQ2: How is structure of explanations in Stack Overflow different from compilers error messages?

As the first step towards answering this research, we wanted to quantify whether whether the argument structure between compilers error messages and Stack Overflow answers are significantly different. To do so, we applied a statistical, permutation testing approach by Simpson, Lyday, Hayasaka, and colleagues [348] that allows comparison across two groups when each observation in the group is an ordered set. If we think of the question-answer pairs from Stack Overflow as tuples, then the first group is the set of error messages from the compiler (such as OpenJDK or LLVM) found in the Stack Overflow Question. The second group consists of the the corresponding accepted, human-authored answer for its associated Stack Overflow question. Essentially, the goal of this analysis is to identify if these two groups are statistically different.

However, since error message aren't numbers, they must be first represented in an approximated form for statistical analysis. Thus, an error message, whether it is a compiler error message extracted from a Stack Overflow question or a Stack Overflow accepted answer, is represented as an ordered set in terms of argumentation components, E:

$$E = \langle a_1, a_2, \ldots, a_n, r \rangle \tag{6.1}$$

where $a_1, a_2, \ldots, a_n$ are the labels for the argument components, such as grounds, warrants, and backing, and r is an extended resolution component. For each component, a binary `true` or `false` indicates the presence or absence of the component within the argument.

Then, given any two error messages, E_1 and E_2, we now need a metric that represents the similarity between two sets: this is the Jaccard index, and intuitively the Jaccard index is the intersection over the union of sets [348]. Next, we perform a permutation testing calculation, fully described in Simpson, Lyday, Hayasaka, and colleagues [348]. The explanation of this algorithm is fairly intricate, but the essential result of this computation is an empirically-obtained p-value.

To characterize what types of arguments structures are found in accepted answers, we used quasi-statistics—essentially, a process of transforming qualitative data to simple counts—to aid the interpretations of the Stack Overflow data [235]. We once again used the error messages as ordered sets to perform this task. First, we removed negligible components in the set—those components with few counts—and ignore them in any subsequent operations. Second, we grouped identical sets—that is, sets with the same ordered values and counted them. In practice, because there are only a finite number of reasonable ways to present explanations, we expect there to be few variations in argument structure from Toulmin's prototypical structures (Section 6.3).

6.5.3 RQ3: How is the content of explanations in Stack Overflow different from compiler error messages?

To identify the *content* of argument, that is, the techniques developers use within the argumentation components, we performed a second qualitative coding exercise over the first closed coding. For this analysis, we performed descriptive coding to label the types of evidence provided within the accepted answers [321]. As a concrete example, the argumentation component of *backing* can be provided by

Table 6.3 OpenJDK and Jikes Error Message Preferences

		OpenJDK		Jikes	
Tag	p[1]	n	%	n	%
E1	.001*	2	3%	66	97%
E2	.014*	20	29%	48	71%
E3	.037*	46	68%	22	32%
E4	.014*	20	29%	48	70%
E5	.732	36	53%	32	47%

[1] * Indicates a statistically significant result.

through pointing to a local or program element in the code (blame), through a code example that provides evidence for the problem, or through external resources, such as language-specification documentation.

In addition to random sampling, we performed *purposive sampling*, or non-probabilistic sampling, on question-answer pairs to compose *memos* [35]. These memos captured interesting exchanges or properties of the question-answer pairs to promote depth and credibility, and to frame the information needs and responses posters' through their reported experiences. That is, they provide a *thick description* to contextualize the findings [293].

6.6 Results

6.6.1 RQ1: Are compiler errors presented as explanations helpful to developers?

From Table 6.3, developers significantly preferred Jikes over OpenJDK for E1, E2, and E4; they preferred OpenJDK over Jikes for E3. Green bars indicate the greater preference of error tags with a significant difference. We did not identify significant differences in E5.

E1 *Deficient argument vs. simple argument.* As we expected, developers significantly preferred the simple argument from Jikes to the deficient argument in OpenJDK.

E2 *Deficient argument vs. extended argument.* As we expected, developers significantly preferred the extended argument from Jikes to the deficient argument in OpenJDK.

E3 *Claim-resolution vs. extended argument* We did not know if developers would prefer a claim-resolution structure or an extended argument, given that the extended argument did not provide a resolution. Developers significantly preferred having a resolution over a more elaborate argument.

E4 *Different claim, same extended argument.* Given two different claims, we expected the developer to prefer Jikes, because the argument is present in natural language. In other words, given the same claim, the content of the argument would influence their preference, not the structure. Developers significantly preferred the natural language presentation of the content.

E5 *Same claim, same simple argument.* Given only minor variations in the wording of the content, we expected that the preference would essentially be a coin flip. Indeed, developers did not significantly prefer OpenJDK or Jikes; the distribution is also nearly 50%/50%, as we would expect from a random selection.

6.6.2 RQ2: How is structure of explanations in Stack Overflow different from compilers error messages?

The Jaccard ratio of the two groups are $R_j = 1.6441$, with permutation testing yielding a significant difference between the two groups (for repeated iterations, $p = 0.008 \pm 0.001$). Because we have computed a pair-wise statistic, the implication is that the compiler error message and Stack Overflow accepted answers are significantly different in terms of argumentation layout than the compiler error message provided in the question.

Because the questioner asked a question about the compiler error message, this indicates some confusion with the error messages they were presented with. Because the same questioner also marked the Stack Overflow answer as accepted, we can assume that the answer has resolved whatever confusion they had in the original question. Since the argument structure between the compiler error message and the accepted answer are significantly different in terms of argument layout, we can conclude that differences in structure argument layout can be attributed to the acceptance of the Stack Overflow answer.

The resulting argument structures are found in Figure 6.3 and Figure 6.4, for compiler error messages and for Stack Overflow accepted answers, respectively. For each the group, the argument layouts are ordered from most frequently observed to least frequently observed. In this quasi-statistical reporting, it is clear to see why the argument layout for compiler errors and Stack Overflow accepted answers were found to be significantly different in RQ1: compiler error messages predominantly present a claim with no additional information, and occasionally present a resolution (that is, a fix) to resolve the claim. In contrast, Stack Overflow accepted answers are inverted in argumentation layout frequency; the most frequent argument layout extends simple argument layout with backing, and least frequently provides solely a resolution for the claim. In our investigation, we did not find any instances in which Stack Overflow accepted answers solely rephrased the compiler error message (that is, the claim-only layout).

Thus, not only do Stack Overflow accepted answers more closely align with Toulmin model's of argumentation, these answers satisfactory resolved the confusion of the developer when the original compiler error message did not.

6.6.3 RQ3: How is the content of explanations in Stack Overflow different from compiler error messages?

In this section, we describe the content of the components of argument structure An overview of the argument structure is presented in Table 6.4.

6.6.3.1 Claim

Because of the layout of Stack Overflow, accepted answers assume that the developer has read the error message in the question, and will refer to the claim without explicit antecedent. For instance, the answer may say, "This problem" (A1225726) or "This issue" (A32831677) or immediately chain from the question to the connect their ground and warrant (A28880386). We did however, encounter instances where developers explained error messages through first *rephrasing*, such as "it means that" (A16686282) and "is saying" (A20858493)—usually for the purpose of simplifying the jargon in the message or making an obtuse message more conversational. Incidentally, compiler authors like Czaplicki (of the Elm programming language) have also noted that error messages should be more conversational and human-like [70]. For example, the compiler error message:

Table 6.4 Argument Layout Components for Error Messages

Attribute	Description
Simple Argument Components	
Claim (Section 6.6.3.1)	The claim is the concluding assertion or judgment about a problem in the code.
Resolution (Section 6.6.3.2)	Resolutions suggest concrete actions to the source code to remediate the problem.
Ground (Section 6.6.3.3)	Facts, rules, and evidence to support the claim.
Warrant (Section 6.6.3.4)	Bridging statements that connect the grounds to the claim. Provides justification for using the grounds to support the claim.
Extended Argument Components	
Backing (Section 6.6.3.5)	Additional evidence to support the warrant, if the warrant is not accepted.
Qualifier (Section 6.6.3.6)	This is the degree of belief for a claim, used to weaken the claim.
Rebuttal (Section 6.6.3.7)	Exceptions to the claim or other components of the argument.

```
foreach statement cannot operate on variables of type 'E' because 'E' does
↪  not contain a public definition for 'GetEnumerator'}
```

is rephrased by the accepted answer as "It means that you cannot do *foreach* on your desired object, since it does not expose a `GetEnumerator` method."

6.6.3.2 Resolution

Although not explicitly present in Toulmin's model of argument, one way in which error messages can convince developers of an argument is to offer the solution to that argument that resolves their issue. Typically, Stack Overflow accepted answers provide these resolution in a style similar to "Quick Fixes" in IDEs—they briefly describe what will be changed, show the resulting code after applying the change, and demonstrate that the compiler defect will be removed as a result of applying

the change. A prototypical example of how answers provide resolutions is found in A8783019. Here, the answer notes, "You're missing an `&` in the definition." The answer then proceeds to show the original code:

```
float computeDotProduct3(Vector3f& vec_a,
  Vector3f vec_b) {
```

against the suggested fix:

```
float computeDotProduct3(Vector3f& vec_a,
  ector3f& vec_b) {
```

6.6.3.3 Ground

Grounds are an essential building block for convincing arguments; they are the substrate of declarative facts—which bridged by the warrant—support the claim, that is, the compiler error message. For example, "the variable is non-static private field" (A4114006), "`clone()` returns an `Object`" (A3941850), "`foo<T>` is a base class of `bar<T>`" (A27412912), "[t]he only supertype of `Int` and `Point` is `Any`" (A2871344), "local variables cannot have external linkage" (A5185833) all refer to grounds about the state of the program or rules about what the compiler will accept.

Consider the use of `gets()` in a C program, which in the `gcc` compiler generates the message:

```
test.c:27:2: warning: ‘gets’ is deprecated
  (declared at /usr/include/stdio.h:638)
  [-Wdeprecated-declarations]

gets(temp);
^
```

The poster of the compiler error wants to suppress this warning (Q26192934), but the accepted answer explains the grounds for this warning (A26192934): "gets is deprecated *because* it's dangerous, it may cause buffer overflow."

6.6.3.4 Warrant

In argumentation theory, warrants are bridge terms, such as "since" or "because" that connect the ground to the claim. Often, the warrant is not explicitly expressed,

and the connection between the ground and the claim must be implicity identified [110]. During our analysis, we would insert implicit “since” or “because” phrases during reading of the error message or Stack Overflow answer to identify implicit warrants.

In some compilers, messages can bridge grounds with warrants through explicit concenations, such as with the ”reason:” error template in Java:

```
Test.java:6: error: method b in class Test cannot
be applied to given types;
        b(newList(type));
        ^
  required: List<T>
  found: CAP#1
  reason:
    inference variable L has incompatible bounds
    equality constraints: CAP#2
    upper bounds: List<CAP#3>,List<?>
  where T,L are type-variables:
     T extends Object declared in method <T>b(List<T>)
     ...
```

Unfortunately, the grounds for this warrant are particularly dense in itself. However, warrants needs not always be this obtuse, as the following C++ message from OpenCV indicates:

```
OpenCV Error: Image step is wrong
    (The matrix is not continuous,
    thus its number of rows can not be changed).
```

Here, the warrant is bridged through the use of the parenthetical statement.

6.6.3.5 Backing

A backing may be required in an argument if the warrant is not accepted; in this case, the backing is additional evidence needed to support the warrant. In practice, one should selectively support warrants; otherwise, the argument structure grows recursively and quickly becomes intractable [110]. For presenting error messages, we found that while warrants were typically additional statements, backing was

provided through the use of resources. These resources include code examples or code snippets (A2640738, A1811168), references to the programming language specifications (A5005384), and occasionally, bug reports (A37830382) as well as tutorials (A2640738).

6.6.3.6 Qualifiers

Despite the usefulness of static analysis techniques for reporting compiler error messages to developer, many classes of analysis feedback are undecidable or computationally hard, which necessitate the use of unsound simplifications [202]. Qualifiers include statements like "should" (A29189727), "likely" (A17980236), "try" (A7316513), and "probably" (A2841647, A7328052. A7942837). Although we found such usages throughout Stack Overflow, it was difficult for us to determine if these usages are simply used as casual linguistic constructs (essentially, fillers) or if the answer actually intended to convey a judgment about belief. We did, however, find several examples when developers were confused because the wording of the compiler error made the developer believe that their own judgment was in error: questions such as Q5013194 and Q36476599.

6.6.3.7 Rebuttal

We found few instances of rebuttals within Stack Overflow accepted answers, and one of the reasons we believe rebuttals to be relatively infrequency is that an author must have an expectation of what to rebut in order to provide a rebuttal in the first place. Thus, we interpreted rebuttals liberally as statements in which an answer would retract a particular ground or resolution due to a particular constraint—for example, due to a bug in the compiler (A2858799, A1167204). Another means of rebuttal occurs when the accepted answer provides reasons for *ignoring* a claim, as in A11180068. Here, the accepted answer suggests downgrading a ReSharper warning from a warning to a hint in order to not get "desensitized to their warnings, which are usually useful."

6.7 Limitations

The selection of error messages in our comparative study, along with the qualitative research approaches used in the Stack Overflow study, introduces trade-offs in the design and reporting of our study.

Comparative evaluation with Jikes and OpenJDK. In order to keep the study brief, we were not able to evaluate all combinations of the argument design space. In particular, we did not evaluate whether developer prefers simple arguments against extended arguments. Although our results—where expected—were statistically significant, the error messages we asked participants to evaluate are not necessarily representative in terms of either the difficulty or type of error message. Because the authors selected the explanations to present, we may also unintentionally introduced a bias in the selection process, favoring certain argument structures over others. The subsequent Stack Overflow study to some extent mitigates this threat, but does not explicit.

Identifying argument content. The design space of argument content is constrained to available affordances in Stack Overflow. For example, answers in Stack Overflow must use mostly text notation, although past research has found that developers sometimes place diagrammatic annotations on their code to help with comprehension [18]. Similarly, Flanagan, Flatt, Krishnamurthi, and colleagues [121] uses a diagrammatic representation on the source code to help developers understand code flow for an error. Other tools like Path Projection [191] and Theseus [218] use visual overlays on the source code, which are not expressible within Stack Overflow except through rudimentary methods like adding comments to the source. Thus, the design space of attributes is biased towards linear, text-based representations of compiler error messages.

Generalizability. As a qualitative approach, our findings do not offer external validity in the traditional sense of nomothetic, sample-to-population, or *statistical generalization*. For example, we cannot claim that the differences in design space usage between error messages and Stack Overflow accepted answers generalize to those outside the ones we observed within our study. That is, our findings are embedded within Stack Overflow and contextualized to understand a particular aspect of developer experiences as they comprehend and resolve compiler error messages within these question-answer pairs. As one example, the argument layout for compiler error messages is likely to significantly underrepresent claim-resolution layouts, as resolutions in integrated development environments appear in a different location—such as Quick Fixes in the editor margin—than the compiler error message.

In place of statistical generalization, our qualitative findings support an idiographic means of satisfying external validity: *analytic generalization* [291]. In analytic generalization, we generalize from individual statements within question-answer pairs to broader concepts or higher-order abstractions through the applica-

tion of argumentation theory.

6.8 Related Work

The work by Nasehi, Sillito, Maurer, and colleagues [259] is the closest related work in terms of research approach and methodology. Nasehi and colleagues, inspected Stack Overflow questions and accepted answers to identify attributes of good code examples: we use a similar methodology to verify if attributes of good compiler error messages correspond to structure and content drawn from argumentation theory.

6.8.1 Design Criteria and Guidelines

Several researchers have identified guidelines for compile errors. However, the history of design criteria for improving compiler error messages is both long and sometimes sordid; many of these guidelines are today are considered to be pedestrian [48, 251].

Early work by Horning [165] suggested guidelines for the display of error messages, such as the use of headings that identify the version of the compiler being used, a "coordinate system" for relating the error message back to the source code listing, and the "memory addresses" relating to the error message. Shneiderman [337] focused less on the structural design of the error message and more on the holistic presentation, recommending that errors should have a positive tone, be specific using the developer's language, provide actionable information, and have a consistent, comprehensible format. In 1982, Dean [89] argues for design guidelines that emphasize humans goals, such as helping people correct errors as easily as they make them, and giving people control over the messages they receive.

More recently, Traver [369] adapted criteria for Nielson's heuristic evaluation [266] to compiler error messages, suggesting principles such as clarity, specificity, context-insensitivity, as well as previously-seen guidelines such as positive tone and matching the developers' language. Similarly, Sadowski, Gogh, Jaspan, and colleagues [319] present four design guidelines that inform when and how to incorporate new error messages into a static analysis platform, Tricoder. Like Kantorowitz and Laor [188], they suggest that program analysis tools minimize errors messages that are false positives.

6.8.2 Barriers to Error Message Comprehension

Johnson, Song, Murphy-Hill, and colleagues [183] conducted an interview study with developers to identify barriers to using static analysis tools. Their interviewers reported barriers such as "poorly presented" tool output and "false positives" as contributing to comprehension difficulties. One interviewee suggested that that error messages be presented in some "distinct structure" to facilitate comprehension [183]. A follow-up study by Johnson, Pandita, Smith, and colleagues [182] identified that mismatches between developers' programming knowledge against information provided by the error message contribute to this confusion. A large-scale multi-method study at Microsoft identified several presentation "pain points," such as "bad warnings messages" and "bad visualization of warnings" [64].

Ko and Myers [196] found that many comprehension difficulties are due to programmers' false assumptions formed while trying to resolve errors [196]. Similarly, Lieber, Brandt, and Miller [218] postulated that difficulties in resolving errors were due to faulty mental models, or misconceptions, that remained uncorrected until the developer manually requested information explicitly from their programming environment; they developed an always-on visualization in the integrated development to proactively address misconceptions.

Still other work has focused on barriers novice developers face. For example, Marceau, Fisler, and Krishnamurthi [231] found that students struggle with the "carefully-designed vocabulary of the error message" and often misinterpret the highlighted source code. And a large-scale study of novice compilations found that minor syntax issues and typos are attributable to why compiler error messages are emitted [6].

6.9 Design Principles

We synthesize our findings as design principles, which compiler authors can use to inform the design and evaluation of error messages. We discuss our findings within the context of these principles:

Principle I—Distinguish fixes from explanations. Accepted answers from Stack Overflow identified a dichotomy in argument structure: 1) claim-resolutions, which we can think of essential as quick fixes that immediately *resolve the problem* for the developer, and 2) simple to extended argument structures, which provide an

explanation of the problem (Figure 6.4).

Both styles of argument structure are useful. A claim-resolution structure is appropriate when the resolution is obvious. For example, consider a C file with a missing semi-colon, as presented through the LLVM:

```
hello.c:4:28: error: expected ';' after expression
  printf("Hello, world!\n")
                           ^
                           ;
```

For an expert, it is clear what the problem is and the developer does not need an explanation for how semi-colons work in C. By constrast, consider the error message E4 from our comparative evaluation. Here, there is a design choice that depends on which `remove` method the developer intends to call; consequently, a simple argument structure is appropriate.

Principle II—Allow developers the autonomy to elaborate arguments. From our comparative evaluation in Phase I, we found that with E3 developers preferred the OpenJDK version of the error messages, despite the fact that the Jikes version is more explanatory. However, novice developers may still find the explanation from Jikes useful, and expert developers may still find the Jikes presentation useful if they want to understand the rationale for the fix. Thus, developers may selectively need more or less help in comprehending the problem, and we should support mechanisms to progressively elaborate error messages. The static analysis tool Error Prone implements such an approach; the tool provides a simple argument for the error messages, but also enables additional backing through providing an supporting link:

```
1 ShortSet.java:9: error: [CollectionIncompatibleType]
2 Argument 'i - 1' should not be passed to this method;
3 its type int is not compatible with its collection's
4 type argument Short
5         s.remove(i - 1);
6                  ^
7 (see http://errorprone.info/bugpattern/
8             CollectionIncompatibleType)
```

Similarly, the Rust and Dotty compilers initially provide a simple argument, but the developer can invoke an extended argument by passing a `--explain` flag to the compiler.

Principle III—Apply argument structure and content to the design and evaluation of error messages. The theory of argumentation can guide the design of error messages, as well as assess potential problems with existing error messages. For instance, considering the following Haskell code snippet:

```
let y = [True, 'a']
```

which produces this error message in the Haskell interpreter, `ghci`:

```
Couldn't match expected type ‘Bool’ with actual type ‘Char’
* In the expression: 'a'
* In the expression: [True, 'a']
  In an equation for ‘y’: y = [True, 'a']
```

Inspecting this error message through the lens of argumentation theory immediately reveals a problem: the error message does not present a claim. `Couldn't match expected type ‘Bool’ with actual type ‘Char’` is actually a ground *masquerading* as a claim. The rest of the explanation is backing to support the ground. This deficiency is easy to spot if we compare against a similarly-produced F# (`let y = [true; 'a'];;`) error message:

```
let y = [true; 'a'];;
---------------^^^

error FS0001: All elements of a list constructor expression must have the
↪  same type. This expression was expected to have type 'bool', but here
↪  has type 'char'.
```

Now, it is apparent the actual claim is that all `all elements of a list expression must have the same type`, and the remainder of the error message is evidence to support that claim. There are also content differences: F# helpfully provides a code snippet that points to the position of the error in the result, where `ghci` indicates the location narratively through a series of `In the expression` statements.

6.10 Conclusion

In this paper, we conducted studies on error messages, through Toulmin's model of argumentation. Our results suggest that generalizable, theory-driven approaches to the design and evaluation of error messages will lead to more explainable, human-friendly errors—across programming languages and compilers.

7 | Related Work

> As we learn about each other, so we learn about ourselves.
>
> The Doctor

How does the work in this dissertation fit within the ontology and ontogeny of related research areas in computer science and other disciplines? This chapter delineates collections of research pertinent to the design of error messages, in program comprehension (Section 7.1), human factors and warning design (Section 7.2), experts systems (Section 7.3), structure editors (Section 7.4), and error messages for novices (Section 7.5). Through examining the similarities and differences of my own research within these collections, I defend the novelty of my thesis.

7.1 Program Comprehension in Debugging

Program comprehension is a cognitively-demanding activity, coordinating and competing with a number of parallel mental processes that include language [343], learning [305], attention [396], problem-solving [199], and memory [279]. Given the complexity of human cognition and the different levels of abstraction under which cognition has been studied, there are several concurrent theories to explain program comprehension. The multiplicity of theoretical parameters can complement, integrate, and sometimes contradict theories such as the theory of rational reconstruction proposed in this dissertation; for a literature review of confounding parameters in program comprehension, see Siegmund and Schumann [345]. In this section, I sample theories three established theories of program comprehension during debugging and maintenance activities: plans (Section 7.1.1), beacons

(Section 7.1.2), and information foraging theory (Section 7.1.3). I conclude with an explanation of how rational reconstruction can support these theories (Section 7.1.4).

7.1.1 Plans

Shneiderman and Mayer [338] presented an information processing model that comprises a long-term store of semantic and syntactic knowledge in conjunction with a working memory through which developers construct problem solutions. For debugging an error message, Shneiderman and Mayer [338] proposed that developers are unable to resolve debugging output in two possible situations. In the first situations, the developer has a correct plan for what the program is supposed to do, but has made a mistake in representing that idea in the source code. In the second situation, the developer makes a mistake when they use the error message to construct a plan. This second situation is more insidious: confusion arises because the error message conflicts in some way with the developers' internal semantics. Other researchers have also considered debugging as the construction or execution of programming plans, or schemas [45, 213, 311, 352, 353, 359, 381]. The planning theories have several differences: they are sometimes top-down comprehension models [44, 352], bottom-up comprehension models [214, 282], or integrated models [237]. Nevertheless, there are many commonalities among them: namely, they are all some cognitive process that constructs a mapping between the source code and a cognitive representation of the problem, and understanding involves reconstructing some or all of these mappings.

7.1.2 Beacons

But what is the link between source code and mapping during planning and reconstruction? One theory is that developers search for beacons [44, 78, 396]—lines of code that serve as indicators of a particular structure or operation, and developers use the presence or absence of these beacons to confirm or reject hypotheses about the message. To search for these beacons, developers employ a process called program slicing, in which developers break apart large programs into smaller coherent, though not necessarily textually contiguous, pieces [391]. During slicing, if the developer fails to confirm the presence of beacons in source code, they may reject their hypothesis about the error, and possibly even the error message itself. A recent functional magnetic resonance imaging (fMRI) study by Siegmund, Peitek, Parnin, and colleagues [344] confirmed lower activation of brain areas during comprehension

based on semantic cues, confirming that beacons ease comprehension.

7.1.3 Information Foraging Theory

An alternative theory for programmer comprehension that contrasts with the classical information processing theories we have so far described is information foraging theory [208]. Information foraging explains and predicts how developers navigation as a problem of information seeking [122]. Information foraging proposes a parsimonious predator-prey analogy in which a developer (predator) uses information patches, cues or signposts, and information scents—essentially, a perceived likelihood that a cue will lead to the fix (prey).

A salient property of information foraging theory is that it assumes that the environment drives the developer's behavior. That is to say, the theory does not require knowledge of what is inside the developer's mind [208]. We can think of information foraging theory as a greedy search: at each decision point, the developer chooses the interaction that they estimate would maximize information gain.

7.1.4 Relation to Rational Reconstruction

Although the theory of rational reconstruction is at a higher level of abstraction than either the cognitive theories of plans and beacons or the ecological theory of information foraging, rational reconstruction can be used as a support for both. For instance, rational reconstruction as explanations can help developers construct plans, or they can act as arguments to help developers justify their already-constructed plans. Explanations situated within source code are essentially explicit beacons; without rational reconstruction, the developer would likely need to identify these beacons implicitly anyway. Rational reconstruction points may also act as a navigational aid for information foraging theory, in cases where the developer's information estimate is contradictory to the reconstruction. However, I'm not sure how useful the explanations themselves would be if information foraging is the sole driver of comprehension.

7.2 Human Factors in Error and Warning Design

Of course, warnings messages aren't something people encounter exclusively in digital systems; they can be found in everyday situations and places, such as product

instruction manuals [407], medication [118, 249], cigarettes and alcohol labels [30, 207], and on signs at beaches [234]. But are theories of warning design [108] for *physical objects* applicable to the design of error messages in *digital systems* like program analysis tools? Research suggests that it is [148, 288]: Pieterse and Gelderblom [288], for example, successfully applied warning design theory—originally studied in the context of the physical world—to the design of error messages in a digital system like online Internet banking. As part of their study, Pieterse and Gelderblom [288] adopted guidelines from warning theory design theory, and conducted a heuristic evaluation in which experts using these guidelines to effectively identify problems with error messages. There are several noteworthy literature review in human factors which cover these and other dimensions for warnings and errors [92, 108, 206, 209, 280, 315, 400].

Mapping findings from warning design theory in the physical world to digital systems additional factors through which we can investigate the comprehension of error messages—beyond the theory of rational reconstruction that I've considered within this dissertation. As one example, the Communication-Human Information Processing (C-HIP) model is a communication theory from warning design with three conceptual stages: source, channel, and receiver [72]. Within the receiver stage, the model describes not only comprehension, but also attention, attitudes, and motivation for why a user may or may not successfully interpret the message. Similarly, the model also explains why a user might not notice the warning, due to processing bottlenecks in one of the stages of this pipeline. For error messages in program analysis tools, the C-HIP model could be applied to describe how information flows from one stage to the next.

In particular, the C-HIP model brings to attention a broader scope of error message comprehension than the one addressed in this dissertation. The thesis of this dissertation primarily focuses on the error message content, and how developers explicate this information as reconstructions in text or visual forms. But there are other factors that can support or preclude comprehension: for instance, a developer may not notice an error message in the first place. Our studies also considered only error messages that the developer needed to resolve in order to successfully compile the project; that is, the resolution of the error messages were mandatory. Consequently, rational reconstruction theory does not explain why developers might choose to ignore errors or warnings, or how they would prioritize multiple errors. But warning design theory might.

Finally, warning design theory can inform the design space of error message as a medium. These dimensions of medium include color [195], size [22, 41], location [207,

399], language of the warning text [2, 103, 148, 398], the pictorials or icons [86, 175, 363], expertise [42], habituation [150, 193], and social influence [96, 109, 124]. These dimensions might influence how developers interpret the severity of the message, whether they choose to take action for the message, which message they are likely to use if multiple presentations are available, and the saliency of the message.

7.3 Expert Systems

Expert systems are computer programs [219]. They are computer programs for reconstructing the expertise and reasoning capabilities of qualified specialists within limited domains [299], for emulating human expertise in well-defined problem domains [309], and for computationally representing task-specific human-knowledge of experts to systems [340]. While first-generation expert systems in the 1970s, such as DENDRAL [117] and MACSYMA [232], focused primarily on performance and problem-solving, second-generation systems, notably MYCIN [65, 339], introduced the distinguishing feature of explanation facilities as part of the problem-solving process [88].

MYCIN became a model for succeeding systems [299], and the development of expert systems became a forcing function in artificial intelligence research for computational explanations and problem-solving [357]. Davis [88] writes, “problem solving is only the most obvious [behavior of expert systems] and while necessary, it is clearly insufficient. Would we be willing to call someone an expert if he could solve a problem, but was able to explain the result? …I think not.” And Chandrasekaran and Swartout [56] observed that “explanation of a knowledge system’s conclusions can be as important to the conclusions themselves”.

In addition to the technical contributions of expert systems, people find their explanations to be useful across several dimensions [140]. An evaluation by Ye and Johnson [406] indicated that explanation-facilities can make advice from expert-systems more acceptable to users. They evaluated three types of explanation: inferential steps taken by the expert-system (trace or line reasoning), explicit description of the casual argument or rationale for each step taken by the expert system (justification), high-level goal structures to determine how the expert system uses its domain knowledge to accomplish the task (strategy). Of the three, justification was found to be the most effective in changing the user attitudes towards the system. Lim, Dey, and Avrahami [220] found explanations describing why a system behaved a certain way resulted in better understanding and stronger feelings trust. Wick

and Thompson [395] proposed a computational model of reconstructive explanations for expert systems: that effectiveness explanations often needs to substantially reorganize the actual line of reasoning as a line of explanation. However, Wick and Thompson [395] did not actually evaluate these explanations.

Though both novices and experts found explanations to be useful, they use the explanations in different ways. Mao and Benbasat [229] show that people requested explanation to deal with comprehension difficulties caused by perceived anomalies—not necessarily real anomalies—in expert system output messages, and that there were both qualitative and quantitative differences in the nature of explanation between novices and experts. Specifically, they found that novices relied on explanations for understanding the basic meaning, implication, and reasoning process of the expert system. In contrast, experts rapidly created their own rationalizations and used explanations as a source of confirmation. Fischer, Mastaglio, Reeves, and colleagues [120] proposed that systems should not attempt to generate a perfect, one-shot explanation: instead, systems should initially provide brief explanation and then allow the user to elaborate through several levels of explanation. Other researchers have also suggested that tailoring explanations to users improves the quality of explanation [24, 164, 181, 241, 244, 275].

Stand-alone expert systems eventually declined in popularity [10]; nevertheless their ideas becomes embedded in specialized contexts such as recommendation systems [276, 313, 365], and as intelligent tutoring systems [376]. The prolific rise and fall of experts systems brought to attention three critical weaknesses of expert systems, particularly with respect to explanation [184]:

1. There was no unifying theory of explanation, and few papers actually considered what explanations might be, or what might be acceptable as everyday explanations as opposed to scientific explanations.

2. The lack of theory of explanation also gives rise to the absence of any criteria for judging the quality of the explanation.

3. There were relatively few studies which evaluated to any degree the resulting explanations.

The third-point can be debated, as what constitutes relatively few studies is subjective. But the first two problems reveal a knowledge gap, not only for expert systems, but parallel that knowledge gaps regarding comprehensible error messages in program analysis tools. It is precisely these first two questions that this

dissertation investigates: it proposes a theory of rational reconstruction, and in doing so, offers a mechanism for constructing and evaluating error messages.

Given the similarities in goals between expert systems and error messages in program analysis tools, could program analysis tools be investigated as a form of expert system? If so, would explanations be as useful as they appear to be for expert systems? What lessons could we learn from their successes and failures? For example, linters—however rudimentary they may be— are essentially a collection of automated rules about problems in source code, encoded into the tool by expert developers. It may be fruitful to revisit the ideas from the expert systems to inform the implementation of program analysis tools.

7.4 Preventing Errors with Structure Editors

Structure editors, also called projectional editors or syntax-directed editors, make it difficult-to-impossible for developers to insert costly syntax errors into the source code in the first place [7]. Rather than having developers work with source code as text composed of strings of characters, they work directly with the syntax-tree structure of the program. Structure editors have a long history, with early efforts such as MENTOR [101] and the Cornell Program Synthesizer [364] in the 1980s. Although structure editors are not mainstream, there are a few contemporary and boutique structure editors intended for professional developers, such as Eco [97], Lamdu [226], Unison [63], and isomor*f* [190].

There are some arguments in favor of structure editing approaches, beyond preventing syntax errors [203, 331]. First, if developers think in terms of tree structures in their own mind, editors should support this mode of thinking [133, 192]. Second, structure editors express a typically-desirable property of closeness of mapping in that developers work with the underlying syntax-tree directly, rather than an abstraction of text [380]. Third, proponents of structure editing argue that antiquated text-based representations are inefficient ways of composing programs [284].

Unfortunately, the arguments in favor have not held up to scrutiny [374, 387]. Critics of structure editors have described them as restrictive, inflexible, and inefficient [192]. Thinking in terms of tree-structures turns out not to be natural at all [160]: complex tasks require substantial experience with the underlying abstract tree-structure, and additional developer experience lead doesn't to increased efficiency when composing programs [32]. Even trivial operations, such as entering algebraic expressions, were found to be frustrating and tedious for developers to

compose over traditional text-editing [362]. Finally, developers routinely work in malformed edit states: in the midst of a function call, for instance, "`std(m,`" they may realize that they need an additional helper function, and begin writing this helper function without first completing the original call [271]. Conventional structured editors forbid such common workflows [368].

Light-weight structure editing approaches have found their way into modern development environments, by relaxing the constraint of maintaining syntactically-valid source code. Context-sensitive code templates [389], intelligent copying-and-pasting of code fragments [383], auto-completion [227], and refactoring are examples of tools whose usage can reduce, but not entirely prevent, the introduction of syntax errors [131]. But relaxing the constraint of syntactic-validity brings us right back to this dissertation: there remains a need to support comprehensible error messages for syntax. And even theoretically-perfect structure editors do not prevent other categories of errors, such as semantic errors, that developers introduce in source code.

7.5 Error Messages for Novices

Despite cognitive differences in the way novices and experts problem-solve during debugging (Section 7.1 on page 128), examining literature for novices can provide insights into methods (Section 7.5.1), techniques (Section 7.5.2), and perspectives (Section 7.5.3) on the design of error messages for experts.

There is substantial research literature on making error messages more accessible to novices, spanning many decades, and for many programming languages and programming environments [23, 71, 87, 115, 119, 126, 153, 162, 189, 251, 354, 393, 394, 408]. In this section, I focus on recent literature applicable to modern programming environments as investigated within this dissertation.

7.5.1 Error Message Types and Distributions

An survey of the error distributions for novice error messages indicates that error distributions are long-tailed, such that relatively few errors account for the majority of problems encountered by novices. For example, Jadud [174] found that five most common errors of 42 different types of errors accounted for 58% of all errors: missing semicolons, unknown symbols: variable, bracket expected, illegal start of expressions, and unknown symbol: class. Denny, Luxton-Reilly, Tempero, and

colleagues [94] found that 73% of all submissions failed to compile due to a syntax error; in the top quartile, nearly 50% of all submissions failed to compile due to a syntax error. The median lines of code for the submitted programs was eight. A similar study by Jackson, Cobb, and Carver [172] found that the top ten errors to represent nearly 52% of all errors, and that the top twenty represent 62.5% of all errors. Pritchard [298] found the frequency of error messages to empirically resemble the Zipf-Mandelbrot distribution, a type of long-tailed distribution, across both Java and Python languages.

Further inspections of these data sets provide visibility into comprehension difficulties with error messages. For example, Jadud [174] noted, "instead of taking in the error or consulting additional help regarding the meaning of the error, the students would immediately hit the 'compile' button a second time." Instructors postulated that students "don't believe" the compilation error BlueJ was reporting to them in many cases. Brown and Altadmri [46] used the Blackbox dataset [6] for almost 100 million BlueJ compilations events. Combined with survey data of 76 educators, they found that educators' estimates of student errors did not agree either with one another or with the student data. Their data also reveals little learning effect over the course of 3 to 6 months for most students: the time to fix certain compiler errors—for example, syntax errors—reduces with additional experience, but they found no consistent effects for other errors, even when these errors are unrelated to program complexity, such as string comparisons. Finally, Pritchard [298] identified that while both Python and Java exhibited Zipf-Mandelbrot characteristics, differences in the parameters of the distribution suggested that Python perhaps provides more descriptive error messages than Java.

For experts, there are limited studies on the types of error messages developer encounter in practice [329]. There is a need for additional studies that characterize and pinpoint the space of confusing errors for developers, across both programming languages and tools.

7.5.2 Mini-Languages

Simplifying programming concepts by adopting subsets of a more expressive program language can eliminate confusing error messages, especially if novices aren't expected to use those capabilities in their programs in the first place. Removing language features can also help program analysis provide by disambiguating the space of possible diagnostics. For example, consider the following invalid `if` statement for the integers `x` and `m`:

```
if (x >> m) {
  return true;
}
```

If we interpret `x >> m` as a signed right-shift operator, then the problem is likely a type error: the result of this bit-shift is an integer, but `if` requires a boolean for the conditional. But if bitwise and bit-shift operators are prohibited within the language—perhaps because this concept isn't covered in the course—then it's far more likely than the student actually intended to simply perform a greater-than operation: `x > m`. In the latter case, the problem is likely a syntax error.

An early mini-languages is `LOGO`, a teaching environment that simplified and constrained the `BASIC` programming language to a turtle placed on a grid, through a number of limited movement commands [333]. This inspired a family of derivative visual languages, catalogued by Brusilovsky, Calabrese, Hvorecky, and colleagues [49].

There are also several text-based mini-languages. MiniJava, a teaching-oriented implementation of Java that discards over 700 built-in classes from the Java standard library to a manageable 17 core classes, eliminates inner classes, removes the do-while statement, and simplifies some other language constructs [312]. Language-levels in DrRacket provide full-fledged languages as staged into sub-languages that tailor programming concepts to what students have learned in the course [231]. And Helium, a variation of Haskell, aims to provider higher-quality error messages tailored for students [157]. Because Helium lacks removes type classes from the language, programs written in Helium are not generally compatible with Haskell.

Thinking about languages as subsets of the language can help improve error messages reporting for developers in programming analysis tools. For example, the unwieldily error message generated by LLVM in Section 2.2.4 on page 6 was in part due to the program analysis failing to distinguish standard library code from code the developer wrote. Awareness of the source code as being part of a standard library or developer-generated could improve the error messages from program analysis. For example, MiniJava specifically recognizes the use of the standard `Vector` library in Java, and can tailor messages when this class is used [312].

In contrast to novices, prohibiting expressive and useful concepts in programming languages is undesirable for experts—but sometimes it's necessary to do so because their error messages are unusable. For example, the Google Style Guide explicitly prohibits certain language constructs, instead suggesting alternative and sometimes less expressive implementations [134]. The guide disallows the use of preprocessor macros entirely, because they make it more difficult for programming

tools to perform analysis: “every error message from the compiler when developers incorrectly use that interface now must explain how the macros formed the interface. Refactoring and analysis tools have a dramatically harder time updating the interface.” [134]. Template programming is also strongly discouraged because it leads to poor compile-time messages, even for simple interfaces [134]. In effect, C++ at Google has become a mini-language. Perhaps better error messages would allow developers to freely use these more expressive constructs in their code.

7.5.3 Enhancing Compiler Error Messages

Although there are substantive differences between novices and experts in debugging [142, 240, 377], understanding how researchers enhance error messages for novices could provide avenues for investigating error messages in intermediate-expert developers. With that said, even experts are at times novices: for example, when needing to transfer skills from one learned programming language to another with no formal instruction [325, 403], or when contributing to unfamiliar code [200, 290].

Unfortunately, there is little consensus on what helps novices with error messages, even though researchers agree that error messages from production program analysis is inadequate for novices. Enhanced error messages, then, are the revisions made to the existing error messages in order to make them more appropriate for novices. For example, Denny, Luxton-Reilly, and Carpenter [93] implemented an feedback system in CodeWrite, a web-based tool in which students write the body of methods in Java. The enhanced feedback includes a table showing two code fragments side-by-side: the submitted version of the code with the error, and a corrected version of the code highlighting the syntax differences. CodeWrite presents an explanation of the error, and describes how it is corrected.

However, Denny, Luxton-Reilly, and Carpenter [93] found no significant effects between the enhanced and baseline error messages. They speculated that the types of errors may be simple enough to resolve without needing enhanced messages, or that students did not pay attention to the additional information in the error message. Pettit, Homer, and Gee [286] conducted a similar study, using an automated assessment tool for C++ called Athene. In addition to the original compiler error message from GCC, they added explanatory feedback the error messages and some hints for what to check Pettit, Homer, and Gee [286] found no measurable benefit to students: they and also suggested that perhaps students don’t attentively read the error messages, although students overwhelmingly self-reported reading

them. Nienaltowski, Pedroni, and Meyer [268] also found that more detailed error messages do not necessarily simplify the understanding of error messages; instead, it mattered more where the information was placed and how it was structured. A nicely constructed study by Prather, Pettit, McMurry, and colleagues [297] found that enhanced compiler error messages did not quantitatively show a substantially increase in learning outcomes over existing error messages, but qualitative results did show that students were reading the enhanced error messages and generally making effective changes to their code.

There are several other attempts to improve compiler error messages for students and present interesting ideas, but their studies have either limited or no evaluation, do not offer a principled rationale for how the error messages are improved, or yield inconclusive results [39, 71, 115, 123, 126, 168, 204, 324, 326, 388, 393].

Two notable exceptions are the studies by Becker, Glanville, Iwashima, and colleagues [27] and Marceau, Fisler, and Krishnamurthi [231]. A notable exception is the study by Becker, Glanville, Iwashima, and colleagues [27]. In contrast to Denny, Luxton-Reilly, and Carpenter [93] and Pettit, Homer, and Gee [286], Becker, Glanville, Iwashima, and colleagues [27] found that their enhanced error messages did significantly reduce the frequency of overall errors and errors per student for some types of errors. And Marceau, Fisler, and Krishnamurthi [231] investigated DrRacket, a novice development environment, to understand why enhanced messages remained confusing for students. Their results found that students struggled with the vocabulary of the error messages, and often misinterpreted the source highlighting. They concluded that–despite the considerable effort the development team invested into the design of error messages—the team lacked a clear model of errors and feedback to principally guide the error message design process. They have conducted subsequent to develop a rubric for assessing the performance of error messages [230], and these rubrics have been applied as formal processes for assessing and evaluating error reports, such as in Pyret [401].

Researchers acknowledge that error messages from production program analysis tools for experts shouldn't be the same as error messages for novices [126, 268, 281, 324, 401]. An extensive literature review by Qian and Lehman [300] offers a defense for this position, through examining barriers that novices have with syntactic knowledge, conceptual knowledge, and strategic knowledge. In other words, even when a novice and an expert encounter the same error message, where they get stuck and how they get stuck are likely to be very different. Moreover, feedback for novices in educational contexts are intended for learning, whereas feedback for professional developers is intended to help them productively identify and resolve

the issue [341].

Despite the differences in novices and experts, there are still several lessons we can draw from the study of novices. First, the study methodologies used to understand difficulties novices have with error messages can be adapted to experiments for intermediate and expert developers. Second, it seems that ad-hoc approaches to improving error messages is not all that successfully. Even when the error messages perform better than baseline errors, we gain few insights into why those messages are better. Instead, applying and developing principled models and theories to drive the careful design of the error message seems more productive, even if the process of theory-building initially delays our ability to actually revise error messages.

8 | Conclusion

> I'll be a story in your head. That's okay. We're all stories, in the end.
>
> The Doctor

8.1 Error: Expected Declaration or Statement at End of Input

The thesis statement of this dissertation is:

> Difficulties interpreting error messages produced by program analysis tools are a significant predictor of developers' inabilities to resolve defects: difficulties in interpreting error messages can be explained by framing error messages as insufficient rational reconstructions, in both visual and textual output presentations.

I defended the claims of the thesis statement through three studies. In the first study (Chapter 4), I investigated how developers used the Eclipse IDE to comprehend and resolve Java compiler error messages. I found that difficulties interpreting error messages produced by program analysis tools are a significant predictor of developers' inabilities to resolve defects. In the second study (Chapter 5), I conducted a participatory design activity through which developers constructed diagrammatic representations of error messages, overlaid on source code listings. The results of the second study identify how error messages are insufficiently aligned with developer expectations, particularly with respect to revealing relationships among

relevant program elements. In the third study, I analyzed Stack Overflow questions and answers relating to error messages using argument theory, a form of rational reconstruction. The third study (Chapter 6) finds that human-authored error messages incorporate complementary argument layout structures, and that these layout structures are significantly different from how error messages present errors to developers. Moreover, developers indicate significant preference for structures that either provide a simple resolution, or employ a proper simple or extended argument structure. Together, the results of these studies advance our scientific understanding for how developers comprehend error messages.

In the remainder of this chapter, I address some residual topics to situate the research in this dissertation in terms of its broader impacts. First, I synthesize the results of this research as design guidelines (Section 8.2). The design guidelines allow us to generalize the findings of this work to environments and programming languages beyond those studied in this dissertation. Second, I implement a proof-of-concept compiler in TypeScript that enables the presentation of error messages as rational reconstructions (Section 8.3). The proof-of-concept demonstrates that it is feasible to present rational reconstructions when appropriate introspective capabilities are present in the compiler infrastructure. Third, I discuss future work (Section 8.4). This discussion provides a landscape for what I believe are impactful directions within the discipline of error message comprehension.

8.2 Design Guidelines

All of our studies were conducted within the Java programming language, with the assumption that developers had familiarity with the Eclipse programming environment. To generalize the studies in this dissertation to other programming languages and environments [269], we derive five design guidelines that capture the principal findings from the studies conducted in this dissertation. Design guidelines from studies by other research pertaining to system error messages can be found in Appendix E.

Principle I—Implement rational reconstructions for humans, not tools. If a program analysis error message is convenient to construct from the perspective of the program analysis algorithm, this is a signal that the error message is inappropriate for the developers. Our results demonstrate that developers reason about error messages differently than the underlying mechanism by which program analysis

algorithms identify a problem in the source, and simply exposing internal program analysis processes to developers isn't useful. Consequently, a human-centered error message will likely require reconstruction. and toolsmiths should expect to do so to facilitate error message comprehension.

Principle II—Use code as the medium through which to situate error messages. Provide context for error messages, framed through the primary artifact within which developer work—source code. In graphical environments, consider using affordances which display error message information in proximity to the source code. In text environments, consider inlining the source code context as part of the error message presentation. When using diagrammatic elements, be ware of introducing unfamiliar notation. Stick with basic diagram techniques, such as lines, boxes, and arrows.

Principle III—Distinguish source code from explanation of source code. Avoid mixing natural language and source code within the same sentence, as mixing the two modes of natural language to context-switch incurs a cognitive processing cost. Instead, clearly separate natural language explanation and its accompanying source code as part of the rational reconstruction. If necessary, it's okay to use less-precise language to support readability, if it's clear from the context what is being referred to.

Principle IV—Present rational reconstructions as coherent narrative of causes to symptoms. As with good arguments, error messages are easier to comprehend if there is a coherent narrative of causes to symptoms that logically explain why a problem has occurred. These sequences need not be elaborate: often a single, straight-forward backing can lead a developer to comprehend an otherwise inscrutable message. Developers find it helpful if the error message explicitly relates why different program elements are relevant to a particular problem.

Principle V—Give developers autonomy over error message presentation. Depending on familiarity with the code and experts, developers may need to more or less help in comprehending the problem. Support mechanisms to progressively elaborate error messages which support both expert and occasional developers.

8.3 Toward Engineering a Compiler

8.3.1 Approach

To apply our theory into practice, I operationalized our design guidelines within a production compiler (Section 8.2). Specifically, I modified the Microsoft TypeScript compiler to generate rational reconstructions for `TS2393`, a duplicate function implement error. I assessed this prototype through a formative, solution validation, conducted through a focus group. The prototype demonstrates the feasibility of engineering a compiler to support rational reconstruction, as well as its potential utility if incorporated into practical program analysis tools.

I selected the TypeScript compiler for modification for several, mostly practical reasons. Namely, the compiler is engineered with modern compiler design principles, such as offering compiler toolsmiths an API-as-a-service. This makes it is easy to introspect and to extend the compiler. The TypeScript source code builds relatively quickly, on the order of seconds, and this rapid turn-around facilitates rapid iteration when developing the prototyped. Additionally, the TypeScript compiler is itself written using TypeScript, which means that the tools for developing the compiler and the tools for testing rational reconstructions are unified. These properties make TypeScript a convenient target for error message research.

Similarly, the decision to support a duplicate function implement as the vehicle to investigate rational reconstructions was also not entirely arbitrary, and required balancing several design constraints. I wanted an error message that would highlight relationships between multiple program elements; this eliminated classes of errors, such as syntax errors, whose reporting would only involve a single error. I wanted an error message that would require introspecting the compiler during the creation of the rational reconstruction, yet at the time did not want to spend extraordinary effort to surface the underlying compiler structures. To support a potential study, I also wanted an error message that used concepts that would likely to be familiar to even occasional TypeScript developers. The duplicate function implementation error meets all of these constraints. Moreover, variations of the error message were used in previous studies: T8 in (Chapter 4), and `Brick` in (Chapter 5).

Essentially, there are two approaches to implementing rational reconstruction within the TypeScript architecture. For the first approach, we could modify the TypeScript compiler itself to construct rational reconstructions during the compilation process. This turned out to be impractical, because it required knowing which information the compiler should collect before we knew when (or if) we could even

encounter an error. The second approach is retrospective: we let the compiler run it's course as it would normally do. When the compiler identifies an error message, we suppress the baseline error message. Instead, we interrogate the compiler through introspection and request additional details. The information requested during introspective is then used to construct a rational reconstruction. The implementations details for how this process works is found in Appendix G.

8.3.2 Example: Duplicate Function Implementation

I implemented a rational reconstruction for TypeScript error `TS2393`, a duplication function implemented error. For example, consider the following TypeScript file, which induces `TS2393` because of the function `foo`:

```
1 function baz() { }
2
3 function foo(a: boolean) {
4 }
5
6 function bar(): void { return; }
7
8 function foo(b: boolean, c: boolean) {
9 }
```

Unlike Java, TypeScript does not support multiple function implementations with the same name.[1] Consequently, the above source listing results in the following TypeScript error message:

```
file.ts(3,10): error TS2393: Duplicate function implementation.
file.ts(8,10): error TS2393: Duplicate function implementation.
```

This error message is an insufficient rational reconstruction for a developer, in several ways. First, the presentation of the error message does not explicitly acknowledge that the duplicate function implementation errors are in fact related to each other. The error message for Line 3 is a reciprocal of the error message on Line 8. Second, the error message does not indicate why this is a problem—though such an explanation might not be needed for an expert who routinely encounters this type of message. Third, other than line number and location, there is no contextual beacon from which the developer can relate this error message back to their source

[1] If the developer wants to overload a function, they must use an alternative implementation that involves supplying function types to a single implementation. This single implementation then explicitly performs type checks to guide the behavior of the function.

code. Finally, even if the brief error message is suitable for some circumstances, there is no way for the developer to request a more elaborate explanation of the problem.

In Rational TypeScript, the error message is emitted as:

```
error[TS2393]: duplicate implementation of function `foo`
 --> file.ts:8:10
  |
3 × function foo(a: boolean) {
  |          --- previous implementation of `foo` here
...
8 × function foo(b: boolean, c: boolean) {
  |          ~~~ `foo` reimplemented here
  |
  = hint: `foo` must be implemented only once within the same namespace
  = hint: To overload a function, see
          https://www.typescriptlang.org/docs/handbook/functions.html
```

This error message has several appealing properties that make it suitable for a lightweight. The key differences from the baseline TypeScript is message is that Rational TypeScript collects the duplication function implementations and presents them as a single, related error. Furthermore, the error message informs the developer of how the problem manifests, using the context of their source code in the reconstruction. Through hints, the error messages provides additional warrants for why the problem is a problem. The error message provides a pointer to additional documentation from the TypeScript handbook, for the use case in which the developer is intending to perform function overloading. Finally, the error message uses colorized output to visually partition the different components of the error messages, such as the source code and explanatory text.

8.3.3 Formative Evaluation

Method. I conducted a 30-minute formative evaluation of Rational TypeScript, focusing on the presentation of the error message. *Participants.* I recruited participants (E1-E5) within Machine Learning at Microsoft, across various levels of seniority (Table 8.1). Participants worked primarily in either TypeScript or C#, but occasionally had to contribute code to their secondary language, for example, when adding features or resolving bugs that spanned both front-end and back-end development. Due to the nature of build system, all participants used Visual Studio Code or Visual Studio to write the soft, but compiled code using a command-line

Table 8.1 Participants in Focus Group

ID	Title	Organization	Primary Language
E1	Principal Software Engineer	Machine Learning	C#
E2	Principal Software Engineer	Machine Learning	TypeScript
E3	Senior Software Engineer	Machine Learning	C#
E4	Senior Software Engineer	Office Online	TypeScript
E5	Software Engineer	Machine Learning	TypeScript

Participants attended a 30-minute focus group session. During the session, they evaluated baseline TypeScript error messages and rational reconstructions for a duplicate function implementation error.

console. *Study protocol.* Participants were shown a demo of the duplicate function implementation error, along with baseline and rational reconstruction versions of the error message in TypeScript. Participants were asked about for their feedback about the error message presentations, and when they would prefer one or the other. Both versions of the error messages were made available for the duration of the discussion. *Analysis.* Feedback was collected from the session, and qualitatively summarized by myself.

Results. Participants reported that Rational TypeScript messages were more helpful than baseline TypeScript messages, particularly with developers who only sporadically program with TypeScript (E1, E3). Although full-time TypeScript developers generally preferred the brevity of baseline error messages for routine errors (E2, E4), they nevertheless indicated that rational reconstructions would be useful as a presentation option for error messages when working with unfamiliar code (E2, E4, E5). The results of the formative evaluation, though limited, encourage us to pursue rational reconstructions in the design and implementation of practical program analysis tools.

8.4 Future Work

There are many fruitful avenues of research that are worthwhile to pursue but beyond the scope of this dissertation. Here are just a few directions for rational reconstructions in program analysis tools:

- **Benefits of wrong error messages.** This dissertation investigated the pre-

sentation of error messages, with the assumption that all reported errors are true positives. But most program analysis is an approximation of the actual behavior of the program. As such, it is possible that the rational reconstruction generated by the program analysis tool is actually wrong! Would incorrect rational reconstructions still be useful to developers? Would such reconstructions, for example, allow developers to confidently assess that the error diagnostic is a false positive and dismiss it? Or should we only present rational reconstructions to developers when we have a high degree of confidence that the diagnostic is accurate?

- **Supporting developer diversity.** A limitation of our studies is that we primarily studied entry-level software developers, and only those with experience in Java and the Eclipse IDE. However, our formative evaluation with Rational TypeScript using principal and senior software developers reveals that rational reconstructions may not always be the appropriate form of error message presentation. For example, experts in a programming language would likely be able to easily repair syntax errors in the program without an elaborate error message. How do we construct rational reconstructions to support the spectrum of developers?

- **Error message telemetry and longitudinal research.** One difficulty with assessing whether error messages help developers is that the effect size of any particular instance of an improved error message against a baseline error message is small. Thus, the impact of better error messages is likely to only be noticeable over time. Unfortunately, today we have limited telemetry and longitudinal data on error message distributions. For example, our studies relied solely on the Google error distribution from Seo, Sadowski, Elbaum, and colleagues [329] to inform our research designs, and Google has proprietary build processes that are not necessarily representative of other organizations. From what other sources might be obtain telemetry on program analysis error messages?

- **Conversational program analysis tools.** The interaction model for our study was linear and one-way: given a problem in the source code, the program analysis tool could report and present the problem to the developer. However, there is no mechanism for the developer to subsequently interact with the program analysis tool ask questions about the error message. How could program analysis tools be made to be conversational agents? And would conversational program analysis tools actually help developers?

- **Interviews with authors of program analysis tools.** Presumably, authors of program analysis tools aren't intentionally going out to their way to design inscrutable error messages—at least I would hope not! And some programming language communities, such as LLVM, Rust and Elm, have made commitments to improving the quality of their error reporting for developers. So how do poor error messages arise in practice? Are there tools or techniques that we can develop to reduce the authorial burden of writing good error messages? What are the perceptions of tool authors on the quality of their own messages? Do authors of program analysis tools have realistic assumptions about the developers who use their tools? Understanding the process through which authors construct error messages can help pinpoint where the problem really is, whether social, technological, or both.
- **Computational generation of rational reconstructions.** The rational reconstructions in this study were manually constructed. Although constructing these error messages is possible to do—and it's what program analysis authors currently do—the authorial burden of constructing a good error message is high. Would it possible to build intelligent program analysis tools to automatically explain their own diagnostics to developers? One avenue for pursuing this line of thinking might be to draw inspiration from early ideas in expert systems.

8.5 Epilogue

Curiously, implementing the design guidelines in a medley of program analysis tools—and evaluating their usefulness and effectiveness—seems just the right amount of material for a second dissertation.

```
# and if the sun comes up tomorrow,
# let her be

> python
>>> from __future__ import braces
  File "<stdin>", line 1
SyntaxError: not a chance
>>> import antigravity
```

BIBLIOGRAPHY

[1] G. D. Abowd and R. Beale, "Users, systems and interfaces: A unifying framework for interaction," in *People and Computers VI*, 1991, pp. 73–87 (see pp. 196, 211).

[2] A. Adams, S. Bochner, and L. Bilik, "The effectiveness of warning signs in hazardous work places: cognitive and social determinants," *Applied Ergonomics*, vol. 29, no. 4, pp. 247–254, 1998 (see p. 132).

[3] A.-R. Adl-Tabatabai and T. Gross, "Source-level debugging of scalar optimized code," in *Proceedings of the ACM SIGPLAN 1996 Conference on Programming Language Design and Implementation*, ser. PLDI '96, Philadelphia, Pennsylvania, USA: ACM, 1996, pp. 33–43 (see p. 200).

[4] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, "Compilers: Principles, Techniques, and Tools," 2007 (see p. 29).

[5] S. Ainsworth and A. T. Loizou, "The effects of self-explaining when learning with text or diagrams," *Cognitive Science*, vol. 27, no. 4, pp. 669–681, 2003 (see p. 100).

[6] A. Altadmri and N. C. Brown, "37 million compilations: Investigating novice programming mistakes in large-scale student data," in *SIGCSE*, 2015, pp. 522–527 (see pp. 45, 125, 136).

[7] A. Altadmri, M. Kolling, and N. C. C. Brown, "The Cost of Syntax and How to Avoid It: Text versus Frame-Based Editing," in *2016 IEEE 40th Annual Computer Software and Applications Conference (COMPSAC)*, IEEE, 2016, pp. 748–753 (see p. 134).

[8] B. de Alwis and G. Murphy, "Using visual momentum to explain disorientation in the Eclipse IDE," in *VL/HCC*, 2006, pp. 51–54 (see pp. 72, 74).

[9] G. Ammons, D. Mandelin, R. Bodík, and J. R. Larus, "Debugging temporal specifications with concept analysis," in *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, ser. PLDI '03, San Diego, California, USA: ACM, 2003, pp. 182–195 (see p. 201).

[10] C. Angeli, "Diagnostic expert systems: From expert's knowledge to real-time systems," *Advanced knowledge based systems: Model, applications & research*, vol. 1, pp. 50–73, 2010 (see p. 133).

[11] J. Ansel, C. Chan, Y. L. Wong, M. Olszewski, Q. Zhao, A. Edelman, and S. Amarasinghe, "Petabricks: A language and compiler for algorithmic choice," in *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '09, Dublin, Ireland: ACM, 2009, pp. 38–49 (see p. 204).

[12] H. Arksey and L. O'Malley, "Scoping studies: towards a methodological framework," *Int J Soc Res Methodol*, vol. 8, 2005 (see p. 197).

[13] A. Ayers, R. Schooler, C. Metcalf, A. Agarwal, J. Rhee, and E. Witchel, "Traceback: First fault diagnosis by reconstruction of distributed control flow," in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '05, Chicago, IL, USA: ACM, 2005, pp. 201–212 (see p. 206).

[14] N. Ayewah, D. Hovemeyer, J. D. Morgenthaler, J. Penix, and W. Pugh, "Using Static Analysis to Find Bugs," *IEEE Software*, vol. 25, no. 5, pp. 22–29, 2008 (see p. 25).

[15] T. Ball, M. Naik, S. K. Rajamani, T. Ball, M. Naik, and S. K. Rajamani, "From symptom to cause: Localizing errors in counterexample traces," in *Proceedings of the 30th ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL '03*, vol. 38, New York, New York, USA: ACM Press, 2003, pp. 97–105 (see p. 41).

[16] T. Ball and S. K. Rajamani, "The SLAM Project: Debugging System Software via Static Analysis," in *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '02, New York, NY, USA: ACM, 2002, pp. 1–3 (see p. 40).

[17] T. Barik, Y. Song, B. Johnson, and E. Murphy-Hill, "From quick fixes to slow fixes: Reimagining static analysis resolutions to enable design space exploration," in *IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2016, pp. 211–221 (see p. 72).

[18] T. Barik, K. Lubick, S. Christie, and E. Murphy-Hill, "How developers visualize compiler messages: A foundational approach to notification construction," in *2014 Second IEEE Working Conference on Software Visualization*, 2014, pp. 87–96 (see pp. 77, 123).

[19] T. Barik, C. Parnin, and E. Murphy-Hill, "One λ at a time: What do we know about presenting human-friendly output from program analysis tools?" In *PLATEAU'17 Workshop on Evaluation and Usability of Programming Languages and Tools*, 2017 (see p. 195).

[20] T. Barik, J. Smith, K. Lubick, E. Holmes, J. Feng, E. Murphy-Hill, and C. Parnin, "Do developers read compiler error messages?" In *Proceedings of the 39th International Conference on Software Engineering*, ser. ICSE '17, Buenos Aires, Argentina: IEEE Press, 2017, pp. 575–585 (see pp. 49, 103, 196).

[21] T. Barik, J. Witschey, B. Johnson, and E. 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*, ser. ICSE Companion 2014, Hyderabad, India: ACM, 2014, pp. 536–539 (see pp. 196, 209).

[22] T. Barlow and M. S. Wogalter, "Increasing the surface area on small product containers to facilitate communication of label information and warnings," *Proceedings of Interface*, vol. 91, no. 7, pp. 88–93, 1991 (see p. 131).

[23] M. Barr, S. Holden, D. Phillips, and T. Greening, "An Exploration of Novice Programming Errors in an Object-oriented Environment," in *Working Group Reports from ITiCSE on Innovation and Technology in Computer Science Education*, ser. ITiCSE-WGR '99, New York, NY, USA: ACM, 1999, pp. 42–46 (see p. 135).

[24] J. A. Bateman and C. Paris, "Phrasing a text in terms the user can understand," in *IJCAI*, 1989, pp. 1511–1517 (see p. 133).

[25] (2018). Bazel: Build and test software of any size, quickly and reliably, [Online]. Available: `https://bazel.build/` (visited on 01/01/2018) (see p. 31).

[26] M. Beaven and R. Stansifer, "Explaining type errors in polymorphic languages," *ACM Lett. Program. Lang. Syst.*, vol. 2, no. 1-4, pp. 17–30, 1993 (see p. 39).

[27] B. A. Becker, G. Glanville, R. Iwashima, C. McDonnell, K. Goslin, and C. Mooney, "Effective compiler error message enhancement for novice program-

ming students," *Computer Science Education*, vol. 26, no. 2-3, pp. 148–175, 2016 (see p. 139).

[28] R. Bednarik and M. Tukiainen, "Temporal eye-tracking data: Evolution of debugging strategies with multiple representations," in *ETRA*, Savannah, Georgia, 2008, pp. 99–102 (see p. 74).

[29] I. Beer, S. Ben-David, H. Chockler, A. Orni, and R. Trefler, "Explaining Counterexamples Using Causality," in *Computer Aided Verification*, A. Bouajjani and O. Maler, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 94–108 (see p. 41).

[30] R. F. Beltramini, "Perceived Believability of Warning Label Information Presented in Cigarette Advertising," *Journal of Advertising*, vol. 17, no. 2, pp. 26–32, 1988 (see p. 131).

[31] D. Benyon and D. Murray, "Applying user modeling to human-computer interaction design," *Artificial Intelligence Review*, vol. 7, no. 3, pp. 199–225, 1993 (see p. 15).

[32] T. Berger, M. Völter, H. P. Jensen, T. Dangprasert, and J. Siegmund, "Efficiency of Projectional Editing: A Controlled Experiment," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2016, New York, NY, USA: ACM, 2016, pp. 763–774 (see p. 134).

[33] G. Bierman, M. Abadi, and M. Torgersen, "Understanding TypeScript," in *ECOOP 2014 – Object-Oriented Programming: 28th European Conference, Uppsala, Sweden, July 28 – August 1, 2014. Proceedings*, R. Jones, Ed., Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 257–281 (see p. 36).

[34] D. Binkley, "Source code analysis: A road map," in *Future of Software Engineering, 2007. FOSE '07*, 2007, pp. 104–119 (see p. 24).

[35] M. Birks, Y. Chapman, and K. Francis, "Memoing in qualitative research: Probing data and processes," *Journal of Research in Nursing*, vol. 13, no. 1, pp. 68–75, 2008 (see p. 116).

[36] S. Blackshear and S. K. Lahiri, "Almost-correct specifications: A modular semantic framework for assigning confidence to warnings," in *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design*

and Implementation, ser. PLDI '13, Seattle, Washington, USA: ACM, 2013, pp. 209–218 (see p. 205).

[37] M. D. Bond, G. Z. Baker, and S. Z. Guyer, "Breadcrumbs: Efficient context sensitivity for dynamic bug detection analyses," in *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '10, Toronto, Ontario, Canada: ACM, 2010, pp. 13–24 (see p. 204).

[38] P. N. van den Bosch, "A Bibliography on Syntax Error Handling in Context Free Languages," *SIGPLAN Not.*, vol. 27, no. 4, pp. 77–86, 1992 (see p. 30).

[39] B. D. Boulay and I. Matthew, "Fatal error in pass zero: how not to confuse novices," *Behaviour & Information Technology*, vol. 3, no. 2, pp. 109–118, 1984 (see p. 139).

[40] N. Boustani and J. Hage, "Improving type error messages for generic Java," *Higher-Order and Symbolic Computation*, vol. 24, no. 1-2, pp. 3–39, 2011 (see p. 100).

[41] C. C. Braun, N. C. Silver, and B. R. Stock, "Likelihood of Reading Warnings: The Effect of Fonts and Font Sizes," *Proceedings of the Human Factors and Ergonomics Society Annual Meeting*, vol. 36, no. 13, pp. 926–930, 1992 (see p. 131).

[42] C. Bravo-Lillo, L. F. Cranor, J. Downs, and S. Komanduri, "Bridging the Gap in Computer Security Warnings: A Mental Model Approach," *IEEE Security & Privacy*, vol. 9, no. 2, pp. 18–26, 2011 (see p. 132).

[43] G. Brooks, G. J. Hansen, and S. Simmons, "A new approach to debugging optimized code," in *Proceedings of the ACM SIGPLAN 1992 Conference on Programming Language Design and Implementation*, ser. PLDI '92, San Francisco, California, USA: ACM, 1992, pp. 1–11 (see pp. 200, 201).

[44] R. Brooks, "Towards a theory of the cognitive processes in computer programming," *International Journal of Human-Computer Studies*, vol. 51, no. 2, pp. 197–211, 1977 (see p. 129).

[45] ——, "Towards a theory of the comprehension of computer programs," *International Journal of Man-Machine Studies*, vol. 18, no. 6, pp. 543–554, 1983 (see p. 129).

[46] N. C. C. Brown and A. Altadmri, “Novice Java Programming Mistakes: Large-Scale Data vs. Educator Beliefs,” *Trans. Comput. Educ.*, vol. 17, no. 2, 7:1–7:21, 2017 (see p. 136).

[47] P. J. Brown, “‘My system gives excellent error messages’–or does it?” *Software: Practice and Experience*, vol. 12, no. 1, pp. 91–94, 1982 (see p. 210).

[48] ——, “Error messages: The neglected area of the man/machine interface,” *Commun. ACM*, vol. 26, no. 4, pp. 246–249, 1983 (see pp. 47, 103, 124, 196, 210, 266).

[49] P. Brusilovsky, E. Calabrese, J. Hvorecky, A. Kouchnirenko, and P. Miller, “Mini-languages: a way to learn programming principles,” *Education and Information Technologies*, vol. 2, no. 1, pp. 65–83, 1997 (see p. 137).

[50] S. Burckhardt, M. Fahndrich, P. de Halleux, S. McDirmid, M. Moskal, N. Tillmann, and J. Kato, “It’s alive! continuous feedback in ui programming,” in *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI ’13, Seattle, Washington, USA: ACM, 2013, pp. 95–104 (see p. 203).

[51] T. Busjahn, R. Bednarik, A. Begel, M. Crosby, J. H. Paterson, C. Schulte, B. Sharif, and S. Tamm, “Eye movements in code reading: Relaxing the linear order,” in *ICPC*, Florence, Italy, 2015, pp. 255–265 (see p. 74).

[52] C. Calcagno, D. Distefano, P. W. O’Hearn, and H. Yang, “Compositional shape analysis by means of bi-abduction,” *J. ACM*, vol. 58, no. 6, 26:1–26:66, 2011 (see p. 271).

[53] J. L. Campbell, C. Quincy, J. Osserman, and O. K. Pedersen, “Coding In-depth Semistructured Interviews,” *Sociological Methods & Research*, vol. 42, no. 3, pp. 294–320, 2013 (see p. 112).

[54] J. C. Campbell, A. Hindle, and J. N. Amaral, “Syntax errors just aren’t natural: improving error reporting with language models,” in *Proceedings of the 11th Working Conference on Mining Software Repositories - MSR 2014*, 2014, pp. 252–261 (see pp. 29, 30, 100, 103).

[55] E. Chailloux, P. Manoury, and B. Pagano, “Program Analysis Tools,” in *Developing Applications with Objective Caml*, O’Reilly Media, 2000 (see pp. 24, 25).

[56] B. Chandrasekaran and W. Swartout, “Explanations in knowledge systems: the role of explicit representation of design knowledge,” *IEEE Expert*, vol. 6, no. 3, pp. 47–49, 1991 (see p. 132).

[57] B. Chandrasekaran, M. C. Tanner, and J. R. Josephson, “Explaining control strategies in problem solving,” *IEEE Expert*, vol. 4, no. 1, pp. 9–15, 1989 (see p. 14).

[58] A. Charguéraud, “Improving Type Error Messages in OCaml,” in *Proceedings ML/OCaml*, 2014, pp. 80–97. arXiv: 1512.01897 (see pp. 37, 38).

[59] P. Charles and D. Shields, *Frequently asked questions about jikes*, 1998 (see p. 109).

[60] S. Chen, M. Erwig, and K. Smeltzer, “Let’s hear both sides: On combining type-error reporting tools,” in *VL/HCC ’14*, 2014, pp. 145–152 (see p. 100).

[61] Y. Chen, A. Groce, C. Zhang, W.-K. Wong, X. Fern, E. Eide, and J. Regehr, “Taming compiler fuzzers,” in *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI ’13, Seattle, Washington, USA: ACM, 2013, pp. 197–208 (see p. 205).

[62] S. Cherem, L. Princehouse, and R. Rugina, “Practical memory leak detection using guarded value-flow analysis,” in *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI ’07, San Diego, California, USA: ACM, 2007, pp. 480–491 (see p. 203).

[63] P. Chiusano. (2017). Unison: Next-generation programming platform, [Online]. Available: http://unisonweb.org/ (see p. 134).

[64] M. Christakis and C. Bird, “What developers want and need from program analysis: an empirical study,” in *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering - ASE 2016*, New York, New York, USA: ACM Press, 2016, pp. 332–343 (see pp. 50, 125, 196).

[65] W. J. Clancey, “The epistemology of a rule-based expert system —a framework for explanation,” *Artificial Intelligence*, vol. 20, no. 3, pp. 215–251, 1983 (see p. 132).

[66] *Clang Static Analyzer*, http://clang-analyzer.llvm.org/ (see p. 71).

[67] E. M. Clarke, “The Birth of Model Checking,” in *25 Years of Model Checking: History, Achievements, Perspectives*, O. Grumberg and H. Veith, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 1–26 (see p. 39).

[68] E. M. Clarke, O. Grumberg, and D. A. Peled, *Model Checking*. The MIT Press, 2000 (see p. 39).

[69] E. Clarke, D. Kroening, and F. Lerda, “A Tool for Checking ANSI-C Programs,” in, Springer, Berlin, Heidelberg, 2004, pp. 168–176 (see p. 40).

[70] *Compiler errors for humans* (see pp. 34, 118).

[71] R. W. Conway and T. R. Wilcox, “Design and Implementation of a Diagnostic Compiler for PL/I,” *Commun. ACM*, vol. 16, no. 3, pp. 169–179, 1973 (see pp. 135, 139).

[72] V. C. Conzola and M. S. Wogalter, “A Communication–Human Information Processing (C–HIP) approach to warning effectiveness in the workplace,” *Journal of Risk Research*, vol. 4, no. 4, pp. 309–322, 2001 (see p. 131).

[73] L. Cooke and E. Cuddihy, “Using eye tracking to address limitations in think-aloud protocol,” English, in *International Professional Communication Conference*, 2005, pp. 653–658 (see p. 74).

[74] E. Coppa, C. Demetrescu, and I. Finocchi, “Input-sensitive profiling,” in *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI ’12, Beijing, China: ACM, 2012, pp. 89–98 (see p. 205).

[75] J. C. Corbett, M. B. Dwyer, J. Hatcliff, and Robby, “Bandera: A Source-level Interface for Model Checking Java Programs,” in *Proceedings of the 22Nd International Conference on Software Engineering*, ser. ICSE ’00, New York, NY, USA: ACM, 2000, pp. 762–765 (see p. 40).

[76] D. S. Coutant, S. Meloy, and M. Ruscetta, “Doc: A practical approach to source-level debugging of globally optimized code,” in *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*, ser. PLDI ’88, Atlanta, Georgia, USA: ACM, 1988, pp. 125–134 (see p. 200).

[77] J. W. Creswell, *Research Design: Qualitative, Quantitative, and Mixed Methods Approaches*, Fourth. SAGE Publications, 2014 (see p. 16).

[78] M. E. Crosby, J. Scholtz, and S. Wiedenbeck, "The roles beacons play in comprehension for novice and expert programmers," in *14th Workshop of the Psychology of Programming Interest Group*, 2002, pp. 58–73 (see p. 129).

[79] M. Crotty, *The Foundations of Social Research: Meaning and Perspective in the Research Process*. SAGE Publications, 1998 (see p. 16).

[80] C. Csallner and Y. Smaragdakis, "Check 'N' Crash: Combining Static Checking and Testing," in *Proceedings of the 27th International Conference on Software Engineering*, ser. ICSE '05, New York, NY, USA: ACM, 2005, pp. 422–431 (see p. 25).

[81] V. D'Silva, D. Kroening, and G. Weissenbacher, "A Survey of Automated Techniques for Formal Software Verification," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 27, no. 7, pp. 1165–1178, 2008 (see p. 39).

[82] L. Damas and R. Milner, "Principal type-schemes for functional programs," in *Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '82, Albuquerque, New Mexico: ACM, 1982, pp. 207–212 (see p. 36).

[83] N. Danas, T. Nelson, L. Harrison, S. Krishnamurthi, and D. J. Dougherty, "User Studies of Principled Model Finder Output," in *Software Engineering and Formal Methods*, A. Cimatti and M. Sirjani, Eds., Cham: Springer International Publishing, 2017, pp. 168–184 (see pp. 41, 196).

[84] S. A. Dart, R. J. Ellison, P. H. Feiler, and A. N. Habermann, "Software Development Environments," *Computer*, vol. 20, no. 11, pp. 18–28, 1987 (see p. 42).

[85] I. F. Darwin, *Java Cookbook*. O'Reilly Media, Inc., 2004 (see p. 109).

[86] S. Davies, H. Haines, B. Norris, and J. R. Wilson, "Safety pictograms: are they getting the message across?" *Applied Ergonomics*, vol. 29, no. 1, pp. 15–23, 1998 (see p. 132).

[87] E. A. Davis, M. C. Linn, and M. Clancy, “Learning to Use Parentheses and Quotes in LISP,” *Computer Science Education*, vol. 6, no. 1, pp. 15–31, 1995 (see p. 135).

[88] R. Davis, “Expert Systems: Where Are We? And Where Do We Go from Here?” *AI Magazine*, vol. 3, no. 2, p. 3, 1982 (see p. 132).

[89] M. Dean, “How a computer should talk to people,” *IBM Systems Journal*, vol. 21, no. 4, pp. 424–453, 1982 (see pp. 15, 47, 124, 265).

[90] P. Degano and C. Priami, “Comparison of syntactic error handling in LR parsers,” *Software: Practice and Experience*, vol. 25, no. 6, pp. 657–679, 1995 (see p. 30).

[91] ——, “LR techniques for handling syntax errors,” *Computer Languages*, vol. 24, no. 2, pp. 73–98, 1998 (see p. 30).

[92] D. M. DeJoy, “Consumer Product Warnings: Review and Analysis of Effectiveness Research,” *Proceedings of the Human Factors Society Annual Meeting*, vol. 33, no. 15, pp. 936–940, 1989 (see p. 131).

[93] P. Denny, A. Luxton-Reilly, and D. Carpenter, “Enhancing syntax error messages appears ineffectual,” in *ITiCSE*, 2014, pp. 273–278 (see pp. 50, 100, 138, 139).

[94] P. Denny, A. Luxton-Reilly, E. Tempero, and J. Hendrickx, “Understanding the Syntax Barrier for Novices,” in *Proceedings of the 16th Annual Joint Conference on Innovation and Technology in Computer Science Education*, ser. ITiCSE ’11, New York, NY, USA: ACM, 2011, pp. 208–212 (see p. 135).

[95] N. K. Denzin, “Moments, Mixed Methods, and Paradigm Dialogs,” *Qualitative Inquiry*, vol. 16, no. 6, pp. 419–427, 2010 (see p. 17).

[96] M. A. DeTurck, I.-H. Chih, and Y.-P. Hsu, “Three Studies Testing the Effects of Role Models on Product Users’ Safety Behavior,” *Human Factors*, vol. 41, no. 3, pp. 397–412, 1999 (see p. 132).

[97] L. Diekmann and L. Tratt, “Eco: A Language Composition Editor BT - Software Language Engineering: 7th International Conference, SLE 2014, Västerås, Sweden, September 15-16, 2014. Proceedings,” in, B. Combemale,

D. J. Pearce, O. Barais, and J. J. Vinju, Eds., Cham: Springer International Publishing, 2014, pp. 82–101 (see p. 134).

[98] I. Dillig, T. Dillig, and A. Aiken, "Automated error diagnosis using abductive inference," in *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '12, Beijing, China: ACM, 2012, pp. 181–192 (see p. 203).

[99] D. von Dincklage and A. Diwan, "Explaining failures of program analyses," in *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '08, Tucson, AZ, USA: ACM, 2008, pp. 260–269 (see p. 204).

[100] A. Dix, J. Finlay, G. D. Abowd, and R. Beale, *Human-Computer Interaction*, 3rd ed. Prentice-Hall, 2004 (see p. 46).

[101] V. Donzeau-Gouge, G. Huet, G. Kahn, and B. Lang, "Programming environments based on structured editors: The MENTOR experience," INSTITUT NATIONAL DE RECHERCHE D'INFORMATIQUE ET D'AUTOMATIQUE ROCQUENCOURT (FRANCE), Tech. Rep., 1980 (see p. 134).

[102] *dotCover: A code coverage tool for .NET*, `http://www.jetbrains.com/dotcover/` (see p. 42).

[103] E. Duarte, F. Rebelo, J. Teles, and P. Noriega, "What should i do? - A study about conflicting and ambiguous warning messages," *Work*, vol. 41, no. SUPPL.1, pp. 3633–3640, 2012 (see p. 132).

[104] D. Duggan and F. Bent, "Explaining type inference," *Science of Computer Programming*, vol. 27, no. 1, pp. 37–83, 1996 (see pp. 36, 39).

[105] S. Easterbrook, J. Singer, M.-A. Storey, and D. Damian, "Selecting Empirical Methods for Software Engineering," in *Guide to Advanced Empirical Software Engineering*, F. Shull, J. Singer, and D. Sjøberg, Eds., 2008, ch. 11, pp. 285–311 (see p. 16).

[106] *Eclipse*, `http://www.eclipse.org/luna/` (see pp. 42, 57).

[107] J. Edlund, J. Gustafson, M. Heldner, and A. Hjalmarsson, "Towards human-like spoken dialogue systems," *Speech Communication*, vol. 50, no. 8, pp. 630–645, 2008 (see p. 15).

[108] J. Edworthy, *Warning design: A research prospective*. CRC Press, 1996 (see p. 131).

[109] J. Edworthy and S. Dale, "Extending Knowledge of the Effects of Social Influence in Warning Compliance," *Proceedings of the Human Factors and Ergonomics Society Annual Meeting*, vol. 44, no. 25, pp. 107–110, 2000 (see p. 132).

[110] F. H. van Eemeren, B. Garssen, E. C. W. Krabbe, A. F. Snoeck Henkemans, B. Verheij, and J. H. M. Wagemans, *Handbook of Argumentation Theory*. Dordrecht: Springer Netherlands, 2014 (see pp. 105, 121).

[111] N. El Boustani and J. Hage, "Improving type error messages for generic java," in *Proceedings of the 2009 ACM SIGPLAN workshop on Partial evaluation and program manipulation - PEPM '09*, New York, New York, USA: ACM Press, 2008, p. 131 (see p. 39).

[112] F. Elbabour, O. Alhadreti, and P. Mayhew, "Eye Tracking in Retrospective Think-aloud Usability Testing: Is There Added Value?" *J. Usability Studies*, vol. 12, no. 3, pp. 95–110, 2017 (see p. 73).

[113] M. D. Ernst, "Static and dynamic analysis: Synergy and duality," in *WODA 2003: ICSE Workshop on Dynamic Analysis*, 2003, pp. 24–27 (see pp. 24, 25).

[114] (2017). Error Prone, [Online]. Available: `http://errorprone.info/` (visited on 01/01/2018) (see pp. 31, 272).

[115] G. Evangelidis, V. Dagdilelis, M. Satratzemi, and V. Efopoulos, "X-compiler: yet another integrated novice programming environment," in *Proceedings IEEE International Conference on Advanced Learning Technologies*, 2001, pp. 166–169 (see pp. 135, 139).

[116] M. Faddegon and O. Chitil, "Lightweight computation tree tracing for lazy functional languages," in *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '16, Santa Barbara, CA, USA: ACM, 2016, pp. 114–128 (see p. 206).

[117] E. A. Feigenbaum, B. G. Buchanan, and J. Lederberg, "On generality and problem solving: A case study using the dendral program," STANFORD UNIV CALIF DEPT OF COMPUTER SCIENCE, Tech. Rep., 1970 (see p. 132).

[118] R. Filik, K. Purdy, A. Gale, and D. Gerrett, "Labeling of Medicines and Patient Safety: Evaluating Methods of Reducing Drug Name Confusion," *Human Factors*, vol. 48, no. 1, pp. 39–47, 2006 (see p. 131).

[119] R. B. Findler, C. Flanagan, M. Flatt, S. Krishnamurthi, and M. Felleisen, "DrScheme: A pedagogic programming environment for scheme," in *Programming Languages: Implementations, Logics, and Programs (PLILP '97)*, H. Glaser, P. Hartel, and H. Kuchen, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 1997, pp. 369–388 (see p. 135).

[120] G. Fischer, T. Mastaglio, B. Reeves, and J. Rieman, "Minimalist explanations in knowledge-based systems," in *Twenty-Third Annual Hawaii International Conference on System Sciences*, vol. iii, 1990, 309–317 vol.3 (see p. 133).

[121] C. Flanagan, M. Flatt, S. Krishnamurthi, S. Weirich, and M. Felleisen, "Catching bugs in the web of program invariants," in *Proceedings of the ACM SIGPLAN 1996 Conference on Programming Language Design and Implementation*, ser. PLDI '96, Philadelphia, Pennsylvania, USA: ACM, 1996, pp. 23–32 (see pp. 44, 123, 204).

[122] S. D. Fleming, C. Scaffidi, D. Piorkowski, M. Burnett, R. Bellamy, J. Lawrance, and I. Kwan, "An Information Foraging Theory Perspective on Tools for Debugging, Refactoring, and Reuse Tasks," *ACM Transactions on Software Engineering and Methodology*, vol. 22, no. 2, pp. 1–41, 2013 (see p. 130).

[123] T. Flowers, C. A. Carver, and J. Jackson, "Empowering students and building confidence in novice programmers through Gauntlet," in *34th Annual Frontiers in Education, 2004. FIE 2004.*, 2004, T3H/10–T3H/13 Vol. 1 (see p. 139).

[124] B. J. Fogg, "Persuasive Technology: Using Computers to Change What We Think and Do," *Ubiquity*, vol. 2002, no. December, 2002 (see p. 132).

[125] M. Fowler, *Refactoring: Improving the Design of Existing Code*. 1999, p. 464 (see p. 217).

[126] S. N. Freund and E. S. Roberts, "Thetis: An ANSI C Programming Environment Designed for Introductory Use," in *Proceedings of the Twenty-seventh SIGCSE Technical Symposium on Computer Science Education*, ser. SIGCSE '96, New York, NY, USA: ACM, 1996, pp. 300–304 (see pp. 135, 139).

[127] M. Frické, *Logic and the Organization of Information*. New York, NY: Springer New York, 2012 (see p. 212).

[128] S. Garcia, D. Jeon, C. M. Louie, and M. B. Taylor, "Kremlin: Rethinking and rebooting gprof for the multicore age," in *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '11, San Jose, California, USA: ACM, 2011, pp. 458–469 (see p. 205).

[129] H. Gast, "Explaining ML Type Errors by Data Flows BT - Implementation and Application of Functional Languages: 16th International Workshop, IFL 2004, Lübeck, Germany, September 8-10, 2004 Revised Selected Papers," in, C. Grelck, F. Huch, G. J. Michaelson, and P. Trinder, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 72–89 (see p. 39).

[130] *GazePoint*, `http://www.gazept.com/` (see p. 57).

[131] X. Ge and E. Murphy-Hill, "Manual refactoring changes with automated refactoring validation," in *Proceedings of the 36th International Conference on Software Engineering - ICSE 2014*, New York, New York, USA: ACM Press, 2014, pp. 1095–1105 (see p. 135).

[132] L. M. Given, Ed., *The Sage Encyclopedia of Qualitative Research Methods*. 2008, p. 1043 (see p. 16).

[133] A. Gomolka and B. Humm, "Structure Editors: Old Hat or Future Vision?" In *Evaluation of Novel Approaches to Software Engineering: 6th International Conference*, L. A. Maciaszek and K. Zhang, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 82–97 (see p. 134).

[134] Google. (). Google c++ style guide, [Online]. Available: `https://google.github.io/styleguide/cppguide.html` (visited on 01/01/2018) (see pp. 137, 138).

[135] A. Gosain and G. Sharma, "A survey of dynamic program analysis techniques and tools," in *Proceedings of the 3rd International Conference on Frontiers of Intelligent Computing: Theory and Applications (FICTA) 2014: Volume 1*, S. C. Satapathy, B. N. Biswal, S. K. Udgata, and J. Mandal, Eds. Cham: Springer International Publishing, 2015, pp. 113–122 (see pp. 24, 25).

[136] ——, “Static analysis: A survey of techniques and tools,” in *Intelligent Computing and Applications: Proceedings of the International Conference on ICA, 22-24 December 2014*, D. Mandal, R. Kar, S. Das, and B. K. Panigrahi, Eds. New Delhi: Springer India, 2015, pp. 581–591 (see p. 25).

[137] M. J. Grant and A. Booth, “A typology of reviews: an analysis of 14 review types and associated methodologies,” *Health Information & Libraries Journal*, vol. 26, no. 2, pp. 91–108, 2009 (see p. 198).

[138] T. Green and M. Petre, “Usability analysis of visual programming environments: A ‘Cognitive Dimensions’ framework,” *Journal of Visual Languages & Computing*, vol. 7, no. 2, pp. 131–174, 1996 (see p. 91).

[139] D. Greenaway, J. Lim, J. Andronick, and G. Klein, “Don’t sweat the small stuff: Formal verification of C code without the pain,” in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI ’14, Edinburgh, United Kingdom: ACM, 2014, pp. 429–439 (see p. 200).

[140] S. Gregor and I. Benbasat, “Explanations from Intelligent Systems: Theoretical Foundations and Implications for Practice,” *MIS Quarterly*, vol. 23, no. 4, pp. 497–530, 1999 (see p. 132).

[141] A. Groce and W. Visser, “What Went Wrong: Explaining Counterexamples,” in *Model Checking Software*, T. Ball and S. K. Rajamani, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 121–136 (see p. 41).

[142] L. Gugerty and G. Olson, “Debugging by skilled and novice programmers,” in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, ser. CHI ’86, Boston, Massachusetts, USA: ACM, 1986, pp. 171–174 (see p. 138).

[143] A. Gurfinkel and M. Chechik, “Proof-Like Counter-Examples,” in *Tools and Algorithms for the Construction and Analysis of Systems*, H. Garavel and J. Hatcliff, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 160–175 (see p. 42).

[144] J. Ha, C. J. Rossbach, J. V. Davis, I. Roy, H. E. Ramadan, D. E. Porter, D. L. Chen, and E. Witchel, “Improved error reporting for software that uses black-box components,” in *Proceedings of the 28th ACM SIGPLAN Conference on*

Programming Language Design and Implementation, ser. PLDI '07, San Diego, California, USA: ACM, 2007, pp. 101–111 (see p. 201).

[145] M. van den Haak, M. De Jong, and P. Jan Schellens, "Retrospective vs. concurrent think-aloud protocols: Testing the usability of an online library catalogue," *Behaviour & Information Technology*, vol. 22, no. 5, pp. 339–351, 2003 (see p. 61).

[146] K. Hammond and V. J. Rayward-Smith, "A survey on syntactic error recovery and repair," *Computer Languages*, vol. 9, no. 1, pp. 51–67, 1984 (see p. 30).

[147] S. Hanenberg, "Faith, hope, and love: An essay on software science's neglect of human factors," in *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, ser. OOPSLA '10, Reno/Tahoe, Nevada, USA: ACM, 2010, pp. 933–946 (see pp. 196, 207).

[148] M. Harbach, S. Fahl, P. Yakovleva, and M. Smith, "Sorry, I Don't Get It: An Analysis of Warning Message Texts," in *Financial Cryptography and Data Security*, A. A. Adams, M. Brenner, and M. Smith, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 94–111 (see pp. 131, 132).

[149] C. Hardaker, "Trolling in asynchronous computer-mediated communication: From user discussions to academic definitions," *Journal of Politeness Research. Language, Behaviour, Culture*, vol. 6, no. 2, pp. 215–242, 2010 (see p. 112).

[150] W. A. Harrell, "Effect of Two Warning Signs on Adult Supervision and Risky Activities by Children in Grocery Shopping Carts," *Psychological Reports*, vol. 92, no. 3, pp. 889–898, 2003 (see p. 132).

[151] L. R. Harris and M. Jenkin, "Vision and Attention," in *Vision and Attention*, Springer New York, 2001, pp. 1–17 (see p. 62).

[152] B. Hartmann, D. MacDougall, J. Brandt, and S. R. Klemmer, "What would other programmers do," in *CHI '10*, 2010, pp. 1019–1028 (see pp. 35, 100, 211).

[153] R. W. Hasker, "HiC: A C++ Compiler for CS1," *J. Comput. Sci. Coll.*, vol. 18, no. 1, pp. 56–64, 2002 (see p. 135).

[154] A. Head, C. Appachu, M. A. Hearst, and B. Hartmann, "Tutorons: Generating context-relevant, on-demand explanations and demonstrations of online code," in *2015 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, IEEE, 2015, pp. 3–12 (see p. 35).

[155] B. Heeren, "Top quality type error messages," PhD, Universiteit Utrecht, 2005 (see p. 39).

[156] B. Heeren, J. Hage, and S. D. Swierstra, "Scripting the type inference process," in *Proceedings of the Eighth ACM SIGPLAN International Conference on Functional Programming*, ser. ICFP '03, Uppsala, Sweden: ACM, 2003, pp. 3–13 (see p. 39).

[157] B. Heeren, D. Leijen, and A. van IJzendoorn, "Helium, for learning haskell," in *Proceedings of the 2003 ACM SIGPLAN Workshop on Haskell*, ser. Haskell '03, Uppsala, Sweden: ACM, 2003, pp. 62–71 (see pp. 38, 137, 275).

[158] P. Hejmady and N. H. Narayanan, "Visual attention patterns during program debugging with an IDE," in *ETRA*, Santa Barbara, California, 2012, pp. 197–200 (see p. 74).

[159] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre, "Software Verification with BLAST," in *Model Checking Software*, T. Ball and S. K. Rajamani, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 235–239 (see p. 40).

[160] A. Hindle, E. T. Barr, Z. Su, M. Gabel, and P. Devanbu, "On the naturalness of software," in *2012 34th International Conference on Software Engineering (ICSE)*, 2012, pp. 837–847 (see p. 134).

[161] K. J. Hoffman, P. Eugster, and S. Jagannathan, "Semantics-aware trace analysis," in *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '09, Dublin, Ireland: ACM, 2009, pp. 453–464 (see p. 201).

[162] R. C. Holt, D. B. Wortman, D. T. Barnard, and J. R. Cordy, "SP/K: A System for Teaching Computer Programming," *Commun. ACM*, vol. 20, no. 5, pp. 301–309, 1977 (see p. 135).

[163] G. J. Holzmann, "The model checker SPIN," *IEEE Transactions on Software Engineering*, vol. 23, no. 5, pp. 279–295, 1997 (see p. 40).

[164] H. Horacek, "A Model for Adapting Explanations to the User's Likely Inferences," *User Modeling and User-Adapted Interaction*, vol. 7, no. 1, pp. 1–55, 1997 (see p. 133).

[165] J. J. Horning, "What the compiler should tell the user," in *Compiler Construction: An Advanced Course*, ser. Lecture Notes in Computer Science, vol. 21, Berlin, Heidelberg: Springer, 1974, pp. 525–548 (see pp. 46–48, 124, 264).

[166] S. Horwitz, "Identifying the semantic and textual differences between two versions of a program," in *Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation*, ser. PLDI '90, White Plains, New York, USA: ACM, 1990, pp. 234–245 (see p. 202).

[167] Y. Hoskote, T. Kam, P.-H. Ho, and X. Zhao, "Coverage estimation for symbolic model checking," in *Proceedings 1999 Design Automation Conference (Cat. No. 99CH36361)*, 1999, pp. 300–305 (see p. 42).

[168] M. Hristova, A. Misra, M. Rutter, and R. Mercuri, "Identifying and correcting Java programming errors for introductory computer science students," *ACM SIGCSE Bulletin*, vol. 35, no. 1, p. 153, 2003 (see pp. 139, 211).

[169] (2018). Infer static analyzer, [Online]. Available: `http://fbinfer.com/` (visited on 01/01/2018) (see p. 271).

[170] *IntelliJ*, `https://www.jetbrains.com` (see pp. 42, 57).

[171] C. Isradisaikul and A. C. Myers, "Finding counterexamples from parsing conflicts," in *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '15, Portland, OR, USA: ACM, 2015, pp. 555–564 (see p. 202).

[172] J. Jackson, M. Cobb, and C. Carver, "Identifying top Java errors for novice programmers," in *Proceedings Frontiers in Education 35th Annual Conference*, IEEE, 2005, T4C–24–T4C–27 (see pp. 136, 210).

[173] J. Z. Jacobson and P. Dodwell, "Saccadic eye movements during reading," *Brain and Language*, vol. 8, no. 3, pp. 303–314, 1979 (see pp. 63, 69).

[174] M. C. Jadud, "A First Look at Novice Compilation Behaviour Using BlueJ," *Computer Science Education*, vol. 15, no. 1, pp. 25–40, 2005 (see pp. 135, 136).

[175] L. S. Jaynes and D. B. Boles, "The Effect of Symbols on Warning Compliance," *Proceedings of the Human Factors Society Annual Meeting*, vol. 34, no. 14, pp. 984–987, 1990 (see p. 132).

[176] C. L. Jeffery, "Generating LR syntax error messages from examples," *ACM Transactions on Programming Languages and Systems*, vol. 25, no. 5, pp. 631–640, 2003 (see pp. 31, 100).

[177] R. Jhala and R. Majumdar, "Software Model Checking," *ACM Comput. Surv.*, vol. 41, no. 4, 21:1–21:54, 2009 (see p. 40).

[178] G. Jin, L. Song, W. Zhang, S. Lu, and B. Liblit, "Automated atomicity-violation fixing," in *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '11, San Jose, California, USA: ACM, 2011, pp. 389–400 (see p. 200).

[179] A. Johnson, L. Waye, S. Moore, and S. Chong, "Exploring and enforcing security guarantees via program dependence graphs," in *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '15, Portland, OR, USA: ACM, 2015, pp. 291–302 (see p. 206).

[180] B. Johnson, "A Tool (Mis)communication Theory and Adaptive Approach for Supporting Developer Tool Use," PhD thesis, North Carolina State University, 2017 (see p. 45).

[181] B. Johnson, R. Pandita, E. Murphy-Hill, and S. Heckman, "Bespoke tools: Adapted to the concepts developers know," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2015, Bergamo, Italy: ACM, 2015, pp. 878–881 (see p. 133).

[182] B. Johnson, R. Pandita, J. Smith, D. Ford, S. Elder, E. Murphy-Hill, S. Heckman, and C. Sadowski, "A cross-tool communication study on program analysis tool notifications," in *FSE*, 2016, pp. 73–84 (see pp. 63, 125).

[183] B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge, "Why don't software developers use static analysis tools to find bugs?" English, in *2013*

35th International Conference on Software Engineering (ICSE), IEEE, 2013, pp. 672–681 (see pp. 31, 50, 103, 108, 125, 196).

[184] H. Johnson and P. Johnson, "Explanation Facilities and Interactive Systems," in *Proceedings of the 1st International Conference on Intelligent User Interfaces*, ser. IUI '93, New York, NY, USA: ACM, 1993, pp. 159–166 (see p. 133).

[185] R. B. Johnson and A. J. Onwuegbuzie, "Mixed Methods Research: A Research Paradigm Whose Time Has Come," *Educational Researcher*, vol. 33, no. 7, pp. 14–26, 2004 (see p. 16).

[186] M. Jose and R. Majumdar, "Cause clue clauses: Error localization using maximum satisfiability," in *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '11, San Jose, California, USA: ACM, 2011, pp. 437–446 (see p. 204).

[187] J. M. Joyce, "Kullback-leibler divergence," in *International Encyclopedia of Statistical Science*, M. Lovric, Ed. Springer Berlin Heidelberg, 2011, pp. 720–722 (see p. 62).

[188] E. Kantorowitz and H. Laor, "Automatic generation of useful syntax error messages," *Software: Practice and Experience*, vol. 16, no. 7, pp. 627–640, 1986 (see pp. 47, 48, 100, 124, 266).

[189] C. Kelleher and R. Pausch, "Lowering the Barriers to Programming: A Taxonomy of Programming Environments and Languages for Novice Programmers," *ACM Comput. Surv.*, vol. 37, no. 2, pp. 83–137, 2005 (see p. 135).

[190] A. Kent and B. Kent. (). Isomof, [Online]. Available: `https://isomorf.io/` (visited on 01/01/2018) (see p. 134).

[191] Y. P. Khoo, J. S. Foster, M. Hicks, and V. Sazawal, "Path projection for user-centered static analysis tools," in *PASTE*, 2008, p. 57 (see p. 123).

[192] A. A. Khwaja and J. E. Urban, "Syntax-directed Editing Environments: Issues and Features," in *Proceedings of the 1993 ACM/SIGAPP Symposium on Applied Computing: States of the Art and Practice*, ser. SAC '93, New York, NY, USA: ACM, 1993, pp. 230–237 (see p. 134).

[193] S. Kim and M. S. Wogalter, “Habituation, Dishabituation, and Recovery Effects in Visual Warnings,” *Proceedings of the Human Factors and Ergonomics Society Annual Meeting*, vol. 53, no. 20, pp. 1612–1616, 2009 (see p. 132).

[194] B. Kitchenham, “Procedures for performing systematic reviews,” *Keele, UK, Keele University*, vol. 33, no. 2004, pp. 1–26, 2004 (see p. 197).

[195] P. B. Kline, C. C. Braun, N. Peterson, and N. C. Silver, “The Impact of Color on Warnings Research,” *Proceedings of the Human Factors and Ergonomics Society Annual Meeting*, vol. 37, no. 14, pp. 940–944, 1993 (see p. 131).

[196] A. Ko and B. Myers, “Development and evaluation of a model of programming errors,” in *IEEE Symposium on Human Centric Computing Languages and Environments*, IEEE, 2003, pp. 7–14 (see p. 125).

[197] A. J. Ko and B. A. Myers, “A framework and methodology for studying the causes of software errors in programming systems,” *Journal of Visual Languages & Computing*, vol. 16, no. 1, pp. 41–84, 2005 (see pp. 77, 80).

[198] ——, “Finding causes of program output with the Java Whyline,” in *Proceedings of the 27th international conference on Human factors in computing systems - CHI 09*, New York, New York, USA: ACM Press, 2009, pp. 1569–1578 (see p. 44).

[199] J. Koenemann and S. P. Robertson, “Expert Problem Solving Strategies for Program Comprehension,” in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, ser. CHI ’91, New York, NY, USA: ACM, 1991, pp. 125–130 (see p. 128).

[200] N. Kulkarni and V. Varma, “Supporting Comprehension of Unfamiliar Programs by Modeling an Expert’s Perception,” in *Proceedings of the 3rd International Workshop on Realizing Artificial Intelligence Synergies in Software Engineering*, ser. RAISE 2014, New York, NY, USA: ACM, 2014, pp. 19–24 (see p. 138).

[201] S. K. Kummerfeld and J. Kay, “The neglected battle fields of syntax errors,” pp. 105–111, 2003 (see p. 29).

[202] W. Landi and William, “Undecidability of static analysis,” *ACM Letters on Programming Languages and Systems*, vol. 1, no. 4, pp. 323–337, 1992 (see p. 122).

[203] B. Lang, “On the usefulness of syntax directed editors,” in *Advanced Programming Environments: Proceedings of an International Workshop*, R. Conradi, T. M. Didriksen, and D. H. Wanvik, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 1986, pp. 47–51 (see p. 134).

[204] B. Lang, “Teaching New Programmers: A Java Tool Set As a Student Teaching Aid,” in *Proceedings of the Inaugural Conference on the Principles and Practice of Programming, 2002 and Proceedings of the Second Workshop on Intermediate Representation Engineering for Virtual Machines, 2002*, ser. PPPJ ’02/IRE ’02, Maynooth, County Kildare, Ireland, Ireland: National University of Ireland, 2002, pp. 95–100 (see p. 139).

[205] C. Lattner and V. Adve, “LLVM: A compilation framework for lifelong program analysis & transformation,” in *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, IEEE, 2004, pp. 75–86 (see p. 25).

[206] K. R. Laughery and M. S. Wogalter, “Designing Effective Warnings,” *Reviews of Human Factors and Ergonomics*, vol. 2, no. 1, pp. 241–271, 2006 (see p. 131).

[207] K. R. Laughery, S. L. Young, K. P. Vaubel, and J. W. Brelsford, “The Noticeability of Warnings on Alcoholic Beverage Containers,” *Journal of Public Policy & Marketing*, vol. 12, no. 1, pp. 38–56, 1993 (see p. 131).

[208] J. Lawrance, C. Bogart, M. Burnett, R. Bellamy, K. Rector, and S. D. Fleming, “How Programmers Debug, Revisited: An Information Foraging Theory Perspective,” *IEEE Transactions on Software Engineering*, vol. 39, no. 2, pp. 197–215, 2013 (see pp. 74, 130).

[209] M. R. Lehto and J. M. Miller, *Warnings. Volume I-Fundamentals, Design, and Evaluation Methodologies.* Fuller Technical Publications, Ann Arbor, Mich., 1986 (see p. 131).

[210] K. R. M. Leino, “Dafny: An automatic program verifier for functional correctness,” in *Logic for Programming, Artificial Intelligence, and Reasoning: 16th International Conference, LPAR-16, Dakar, Senegal, April 25–May 1, 2010, Revised Selected Papers*, E. M. Clarke and A. Voronkov, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 348–370 (see p. 271).

[211] B. S. Lerner, M. Flower, D. Grossman, and C. Chambers, “Searching for type-error messages,” in *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI ’07, San Diego, California, USA: ACM, 2007, pp. 425–434 (see p. 202).

[212] ——, “Searching for type-error messages,” *ACM SIGPLAN Notices*, vol. 42, no. 6, p. 425, 2007 (see p. 38).

[213] S. Letovsky and E. Soloway, “Delocalized Plans and Program Comprehension,” *IEEE Software*, vol. 3, no. 3, pp. 41–49, 1986 (see p. 129).

[214] S. Letovsky, “Cognitive processes in program comprehension,” *Journal of Systems and Software*, vol. 7, no. 4, pp. 325–339, 1987 (see p. 129).

[215] A. Leung, J. Sarracino, and S. Lerner, “Interactive parser synthesis by example,” in *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI ’15, Portland, OR, USA: ACM, 2015, pp. 565–574 (see p. 203).

[216] B. Liblit, A. Aiken, A. X. Zheng, and M. I. Jordan, “Bug isolation via remote program sampling,” in *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, ser. PLDI ’03, San Diego, California, USA: ACM, 2003, pp. 141–154 (see p. 201).

[217] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan, “Scalable statistical bug isolation,” in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI ’05, Chicago, IL, USA: ACM, 2005, pp. 15–26 (see p. 201).

[218] T. Lieber, J. R. Brandt, and R. C. Miller, “Addressing misconceptions about code with always-on programming visualizations,” in *CHI*, 2014, pp. 2481–2490 (see pp. 44, 123, 125).

[219] J. Liebowitz, *The handbook of applied expert systems*. cRc Press, 1997 (see p. 132).

[220] B. Y. Lim, A. K. Dey, and D. Avrahami, “Why and why not explanations improve the intelligibility of context-aware intelligent systems,” in *CHI ’09*, 2009, pp. 2119–2129 (see pp. 100, 132).

[221] C. Litecky, "An expert system for Cobol program debugging," *ACM SIGMIS Database*, vol. 20, no. 1, pp. 1–6, 1989 (see pp. 32, 211).

[222] B. Livshits, A. V. Nori, S. K. Rajamani, and A. Banerjee, "Merlin: Specification inference for explicit information flow problems," in *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '09, Dublin, Ireland: ACM, 2009, pp. 75–86 (see p. 206).

[223] (2017). Llvm: Expressive diagnostics, [Online]. Available: `http://clang.llvm.org/diagnostics.html` (visited on 01/01/2018) (see p. 29).

[224] F. Logozzo, S. K. Lahiri, M. Fähndrich, and S. Blackshear, "Verification modulo versions: Towards usable verification," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '14, Edinburgh, United Kingdom: ACM, 2014, pp. 294–304 (see p. 205).

[225] C. Loncaric, S. Chandra, C. Schlesinger, M. Sridharan, C. Loncaric, S. Chandra, C. Schlesinger, and M. Sridharan, "A practical framework for type inference error explanation," in *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications - OOPSLA 2016*, vol. 51, New York, New York, USA: ACM Press, 2016, pp. 781–799 (see p. 39).

[226] E. Lotem and Y. Chuchem. (2016). Lamdu: Towards a new programming experience, [Online]. Available: `http://www.lamdu.org/` (visited on 01/01/2018) (see p. 134).

[227] M. Madsen, B. Livshits, and M. Fanning, "Practical Static Analysis of JavaScript Applications in the Presence of Frameworks and Libraries," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2013, New York, NY, USA: ACM, 2013, pp. 499–509 (see p. 135).

[228] L. Mamykina, B. Manoim, M. Mittal, G. Hripcsak, and B. Hartmann, "Design lessons from the fastest Q&A site in the west," in *CHI*, 2011, pp. 2857–2866 (see p. 110).

[229] J.-Y. Mao and I. Benbasat, "The Use of Explanations in Knowledge-Based Systems: Cognitive Perspectives and a Process-Tracing Analysis," *Journal*

of Management Information Systems, vol. 17, no. 2, pp. 153–179, 2000 (see p. 133).

[230] G. Marceau, K. Fisler, and S. Krishnamurthi, "Measuring the effectiveness of error messages designed for novice programmers," in *SIGCSE*, 2011, pp. 499–504 (see pp. 50, 139).

[231] ——, "Mind your language: On novices' interactions with error messages," in *ONWARD*, 2011, pp. 3–17 (see pp. 125, 137, 139, 211).

[232] W. A. Martin and R. J. Fateman, "The macsyma system," in *Proceedings of the Second ACM Symposium on Symbolic and Algebraic Manipulation*, ser. SYMSAC '71, Los Angeles, California, USA: ACM, 1971, pp. 59–75 (see p. 132).

[233] N. D. Matsakis and F. S. Klock II, "The rust language," in *Proceedings of the 2014 ACM SIGAda Annual Conference on High Integrity Language Technology*, ser. HILT '14, Portland, Oregon, USA: ACM, 2014, pp. 103–104 (see p. 273).

[234] B. Matthews, R. Andronaco, and A. Adams, "Warning signs at beaches: Do they work?" *Safety Science*, vol. 62, pp. 312–318, 2014 (see p. 131).

[235] J. A. Maxwell, "Using numbers in qualitative research," *Qualitative Inquiry*, vol. 16, no. 6, pp. 475–482, 2010 (see p. 115).

[236] G. R. Mayes, "Argument-Explanation Complementarity and the Structure of Informal Reasoning," *Informal Logic*, vol. 30, no. 1, pp. 92–111, 2010 (see p. 14).

[237] A. von Mayrhauser and A. M. Vans, "From program comprehension to tool requirements for an industrial environment," in *[1993] IEEE Second Workshop on Program Comprehension*, 1993, pp. 78–86 (see p. 129).

[238] B. McAdam, "How to Repair Type Errors Automatically," in *Trends in Functional Programming, Volume 3*, 2002, ch. 8 (see pp. 38, 39).

[239] B. J. McAdam, "On the Unification of Substitutions in Type Inference," in *Implementation of Functional Languages*, K. Hammond, T. Davie, and C. Clack, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 137–152 (see pp. 38, 39).

[240] R. McCauley, S. Fitzgerald, G. Lewandowski, L. Murphy, B. Simon, L. Thomas, and C. Zander, "Debugging: a review of the literature from an educational perspective," *Computer Science Education*, vol. 18, no. 2, pp. 67–92, 2008 (see p. 138).

[241] K. R. McKeown, M. Wish, and K. Matthews, "Tailoring Explanations for the User," in *Proceedings of the 9th International Joint Conference on Artificial Intelligence - Volume 2*, ser. IJCAI'85, San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1985, pp. 794–798 (see p. 133).

[242] M. McLuhan and Q. Fiore, "The medium is the message," *New York*, vol. 123, pp. 126–128, 1967 (see p. 9).

[243] D. Menendez and S. Nagarakatte, "Alive-infer: Data-driven precondition inference for peephole optimizations in llvm," in *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '17, Barcelona, Spain: ACM, 2017, pp. 49–63 (see p. 202).

[244] V. O. Mittal and C. L. Paris, "Generating explanations in context: The system perspective," *Expert Systems with Applications*, vol. 8, no. 4, pp. 491–503, 1995 (see p. 133).

[245] R. Molich and J. Nielsen, "Improving a human-computer dialogue," *Communications of the ACM*, vol. 33, no. 3, pp. 338–348, 1990 (see p. 211).

[246] J. D. Moore, "What makes human explanations effective," in *Proceedings of the fifteenth annual conference of the Cognitive Science Society*, 1993, pp. 131–136 (see p. 15).

[247] E. M. Moreno, K. D. Federmeier, and M. Kutas, "Switching languages, Switching palabras (words): An electrophysiological study of code switching," *Brain and Language*, vol. 80, no. 2, pp. 188–207, 2002 (see p. 70).

[248] D. L. Morgan, "Paradigms Lost and Pragmatism Regained: Methodological Implications of Combining Qualitative and Quantitative Methods," *Journal of Mixed Methods Research*, vol. 1, no. 1, pp. 48–76, 2007 (see p. 16).

[249] D. G. Morrow, C. M. Hier, W. E. Menard, and V. O. Leirer, "Icons improve older and younger adults' comprehension of medication information," *The Journals of Gerontology Series B: Psychological Sciences and Social Sciences*, vol. 53, no. 4, P240–P254, 1998 (see p. 131).

[250] M. A. A. Mosleh, M. A. Alhussein, M. S. Baba, S. Malek, and S. ab Hamid, "Reviewing and Classification of Software Model Checking Tools," in *Advanced Computer and Communication Engineering Technology*, H. A. Sulaiman, M. A. Othman, M. F. I. Othman, Y. A. Rahim, and N. C. Pee, Eds., Cham: Springer International Publishing, 2016, pp. 279–294 (see p. 40).

[251] P. G. Moulton and M. E. Muller, "DITRAN—a compiler emphasizing diagnostics," *Communications of the ACM*, vol. 10, no. 1, pp. 45–52, 1967 (see pp. 46, 124, 135, 211, 264).

[252] F. Mulder. (2016). Awesome error messages for Dotty, [Online]. Available: `https://www.scala-lang.org/blog/2016/10/14/dotty-errors.html` (visited on 01/01/2018) (see pp. 31, 33).

[253] ——, (2016). Shape of errors to come, [Online]. Available: `https://blog.rust-lang.org/2016/08/10/Shape-of-errors-to-come.html` (visited on 01/01/2018) (see pp. 31, 34).

[254] E. Murphy-Hill, T. Barik, and A. P. Black, "Interactive ambient visualizations for soft advice," *Information Visualization*, vol. 12, no. 2, pp. 107–132, 2013 (see pp. 45, 48, 267).

[255] E. Murphy-Hill and A. P. Black, "Programmer-friendly refactoring errors," *IEEE Transactions on Software Engineering*, vol. 38, no. 6, pp. 1417–1431, 2012 (see pp. 44, 48, 267).

[256] K. Muşlu, Y. Brun, R. Holmes, M. D. Ernst, and D. Notkin, "Speculative analysis of integrated development environment recommendations," in *OOPSLA '12*, 2012, pp. 669–682 (see pp. 210, 211).

[257] N. J. D. Nagelkerke, "A note on a general definition of the coefficient of determination," *Biometrika*, vol. 78, no. 3, pp. 691–692, 1991 (see p. 63).

[258] S. Narayanasamy, Z. Wang, J. Tigani, A. Edwards, and B. Calder, "Automatically classifying benign and harmful data races using replay analysis," in *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '07, San Diego, California, USA: ACM, 2007, pp. 22–31 (see p. 201).

[259] S. M. Nasehi, J. Sillito, F. Maurer, and C. 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), 2012, pp. 25–34 (see p. 124).

[260] C. Nass, J. Steuer, and E. R. Tauber, "Computers Are Social Actors," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, ser. CHI '94, New York, NY, USA: ACM, 1994, pp. 72–78 (see p. 15).

[261] National Cryptologic School, "Lesson Four," Ft. George G. Meade, Maryland, Tech. Rep., 1990 (see pp. 47, 267).

[262] *NCrunch: Concurrent testing tool for Visual Studio*, `http://www.ncrunch.net/` (see p. 42).

[263] T. Nelson, N. Danas, D. J. Dougherty, and S. Krishnamurthi, "The Power of "Why" and "Why Not": Enriching Scenario Exploration with Provenance," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2017, New York, NY, USA: ACM, 2017, pp. 106–116 (see p. 41).

[264] T. Nelson, S. Saghafi, D. J. Dougherty, K. Fisler, and S. Krishnamurthi, "Aluminum: Principled scenario exploration through minimality," in *Proceedings of the 2013 International Conference on Software Engineering*, ser. ICSE '13, San Francisco, CA, USA: IEEE Press, 2013, pp. 232–241 (see p. 41).

[265] P. C. Nguyen and D. Van Horn, "Relatively complete counterexamples for higher-order programs," in *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '15, Portland, OR, USA: ACM, 2015, pp. 446–456 (see p. 202).

[266] J. Nielsen, "Heuristic evaluation," *Usability inspection methods*, vol. 17, no. 1, pp. 25–62, 1994 (see p. 124).

[267] J. Nielsen and R. Molich, "Heuristic evaluation of user interfaces," ser. CHI '90, 1990, pp. 249–256 (see p. 91).

[268] M.-H. Nienaltowski, M. Pedroni, and B. Meyer, "Compiler error messages: What can help novices?" In *SIGCSE '08*, 2008, pp. 168–172 (see pp. 100, 108, 139).

[269] D. A. Norman, “Design principles for human-computer interfaces,” in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, ser. CHI ’83, Boston, Massachusetts, USA: ACM, 1983, pp. 1–10 (see p. 142).

[270] P. Ohmann, A. Brooks, L. D’Antoni, and B. Liblit, “Control-flow recovery from partial failure reports,” in *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI ’17, Barcelona, Spain: ACM, 2017, pp. 390–405 (see p. 206).

[271] C. Omar, I. Voysey, M. Hilton, J. Sunshine, C. L. Goues, J. Aldrich, and M. A. Hammer, “Toward Semantic Foundations for Program Editors,” in *2nd Summit on Advances in Programming Languages (SNAPL 2017)*, B. S. Lerner, R. Bodík, and S. Krishnamurthi, Eds., ser. Leibniz International Proceedings in Informatics (LIPIcs), vol. 71, Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2017, 11:1–11:12 (see p. 135).

[272] *OpenCV*, `http://opencv.org/` (see p. 59).

[273] S. Packowski, “A lightweight and flexible process for designing intuitive error handling and effective error messages,” in *CASCON ’09*, 2009, pp. 149–163 (see p. 211).

[274] O. Padon, K. L. McMillan, A. Panda, M. Sagiv, and S. Shoham, “Ivy: Safety verification by interactive generalization,” in *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI ’16, Santa Barbara, CA, USA: ACM, 2016, pp. 614–630 (see p. 202).

[275] C. L. Paris, “Tailoring Object Descriptions to a User’s Level of Expertise,” *Comput. Linguist.*, vol. 14, no. 3, pp. 64–78, 1988 (see p. 133).

[276] D. H. Park, H. K. Kim, I. Y. Choi, and J. K. Kim, “A literature review and classification of recommender systems research,” *Expert Systems with Applications*, vol. 39, no. 11, pp. 10 059–10 072, 2012 (see p. 133).

[277] D. L. Parnas and P. C. Clements, “A rational design process: How and why to fake it,” *IEEE Transactions on Software Engineering*, vol. SE-12, no. 2, pp. 251–257, 1986 (see p. 15).

[278] C. Parnin, “Subvocalization - Toward hearing the inner thoughts of developers,” in *ICPC ’11*, 2011, pp. 197–200 (see p. 77).

[279] C. Parnin and S. Rugaber, “Programmer information needs after memory failure,” in *2012 20th IEEE International Conference on Program Comprehension (ICPC)*, IEEE, 2012, pp. 123–132 (see p. 128).

[280] S. O. Parsons, J. L. Seminara, and M. S. Wogalter, “A Summary of Warnings Research,” *Ergonomics in Design*, vol. 7, no. 1, pp. 21–31, 1999 (see p. 131).

[281] A. Pears, S. Seidman, L. Malmi, L. Mannila, E. Adams, J. Bennedsen, M. Devlin, and J. Paterson, “A survey of literature on the teaching of introductory programming,” in *Working Group Reports on ITiCSE on Innovation and Technology in Computer Science Education*, ser. ITiCSE-WGR ’07, New York, NY, USA: ACM, 2007, pp. 204–223 (see p. 139).

[282] N. Pennington, “Comprehension strategies in programming,” in *Empirical studies of programmers: second workshop*, Ablex Publishing Corp., 1987, pp. 100–113 (see p. 129).

[283] D. Perelman, S. Gulwani, T. Ball, and D. Grossman, “Type-directed completion of partial expressions,” in *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI ’12, Beijing, China: ACM, 2012, pp. 275–286 (see p. 205).

[284] R. H. Perrott, “Syntax-directed editing,” *Software Engineering Journal*, vol. 3, no. 2, 37–46(9), 1988 (see p. 134).

[285] T. Peters. (2014). Pep 20: The zen of python, [Online]. Available: `https://www.python.org/dev/peps/pep-0020/` (see p. 269).

[286] R. S. Pettit, J. Homer, and R. Gee, “Do Enhanced Compiler Error Messages Help Students?: Results Inconclusive.,” in *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education*, ser. SIGCSE ’17, New York, NY, USA: ACM, 2017, pp. 465–470 (see pp. 138, 139).

[287] B. C. Pierce and D. N. Turner, “Local type inference,” *ACM Trans. Program. Lang. Syst.*, vol. 22, no. 1, pp. 1–44, 2000 (see p. 36).

[288] H. Pieterse and H. Gelderblom, “Guidelines for Error Message Design,” *International Journal of Technology and Human Interaction (IJTHI)*, vol. 14, no. 1, pp. 80–98, 2018 (see p. 131).

[289] N. Pinkwart, K. Ashley, C. Lynch, and V. Aleven, "Evaluating an intelligent tutoring system for making legal arguments with hypotheticals," *Int. J. Artif. Intell. Ed.*, vol. 19, no. 4, pp. 401–424, 2009 (see p. 105).

[290] D. Piorkowski, S. D. Fleming, C. Scaffidi, M. Burnett, I. Kwan, A. Z. Henley, J. Macbeth, C. Hill, and A. Horvath, "To fix or to learn? How production bias affects developers' information foraging during debugging," in *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2015, pp. 11–20 (see p. 138).

[291] D. F. Polit and C. T. Beck, "Generalization in quantitative and qualitative research: Myths and strategies," *International Journal of Nursing Studies*, vol. 47, no. 11, pp. 1451–1458, 2010 (see p. 123).

[292] J. Pombrio and S. Krishnamurthi, "Resugaring: Lifting evaluation sequences through syntactic sugar," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '14, Edinburgh, United Kingdom: ACM, 2014, pp. 361–371 (see p. 200).

[293] J. Ponterotto, "Brief note on the origins, evolution, and meaning of the qualitative research concept thick description," *The Qualitative Report*, vol. 11, no. 3, 2006 (see p. 116).

[294] L. Ponzanelli, A. Bacchelli, and M. Lanza, "Seahawk: Stack Overflow in the IDE," in *2013 35th International Conference on Software Engineering (ICSE)*, 2013, pp. 1295–1298 (see p. 35).

[295] L. Ponzanelli, G. Bavota, M. Di Penta, R. Oliveto, and M. Lanza, "Mining StackOverflow to Turn the IDE into a Self-confident Programming Prompter," in *Proceedings of the 11th Working Conference on Mining Software Repositories*, ser. MSR 2014, New York, NY, USA: ACM, 2014, pp. 102–111 (see p. 35).

[296] F. Pottier, "Reachability and Error Diagnosis in LR(1) Parsers," in *Proceedings of the 25th International Conference on Compiler Construction*, ser. CC 2016, New York, NY, USA: ACM, 2016, pp. 88–98 (see p. 31).

[297] J. Prather, R. Pettit, K. H. McMurry, A. Peters, J. Homer, N. Simone, and M. Cohen, "On Novices' Interaction with Compiler Error Messages: A Human Factors Approach," in *Proceedings of the 2017 ACM Conference on Interna-*

tional Computing Education Research, ser. ICER '17, New York, NY, USA: ACM, 2017, pp. 74–82 (see p. 139).

[298] D. Pritchard, "Frequency distribution of error messages," in *PLATEAU*, 2015, pp. 1–8 (see pp. 46, 50, 72, 136).

[299] F. Puppe, Ed., *Systematic Introduction to Expert Systems: Knowledge Representations and Problem-Solving Methods*, 2. Springer-Verlag, 1993, vol. 8 (see p. 132).

[300] Y. Qian and J. Lehman, "Students' Misconceptions and Other Difficulties in Introductory Programming: A Literature Review," *ACM Trans. Comput. Educ.*, vol. 18, no. 1, 1:1–1:24, 2017 (see p. 139).

[301] X. Qiu, P. Garg, A. Ştefănescu, and P. Madhusudan, "Natural proofs for structure, data, and separation," in *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '13, Seattle, Washington, USA: ACM, 2013, pp. 231–242 (see p. 200).

[302] *Racket*, `https://racket-lang.org/` (see p. 44).

[303] V. Rahli, J. Wells, J. Pirie, and F. Kamareddine, "Skalpel: A Type Error Slicer for Standard ML," *Electronic Notes in Theoretical Computer Science*, vol. 312, pp. 197–213, 2015 (see p. 39).

[304] M. M. Rahman and C. K. Roy, "SurfClipse: Context-Aware Meta-search in the IDE," in *2014 IEEE International Conference on Software Maintenance and Evolution*, 2014, pp. 617–620 (see p. 35).

[305] V. Rajlich and N. Wilde, "The role of concepts in program comprehension," in *Proceedings 10th International Workshop on Program Comprehension*, 2002, pp. 271–278 (see p. 128).

[306] J. W. Ratcliff and D. E. Metzener, "Pattern matching: The Gestalt approach," *Dr Dobbs Journal*, vol. 13, no. 7, p. 46, 1988 (see p. 59).

[307] K. Rayner, "Eye movements in reading and information processing: 20 years of research.," *Psychological Bulletin*, vol. 124, no. 3, pp. 372–422, 1998 (see pp. 62, 66).

[308] J. Regehr, Y. Chen, P. Cuoq, E. Eide, C. Ellison, and X. Yang, "Test-case reduction for c compiler bugs," in *Proceedings of the 33rd ACM SIGPLAN Con-*

ference on Programming Language Design and Implementation, ser. PLDI '12, Beijing, China: ACM, 2012, pp. 335–346 (see p. 202).

[309] G. Riley, "Clips: An expert system building tool," 1991 (see p. 132).

[310] C. C. Risley and T. J. Smedley, "Visualization of compile time errors in a Java compatible visual language," in *Proceedings. 1998 IEEE Symposium on Visual Languages (Cat. No.98TB100254)*, 1998, pp. 22–29 (see p. 45).

[311] R. S. Rist and colleagues, "Plans in programming: Definition, demonstration, and development," in *first workshop on empirical studies of programmers on Empirical studies of programmers*, 1986, pp. 28–47 (see p. 129).

[312] E. Roberts, "An overview of MiniJava," in *SIGCSE '01*, vol. 33, 2001, pp. 1–5 (see pp. 101, 137).

[313] M. Robillard, R. Walker, and T. Zimmermann, "Recommendation Systems for Software Engineering," *IEEE Software*, vol. 27, no. 4, pp. 80–86, 2010 (see p. 133).

[314] P. Rodeghero, C. McMillan, P. W. McBurney, N. Bosch, and S. D'Mello, "Improving automated source code summarization via an eye-tracking study of programmers," in *ICSE*, 2014, pp. 390–401 (see p. 73).

[315] W. A. Rogers, N. Lamson, and G. K. Rousseau, "Warning Research: An Integrative Perspective," *Human Factors*, vol. 42, no. 1, pp. 102–139, 2000 (see p. 131).

[316] P. Romero, R. Cox, B. du Boulay, and R. Lutz, "Visual attention and representation switching during Java program debugging: A study using the restricted focus viewer," in *Diagrammatic Representation and Inference: Second International Conference, Diagrams 2002 Callaway Gardens, GA, USA, April 18–20, 2002 Proceedings*, M. Hegarty, B. Meyer, and N. H. Narayanan, Eds. 2002, pp. 221–235 (see p. 73).

[317] C. Rubio-González, H. S. Gunawi, B. Liblit, R. H. Arpaci-Dusseau, and A. C. Arpaci-Dusseau, "Error propagation analysis for file systems," in *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '09, Dublin, Ireland: ACM, 2009, pp. 270–280 (see p. 206).

[318] *Rust Enhanced*, https://github.com/rust-lang/rust-enhanced (see p. 44).

[319] C. Sadowski, J. van Gogh, C. Jaspan, E. Soderberg, and C. Winter, "Tricorder: Building a Program Analysis Ecosystem," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, IEEE, 2015, pp. 598–608 (see pp. 24, 48, 124, 268).

[320] S. Saghafi, R. Danas, and D. J. Dougherty, "Exploring Theories with a Model-Finding Assistant," in *Automated Deduction - CADE-25*, A. P. Felty and A. Middeldorp, Eds., Cham: Springer International Publishing, 2015, pp. 434–449 (see p. 41).

[321] J. Saldaña, *The Coding Manual for Qualitative Researchers*. SAGE Publications, 2009 (see p. 115).

[322] I. Salman, A. T. Misirli, and N. Juristo, "Are students representatives of professionals in software engineering experiments?" In *ICSE*, 2015, pp. 666–676 (see p. 73).

[323] E. A. Santos, J. C. Campbell, D. Patel, A. Hindle, and J. N. Amaral, "Syntax and sensibility: Using language models to detect and correct syntax errors," in *25th IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER 2018)*, (25th IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER 2018)), Campobasso, Italy, 21, 2018, pp. 1–11 (see p. 30).

[324] M. Satratzemi, S. Xinogalos, and V. Dagdilelis, "An environment for teaching object-oriented programming: objectKarel," in *Proceedings 3rd IEEE International Conference on Advanced Technologies*, 2003, pp. 342–343 (see p. 139).

[325] J. Scholtz and S. Wiedenbeck, "Learning a new programming language: a model of the planning process," in *Proceedings of the Twenty-Fourth Annual Hawaii International Conference on System Sciences*, vol. ii, 1991, 3–12 vol.2 (see p. 138).

[326] T. Schorsch, Tom, Schorsch, and Tom, "CAP: An automated self-assessment tool to check Pascal programs for syntax, logic and style errors," in *Proceedings of the twenty-sixth SIGCSE technical symposium on Computer science education - SIGCSE '95*, vol. 27, New York, New York, USA: ACM Press, 1995, pp. 168–172 (see p. 139).

[327] H. J. Schünemann and L. Moja, “Reviews: Rapid! Rapid! Rapid! …and systematic,” *Systematic Reviews*, vol. 4, no. 1, p. 4, 2015 (see p. 197).

[328] E. L. Seidel, H. Sibghat, K. Chaudhuri, W. Weimer, and R. Jhala, “Learning to blame: localizing novice type errors with data-driven diagnosis,” *Proceedings of the ACM on Programming Languages*, vol. 1, no. OOPSLA, pp. 1–27, 2017 (see p. 39).

[329] H. Seo, C. Sadowski, S. Elbaum, E. Aftandilian, and R. Bowdidge, “Programmers’ build errors: a case study (at google),” in *Proceedings of the 36th International Conference on Software Engineering - ICSE 2014*, New York, New York, USA: ACM Press, 2014, pp. 724–734 (see pp. 45, 46, 50, 53, 55, 56, 62, 65, 72, 103, 136, 148, 196).

[330] O. Shacham, M. Vechev, and E. Yahav, “Chameleon: Adaptive selection of collections,” in *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI ’09, Dublin, Ireland: ACM, 2009, pp. 408–418 (see p. 203).

[331] U. Shani, “Should Program Editors Not Abandon Text Oriented Commands?” *SIGPLAN Not.*, vol. 18, no. 1, pp. 35–41, 1983 (see p. 134).

[332] B. Sharif, M. Falcone, and J. I. Maletic, “An eye-tracking study on the role of scan time in finding source code defects,” in *ETRA*, 2012, p. 381 (see p. 74).

[333] D. G. Shaw, “Effects of Learning to Program A Computer in BASIC or Logo on Problem-solving Abilities,” *AEDS Journal*, vol. 19, no. 2-3, pp. 176–189, 1986 (see p. 137).

[334] W. J. Shaw, “Making APL error messages kinder and gentler,” *ACM SIGAPL APL Quote Quad*, vol. 19, no. 4, pp. 320–324, 1989 (see pp. 47, 266).

[335] B. Shneiderman, “Measuring computer program quality and comprehension,” *International Journal of Man-Machine Studies*, vol. 9, no. 4, pp. 465–478, 1977 (see pp. 17, 92, 93).

[336] ——, “System message design: Guidelines and experimental results,” in *Directions in Human-Computer Interaction*, 1982, pp. 55–78 (see p. 211).

[337] B. Shneiderman, “Designing computer system messages,” *Communications of the ACM*, vol. 25, no. 9, pp. 610–611, 1982 (see pp. 47, 124, 266).

[338] B. Shneiderman and R. Mayer, "Syntactic/semantic interactions in programmer behavior: A model and experimental results," *International Journal of Computer & Information Sciences*, vol. 8, no. 3, pp. 219–238, 1979 (see p. 129).

[339] E. Shortliffe, *Computer-based medical consultations: MYCIN*. 1976 (see p. 132).

[340] Shu-Hsien Liao, "Expert system methodologies and applications—a decade review from 1995 to 2004," *Expert Systems with Applications*, vol. 28, no. 1, pp. 93–103, 2005 (see p. 132).

[341] V. J. Shute, "Focus on Formative Feedback," *Review of Educational Research*, vol. 78, no. 1, pp. 153–189, 2008 (see p. 140).

[342] J. Siegmund, "Framework for measuring program comprehension," PhD thesis, University of Magdeburg, 2012 (see p. 56).

[343] J. Siegmund, C. Kästner, S. Apel, C. Parnin, A. Bethmann, T. Leich, G. Saake, and A. Brechmann, "Understanding Understanding Source Code with Functional Magnetic Resonance Imaging," in *Proceedings of the 36th International Conference on Software Engineering*, ser. ICSE 2014, New York, NY, USA: ACM, 2014, pp. 378–389 (see p. 128).

[344] J. Siegmund, N. Peitek, C. Parnin, S. Apel, J. Hofmeister, C. Kästner, A. Begel, A. Bethmann, and A. Brechmann, "Measuring Neural Efficiency of Program Comprehension," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2017, New York, NY, USA: ACM, 2017, pp. 140–150 (see p. 129).

[345] J. Siegmund and J. Schumann, "Confounding parameters on program comprehension: a literature survey," *Empirical Software Engineering*, vol. 20, no. 4, pp. 1159–1192, 2015 (see p. 128).

[346] J. Siek and A. Lumsdaine, "Concept checking: Binding parametric polymorphism in C++," in *First Workshop on C++ Template Programming*, Germany, 2000 (see p. 103).

[347] A. Simon, O. Chitil, and F. Huch, "Typeview: A tool for understanding type errors," in *Draft Proceedings of the 12th International Workshop on*

Implementation of Functional Languages, Citeseer, 2000, pp. 63–69 (see p. 39).

[348] S. L. Simpson, R. G. Lyday, S. Hayasaka, A. P. Marsh, and P. J. Laurienti, "A permutation testing framework to compare groups of brain networks," *Frontiers in Computational Neuroscience*, vol. 7, p. 171, 2013 (see pp. 114, 115).

[349] R. Singh, S. Gulwani, and A. Solar-Lezama, "Automated feedback generation for introductory programming assignments," in *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '13, Seattle, Washington, USA: ACM, 2013, pp. 15–26 (see p. 200).

[350] R. Smith, "An overview of the Tesseract OCR engine," in *ICDAR*, vol. 2, 2007, pp. 629–633 (see p. 59).

[351] A. Solar-Lezama, C. G. Jones, and R. Bodik, "Sketching concurrent data structures," in *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '08, Tucson, AZ, USA: ACM, 2008, pp. 136–148 (see p. 202).

[352] E. Soloway, B. Adelson, and K. Ehrlich, "Knowledge and processes in the comprehension of computer programs," *The nature of expertise*, pp. 129–152, 1988 (see p. 129).

[353] E. Soloway and K. Ehrlich, "Empirical Studies of Programming Knowledge," *IEEE Transactions on Software Engineering*, vol. SE-10, no. 5, pp. 595–609, 1984 (see p. 129).

[354] J. C. Spohrer and E. Soloway, "Novice mistakes: are the folk wisdoms correct?" *Communications of the ACM*, vol. 29, no. 7, pp. 624–632, 1986 (see p. 135).

[355] M. Sridharan, S. J. Fink, and R. Bodik, "Thin slicing," in *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '07, San Diego, California, USA: ACM, 2007, pp. 112–122 (see p. 204).

[356] (2017). Stack overflow developer survey results 2017, [Online]. Available: `https://insights.stackoverflow.com/survey/2017` (visited on 01/01/2018) (see p. 42).

[357] M. Stefik, J. Aikins, R. Balzer, J. Benoit, L. Birnbaum, F. Hayes-Roth, and E. Sacerdoti, “The organization of expert systems, a tutorial,” *Artificial Intelligence*, vol. 18, no. 2, pp. 135–173, 1982 (see p. 132).

[358] J. Stolarek, “Understanding Basic Haskell Error Messages,” *The Monad.Reader*, no. 20, pp. 21–41, 2012 (see p. 37).

[359] M.-A. Storey, “Theories, tools and research methods in program comprehension: past, present and future,” *Software Quality Journal*, vol. 14, no. 3, pp. 187–208, 2006 (see p. 129).

[360] P. J. Stuckey, M. Sulzmann, and J. Wazny, “Interactive type debugging in haskell,” in *Proceedings of the 2003 ACM SIGPLAN Workshop on Haskell*, ser. Haskell ’03, Uppsala, Sweden: ACM, 2003, pp. 72–83 (see p. 39).

[361] ——, “Improving type error diagnosis,” in *Proceedings of the 2004 ACM SIGPLAN Workshop on Haskell*, ser. Haskell ’04, Snowbird, Utah, USA: ACM, 2004, pp. 80–91 (see p. 39).

[362] B. Sufrin and O. De Moor, “Modeless structure editing,” in *Proceedings of the Oxford-Microsoft symposium in Celebration of the work of Tony Hoare*, 1999 (see p. 135).

[363] J. Tao, N. Yafeng, and Z. Lei, “Are the warning icons more attentional?” *Applied Ergonomics*, vol. 65, pp. 51–60, 2017 (see p. 132).

[364] T. Teitelbaum and T. Reps, “The Cornell Program Synthesizer: A Syntax-directed Programming Environment,” *Commun. ACM*, vol. 24, no. 9, pp. 563–573, 1981 (see p. 134).

[365] N. Tintarev and J. Masthoff, “A Survey of Explanations in Recommender Systems,” in *2007 IEEE 23rd International Conference on Data Engineering Workshop*, 2007, pp. 801–810 (see p. 133).

[366] E. Torlak, M. Vaziri, and J. Dolby, “Memsat: Checking axiomatic specifications of memory models,” in *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI ’10, Toronto, Ontario, Canada: ACM, 2010, pp. 341–350 (see p. 206).

[367] S. Toulmin, *The Uses of Argument*. Cambridge University Press, 2003. arXiv: `arXiv:1011.1669v3` (see p. 105).

[368] L. Tratt. (2004). An editor for composed programs, [Online]. Available: https://tratt.net/laurie/blog/entries/an_editor_for_composed_programs.html (see p. 135).

[369] V. J. Traver, "On compiler error messages: What they say and what they mean," *Advances in Human-Computer Interaction*, vol. 2010, pp. 1–26, 2010 (see pp. 47, 50, 77, 103, 124, 196, 210, 211, 267).

[370] C. Treude, O. Barzilay, and M.-A. Storey, "How do programmers ask and answer questions on the web?" In *ICSE*, 2011, pp. 804–807 (see p. 110).

[371] O. Tripp, M. Pistoia, S. J. Fink, M. Sridharan, and O. Weisman, "Taj: Effective taint analysis of web applications," in *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '09, Dublin, Ireland: ACM, 2009, pp. 87–97 (see p. 206).

[372] H. Uwano, M. Nakamura, A. Monden, and K.-i. Matsumoto, "Analyzing individual performance of source code review using reviewers' eye movement," in *ETRA*, San Diego, California, 2006, pp. 133–140 (see p. 74).

[373] (2017). Valgrind user manual (valgrind-3.13.0), [Online]. Available: http://valgrind.org/docs/ (see p. 282).

[374] M. L. Van De Vanter, "Practical language-based editing for software engineers," in *Software Engineering and Human-Computer Interaction (ICSE '94) Workshop on SE-HCI: Joint Research Issues*, R. N. Taylor and J. Coutaz, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 1995, pp. 251–267 (see p. 134).

[375] G. Van Rossum and colleagues, "Python programming language.," in *USENIX Annual Technical Conference*, vol. 41, 2007, p. 36 (see p. 269).

[376] K. VanLehn, "The Relative Effectiveness of Human Tutoring, Intelligent Tutoring Systems, and Other Tutoring Systems," *Educational Psychologist*, vol. 46, no. 4, pp. 197–221, 2011 (see p. 133).

[377] I. Vessey, "Expertise in Debugging Computer Programs: An Analysis of the Content of Verbal Protocols," *IEEE Transactions on Systems, Man, and Cybernetics*, vol. 16, no. 5, pp. 621–637, 1986 (see p. 138).

[378] W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda, “Model Checking Programs,” *Automated Software Engineering*, vol. 10, no. 2, pp. 203–232, 2003 (see p. 40).

[379] *Visual Studio*, `https://www.visualstudio.com/` (see pp. 42, 57).

[380] M. Voelter, J. Siegmund, T. Berger, and B. Kolb, “Towards User-Friendly Projectional Editors,” in *Software Language Engineering: 7th International Conference (SLE 2014)*, B. Combemale, D. J. Pearce, O. Barais, and J. J. Vinju, Eds., Cham: Springer International Publishing, 2014, pp. 41–61 (see p. 134).

[381] A. Von Mayrhauser and A. Vans, “Program comprehension during software maintenance and evolution,” *Computer*, vol. 28, no. 8, pp. 44–55, 1995 (see p. 129).

[382] P. Wadler, “A Complement to Blame,” in *1st Summit on Advances in Programming Languages (SNAPL 2015)*, T. Ball, R. Bodik, S. Krishnamurthi, B. S. Lerner, and G. Morrisett, Eds., ser. Leibniz International Proceedings in Informatics (LIPIcs), vol. 32, Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2015, pp. 309–320 (see p. 39).

[383] G. Wallace, R. Biddle, and E. Tempero, “Smarter Cut-and-paste for Programming Text Editors,” in *Proceedings of the 2nd Australasian Conference on User Interface*, ser. AUIC ’01, Washington, DC, USA: IEEE Computer Society, 2001, pp. 56–63 (see p. 135).

[384] D. Walton, “Explanations and arguments based on practical reasoning,” 2009 (see p. 105).

[385] M. Wand, “Finding the source of type errors,” in *POPL*, 1986, pp. 38–43 (see pp. 39, 103).

[386] C. Wang, Z. Yang, F. Ivančić, and A. Gupta, “Whodunit? Causal Analysis for Counterexamples,” in *Automated Technology for Verification and Analysis*, S. Graf and W. Zhang, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 82–95 (see p. 42).

[387] R. C. Waters, “Program Editors Should Not Abandon Text Oriented Commands,” *SIGPLAN Not.*, vol. 17, no. 7, pp. 39–46, 1982 (see p. 134).

[388] C. Watson, F. W. B. Li, and J. L. Godwin, "BlueFix: Using Crowd-Sourced Feedback to Support Programming Students in Error Diagnosis and Repair," in *Advances in Web-Based Learning - ICWL 2012: 11th International Conference, Sinaia, Romania, September 2-4, 2012*, E. Popescu, Q. Li, R. Klamma, H. Leung, and M. Specht, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 228–239 (see p. 139).

[389] G. Weber, "Code is Not Just Text: Why Our Code Editors Are Inadequate Tools," in *Companion to the First International Conference on the Art, Science and Engineering of Programming*, ser. Programming '17, New York, NY, USA: ACM, 2017, 35:1–35:3 (see p. 135).

[390] G. M. Weinberg, *The Psychology of Computer Programming*. Dorset House Publications, 1998 (see p. 209).

[391] M. Weiser, "Programmers use slices when debugging," *Communications of the ACM*, vol. 25, no. 7, pp. 446–452, 1982 (see p. 129).

[392] D. T. Welsh and L. D. Ordonez, "Conscience without cognition: The effects of subconscious priming on ethical behavior," *Academy of Management Journal*, vol. 57, no. 3, pp. 723–742, 2014 (see p. 61).

[393] J. Whittle, A. Bundy, R. Boulton, and H. Lowe, "An ML editor based on proofs-as-programs," in *14th IEEE International Conference on Automated Software Engineering*, 1999, pp. 166–173 (see pp. 135, 139).

[394] J. Whittle, A. Bundy, and H. Lowe, "An editor for helping novices to learn standard ML BT - Programming Languages: Implementations, Logics, and Programs: 9th International Symposium, PLILP '97 Including a Special Track on Declarative Programming Languages in Education Southampton, UK, Septe," in, H. Glaser, P. Hartel, and H. Kuchen, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 1997, pp. 389–405 (see p. 135).

[395] M. R. Wick and W. B. Thompson, "Reconstructive expert system explanation," *Artificial Intelligence*, vol. 54, no. 1, pp. 33–70, 1992 (see pp. 132, 133).

[396] S. Wiedenbeck, "Beacons in computer program comprehension," *International Journal of Man-Machine Studies*, vol. 25, no. 6, pp. 697–709, 1986 (see pp. 128, 129).

[397] S. L. Wise and X. Kong, “Response time effort: A new measure of examinee motivation in computer-based tests,” *Applied Measurement in Education*, vol. 18, no. 2, pp. 163–183, 2005 (see p. 61).

[398] M. S. Wogalter and N. C. Silver, “Warning signal words: connoted strength and understandability by children, elders, and non-native English speakers,” *Ergonomics*, vol. 38, no. 11, pp. 2188–2206, 1995 (see p. 132).

[399] M. S. Wogalter, S. S. Godfrey, G. A. Fontenelle, D. R. Desaulniers, P. R. Rothstein, and K. R. Laughery, “Effectiveness of Warnings,” *Human Factors*, vol. 29, no. 5, pp. 599–612, 1987 (see p. 131).

[400] M. Wogalter and C. Mayhorn, “The future of risk communication: Technology-based warning systems,” *Handbook of Warnings, MS Wogalter, ed., Lawrence Erlbaum Associates, Inc., Mahwah, NJ*, pp. 783–794, 2006 (see p. 131).

[401] J. Wrenn and S. Krishnamurthi, “Error messages are classifiers: a process to design and evaluate error messages,” in *Proceedings of the 2017 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software - Onward! 2017*, New York, New York, USA: ACM Press, 2017, pp. 134–147 (see p. 139).

[402] B. Wu, J. P. Campora III, and S. Chen, “Learning User Friendly Type-error Messages,” *Proc. ACM Program. Lang.*, vol. 1, no. OOPSLA, 106:1–106:29, 2017 (see p. 38).

[403] Q. Wu and J. R. Anderson, “Problem-solving transfer among programming languages,” 1990 (see p. 138).

[404] G. Xu, M. D. Bond, F. Qin, and A. Rountev, “Leakchaser: Helping programmers narrow down causes of memory leaks,” in *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI ’11, San Jose, California, USA: ACM, 2011, pp. 270–282 (see p. 203).

[405] J. Yang, “Explaining type errors by finding the source of a type conflict.,” in *Scottish Functional Programming Workshop*, Citeseer, vol. 1, 1999, pp. 59–67 (see p. 39).

[406] L. R. Ye and P. E. Johnson, “The Impact of Explanation Facilities on User Acceptance of Expert Systems Advice,” *MIS Quarterly*, vol. 19, no. 2, pp. 157–172, 1995 (see p. 132).

[407] S. L. Young and M. S. Wogalter, “Comprehension and Memory of Instruction Manual Warnings: Conspicuous Print and Pictorial Icons,” *Human Factors*, vol. 32, no. 6, pp. 637–649, 1990 (see p. 131).

[408] E. A. Youngs, “Human errors in programming,” *International Journal of Man-Machine Studies*, vol. 6, no. 3, pp. 361–376, 1974 (see pp. 135, 210).

[409] D. Zhang, A. C. Myers, D. Vytiniotis, and S. Peyton-Jones, “Diagnosing type errors with class,” in *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI ’15, Portland, OR, USA: ACM, 2015, pp. 12–21 (see p. 204).

[410] Q. Zhang, C. Sun, and Z. Su, “Skeletal program enumeration for rigorous compiler testing,” in *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI ’17, Barcelona, Spain: ACM, 2017, pp. 347–361 (see p. 202).

[411] X. Zhang, N. Gupta, and R. Gupta, “Pruning dynamic slices with confidence,” in *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI ’06, Ottawa, Ontario, Canada: ACM, 2006, pp. 169–180 (see p. 204).

APPENDICES

A What Do We Know About Presenting Human-Friendly Output from Program Analysis Tools?

A.1 Abstract

Program analysis tools perform sophisticated analysis on source code to help programmers resolve compiler errors, apply optimizations, and identify security vulnerabilities. Despite the utility of these tools, research suggests that programmers do not frequently adopt them in practice—a primary reason being that the output of these tools is difficult to understand. Towards providing a synthesis of what researchers know about the presentation of program analysis output to programmers, we conducted a scoping review of the PLDI conference proceedings from 1988-2017. The scoping review serves as interim guidance for advancing collaborations between research disciplines. We discuss how cross-disciplinary communities, such as PLATEAU, are critical to improving the usability of program analysis tools.[1]

[1]This chapter was previously published in: T. Barik, C. Parnin, and E. Murphy-Hill, "One λ at a time: What do we know about presenting human-friendly output from program analysis tools?" In *PLATEAU'17 Workshop on Evaluation and Usability of Programming Languages and Tools*, 2017.

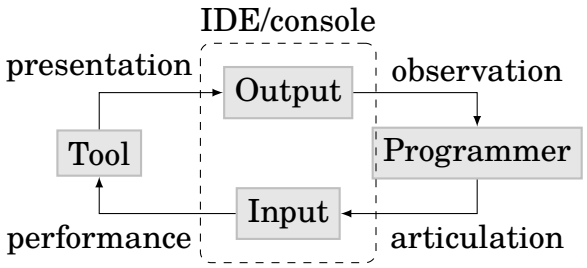


Figure A.1 The interaction framework.

A.2 Introduction

In 1983, Brown [48] lamented that one of the most neglected aspects of the human-machine interface was the quality of the error messages produced by the machine. Today, it appears that many of Brown's laments still hold true with regard to program analysis tools—tools that are intended to help programmers resolve defects in their code. For example, interview and survey studies conducted at Microsoft reveal that poor error messages remain one of the top pain points when using program analysis tools [64], and other studies show similar frustration with error messages in tools [20, 183, 329]. In academia, the situation seems even more dire. As Hanenberg [147] notes in his essay on programming languages research: "developers, which are the main audience for new language constructs, are hardly considered in the research process." And Danas, Nelson, Harrison, and colleagues [83] note that in some cases, the output of program analysis tools, such as in model-finders and SAT-solvers, are generated arbitrarily and in an unprincipled way, without regard to the friendliness towards the programmer who might actually use them.

In prior work [21], we have modeled the interaction of programmers with their program analysis tools in terms of an interaction framework, conceptualized by Abowd and Beale [1] and adapted to tools by Traver [369] (Figure A.1). The interaction framework describes the different interactions between the tool and the programmer, with the tool performing some sophisticated analysis, presenting the information to the programmer for observation through a console or IDE, and then allowing the programmer to articulate their intentions back to the tool. In this paper, we are interested specifically in the *presentation* aspect of the framework, and what we know about presenting human-friendly output from program analysis tools.

Towards the longer-term goal of providing a comprehensive knowledge synthesis about program analysis output, we conducted an interim scoping review of the proceedings from Programming Language Design and Implementation (PLDI), from 1988-2017. The scoping review is intended to be accessible to human-computer interaction (HCI) researchers who want to understand how the PL community is currently applying program analysis output, in order to eventually bridge HCI research with program analysis tools. Consequently, while PLDI papers are typically written to emphasize the formal properties of their program analysis tools as their primary goal, our scoping review reframes these papers in terms of the their *program analysis output* as the primary investigation.

The contributions of this scoping review are:

- A *quasi-gold set* of manually-identified papers from PLDI that relate to program analysis output, to bootstrap future, comprehensive literature reviews on the subject of human-friendly program analysis output.
- A knowledge synthesis of the features of program analysis output that researchers employ to present output to programmers, instantiated as a taxonomy (Appendix A.4). Our taxonomy is agnostic to a particular mode of output, such as text or graphics.

A.3 Methodology

A.3.1 What is a Scoping Review?

In this study, we conduct a *scoping review*—a reduced form of the traditional systematic literature review [12, 194]. Scoping reviews have many of the same characteristics of traditional literature reviews: their purpose is to collect, evaluate, and present the available research evidence for a particular investigation. However, because of their reduced form, they can also be executed more rapidly than traditional literature reviews [327]. For example, reductions to scoping reviews include limiting the types of literature databases, constraining the date range under investigation, or eliding consistency measures such as inter-rater agreement. A notable weakness of scoping reviews is that they are not a final output; instead, they provide interim guidance towards what to expect if a comprehensive literature review were to be conducted. Scoping reviews are particularly useful in this interim stage for soliciting guidance on conducting a more formal review, as is our intention in this paper.

A.3.2 Execution of SALSA Framework

We conducted our scoping review using the traditional SALSA framework: *S*earch, *A*ppraisa*L*, *S*ynthesis, and *A*nalysis. Here, we discuss the additional constraints we adopted in using SALSA for our scoping review.

Search. We scoped our search to all papers within a single conference: Programming Language Design and Implementation (PLDI), for all years (1988-2017). As HCI researchers, we selected PLDI because it is considered to be a top-tier conference for programming languages research, because it contains a variety of program analysis tools, and because these tools tend to have formal properties of soundness and completeness that are not typically found in prototype tools within HCI. Discussions with other researchers within PLDI also revealed that researchers are interested in having their tools adopted by a broader community, but confusing program analysis output hinders usability of the tools to users outside their own research groups.

Appraisal. We manually identified papers through multiple passes. In the first pass, we skimmed titles and abstracts and included any papers which mentioned a program analysis tool and indicated output intended to be consumed by a programmer other than the authors of the tool. In this pass, our goal was to be liberal with paper inclusion, and to minimize false negatives. We interpreted program analysis tools in the broadest sense, to include model checkers, verifiers, static analysis tools, and dynamic analysis tools. In the second pass, we examined the contents of the paper to identify if the paper contributed or discussed its output. Finally, we removed papers that were purely related to reducing false positives, unless those papers used false positives as part of their output to provide additional information to the programmer. For some papers, the output was measured in terms of manual patches submitted to bug repositories. We excluded such papers since the output was manually constructed, and not obtained directly from the tool.

Synthesis and Analysis. We synthesized the papers into a taxonomy of presentation features (Appendix A.4). For analysis, we opted for a narrative-commentary approach [137] in which we summarized the contributions of each of the papers with respect to human-friendly presentations.

A.3.3 Limitations

As a form of interim guidance, a scoping review has several important limitations. First, the review is biased in several ways. Being scoped only to PLDI means

Table A.1 Taxonomy of Presentation

Feature	Section
Alignment	Appendix A.4.1
Clustering and Classification	Appendix A.4.2
Comparing	Appendix A.4.3
Example	Appendix A.4.4
Interactivity	Appendix A.4.5
Localizing	Appendix A.4.6
Ranking	Appendix A.4.7
Reducing	Appendix A.4.8
Tracing	Appendix A.4.9

that the identified taxonomy is likely to be incomplete. Second, the scoping review by definition misses key contributions found in other conferences, such as the International Conference on Software Engineering (ICSE), Foundations of Software Engineering (FSE), and the Conference on Human Factors in Computing Systems (CHI), just to name a few. Third, the paper summaries are intended to be accessible to HCI researchers who may not have formal PL experience. As a result, in the interest of being broadly accessible, some of the summaries of the papers may be oversimplified in terms of their PL contributions. Finally, any conclusions made from this interim work should be treated as provisional and subject to revision as more comprehensive reviews are conducted.

A.4 Taxonomy of Presentation

In this section, we classify and summarize all of the papers from PLDI from 1988-2017 that discuss or contribute to program analysis output intended for programmers. Intentionally, we labeled the taxonomy features such that they do not commit to a particular textual or visual affordance. For example, in a text interface, the feature of ranking (Appendix A.4.7) may be implemented as an enumerated list of items in the console, with a prompt for selection if interactivity is required. In a graphical interface, ranking might instead be implemented using a drop-down, through which the programmer would select their desired option.

The identified taxonomy of presentation features are summarized in Table A.1.

Some papers describe output that use multiple features; in these cases, we selected the feature which we felt best represented the contribution of the output.

A.4.1 Alignment

In alignment, program analysis output is presented in a representation that is already familiar to the programmer.

Within this feature, Pombrio and Krishnamurthi [292] tackle the problem of syntactic sugar: programming constructs that make things easier to express, but are ultimately reducible to alternative constructs. For example, in C, the array access notation `a[i]` is syntactic sugar for (the sometimes less convenient notation) `*(a + i)`. Unfortunately, syntactic sugar is eliminated by many transformation algorithms, making the resulting program unfamiliar to the programmer. Pombrio and Krishnamurthi [292] introduce a process of *resugaring* to allow computation reductions in terms of the surface syntax. With similar aims, the AutoCorres tool uses a technique of *specification abstraction*, to present programmers with a representation of the program at a human-readable abstraction while additionally producing a formal refinement of the final presentation [139].

Notions of natural language and readability find their place in several PLDI papers. Qiu, Garg, Ştefănescu, and colleagues [301] propose *natural proofs*, in which automated reasoning systems restrict themselves to using common patterns found in human proofs. Given a reference implementation, and an error model of potential corrections, Singh, Gulwani, and Solar-Lezama [349] introduce a method for automatically deriving *minimal corrections* to students' incorrect solutions, in the form of a itemized list of changes, expressed in natural language. And the AFix tool uses a variety of static analysis and static code transformations to design bug fixes for a type of concurrency bug, *single-variable atomicity violations* [178]. The bug fixes are human-friendly in that they attempt to provide a fix that, in addition to other metrics, does not harm code *readability*. To support readability, the authors manually evaluated several possible locking policies to determine which ones were most readable.

Issues of alignment and representation become important to programmers during understanding of optimizations in source-level debugging of optimized code [3]; in their approach, Adl-Tabatabai and Gross [3] detect *engendered variables* that would cause the programmer to draw incorrect conclusions as a result of internal optimizations by the compiler. Earlier work by Brooks, Hansen, and Simmons [43] and Coutant, Meloy, and Ruscetta [76] also provide techniques that allow program-

mers to reason about optimized code through mapping the state of the optimized execution back to the original source. For example, Brooks, Hansen, and Simmons [43] use highlighting, boxing, reverse video, grey-scale shading, boxing, and underlining to animate and convey runtime program behavior, overlaid on the original source code, to the programmer.

A.4.2 Clustering and Classification

Clustering and classification output aims to organize or separate information in a way that reduces the cognitive burden for programmers. For example, Narayanasamy, Wang, Tigani, and colleagues [258] focus on a dynamic analysis technique to automatically classify *data races*—a type of concurrency bug in multi-threaded programs—as being potentially benign or potentially harmful. Furthermore, the tool provides the programmer with a reproducible scenario of the data race to help the programmer understand how it manifests.

Liblit, Naik, Zheng, and colleagues [217] present a statistical dynamic debugging technique that *isolates* bugs in programs containing multiple undiagnosed bugs; importantly, the algorithm separates the effects of different bugs and identifies predictors that are associated with individual bugs. An earlier technique using statistical sampling is also presented by the authors [216]. Ha, Rossbach, Davis, and colleagues [144] introduce a classification technique in CLARIFY, a system which classifies *behavior profiles*—essentially, an application's behavior—for black box software components where the source code is not available. And Ammons, Mandelin, Bodík, and colleagues [9] consider the problem of specifications on programs in that the specifications themselves need methods for debugging; they present a method for debugging formal, temporal specifications through *concept analysis* to automatically group traces into highly similar clusters.

A.4.3 Comparing

Comparisons occur in program analysis tools when the programmer has a need to examine or understand differences between two or more versions of their code. Within this feature, Hoffman, Eugster, and Jagannathan [161] introduce a technique of *semantic views* of program executions to perform trace analysis; they apply their technique to identify regressions in large software applications. Through a differencing technique, their RPRISM tool outputs a semantic "diff" between the original and new versions, to allow potential causes to be viewed in their full

context. Similarly, early work by Horwitz [166] identifies both semantic and textual differences between two versions of a program, in contrast to traditional diff-tools that treat source as plain text.

A.4.4 Example

Examples and counterexamples are forms of output that provide evidence for why a situation can occur or how a situation can be violated. Examples are usually provided in conjunction with other presentation features.

The Alive-Infer tool infers preconditions to ensure the validity of a peephole compiler optimization [243]. To the user, it reports both a *weakest* precondition and a set of "succinct" partial preconditions. For wrong optimizations, the tool provides counterexamples. Zhang, Sun, and Su [410] apply a technique of *skeletal program enumeration* to generate small test programs for reporting bugs about in GCC and clang compilers; the generated test programs contain fewer than 30 lines on average. Still other work with test programs devise a test-case reducer for C compiler bugs to obtain small and valid test-cases consistently [308]; the underlying machinery is based on *generic fixpoint computations* which invokes a *modular reducer*.

Padon, McMillan, Panda, and colleagues [274] hypothesize that one of the reasons automated methods are difficult to use in practice is because they are opaque. As Padon, McMillan, Panda, and colleagues [274] state, "they fail in ways that are difficult for a human user to understand and to remedy." Their system, Ivy, graphically displays concrete counterexamples to induction, and allows the user to interactively guide generation from these counterexamples. Nguyen and Van Horn implement a tool in Racket to generate counterexamples for erroneous modules and Isradisaikul and Myers [171] design an algorithm that generates helpful counterexamples for parsing ambiguities; for every parsing conflict, the algorithm generates a compact counterexample illustrating the ambiguity.

PSketch is a program synthesis tool that helps programmers implement concurrent data structures; it uses a *counter example guided inductive synthesis algorithm* (CEGIS) to converge to a solution within a handful of iterations [351]. Given a partial program example, or a *sketch*, PSketch outputs a completed sketch that matches a given correctness criteria.

For type error messages, Lerner, Flower, Grossman, and colleagues [211] pursue an approach in which the type-checker itself does not produce error messages, but instead relies on an oracle for a search procedure that finds similar programs that *do* type-check; to bypass the typically-inscrutable type error messages, their system

provides examples of code (at the same location) that would type check.

And for memory-related output, Cherem, Princehouse, and Rugina [62] implement an analysis algorithm for detecting memory leaks in C programs; their analysis uses *sparse value-flows* to present *concise* error messages containing only a few relevant assignments and path conditions that cause the error to happen.

A.4.5 Interactivity

We identified several papers whose tools support interactivity. That is, the programmer can interact with the tool either before the output is produced, in order to customize the output—or work with the output of the tool in a *mixed-initiative* fashion, where both the programmer and the tool collaborate to arrive at a solution.

Within this feature, Parsify is a program synthesis tool that synthesizes a parser from input and output examples. The tool interface provides immediate visual feedback in response to changes in the grammar being refined, as well as a graphical mechanism for specifying example parse trees using only textual selections [215]. As the programmer adds production rules to the grammar, Parsify uses colored regions overlaid on the examples to convey progress to the programmer.

Live programming is a user interface capability that allows a programmer to edit code and immediately see the effect of the code changes. Burckhardt, Fahndrich, Halleux, and colleagues [50] introduce a *type and effect* formalization that separates the *rendering* of UI components as a side effect of the *non-rendering* logic of the program. This formalization enables responsive feedback and allows the programmer to make code changes without needing to restart the debugging process to refresh the display.

Dillig, Dillig, and Aiken [98] present a technique called *abductive inference*—that is, to find an explanatory hypothesis for a desired outcome—to assist programmers in classifying error reports. The technique computes small, relevant queries presented to a user that capture exactly the information the analysis is missing to either discharge or validate the error.

LeakChaser identifies unnecessarily-held memory references which often result in memory leaks and performance issues in manages languages such as Java [404]. The tool allows an *iterative* process through three *tiers* which assist programmers at different levels of abstraction, from *transactions* at the highest-level tier to *lifetime relationships* at the lowest level tier.

Chameleon assists programmers in choosing an abstract collection implementation in their algorithm [330]. During program execution, Chameleon computes

trace metrics using *semantic profiling*, together with a set of collection selection rules, to present recommended collection adaptation strategies to the programmers. Similarly, the PetaBricks tool makes algorithm choice a first-class construct of the language [11].

Dincklage and Diwan [99] identify how tools can benefit from guidance from the programmer in cases where incorrect tool results would otherwise compromise its usefulness. For example, many refactoring operations in the Eclipse IDE are optimistic, and do not fully check that the result is fully legal. They propose a method to produce necessary and sufficient reasons, that is, a *why* explanation, for a potentially undesirable result; the programmer can then—through applying predicates—provide feedback on whether the given analysis result is desirable.

Finally, MrSpidey is a user-friendly, static debugger for Scheme [121]; the program analysis computes *value set descriptions* for each term in the program and constructs a *value flow graph* connecting the set descriptions; these flows are made visible to the programmer through a value flow browser which overlays arrows over the program text. The programmer can interactively expose portions of the value graph.

A.4.6 Localizing

Tools present the relevant program locations for an error, or *localize* errors, through two forms: 1) a *point* localization, in which a program analysis tool tries to identify a single region or line as relevant to the error, and 2) as *slices*, where multiple regions are responsible for the error.

Point. Zhang, Myers, Vytiniotis, and colleagues [409] implement, within the GHC compiler, a simple Bayesian type error diagnostic that identifies the *most likely* source of the type error, rather than the *first source* the inference engine "trips over." The BugAssist tool implements an algorithm for error cause localization based on a reduction to *the maximal satisfiability problem* to identify the *cause* of an error from failing execution traces [186]. The Breadcrumbs tool uses a *probabilistic calling context* (essentially, a stack trace) to identify the root cause of bug, by recording extra information that *might* be useful in explaining a failure [37].

Slices. Program slicing identifies parts of the program that may affect a point of interest—such as those related to an error message; Sridharan, Fink, and Bodik [355] propose a technique called *thin slicing* which helps programmers better identify bugs because it identifies more relevant lines of code than traditional slicing. Analogous to thin slicing, Zhang, Gupta, and Gupta [411] developed a strategy for

pruning dynamic slices to identify subsets of statements that are likely responsible for producing an incorrect value; for each statement executed in the dynamic slice, their tool computes a confidence value, with higher values corresponding to greater likelihood that the execution of the statement produced a correct value.

A.4.7 Ranking

Ranking is a presentation feature that orders the output of the program analysis in a systematic way. For example, random testing tools, that is, *fuzzers*, can be frustrating to use because they "indiscriminately and repeatedly find bugs that may not be severe enough to fix right away" [61]. Chen, Groce, Zhang, and colleagues [61] propose a technique that *orders* test cases in a way that diverse, interesting cases (defined through a machine technique called *furthest point first*) are highly ranked. And the AcSpec tool prioritizes alarms for automatic program verifiers through *semantic inconsistency detection* in order to report high-confidence warnings to the programmer [36].

Coppa, Demetrescu, and Finocchi [74] present a profiling methodology and toolkit for helping programmers discover asymptotic inefficiencies in their code. The output of the profiler is, for each executed routine of the program, a set of tuples that aggregate performance costs by input size—these outputs are intended to be used as input to performance plots. The Kremlin tool makes recommendations about which parts of the program a programmer should spend effort parallelizing; the tool identifies these regions through a *hierarchical critical path analysis* and presents to the programmer an ordered (by speedup) parallelism plan as a list of files and lines to modify [128].

Perelman, Gulwani, Ball, and colleagues [283] provide ranked expressions for completions in API libraries through a language of *partial expressions*, which allows the programmer to leave "holes" for the parts they do not know.

A.4.8 Reduction

Reduction approaches take a large design space of allowable program output and reduce that space using some systematic rule. Within this feature, Logozzo, Lahiri, Fähndrich, and colleagues introduce a static analysis technique of *Verification Modulo Versions* (VMV), which reduces the number of alarms reported by verifiers while maintaining semantic guarantees [224]. Specifically, VMV is designed for

scenarios in which programmers desire to fix *new* defects introduced since a previous release.

A.4.9 Tracing

Tracing is a form of slicing that involves flows of information, and understanding how information propagates across source code. As one example, Ohmann, Brooks, D'Antoni, and colleagues [270] present a system that answers control-flow queries posed by programmers as formal languages. The tool indicates whether the query expresses control flow that is *possible* or *impossible* for a given failure report. As another example, PIDGIN is a program analysis and understanding tool that allows programmers to interactively explore *information flows*—through *program dependence graphs* within their applications—and investigate counterexamples [179]. Taint analysis is another information-flow analysis that establishes whether values from unstructured parameters may flow into security-sensitive operations [371]; implemented as TAJ, the tool additionally eliminates redundant reports through hybrid thin slicing and *remediation logic* over library local points. Other techniques, such as those by Rubio-González, Gunawi, Liblit, and colleagues [317], use data-flow analysis techniques to track errors as they propagate through file system code.

To support algorithmic debugging, Faddegon and Chitil [116] developed a library in Haskell, that, after annotating suspected functions, presents a detailed *computational tree*. Computational trees are essentially a trace to help programmers understand how a program works or why it does not work. The tool TraceBack provides debugging information for production systems by providing execution history data about program problems [13]; it uses *first-fault diagnosis* to discover what went wrong the *first* time the fault is encountered.

MemSAT helps programmers debug and reason about memory models: given an *axiomatic* specification, the tool outputs a *trace*—sequences of reads and writes—of the program in which the specification is satisfied, or a *minimal* subset of the memory model and program constraints that are unsatisfiable [366].

The Merlin security analysis tool infers *information flows* in a program to identify security vulnerabilities, such as cross-site scripting and SQL inject attacks [222]. Internally, the inference is based on modeling a *data propagation graph* using *probabilistic constraints*.

A.5 Discussion

Lack of user evaluations in PL. Although we identified and classified papers within PLDI in terms of a taxonomy of presentation, our investigation confirms that papers either perform no usability evaluation with programmers, or the claims of usability of the tool are made through intuition—using the authors of the paper as subjects. For example, consider the presentation feature of alignment (Appendix A.4.1), in which several assumptions are made about how output should be presented in familiar representations to the programmer. All of these assumptions appear to be intuitive—give output in the same level of syntactic sugar as their source code for consistency, use proof constructions commonly found in human proofs, and support readability. Unfortunately, none of these assumptions are tested with actual programmers, reminding us of the concerns noted by Hanenberg and others in the introduction. It seems likely that some of these assumptions *are* actually incorrect, which may explain the lack of adoption in practice and the confusing tool output programmers report for many of these sophisticated program analysis tools.

Lack of operational tools in HCI. At the same time, HCI researchers perform usability studies on user interfaces, yet the experiments they conduct are understandably evaluated against representative tool experiences, rather than the multiplicity of corner cases that occur in practice. Consequently, even if the user interfaces are found to be effective or usable for some measures, the tools themselves cannot actually be used in practice. Regrettably, this means that user interface advances remain within academic papers, and do not ever make it to actual programmers without significant investment in tools that may not even be possible to build due to fundamental, technical limitations.

Bridging PL and HCI. In our view, both deficiencies in PL and HCI can be reduced by fostering collaborations between the disciplines. A cross-disciplinary approach to tool development would enable usable program analysis tools, by having a pipeline from program analysis tools to user evaluations in HCI. HCI contributions could then feedback to PL to further improve the output of program analysis tools. But doing so requires a cross-disciplinary community that can provide such opportunities for collaboration. We suggest that PLATEAU has the potential to become this community.

A.6 Conclusions

In this paper, we conducted a scoping review of PLDI from the period of 1988-2017. In the review, we identified and cataloged papers for program analysis tools that discussed or made contributions to the presentation of output towards programmers. Admittedly, a scoping review is only a starting point for investigation, and can only provide interim guidance. Nevertheless, our hope is that the scoping review we have conducted can serve to bootstrap future, comprehensive systematic literature reviews. We are open to feedback on practical methods to realizing that goal.

A.7 Acknowledgments

This material is based upon work supported by the National Science Foundation under Grant No. 1714538.

B | An Interaction-First Approach for Helping Developers Comprehend and Resolve Error Notifications

B.1 Abstract

Error notifications and their resolutions, as presented by modern IDEs, are still cryptic and confusing to developers. We propose an interaction-first approach to help developers more effectively comprehend and resolve compiler error notifications through a conceptual interaction framework. We propose novel taxonomies that can serve as controlled vocabularies for compiler notifications and their resolutions. We use preliminary taxonomies to demonstrate, through a prototype IDE, how the taxonomies make notifications and their resolutions more consistent and unified.[1]

B.2 Introduction

Programming is a cognitively demanding task [390]. One demanding subtask is the cycle of comprehending feedback as presented through compiler error notifications

[1]This chapter was previously published in: T. Barik, J. Witschey, B. Johnson, and colleagues, "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*, ser. ICSE Companion 2014, Hyderabad, India: ACM, 2014, pp. 536–539.

and articulating appropriate resolutions for each. In our experience, it seems nearly every developer has stories about impenetrable error notifications that caused them consternation. Unfortunately, today's integrated development environments (IDEs) emit cryptic and confusing messages, leaving their resolutions equally elusive, even for experts [256, 369].

When the IDE was itself a new idea, researchers recognized its potential to present and resolve compiler messages in innovative ways, by taking advantage of graphics and multiple windows [48]. Consider *quick fixes*, which augment text-based notifications by offering candidate resolutions. While more useful than text notifications alone, even quick fixes do not fully realize the potential of the IDE. Quick fixes only offer one mechanism for all errors, one that may not be appropriate for every error.

This leads us to ask: How should error notifications be presented, and for what errors? Does a single notification style work for all errors? Should every error have a unique notification, or is the best solution somewhere in between, where some errors should be presented in the same way? Do the resolutions offered by these tools actually align with the way developers articulate them? We suggest a principled approach to understanding this space will help us develop more effective tools.

We propose two taxonomies that formalize an interaction framework from human-computer interaction (HCI) research. These taxonomies model notifications and resolutions and can be used as a controlled vocabulary, which can be reasoned about both cognitively and computationally. Though preliminary, we believe this research is useful and will provide tool developers with new tools to design consistent, unified presentations of notifications and their resolutions. We show how abstract representations of notifications can be *computationalized*, or expressed in a form that machines can interpret, and how this representation supports tools that help developers comprehend and resolve error notifications. To demonstrate how tools can implement our formalism, we introduce a language-agnostic prototype system in which new visualizations, as well as quick fixes and other resolution strategies, can be implemented.

B.3 Related Work

Though researchers have previously identified the problems developers encounter when interpreting and resolving error notifications [47, 172, 408], few provide concrete solutions. As early as the 1960s, researchers designed systems to support

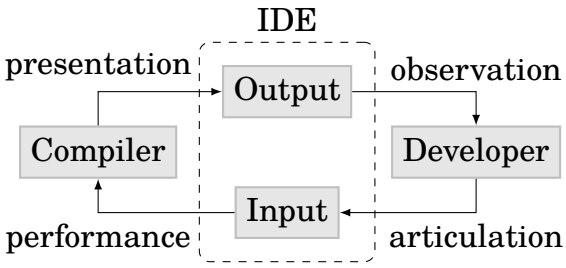


Figure B.1 The interaction framework, instantiated for IDEs.

developer comprehension of compiler error messages [168, 221, 251], but much of this work has focused on text-only solutions that do not take advantage of IDEs. Other researchers have written guidelines for the development of error notifications [245, 273, 336], but these guidelines cannot be directly understood by machines. There has also been recent research on systems to help developers comprehend and resolve errors [152, 231, 256]. In particular, Hartmann and colleagues use a social recommender system which provides suggestions to developers that cognitively align with their comprehension and resolution process [152]. However, a limitation of this system is that the suggestions require a corpus of human-generated fixes.

B.4 Our Approach

We describe Abowd and Beale's interaction framework [1], how it models the interaction between developers and IDEs (Section B.4.1), and through which we identify areas for improvement. We propose two cognitively and computationally expressive taxonomies for building tools that allow developers to more effectively comprehend and resolve error notifications (Section B.4.2).

B.4.1 First Principles: Interaction Framework

Traver considered the difficulty of error message comprehension and resolution from an HCI perspective [369]. This perspective offers a useful insight: Traver describes modern compiler use, as enabled by IDEs, as a specialized instantiation of the Abowd and Beale's interaction framework [1]. However, conventional tools are modeled by the framework only incidentally, and not by design. We believe designing tools for interaction in a principled way will help us make better tools. We briefly

describe here the framework (Figure B.1) and its applicability to IDEs.

This framework comprises four components: a compiler, a developer, an input, and an output. The input and output constitute the interface, which, for developers, is the IDE. For example, imagine a compiler has identified an error in some C# code. The compiler *presents* this error to the developer through the IDE, and the IDE can augment this presentation by, e.g., visually underlining the offending code. This notification must then be *observed* and comprehended by the developer. The developer must then *articulate* a resolution through the IDE, either as a manual edit to the code or through an automated tool in the IDE. The IDE then invokes the compiler, which *performs* the compilation process, and the cycle repeats. Abowd and Beale call these four steps *translations*. This framework exposes interdependence between the translations: if the developer observes a cryptic message and misinterprets it, they are more likely to articulate an incorrect resolution.

B.4.2 Formalizing Translations: Taxonomies

While all four translations in the interaction cycle are important, we think research on some of these translations have made greater computational progress than others. In our opinion, the translations that have made less progress with regards to improving error notifications are presentation and articulation. As with other translations, there are likely many useful formalizations. We chose taxonomies from knowledge management theory as a starting point for designing our formalization, because they help people retrieve, manage, and improve complex problem spaces [127]. We intend to develop a *notification taxonomy* describing information content to include in the presentation of errors, and a *resolution taxonomy* describing how programmers articulate error resolutions to the compiler.

B.4.2.0.1 Presentation: A Notification Taxonomy

We have conducted preliminary research on the first taxonomy by randomly sampling and categorizing roughly 40% of the 500 possible OpenJDK compiler error notifications. The notifications are categorized based on information needed for developer comprehension. We categorized error notifications in a way that identifies the important information that would help a developer understand the underlying error irrespective of any potential resolution strategies. A subset of this taxonomy, which is still in preliminary stages of development, is shown in Figure B.2.

One category in this taxonomy is a Clash, which informs a developer that two

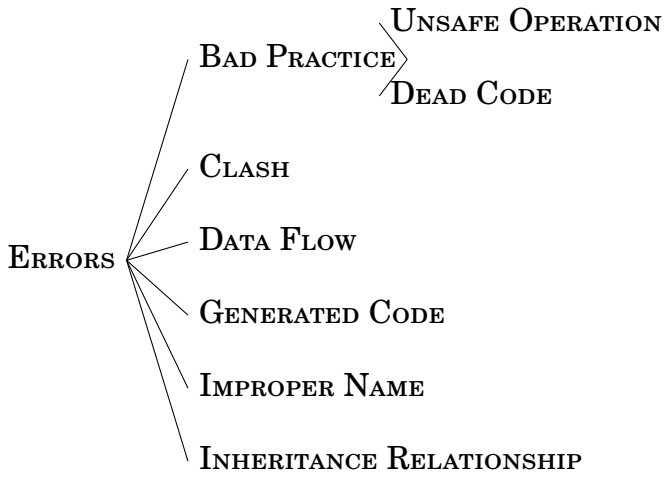
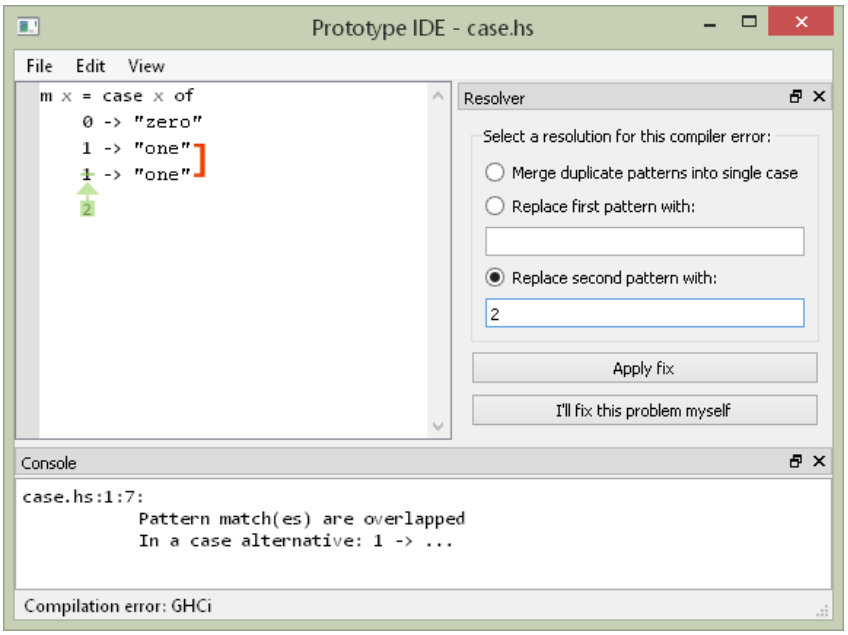


Figure B.2 A partial taxonomy for categorizing notifications by presentation.

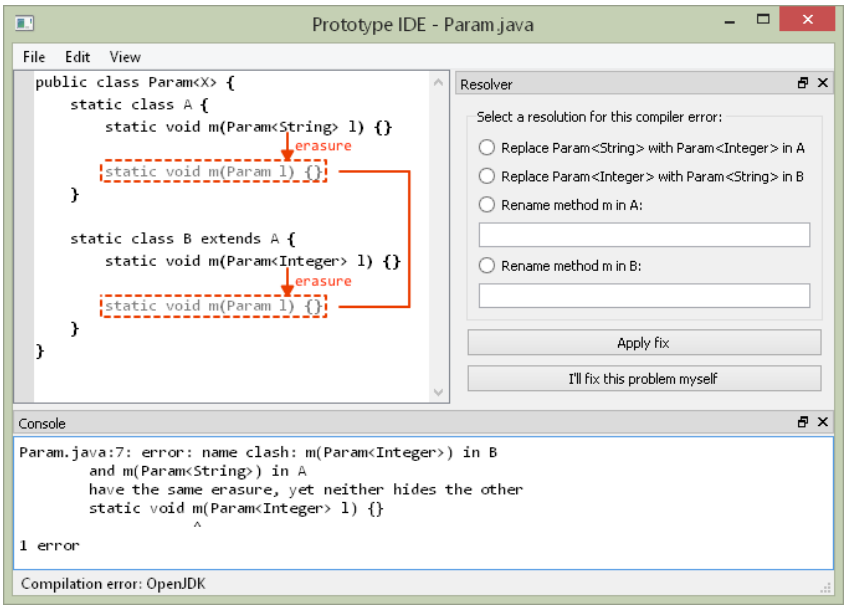
elements cannot coexist in the program. As a result, a notification for a Clash should inform the developer which two program elements are in conflict. For instance, two local variables declared in the same scope in Java with the same name would cause a Clash.

B.4.2.0.2 Articulation: A Resolution Taxonomy

This taxonomy describes strategies developers use to articulate resolutions to errors. By observing and capturing developer resolution strategies, we can potentially generate interfaces which help developers articulate resolutions more naturally and at the appropriate level of abstraction. Table B.1 shows a subset of these taxonomy elements as *tasks*, and how they might be used as GUI widgets or operations. The ChooseOneOf task occurs because a developer must delete all but one of a set of elements from a program, as when a method has been inadvertently assigned both `public` and `private` qualifiers. In this case, the compiler can populate the ChooseOneOf task with the two qualifiers as arguments. This task may be offered as a resolution through an error notification during presentation. Future empirical studies will evaluate the appropriateness of these tasks.



(a) Pattern matches are overlapped



(b) Name clash: Methods have same erasure

Figure B.3 A prototype IDE for notifications and resolutions. The prototype leverages the notification and resolution taxonomies to reuse visualization components. The resolver is a single component, and generates appropriate resolutions using the resolution taxonomy. The text with a red, dashed border is generated code added by the system to help explain the error.

Table B.1 A Partial Taxonomy of Developer Resolution Tasks

Semantic Task	Description	GUI Example
ChooseOneOf$(X, Y, \ldots)$	Chooses an argument from the provided arguments.	Dropdown
Merge$(X, Y, \ldots)$	Merges a set of identical arguments.	Radio button
Remove(X)	Removes a subtree from the source, e.g., dead code.	Radio button
Replace(X, Y)	Replaces the first argument with the second.	Radio button or text field
Move(X)	Moves the argument to a different location in the code.	Drag and drop

B.5 Emerging Results

We present a prototype IDE, shown in Figure B.3. The prototype consists of a text editor pane that can be augmented with visualizations, a resolver pane that the IDE can use to present candidate resolutions, and a traditional console pane containing the raw text error message. Internally, the prototype leverages the presentation and articulation taxonomies by passing notifications as *error objects*. For this paper, it is sufficient to think of these objects as elements from the taxonomies augmented with additional semantic information such as line number, location, or other relevant information for use by the IDE. Here, we demonstrate the use of these taxonomies with two examples.

The IDE in Figure B.3a presents an overlapping pattern error in Haskell, where the first '1' pattern overlaps the second '1' completely, such that the second '1' case can never execute. This error is presented to the IDE as a Clash error object (Figure B.2). It displays the clash between the conflicting cases with a red bracket. The error object also contains resolution tasks (Table B.1) for the developer to articulate: Merge, Replace (for the first conflicting pattern), and Replace (for the second conflicting pattern). With this information, the IDE presents a set of resolution tasks, with graphical widgets (radio buttons, text fields, and so on) as appropriate for articulating each task.

In this example, the developer has chosen the third resolution, so the IDE also displays, in green, the effect of articulating a Replace task: a strike-through for

the case that will change, along with boxed text indicating what the case will be changed to.

A second, more complex Java example, is shown in Figure B.3b. This error is tricky in that it results from *generics*-related type information that is available at compilation time but not at runtime – this is called *type erasure*. Roughly, during the compilation process, parameterized classes are turned into raw classes – for example, `Param<Integer>` and `Param<String>` will both resemble `Param`, at which point, their signatures would be the same (or `have the same erasure`, according to OpenJDK) and the runtime would not be able to tell them apart because both are still available (`neither hides the other`).

At first glance this notification may appear to be a completely different error than that in Figure B.3a. However, if we consider it from a user-centric perspective, we can reduce it to a Clash error – a clash between method signatures after type erasure. Because the effect of the error is not apparent in the source code, but is manifested in the compiled code, we also consider it a Generated Code notification.

The error text provided by OpenJDK describes this error, but in a cryptic way. Our visualization is clearer because it directly shows the effect of the type erasure using red arrows labeled `erasure` and inserts a representation of the generated code, which is displayed in a box with a red, dashed border to differentiate it from the developer's code. The IDE also displays red brackets, as before, to show that the methods are in conflict after erasure. Without this visualization, a developer would have thought through the erasure step of compilation and come to the same conclusion, but our visualization reduces the cognitive burden by performing this reasoning for them.

Our taxonomy helped us recognize the semantic similarity between these two notifications. Then, we used it to represent them both computationally. Our visualization can reuse the same infrastructure for the Clash semantics of the errors, despite being different errors in different languages. In addition, our example demonstrates the composability of the notification semantics, which allows Figure B.3b to augment the Clash visualization with an explanation that the Clash occurs in Generated Code.

This demonstrates the immediate benefit of using these taxonomies – they give consistent and unified semantics to notifications and resolutions. This, in turn, allows IDE developers to add presentation and articulation features to their tools for minimal incremental cost.

B.6 Challenges

The discovery of tasks within the resolution taxonomy may reveal non-reusable tasks that are only applicable to a particular error notification. If special cases occur frequently, then it will negate the advantages of unification that taxonomies would otherwise provide. We must make design tradeoffs in the number of categories. A small number of highly abstract categories allows for greater consistency between notifications, but at the cost of detailed information about individual errors. It is also possible that some elements of these taxonomies cannot be computationalized. For example, consider possible Bad Practice notifications relating to "code smells", which are not necessarily errors, but may indicate general flaws [125]. Resolving code smells is often wholly subjective, preventing any computational solution. So far, we have only used the OpenJDK in creating our taxonomies, though we intend to incorporate more languages in the future. Though we expect our taxonomies to capture a broad range of languages, new language features may require revisions to the taxonomies. A final challenge of our approach is that its effectiveness is constrained by the accuracy of error diagnostics provided by the compiler.

B.7 Conclusions

We think our taxonomies will provide developers and compilers with controlled and expressive vocabularies with which to communicate about errors and their resolutions. The taxonomies give consistent and unified semantics to error objects, which in turn allows IDE developers to easily add presentation and articulation features to their tools. More importantly, the taxonomies allow IDE developers to also design presentation and articulation of notifications in a consistent and unified way. In doing so, tools can offer error notifications and resolutions that align more closely with the way in which developers observe and resolve notifications in their programming activities.

B.8 Acknowledgments

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

C | Study Materials for "Do Developers Read Compiler Error Messages?" (Chapter 4)

C.1 Interview Protocol

C.1.1 Outline

1. Pre-participant steps.
2. Initial steps (namely, calibration).
3. Tasks (5 minutes x 10 tasks = 50 minutes).
4. Post-questionnaire.

C.1.2 Pre-arrival Steps

1. Open both `GazeControl` and `GazeAnalysis`. Make sure the screen is on the proper one (`GazeControl`). Make a new project for each participant with name `PXXX` (which you obtain from the `init` script). You will pause in between tasks.
2. Open Eclipse and reset to defaults.
3. Note if the participant has glasses.

C.1.3 Arrival Steps

1. Have participant sign consent form.
2. Have participant sit in chair and get comfy. Hand them training picture and tell them these are all the places a notification would appear. Ask them not to move too much.
3. Adjust eye tracker. Calibrate eye tracker (in GazeControl) using 9 point alignment. If any lines are outside the blue area, redo those points. Also check 9 of 9, each area should have a red and green point/line. If there are way too many (n>3), try entire recalibration. If still bad after 3 attempts, dismiss participant.
4. **Sanity Check:** In Gaze Analysis, you should see eyes on screen and the right desktop. The camera will also be live. Pull up Eclipse (with okay code).
5. **Secondary offset:** Find what version of Eclipse is running (Go to help, about Eclipse)). Then have them read the one warning aloud.

C.1.4 Instructions for Participant

In this experiment, we are interested in how developers identify and resolve compiler defects.

In this experiment, you will be identifying and resolving 10 compiler defects, which are presented as compiler error messages. You'll get five minutes for each task. If you finish early, you may indicate this me and we can move to the next task. After two minutes, if you feel that you will not be able to complete this task, regardless of any additional time, you may also ask me to move on to the next task.

You should attempt to provide a reasonable solution for the defect that you feel best captures the intention of the code (for example, deleting all the files in the project might remove the compiler defect, but that is highly unlikely to be an acceptable resolution). Note that you are not expected to successfully fix all the defects, and that some defects may be more difficult than others.

As a limitation of the eye tracking equipment, please leave the Eclipse window full-screen, and do not use any resources (such as a web browser) outside of the Eclipse IDE. You may use any of the features available within Eclipse to help you with troubleshooting, but do not change any of the Eclipse preferences or install any new Eclipse packages.

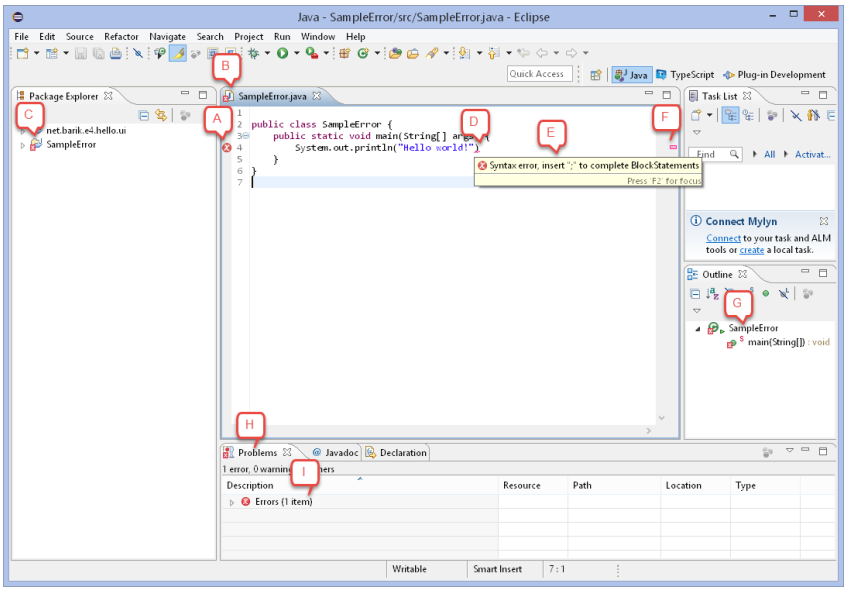


Figure C.1 Notifications sheet provided to participants to familiarize them with all notification sources in the Eclipse IDE.

[Show notifications sheet to user] (Figure C.1)

Do you have any questions at this time?

C.1.5 Post-study questionnaire

Give participant the questionnaire after they have completed the tasks.

C.1.6 Closing

You just took a study on how people understand and resolve compile error messages. At this time, you can also decline to participant in the study. Otherwise, do you have any questions?

C.2 Tasks

Faults are injected into Apache Commons 4.4.0. We present each fault using a unified `diff` format.[1]

[1] `http://www.gnu.org/software/diffutils/manual/html_node/Unified-Format.html`

C.2.1 Task 1: SUBLIST

C.2.1.1 Source Listings

```
@@ -134,5 +134,5 @@

     @Override
-    public List<E> subList(final int fromIndex, final int toIndex) {
+    public List<E> sublist(final int fromIndex, final int toIndex) {
         final List<E> sub = decorated().subList(fromIndex, toIndex);
         return new LazyList<E>(sub, factory);
```

Listing C.1 Injected fault in `LazyList.java`.

```
135     @Override
136     public List<E> sublist(final int fromIndex, final int toIndex) {
137         final List<E> sub = decorated().subList(fromIndex, toIndex);
138         return new LazyList<E>(sub, factory);
139     }
```

Listing C.2 Partial source listing for `LazyList.java` (in IDE).

C.2.1.2 Error Message

Description. `The method sublist(int, int) of type LazyList<E> must override or implement a supertype method`
Resource. `LazyList.java`
Path. `/commons-collections4/src/main/java/org/apache/commons/collections4/list`
Location. `line 136`
Type. `Java Problem`

C.2.2 Task 2: NODECACHE

C.2.2.1 Source Listings

```
@@ -107,4 +106,0 @@
-    public boolean isEmpty() {
-        return size() == 0;
-    }
-
```

Listing C.3 Injected fault in `AbstractLinkedList.java`.

```
58 public class CursorableLinkedList<E> extends AbstractLinkedList<E>
   ↪ implements Serializable {
59
60     /** Ensure serialization compatibility */
61     private static final long serialVersionUID = 8836393098519411393L;
62
63     /** A list of the cursor currently open on this list */
64     private transient List<WeakReference<Cursor<E>>> cursors;
```

Listing C.4 Partial source listing for `CursorableLinkedList.java` (in IDE).

```
42 public class NodeCachingLinkedList<E> extends AbstractLinkedList<E>
   ↪ implements Serializable {
43
44     /** Serialization version */
45     private static final long serialVersionUID = 6897789178562232073L;
46
47     /**
48      * The default value for {@link #maximumCacheSize}.
49      */
50     private static final int DEFAULT_MAXIMUM_CACHE_SIZE = 20;
```

Listing C.5 Partial source listing for `NodeCachingLinkedList.java` (in IDE).

C.2.2.2 Error Messages (2 Items)

Description. `The type CursorableLinkedList<E> must implement the inherited abstract method List<E>.isEmpty()`
Resource. `CursorableLinkedList.java`
Path. `/commons-collections4/src/main/java/org/apache/commons/collections4/list`

Location. line 58
Type. Java Problem

Description. The type NodeCachingLinkedList<E> must implement the inherited abstract method List<E>.isEmpty()
Resource. NodeCachingLinkedList.java
Path. /commons-collections4/src/main/java/org/apache/commons/collections4/list
Location. line 42
Type. Java Problem

C.2.3 Task 3: `IMPORT`

C.2.3.1 Source Listings

```
@@ -21,12 +21,12 @@
 import java.util.Map;
 import java.util.NoSuchElementException;

-import org.apache.commons.collections4.OrderedIterator;
-import org.apache.commons.collections4.OrderedMap;
-import org.apache.commons.collections4.OrderedMapIterator;
-import org.apache.commons.collections4.ResettableIterator;
-import org.apache.commons.collections4.iterators.EmptyOrderedIterator;
-import org.apache.commons.collections4.iterators.EmptyOrderedMapIterator;
+import org.apache.commons.collections3.OrderedIterator;
+import org.apache.commons.collections3.OrderedMap;
+import org.apache.commons.collections3.OrderedMapIterator;
+import org.apache.commons.collections3.ResettableIterator;
+import org.apache.commons.collections3.iterators.EmptyOrderedIterator;
+import org.apache.commons.collections3.iterators.EmptyOrderedMapIterator;

 /**
  * An abstract implementation of a hash-based map that links entries to create an
```

Listing C.6 Injected fault in `AbstractLinkedMap.java`.

```
22 import java.util.NoSuchElementException;
23
24 import org.apache.commons.collections3.OrderedIterator;
25 import org.apache.commons.collections3.OrderedMap;
26 import org.apache.commons.collections3.OrderedMapIterator;
27 import org.apache.commons.collections3.ResettableIterator;
28 import org.apache.commons.collections3.iterators.EmptyOrderedIterator;
29 import org.apache.commons.collections3.iterators.EmptyOrderedMapIterator;
```

Listing C.7 Partial source listing for `AbstractLinkedMap.java` (in IDE).

C.2.3.2 Error Messages (48 Items)

Description. `The import org.apache.commons.collections3 cannot be resolved`
Resource. `AbstractLinkedMap.java`
Path. `/commons-collections4/src/main/java/org/apache/commons/collections4/map`
Location. `line 24`

Type. Java Problem

Description. The import org.apache.commons.collections3 cannot be resolved
Resource. AbstractLinkedMap.java
Path. /commons-collections4/src/main/java/org/apache/commons/collections4/map
Location. line 25
Type. Java Problem

Description. The import org.apache.commons.collections3 cannot be resolved
Resource. AbstractLinkedMap.java
Path. /commons-collections4/src/main/java/org/apache/commons/collections4/map
Location. line 26
Type. Java Problem

Description. The import org.apache.commons.collections3 cannot be resolved
Resource. AbstractLinkedMap.java
Path. /commons-collections4/src/main/java/org/apache/commons/collections4/map
Location. line 27
Type. Java Problem

Description. The import org.apache.commons.collections3 cannot be resolved
Resource. AbstractLinkedMap.java
Path. /commons-collections4/src/main/java/org/apache/commons/collections4/map
Location. line 28
Type. Java Problem

Description. The import org.apache.commons.collections3 cannot be resolved
Resource. AbstractLinkedMap.java
Path. /commons-collections4/src/main/java/org/apache/commons/collections4/map
Location. line 29
Type. Java Problem

(...42 other secondary errors introduced by missing import statements across files (.java): AbstractLinkedMap, LRUMap, LRUMapTest, LinkedMap, and LinkedMapTest.)

C.2.4 Task 4: QUEUEGET

C.2.4.1 Source Listings

```
@@ -106,11 +106,11 @@
     public E peek() {
         return decorated().peek();
     }

     public E element() {
-        return decorated().element();
+        return decorated().get();
     }

     public E remove() {
         return decorated().remove();
     }
```

Listing C.8 Injected fault in `PredicatedQueue.java`.

```
110     public E element() {
111         return decorated().get();
112     }
113
114     public E remove() {
115         return decorated().remove();
116     }
```

Listing C.9 Partial source listing for `PredicatedQueue.java` (in IDE).

C.2.4.2 Error Message

Description. `The method get() is undefined for the type Queue<E>`
Resource. `PredicatedQueue.java`
Path. `/commons-collections4/src/main/java/org/apache/commons/collections4/queue`
Location. `line 111`
Type. `Java Problem`

C.2.5 Task 5: SETADD

C.2.5.1 Source Listing

```
@@ -82,3 +82,3 @@
             for (final E value : values) {
-                decorated.decorated().add(transformer.transform(value));
+                decorated.decorated().add(0, transformer.transform(value));
             }
```

Listing C.10 Injected fault in `TransformedSet.java`.

```
76      public static <E> Set<E> transformedSet(final Set<E> set, final
        ↪  Transformer<? super E, ? extends E> transformer) {
77          final TransformedSet<E> decorated = new TransformedSet<E>(set,
            ↪  transformer);
78          if (transformer != null && set != null && set.size() > 0) {
79              @SuppressWarnings("unchecked") // set is type E
80              final E[] values = (E[]) set.toArray(); // NOPMD - false
                ↪  positive for generics
81              set.clear();
82              for (final E value : values) {
83                  decorated.decorated().add(0, transformer.transform(value));
84              }
85          }
86          return decorated;
87      }
```

Listing C.11 Partial source listing for `TransformedSet.java` (in IDE).

C.2.5.2 Error Message

Description. `The method add(E) in the type Collection<E> is not applicable for the arguments (int, capture#8-of ? extends E)`
Resource. `TransformedSet.java`
Path. `/commons-collections4/src/main/java/org/apache/commons/collections4/set`
Location. `line 83`
Type. `Java Problem`

C.2.6 Task 6: KEYSETKV

C.2.6.1 Source Listing

```
@@ -144,3 +144,3 @@
     public Set<Map.Entry<K, V>> entrySet() {
-        final Set<Map.Entry<K, V>> set = map.entrySet();
+        final Set<Map.Entry<V, K>> set = map.entrySet();
         // unmodifiable set will still allow modification via Map.Entry objects
```

Listing C.12 Injected fault in `FixedSizeMap.java`.

```
143     @Override
144     public Set<Map.Entry<K, V>> entrySet() {
145         final Set<Map.Entry<V, K>> set = map.entrySet();
146         // unmodifiable set will still allow modification via Map.Entry objects
147         return UnmodifiableSet.unmodifiableSet(set);
148     }
```

Listing C.13 Partial source listing for `FixedSizeMap.java` (in IDE).

Error Messages (2 Items)

Description. `Type mismatch: cannot convert from Set<Map.Entry<K,V>> to Set<Map.Entry<V,K`
Resource. `FixedSizeMap.java`
Path. `/commons-collections4/src/main/java/org/apache/commons/collections4/map`
Location. `line 145`
Type. `Java Problem`

Description. `Type mismatch: cannot convert from Set<Map.Entry<V,K>> to Set<Map.Entry<K,V`
Resource. `FixedSizeMap.java`
Path. `/commons-collections4/src/main/java/org/apache/commons/collections4/map`
Location. `line 147`
Type. `Java Problem`

C.2.7 Task 7: CLAZZ

C.2.7.1 Source Listing

```
@@ -550,7 +550,7 @@
          public T create() {
              try {
                  return clazz.newInstance();
-             } catch (final Exception ex) {
+             } catch (final IllegalAccessException ex) {
                  throw new FunctorException("Cannot instantiate class: " + clazz, ex);
              }
          }
```

Listing C.14 Injected fault in `MultiValueMap.java`.

```
550          public T create() {
551              try {
552                  return clazz.newInstance();
553              } catch (final IllegalAccessException ex) {
554                  throw new FunctorException("Cannot instantiate class: " + clazz, ex);
555              }
556          }
```

Listing C.15 Partial source listing for `MultiValueMap.java` (in IDE).

C.2.7.2 Error Message

Description. `Unhandled exception type InstantiationException`
Resource. `MultiValueMap.java`
Path. `/commons-collections4/src/main/java/org/apache/commons/collections4/map`
Location. `line 552`
Type. `Java Problem`

C.2.8 Task 8: NEXT

C.2.8.1 Source Listing

```
@@ -152,6 +152,18 @@
          last = null;
          canRemove = false;
      }
+
+     /**
+      * Gets the next key from the <code>Map</code>.
+      *
+      * @return the next key in the iteration
+      * @throws java.util.NoSuchElementException
+      */
+     public K next() {
+         last = iterator.next();
+         canRemove = true;
+         return last.getKey();
+     }

      /**
       * Gets the iterator as a String.
```

Listing C.16 Injected fault in `EntrySetMapIterator.java`.

```
74      public K next() {
75          last = iterator.next();
76          canRemove = true;
77          return last.getKey();
78      }
79
80      //-----------------------------------------------------------------------
81      /**
82       * Removes the last returned key from the underlying <code>Map</code>.

162     public K next() {
163         last = iterator.next();
164         canRemove = true;
165         return last.getKey();
166     }
167
168     /**
169      * Gets the iterator as a String.
```

Listing C.17 Partial source listing for `EntrySetMapIterator.java` (in IDE).

C.2.8.2 Error Message

Description. Duplicate method next() in type EntrySetMapIterator<K,V>
Resource. EntrySetMapIterator.java
Path. /commons-collections4/src/main/java/org/apache/commons/collections4/iterators
Location. line 74
Type. Java Problem

Description. Duplicate method next() in type EntrySetMapIterator<K,V>
Resource. EntrySetMapIterator.java
Path. /commons-collections4/src/main/java/org/apache/commons/collections4/iterators
Location. line 162
Type. Java Problem

C.2.9 Task 9: READOBJSTATIC

C.2.9.1 Source Listing

```
@@ -94,3 +94,3 @@
     @SuppressWarnings("unchecked")
-    private void readObject(final ObjectInputStream in) throws ... {
+    private static void readObject(final ObjectInputStream in) throws ... {
         in.defaultReadObject();
```

Listing C.18 Injected fault in `UnmodifiableQueue.java`.

`throws IOException, ClassNotFoundException` has been truncated to `throws ...` due to page length limitations.

```
94      @SuppressWarnings("unchecked")
95      private static void readObject(final ObjectInputStream in) throws
         ↪  IOException, ClassNotFoundException {
96          in.defaultReadObject();
97          setCollection((Collection<E>) in.readObject());
98      }
```

Listing C.19 Partial source listing for `UnmodifiableQueue.java` (in IDE).

C.2.9.2 Error Message

Description. `Cannot make a static reference to the non-static type E`
Resource. `UnmodifiableQueue.java`
Path. `/commons-collections4/src/main/java/org/apache/commons/collections4/queue`
Location. `line 97`
Type. `Java Problem`

C.2.10 Task 10: SWITCH

C.2.10.1 Source Listing

```
@@ -1241,3 +1241,3 @@
              // case 0: has already been dealt with
-             default:
+             default case 0:
                  throw new IllegalStateException("Invalid map index: " + size);
```

Listing C.20 Injected fault in `Flat3Map.java`.

```
1236              case 1:
1237                  buf.append(key1 == this ? "(this Map)" : key1);
1238                  buf.append('=');
1239                  buf.append(value1 == this ? "(this Map)" : value1);
1240                  break;
1241              // case 0: has already been dealt with
1242              default case 0:
1243                  throw new IllegalStateException("Invalid map index: " + size);
```

Listing C.21 Partial source listing for `Flat3Map.java` (in IDE).

C.2.10.2 Error Message

Description. `Syntax error on token "default", : expected after this token`
Resource. `Flat3Map.java`
Path. `/commons-collections4/src/main/java/org/apache/commons/collections4/map`
Location. `line 1242`
Type. `Java Problem`

C.3 Post-study Questionnaire

Post-questionnaire

Participant ID: _____

What is your gender?

[] Male [] Female

What is your age? _____

How knowledgeable are you about the Java programming language?
[] Not knowledgeable about Java
[] Somewhat knowledgeable about Java
[] Knowledgeable about Java
[] Very knowledgeable about Java

How familiar are you with Eclipse?

[] Not familiar with Eclipse
[] Somewhat familiar with Eclipse
[] Familiar with Eclipse
[] Very familiar with Eclipse

How many months and years of industry programming experience do you have?
___ years ___ months

D | Study Materials for "How Do Developers Visualize Compiler Error Messages?" (Chapter 5)

D.1 Interview Protocol

D.1.1 Pre-tasks

1. For this experiment, we will use the following outline: IRB (2 minutes), Questionnaire (2 minutes), Task 1 (25 minutes), Survey (5 minutes), Task 2 (25 minutes).
2. Give participant IRB Consent Form.
3. Give participant Demographics Questionnaire.
4. Start audio and state: "Participant _____. You indicated that we may _____" (optional elected for recording). Get confirmation. Write this number down somewhere.

D.1.2 Task 1

1. Have printout of all tasks, for either control or treatment group.
2. Give them the tasks alphabetically: Apple, Brick, Kite, Melon, Trumpet, Zebra.

D.1.2.1 Instructions to Participant

For the first task, you will be examining six screens that are representative of an IDE. On the top of the screen, you will see a source file. Each source file has one or more errors. Below the source file, you will see the compiler output of the error. The source file also has visual annotations to assist with the error message. In your IDE, you will see visualizations as shown in your Legend file (if in treatment). [Take 1 minute to look over this].

You'll have 30 seconds to just read over the message. You may or may understand every message, and that's okay too. Then, to the best of your ability you will explain what you think the problem is to me. At the same time, I'd like to use the provided sheet to draw and visually annotate your explanation **on top of the source code**. As a guideline, if you find yourself pointing to something that might be worth annotating in some way. You will have at most 2 minutes to give your explanation.

At the end of each exercise, you will have two short questions on your perceptions about the problem.

D.1.3 Questionnaire

You will now take a short survey (5 minutes) that evaluates the visual annotations that you've just seen. This survey introduces some new concepts so please read it carefully. You can use the source code and your notes to help with completing this evaluation.
[Take materials from participant.]

If you need a break, you can have one now.

D.1.4 Task 2

1. In the ActivePresenter recording software, start recording.
2. Turn off the toolbar so the participant can't accidentally close it.
3. When ready to start, type `begin.bat` to prepare source code capture. You can go ahead and do this before giving the instructions.

In this task, you will write 6 programs on a computer. You will have five minutes for each problem. You will be given an expected compiler error message that you've already seen in Task 1. Unlike before, you will now have to write source code to

generate a particular error message. For this exercise, you will work on your own. You should generate an error message that is close to the provided error message, though there may be some small variations in line number. You will not need to create an additional files to solve this problem.

[Show them the interface]. Explain the compile command. Do not use any other commands. Remind them to save.

When you're done, ask the investigator to check your answer. The rules of this experiment don't allow me to provide you with assistance on this problem, but if something unexpected breaks, feel free to ask.

D.1.5 Wrap-up

You just took a study on how people understand messages. At this time, you can also decline to participant in the study. Otherwise, do you have any questions?

D.2 Questionnaire

```
All questions are optional.

What is your gender?
[] Male     [] Female

What is your age?
_______

How knowledgeable are you about the Java programming language?
[] Not knowledgeable about Java
[] Somewhat knowledgeable about Java
[] Knowledgeable about Java
[] Very knowledgeable about Java

What Integrated Development Environment (IDE) or text editor do you
primarily use when programming in Java?


_____________________________________
```

How many months and years of industry programming experience do you have?

___ years ___ months

How would you rate your overall programming ability?
[] Novice
[] Intermediate
[] Advanced
[] Expert

D.3 Visual Markings Cheatsheet

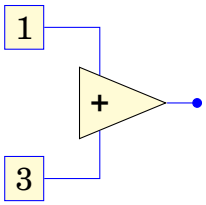
Symbol	Description
code	The starting location of the error.
code	Indicates issues related to the error.
	Arrows can be followed. They indicate the next relevant location to check.
❶	Enumerations are used to number items of potential interest, especially when the information doesn't fit within the source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

D.4 Dimensions Survey for All Six Tasks

For all of the questions below, **1 = lowest** and **5 = highest** (circle one).

Hidden Dependencies

A hidden dependency is a relationship between two components such that one of them is dependent on the other, but that dependency is not fully visible. For example, Excel spreadsheets often contain cells are referenced from other sheets and so you can't easily tell if modifying a given cell will or will not have an impact elsewhere (many hidden dependencies). The LabView programming language (shown below) makes data flow explicit, and so there are fewer hidden dependencies:



On a scale of 1-5, how prevalent are these dependencies in the annotations you just evaluated?

1 2 3 4 5

Consistency

Things are consistent when similar semantics are expressed in similar syntactic forms, that is, things that look similar either behave similarly or mean similar things. For example, the green triangle that means "play" is a nearly universal signal, so it is very consistent. The grammar rules for English have lots of exceptions and irregular words, so they are less consistent.

On a scale of 1-5, how consistent were the annotations you just evaluated?

1 2 3 4 5

Hard Mental Operations

When it feels like you have to juggle many things in your head to keep things straight or to properly use something, that is an indication of hard mental operations. For example, if you had to file your taxes without being able to write anything down

except your final amounts, doing taxes would require many hard mental operations. If you use tax-assistant software that keeps track of the intermediate steps for you and makes sure the correct boxes are filled, there are fewer hard mental operations.

On a scale of 1-5, to what extent did the annotations require hard mental operations?

1 2 3 4 5

Role Expressiveness

A system with high role expressiveness has an intuitive design and feel - it is easy to tell why the respective design decisions were chosen. For example, a well organized machine shop has all the supplies and tools for a given task in the same spot - painting supplies on one table, cutting machines near each other, drill bits next to the drill, etc. The QWERTY keyboard layout has been criticized for having low role expressiveness because of the scattered keys and a lack of cohesive grouping.

On a scale of 1-5, how was the role expressiveness of the annotations you saw previously?

1 2 3 4 5

D.5 Pages

The following pages contain the material forms that the participants get.

D.5.1 Explanatory Visualizations

Screen Listing for Apple.java

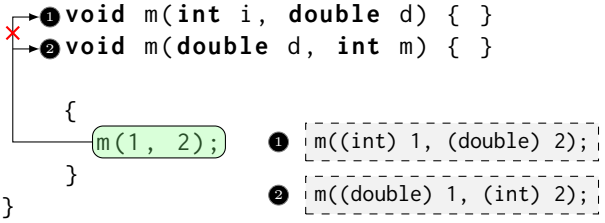
```
1    class Apple {
2        public public String toString() {
3            return "Red";
4        }
5    }
```

Compiler Output

```
Apple.java:2: error: repeated modifier
    public public String toString() {
           ^
1 error
```

Screen Listing for Brick.java

```
1    class Brick {
2        void m(int i, double d) { }
3        void m(double d, int m) { }
4
5        {
6            m(1, 2);
7        }
8    }
```



Compiler Output

```
Brick.java:6: error: reference to m is ambiguous,
both method m(int,double) in Brick and method m(double,int) in Brick match
      m(1, 2);
      ^
1 error
```

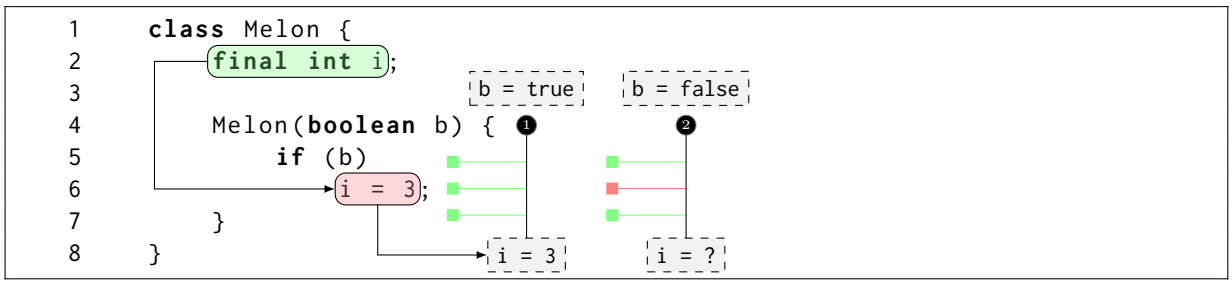
Screen Listing for Kite.java

```
1    class Toy {
2        Toy() throws Exception {}
3    }
4
5    class Kite extends Toy {
         Kite() { super(); }
6    }
```

Compiler Output

```
Kite.java:5: error: unreported exception Exception in default constructor
class Kite extends Toy {
^
1 error
```

Screen Listing for Melon.java



Compiler Output

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

Screen Listing for Trumpet.java

```
 1    import java.io.*;
 2
 3    class Trumpet {
 4
 5        void play() {
 6            try {
 7                if (true) {
 8                    throw new FileNotFoundException();
 9                }
10                else {
11                    throw new EOFException();
12                }
13            }
14            catch(FileNotFoundException fnf) { }
15            catch(EOFException eof) { }
16            catch(IOException ex) { }
17        }
18    }
```

Compiler Output

```
Trumpet.java:16: warning: unreachable catch clause
        catch(IOException ex) { }
        ^
  thrown types FileNotFoundException,EOFException have already been caught
1 warning
```

Screen Listing for Zebra.java

```
1    class Zebra {
2       static class Stripe<X> {}
3
4       static class BlackStripe<X extends Number> extends Stripe<X> {
5            BlackStripe(X x) {}
6       }
7
8       Stripe<String> sf1 = new BlackStripe<>("Marty");
9     }
```

Compiler Output

```
Zebra.java:8: error: cannot infer type arguments for BlackStripe<>;
   Stripe<String> sf1 = new BlackStripe<>("Marty");
                        ^
  reason: inferred type does not conform to declared bound(s)
    inferred: String
    bound(s): Number
1 error
```

D.5.2 Baseline Visualizations

Screen Listing for Apple.java

```
1 class Apple {
2     public public String toString() {
3       return "Red";
4     }
5 }
```

Compiler Output

```
Apple.java:2: error: repeated modifier
    public public String toString() {
           ^
1 error
```

Screen Listing for Brick.java

```
1 class Brick {
2     void m(int i, double d) { }
3     void m(double d, int m) { }
4
5     {
6       m(1, 2);
7     }
8 }
```

Compiler Output

```
Brick.java:6: error: reference to m is ambiguous,
both method m(int,double) in Brick and method m(double,int) in Brick match
      m(1, 2);
      ^
1 error
```

Screen Listing for Kite.java

```
1 class Toy {
2     Toy() throws Exception { }
3 }
4
5 class Kite extends Toy {
6 }
```

Compiler Output

```
Kite.java:5: error: unreported exception Exception in default constructor
class Kite extends Toy {
^
1 error
```

Screen Listing for Melon.java

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

Compiler Output

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

Screen Listing for Trumpet.java

```
1  import java.io.*;
2
3  class Trumpet {
4
5      void play() {
6          try {
7              if (true) {
8                  throw new FileNotFoundException();
9              }
10             else {
11                 throw new EOFException();
12             }
13         }
14         catch(FileNotFoundException fnf) { }
15         catch(EOFException eof) { }
16         catch(IOException ex) { }
17     }
18 }
```

Compiler Output

```
Trumpet.java:16: warning: unreachable catch clause
        catch(IOException ex) { }
        ^
  thrown types FileNotFoundException,EOFException have already been caught
1 warning
```

Screen Listing for Zebra.java

```
1 class Zebra {
2    static class Stripe<X> {}
3
4    static class BlackStripe<X extends Number> extends Stripe<X> {
5        BlackStripe(X x) {}
6    }
7
8    Stripe<String> sf1 = new BlackStripe<>("Marty");
9 }
```

Compiler Output

```
Zebra.java:8: error: cannot infer type arguments for BlackStripe<>;
   Stripe<String> sf1 = new BlackStripe<>("Marty");
                        ^
  reason: inferred type does not conform to declared bound(s)
    inferred: String
    bound(s): Number
1 error
```

D.5.3 Printed

Annotations for Apple.java

```
1   class Apple {

2       public public String toString() {

3         return "Red";

4       }

5   }
```

Compiler Output

```
Apple.java:2: error: repeated modifier
    public public String toString() {
           ^
1 error
```

Questionnaire

1. Have you ever encountered this error message before?
[ ] Yes [ ] No [ ] Unsure

2. How confident are you about the accuracy of your explanation for this error message?

[ ] Not at all confident
[ ] Somewhat confident
[ ] Moderately confident
[ ] Highly confident
[ ] Completely confident

Annotations for Brick.java

```
1   class Brick {
2       void m(int i, double d) { }
3       void m(double d, int m) { }
4
5       {
6         m(1, 2);
7       }
8   }
```

Compiler Output

```
Brick.java:6: error: reference to m is ambiguous,
both method m(int,double) in Brick and method m(double,int) in Brick match
      m(1, 2);
      ^
1 error
```

Questionnaire

1. Have you ever encountered this error message before?
[ ] Yes [ ] No [ ] Unsure

2. How confident are you about the accuracy of your explanation for this error message?

[ ] Not at all confident
[ ] Somewhat confident
[ ] Moderately confident
[ ] Highly confident
[ ] Completely confident

Annotations for Kite.java

```
1   class Toy {

2       Toy() throws Exception { }

3   }

4

5   class Kite extends Toy {

6   }
```

Compiler Output

```
Kite.java:5: error: unreported exception Exception in default constructor
class Kite extends Toy {
^
1 error
```

Questionnaire

1. Have you ever encountered this error message before?
[ ] Yes [ ] No [ ] Unsure

2. How confident are you about the accuracy of your explanation for this error message?

[ ] Not at all confident
[ ] Somewhat confident
[ ] Moderately confident
[ ] Highly confident
[ ] Completely confident

Annotations for Melon.java

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

Compiler Output

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

Questionnaire

1. Have you ever encountered this error message before?
[ ] Yes [ ] No [ ] Unsure

2. How confident are you about the accuracy of your explanation for this error message?

[ ] Not at all confident
[ ] Somewhat confident
[ ] Moderately confident
[ ] Highly confident
[ ] Completely confident

Annotations for Trumpet.java

```
1   import java.io.*;
2
3   class Trumpet {
4
5       void play() {
6           try {
7               if (true) {
8                   throw new FileNotFoundException();
9               }
10              else {
11                  throw new EOFException();
12              }
13          }
14          catch(FileNotFoundException fnf) { }
15          catch(EOFException eof) { }
16          catch(IOException ex) { }
17      }
18  }
```

Compiler Output

```
Trumpet.java:16: warning: unreachable catch clause
        catch(IOException ex) { }
        ^
  thrown types FileNotFoundException,EOFException have already been caught
1 warning
```

Questionnaire

1. Have you ever encountered this error message before?
[ ] Yes [ ] No [ ] Unsure

2. How confident are you about the accuracy of your explanation for
this error message?

[ ] Not at all confident
[ ] Somewhat confident
[ ] Moderately confident
[ ] Highly confident
[ ] Completely confident

Annotations for Zebra.java

```
1  class Zebra {
2      static class Stripe<X> {}
3
4      static class BlackStripe<X extends Number> extends Stripe<X> {
5          BlackStripe(X x) {}
6      }
7
8      Stripe<String> sf1 = new BlackStripe<>("Marty");
9  }
```

Compiler Output

```
Zebra.java:8: error: cannot infer type arguments for BlackStripe<>;
   Stripe<String> sf1 = new BlackStripe<>("Marty");
                        ^
  reason: inferred type does not conform to declared bound(s)
    inferred: String
    bound(s): Number
1 error
```

Questionnaire

1. Have you ever encountered this error message before?
[ ] Yes [ ] No [ ] Unsure

2. How confident are you about the accuracy of your explanation for this error message?

[ ] Not at all confident
[ ] Somewhat confident
[ ] Moderately confident
[ ] Highly confident
[ ] Completely confident

E | Error Message Design Guidelines

Table E.1 Chronological Summary of Guidelines for Designing Error Messages

Reference	Guidelines
Moulton and Muller (1967)	All errors other than logical errors are to be detected and described to the programmer All compilation and execution diagnostic messages and descriptions of errors are to be in terms of the source language The formation of error messages and the analysis of errors are to be made in such a way that they will provide the user with as much information as possible, giving him direct cues aiding the correction of errors As many errors as possible are to be detected during compilation Provision is to be made for the use of diagnostic routines for tracing control of execution of a program and auditing the assignment of values to variables
Horning (1974)	User-directed Source-oriented

Reference	Guidelines
	Specific Localize the problem Complete Readable Restrained and polite
Dean (1982)	Set human goals for messages: Be tolerant of "user errors" Help people correct errors as easily as they make them Give people control over the messages they receive Do not make messages arbitrarily short Identify messages that people need Apply psychology in writing messages: Anticipate people's expectations Help people fit the pieces together Do not force people to re-read Put people at ease Write messages for the audience and the situation Report on the program's reaction to the input Report on the program's assumption about input Report on a program error or adverse condition Request for a go-ahead Request to choose among alternatives Request for missing information Request for correction or clarification of input Edit the messages for appropriate language, using: good writing, vocabulary that is familiar, standard conversational language, consistent messages, and standard punctuation.

Reference	Guidelines
Shneiderman (1982)	Have a positive tone indicating what must be done Be specific and address the problem in the user's terms Place the user in control of the situation Have a neat, consistent, and comprehensible format
Brown (1983)	Exploit the capabilities of the display (e.g., color, reverse-video) to identify the offending symbol Print several lines of the source, both before and after the point of error, to supply a contextual window Allow the user the option of increasing the size of the contextual window Provide some visual scale to show where in the program the error occurred Integrate with an editing facility to correct the source
Kantorowitz and Laor (1986)	A message that proposes how to correct an encountered error is most useful, but should only be produced when there is a high degree of certainty for its correctness For errors that may be corrected in more than one way no attempt should be made to guess which of them is the right one Pieces of code that the compiler cannot analyse correctly because of an error should be underlined. The programmer will then know that unreported errors may exist in these unchecked pieces of code The error messages should reflect a simple error handling mechanism that the programmer may readily understand
Shaw (1989)	The nature of the error is stated

Reference	Guidelines
	The errant data are identified Corrective action is prescribed
Natl. Cryptologic School (1990)	Design error messages for the intended users Write error messages that are specific and constructive Avoid anthropomorphism Write error messages using a positive tone Put the main idea first Make error messages timely Be consistent in grammatical form, terminology, abbreviation, and visual format and placement Provide multiple levels of error messages
Traver (2010)	Clarity & brevity Specificity Context-insensitivity Locality Proper phrasing: Positive tone Constructive guidance Programmer language
Murphy-Hill and Black (2012)	Expressiveness Locatability Completeness Estimability Relationality Perceptibility Distinguishability
Murphy-Hill, Barik, and Black (2013)	Restraint Relationality

Reference	Guidelines
	Partiality Nondistracting Estimability Availability Unobtrusiveness Context-sensitivity Lucidity
Sadowski, Gogh, Jaspan, Soderberg, and Winter (2015)	The error message should be easy to understand and the fix should be clear The error message should have very few false positives The error message should be for something that has the potential for significant impact The error message should occur with a small but noticeable frequency. There is no point in detecting errors that never actually occur

F | Error Message Samples

This appendix contains samples of concrete error messages for the categories of program analysis tools presented in Chapter 3.

F.1 Template Diagnostic

F.2 Python

Python [375] is an interpreted programming language. A notable characteristic of the language is that whitespace is significant. An important design philosophy of the language is readability, as described in the Zen of Python [285].

Here's an example of a Python 3 "Hello world", containing a missing parentheses around `print`:

```
>>> print "Hello world"
  File "<stdin>", line 1
    print "Hello world"
                       ^
SyntaxError: Missing parentheses in call to 'print'. Did you mean
↪  print("Hello world")?
```

This example is executed in the Python REPL. An alternative REPL is `ipython`, which colorizes the error output:

```
In [1]: print "Hello world"
  File "<ipython-input-1-3c090b498326>", line 1
    print "Hello world"
                       ^
SyntaxError: Missing parentheses in call to 'print'. Did you mean
↪  print("Hello world")?
```

F.2.1 Eclipse Compiler for Java

Consider the following source file:

```
1 class Brick {
2     void m(int i, double d) { }
3     void m(double d, int m) { }
4 
5     {
6       m(1, 2);
7     }
8 }
```

The output of Oracle JDK:

```
Brick.java:6: error: reference to m is ambiguous
      m(1, 2);
      ^
  both method m(int,double) in Brick and method m(double,int) in Brick match
1 error
```

The output of Eclipse Compiler for Java (ecj):

```
----------
1. ERROR in Brick.java (at line 6)
        m(1, 2);
        ^
The method m(int, double) is ambiguous for the type Brick
----------
1 problem (1 error)
```

The output of the IBM Jikes compilers:

```
Found 1 semantic error compiling "Brick.java":

     6.       m(1, 2);
              ^-----^
*** Semantic Error: Ambiguous invocation of method "m". At least two methods
 ↪  are accessible from here: "void m(int i, double d);" declared in type
 ↪  "Brick" and "void m(double d, int m);" declared in type "Brick".
```

F.2.2 Infer

Infer [169] is a static program analysis tool for Java, C, and Objective-C. The tool is based on academic research on compositional shape analysis [52]. Essentially, shape analysis is a form of program analysis that attempts to infer descriptions of the data structure. The analysis rests on a generalized form of abduction, or inference of explanatory hypotheses.

For example, given the following source file:

```
/*
 * Copyright (c) 2015 - present Facebook, Inc.
 * All rights reserved.
 *
 * This source code is licensed under the BSD style license found in the
 * LICENSE file in the root directory of this source tree. An additional grant
 * of patent rights can be found in the PATENTS file in the same directory.
 */

class Hello {
  int test() {
    String s = null;
    return s.length();
  }
}
```

Infer will detect a possible `null` dereference:

```
Hello.java:13: error: NULL_DEREFERENCE
  object `s` last assigned on line 12 could be null and is dereferenced at
   ↪  line 13.
  11.     int test() {
  12.       String s = null;
  13. >     return s.length();
  14.     }
  15.   }

Summary of the reports

  NULL_DEREFERENCE: 1
```

F.2.3 Dafny

Dafny [210] uses verification-condition generation to present error messages as a template diagnostic. The developer interacts with Dafny in much the same way as they would a compiler. For example, consider the following Dafny program:

```
function Fibonacci(n: int): int
  decreases n
{
  if n < 2 then n else Fibonacci(n+2) + Fibonacci(n+1)
}
```

Dafny identifies a violation in the program given the indicated `decreases n` termination measure:

```
stdin.dfy(4,23): Error: failure to decrease termination
measure

verifier finished with 0 verified, 1 error
```

F.3 Extended Explanations

F.3.1 Error Prone

The Error Prone [114] tool adds checks to the existing Java compilation pipeline. Consider the following Java source file:

```
public class ShortSet {
  public static void main (String[] args) {
    Set<Short> s = new HashSet<>();
    for (short i = 0; i < 100; i++) {
      s.add(i);
      s.remove(i - 1);
    }
    System.out.println(s.size());
  }
}
```

The error produced by Error Prone for the above code listing is:

```
ShortSet.java:6: error: [CollectionIncompatibleType] Argument 'i - 1'
should not be passed to this method; its type int is not compatible with its
collection's type argument Short
      s.remove(i - 1);
               ^
    (see http://errorprone.info/bugpattern/CollectionIncompatibleType)
1 error
```

Notably, the error message provides an additional link with extended explanations for the problem.

F.3.2 Rust

Rust is a systems programming language sponsored by Mozilla Research [233]. The Rust development team has made significant investments in providing user-friendly error messages. Consider the following Rust program:

```
fn foo() {
}

fn foo() {
}
```

In Rust, the following error is emitted:

```
error[E0428]: the name `foo` is defined multiple times
 --> hello.rs:4:1
  |
1 | fn foo() {
  | -------- previous definition of the value `foo` here
...
4 | fn foo() {
  | ^^^^^^^^ `foo` redefined here
  |
  = note: `foo` must be defined only once in the value namespace of this module
```

If additional explanation is needed, the developer can invoke the `--explain` flag with the `E0428` error code. Rust will then present the following extended explanation:

```
A type or module has been defined more than once.

Erroneous code example:

```
struct Bar;
struct Bar; // error: duplicate definition of value `Bar`
```

Please verify you didn't misspell the type/module's name or remove/rename the
duplicated one. Example:

```
struct Bar;
struct Bar2; // ok!
```
```

F.4 Type Errors

F.4.1 elm

The designers of the Elm compiler emphasize usable error messages. Consider the following source code:

```
import Html exposing (..)
import Html.Attributes exposing (..)

alice =
  img [src "/users/alice/pic"] []

bob =
  img = [src "/users/bob/pic"] []

userPics =
  [ alice, bob, "/users/chuck/pic" ]
```

The error message from this source listing is as follows:

```
-- TYPE MISMATCH ----------------------------------------- types/list.elm

The 3rd element of this list is an unexpected type of value.

15|   [ alive, bob, "/users/chuck/pic" ]

All elements should be the same type of value so that we can iterate over the
list without running into unexpected values.

As I infer the type of values flowing through your program, I see a conflict
between these two types:

    Html

    String
```

F.4.2 Helium

Helium is an educational compiler platform that implements a subset of the Haskell programming language [157]. Consider the following Haskell program:

```
myFilter :: (a -> Bool) -> [a] -> [a]
myFilter p [] = [] myFIlter p (x:xs) =
        if p x
         then x : myFilter p xs
         else myFilter p xs
```

The above program is legal Haskell, but Helium identifies potentially dangerous constructions:

```
(4,1): Warning: Tab character encountered;
       may cause problems with the layout rule
  Hint: Configure your editor to replace tabs by spaces
(3,1) : Warning: Missing type signature:
        myFIlter :: (a -> Bool) -> [a] -> [a]
(2,10): Warning: Variable "p" is not used
(2,1), (3,1): Warning: Suspicious adjacent functions
                       "myFilter" and "myFIlter"
```

F.5 Examples and Counterexamples

F.5.1 CBMC

CBMC is a bounded model checking tool for C, C++, and Java. Consider the follow C program, which has a faulty array access:

```
int main(int argc, char **argv) {
  puts(argv[2]);
}
```

Specifically, the argument count `argc` must be greater than or equal to three, otherwise the argument located in `argv[2]` is uninitialized. That is, if the command line is `gcc -o myprog myprog.c`, then `argc = 4`,

```
CBMC version 5.6 64-bit x86_64 linux
Parsing file1.c
file <command-line> line 0: <command-line>:0:0: warning: "__STDC_VERSION__"
 ↪  redefined
<built-in>: note: this is the location of the previous definition
Converting
Type-checking file1
file file1.c line 2 function main: function `puts' is not declared
Generating GOTO Program
Adding CPROVER library (x86_64)
file <command-line> line 0: <command-line>:0:0: warning: "__STDC_VERSION__"
 ↪  redefined
<built-in>: note: this is the location of the previous definition
file <builtin-library-puts> line 7: warning: implicit function declaration
 ↪  "puts"
old definition in module file1 file file1.c line 2 function main
signed int (void)
new definition in module <built-in-library> file <builtin-library-puts> line
 ↪  7
signed int (const char *s)
Removal of function pointers and virtual functions
Partial Inlining
Generic Property Instrumentation
Starting Bounded Model Checking
size of program expression: 54 steps
simple slicing removed 17 assignments
Generated 7 VCC(s), 2 remaining after simplification
Passing problem to propositional reduction
converting SSA
Running propositional reduction
Post-processing
Solving with MiniSAT 2.2.1 with simplifier
898 variables, 2390 clauses
SAT checker: instance is SATISFIABLE
Solving with MiniSAT 2.2.1 with simplifier
898 variables, 0 clauses
SAT checker inconsistent: instance is UNSATISFIABLE
Runtime decision procedure: 0.004s

** Results:
[main.overflow.1] pointer arithmetic overflow on + in argv + (signed long
 ↪  int)2: SUCCESS
[main.pointer_dereference.1] dereference failure: pointer NULL in
 ↪  argv[(signed long int)2]: SUCCESS
[main.pointer_dereference.2] dereference failure: pointer invalid in
 ↪  argv[(signed long int)2]: SUCCESS
[main.pointer_dereference.3] dereference failure: deallocated dynamic object
 ↪  in argv[(signed long int)2]: SUCCESS
```

```
[main.pointer_dereference.4] dereference failure: dead object in argv[(signed long int)2]: SUCCESS
[main.pointer_dereference.5] dereference failure: dynamic object bounds in argv[(signed long int)2]: SUCCESS
[main.pointer_dereference.6] dereference failure: object bounds in argv[(signed long int)2]: FAILURE

Trace for main.pointer_dereference.6:

State 18  thread 0
---------------------------------------------------
  INPUT argc: 1 (00000000000000000000000000000001)

State 19  thread 0
---------------------------------------------------
  argv'[1]=((char *)NULL)
   (0000000000000000000000000000000000000000000000000000000000000000)

State 22 file file1.c line 1 thread 0
---------------------------------------------------
  argc=1 (00000000000000000000000000000001)

State 23 file file1.c line 1 thread 0
---------------------------------------------------
  argv=argv'
   (0000001000000000000000000000000000000000000000000000000000000000)

Violated property:
  file file1.c line 2 function main
  dereference failure: object bounds in argv[(signed long int)2]
  16l + POINTER_OFFSET(argv) >= 0 && OBJECT_SIZE(argv) >= 24 +
   POINTER_OFFSET(argv) || DYNAMIC_OBJECT(argv)


** 1 of 7 failed (2 iterations)
VERIFICATION FAILED
```

```
int main(int argc, char **argv) {
  puts(argv[2]);
}
```

```
CBMC version 5.6 64-bit x86_64 linux
Parsing file1-fix.c
file <command-line> line 0: <command-line>:0:0: warning: "__STDC_VERSION__" redefined
<built-in>: note: this is the location of the previous definition
Converting
Type-checking file1-fix
file file1-fix.c line 3 function main: function `puts' is not declared
```

```
Generating GOTO Program
Adding CPROVER library (x86_64)
file <command-line> line 0: <command-line>:0:0: warning: "__STDC_VERSION__"
 ↪  redefined
<built-in>: note: this is the location of the previous definition
file <builtin-library-puts> line 7: warning: implicit function declaration
 ↪  "puts"
old definition in module file1-fix file file1-fix.c line 3 function main
signed int (void)
new definition in module <built-in-library> file <builtin-library-puts> line
 ↪  7
signed int (const char *s)
Removal of function pointers and virtual functions
Partial Inlining
Generic Property Instrumentation
Starting Bounded Model Checking
size of program expression: 59 steps
simple slicing removed 20 assignments
Generated 7 VCC(s), 2 remaining after simplification
Passing problem to propositional reduction
converting SSA
Running propositional reduction
Post-processing
Solving with MiniSAT 2.2.1 with simplifier
933 variables, 2524 clauses
SAT checker: instance is UNSATISFIABLE
Runtime decision procedure: 0.003s

** Results:
[main.overflow.1] pointer arithmetic overflow on + in argv + (signed long
 ↪  int)2: SUCCESS
[main.pointer_dereference.1] dereference failure: pointer NULL in
 ↪  argv[(signed long int)2]: SUCCESS
[main.pointer_dereference.2] dereference failure: pointer invalid in
 ↪  argv[(signed long int)2]: SUCCESS
[main.pointer_dereference.3] dereference failure: deallocated dynamic object
 ↪  in argv[(signed long int)2]: SUCCESS
[main.pointer_dereference.4] dereference failure: dead object in
 ↪  argv[(signed long int)2]: SUCCESS
[main.pointer_dereference.5] dereference failure: dynamic object bounds in
 ↪  argv[(signed long int)2]: SUCCESS
[main.pointer_dereference.6] dereference failure: object bounds in
 ↪  argv[(signed long int)2]: SUCCESS

** 0 of 7 failed (1 iteration)
```

```
void f(int a, int b, int c) {
        int temp;
        if (a > b) {temp = a; a = b; b = temp;}
```

```
        if (b > c) {temp = b; b = c; c = temp;}
        if (a < b) {temp = a; a = b; b = temp;}
        assert (a<=b && b<=c);
}
```

```
CBMC version 5.6 64-bit x86_64 linux
Parsing file2.c
file <command-line> line 0: <command-line>:0:0: warning: "__STDC_VERSION__"
 ↪  redefined
<built-in>: note: this is the location of the previous definition
Converting
Type-checking file2
file file2.c line 7 function f: function `assert' is not declared
Generating GOTO Program
Adding CPROVER library (x86_64)
Removal of function pointers and virtual functions
Partial Inlining
Generic Property Instrumentation
Starting Bounded Model Checking
size of program expression: 63 steps
simple slicing removed 2 assignments
Generated 1 VCC(s), 1 remaining after simplification
Passing problem to propositional reduction
converting SSA
Running propositional reduction
Post-processing
Solving with MiniSAT 2.2.1 with simplifier
867 variables, 3180 clauses
SAT checker: instance is SATISFIABLE
Runtime decision procedure: 0.005s

** Results:
[f.assertion.1] assertion a <= b && b <= c: FAILURE

Trace for f.assertion.1:

State 17 file file2.c line 1 thread 0
------------------------------------------------------
  INPUT a: 124955 (00000000000000011110100000011011)

State 19 file file2.c line 1 thread 0
------------------------------------------------------
  INPUT b: 256027 (00000000000000111110100000011011)

State 21 file file2.c line 1 thread 0
------------------------------------------------------
  INPUT c: 124954 (00000000000000011110100000011010)

State 24 file file2.c line 1 thread 0
```

```
44 ------------------------------------------------
45   a=124955 (00000000000000011110100000011011)
46
47 State 25 file file2.c line 1 thread 0
48 ------------------------------------------------
49   b=256027 (00000000000000111110100000011011)
50
51 State 26 file file2.c line 1 thread 0
52 ------------------------------------------------
53   c=124954 (00000000000000011110100000011010)
54
55 State 27 file file2.c line 2 function f thread 0
56 ------------------------------------------------
57   temp=0 (00000000000000000000000000000000)
58
59 State 30 file file2.c line 4 function f thread 0
60 ------------------------------------------------
61   temp=256027 (00000000000000111110100000011011)
62
63 State 31 file file2.c line 4 function f thread 0
64 ------------------------------------------------
65   b=124954 (00000000000000011110100000011010)
66
67 State 32 file file2.c line 4 function f thread 0
68 ------------------------------------------------
69   c=256027 (00000000000000111110100000011011)
70
71 Violated property:
72   file file2.c line 7 function f
73   assertion a <= b && b <= c
74   a <= b && b <= c
75
76
77 ** 1 of 1 failed (1 iteration)
78 VERIFICATION FAILED
```

F.5.2 Java Pathfinder

Java Pathfinder (JPF) is a system to verify executable Java bytecode programs. The model checker in JPF uses the standard output from the source program to generate a trace. This is shown in the below example:[1]

```
/*
 * Copyright (C) 2014, United States Government, as represented by the
 * Administrator of the National Aeronautics and Space Administration.
```

[1] https://babelfish.arc.nasa.gov/trac/jpf/wiki/intro/random_example

```
 * All rights reserved.
 *
 * The Java Pathfinder core (jpf-core) platform is licensed under the
 * Apache License, Version 2.0 (the "License"); you may not use this file
 * except in compliance with the License. You may obtain a copy of the
 * License at
 *
 *        http://www.apache.org/licenses/LICENSE-2.0.
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 */

import java.util.Random;

public class Rand {
  public static void main (String[] args) {
    System.out.println("computing c = a/(b+a - 2)..");
    Random random = new Random(42);      // (1)

    int a = random.nextInt(2);           // (2)
    System.out.printf("a=%d\n", a);

    //... lots of code here

    int b = random.nextInt(3);           // (3)
    System.out.printf("  b=%d       ,a=%d\n", b, a);

    int c = a/(b+a -2);                  // (4)
    System.out.printf("=>  c=%d     , b=%d, a=%d\n", c, b, a);
  }
}
```

The error message is presented as:

```
> bin/jpf Rand
JavaPathfinder v4.1 - (C) 1999-2007 RIACS/NASA Ames Research Center
=====================================================
system under test
application: Rand.java

=====================================================
search started: 1/01/18 11:48 PM
a=1
  b=0
    c=-1
```

```
=====================================================
results
no errors detected

=====================================================
search finished: 1/01/18 11:48 PM
>
```

The trace that is returned depends on the internal properties of the model checking algorithm.

F.5.3 Valgrind

Valgrind [373] automatically detect many memory management and threading bugs, and profile programs in detail at runtime. For example, given the following C program:

```
#include <stdio.h>

int main() {
  int x;
  printf("x = %d\n", x);
}
```

Valgrnid will identify that `x` is uninitialized, and present the error as a trace:

```
==105087== Memcheck, a memory error detector
==105087== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==105087== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==105087== Command: ./a.out
==105087==
==105087== Conditional jump or move depends on uninitialised value(s)
==105087==    at 0x4E900E1: vfprintf (vfprintf.c:1642)
==105087==    by 0x4E98E35: printf (printf.c:33)
==105087==    by 0x108667: main (uninit.c:5)
==105087==
==105087== Use of uninitialised value of size 8
==105087==    at 0x4E8CC9B: _itoa_word (_itoa.c:179)
==105087==    by 0x4E9141F: vfprintf (vfprintf.c:1642)
==105087==    by 0x4E98E35: printf (printf.c:33)
==105087==    by 0x108667: main (uninit.c:5)
==105087==
==105087== Conditional jump or move depends on uninitialised value(s)
==105087==    at 0x4E8CCA5: _itoa_word (_itoa.c:179)
==105087==    by 0x4E9141F: vfprintf (vfprintf.c:1642)
```

```
==105087==    by 0x4E98E35: printf (printf.c:33)
==105087==    by 0x108667: main (uninit.c:5)
==105087==
==105087== Conditional jump or move depends on uninitialised value(s)
==105087==    at 0x4E91521: vfprintf (vfprintf.c:1642)
==105087==    by 0x4E98E35: printf (printf.c:33)
==105087==    by 0x108667: main (uninit.c:5)
==105087==
==105087== Conditional jump or move depends on uninitialised value(s)
==105087==    at 0x4E9018F: vfprintf (vfprintf.c:1642)
==105087==    by 0x4E98E35: printf (printf.c:33)
==105087==    by 0x108667: main (uninit.c:5)
==105087==
==105087== Conditional jump or move depends on uninitialised value(s)
==105087==    at 0x4E90210: vfprintf (vfprintf.c:1642)
==105087==    by 0x4E98E35: printf (printf.c:33)
==105087==    by 0x108667: main (uninit.c:5)
==105087==
x = 0
==105087==
==105087== HEAP SUMMARY:
==105087==     in use at exit: 0 bytes in 0 blocks
==105087==   total heap usage: 1 allocs, 1 frees, 1,024 bytes allocated
==105087==
==105087== All heap blocks were freed -- no leaks are possible
==105087==
==105087== For counts of detected and suppressed errors, rerun with: -v
==105087== Use --track-origins=yes to see where uninitialised values come from
==105087== ERROR SUMMARY: 6 errors from 6 contexts (suppressed: 0 from 0)
```

F.5.4 Frama-C

```
[kernel] Parsing FRAMAC_SHARE/libc/__fc_builtin_for_normalization.i (no
↪  preprocessing)
[kernel] Parsing swap.c (with preprocessing)
[wp] Running WP plugin...
[rte] annotating function swap
[wp] 8 goals scheduled
[wp] [Qed] Goal typed_swap_assert_rte_mem_access_2 : Valid
[wp] [Qed] Goal typed_swap_assert_rte_mem_access_3 : Valid
[wp] [Alt-Ergo] Goal typed_swap_assert_rte_mem_access : Valid
[wp] [Alt-Ergo] Goal typed_swap_post_B : Valid
[wp] [Alt-Ergo] Goal typed_swap_post_A : Unknown (Qed:3ms) (58ms)
[wp] [Qed] Goal typed_swap_assign_part2 : Valid
[wp] [Qed] Goal typed_swap_assign_part1 : Valid
[wp] [Alt-Ergo] Goal typed_swap_assert_rte_mem_access_4 : Valid
[wp] Proved goals:    7 / 8
```

```
     Qed:             4  (0.52ms-2ms-7ms)
     Alt-Ergo:        3  (7ms-16ms) (18) (unknown: 1)
------------------------------------------------------------
  Function swap
------------------------------------------------------------

Goal Post-condition 'A' in 'swap':
Let x = Mint_0[a].
Let x_1 = Mint_0[b].
Assume {
  Type: is_sint32(x) /\ is_sint32(x_1) /\
      is_sint32(Mint_0[b <- x][a <- x][b]).
  (* Heap *)
  Have: (region(a.base) <= 0) /\ (region(b.base) <= 0) /\ linked(Malloc_0).
  (* Pre-condition *)
  Have: valid_rw(Malloc_0, a, 1) /\ valid_rw(Malloc_0, b, 1).
  (* Assertion 'rte,mem_access' *)
  Have: valid_rd(Malloc_0, a, 1).
  (* Assertion 'rte,mem_access' *)
  Have: valid_rd(Malloc_0, b, 1).
}
Prove: x_1 = x.
Prover Alt-Ergo returns Unknown (Qed:3ms) (58ms)

------------------------------------------------------------

Goal Post-condition 'B' in 'swap':
Let x = Mint_0[a].
Let x_1 = Mint_0[b <- x][a <- x][b].
Assume {
  Type: is_sint32(x) /\ is_sint32(Mint_0[b]) /\ is_sint32(x_1).
  (* Heap *)
  Have: (region(a.base) <= 0) /\ (region(b.base) <= 0) /\ linked(Malloc_0).
  (* Pre-condition *)
  Have: valid_rw(Malloc_0, a, 1) /\ valid_rw(Malloc_0, b, 1).
  (* Assertion 'rte,mem_access' *)
  Have: valid_rd(Malloc_0, a, 1).
  (* Assertion 'rte,mem_access' *)
  Have: valid_rd(Malloc_0, b, 1).
}
Prove: x_1 = x.
Prover Alt-Ergo returns Valid (Qed:3ms) (16ms) (18)

------------------------------------------------------------

Goal Assertion 'rte,mem_access' (file swap.c, line 8):
Assume {
  (* Heap *)
  Have: (region(a.base) <= 0) /\ (region(b.base) <= 0) /\ linked(Malloc_0).
```

```
  (* Pre-condition *)
  Have: valid_rw(Malloc_0, a, 1) /\ valid_rw(Malloc_0, b, 1).
}
Prove: valid_rd(Malloc_0, a, 1).
Prover Alt-Ergo returns Valid (Qed:2ms) (12ms) (12)

------------------------------------------------------------

Goal Assertion 'rte,mem_access' (file swap.c, line 9):
Prove: true.
Prover Qed returns Valid (0.52ms)

------------------------------------------------------------

Goal Assertion 'rte,mem_access' (file swap.c, line 10):
Prove: true.
Prover Qed returns Valid (0.68ms)

------------------------------------------------------------

Goal Assertion 'rte,mem_access' (file swap.c, line 10):
Assume {
  (* Heap *)
  Have: (region(a.base) <= 0) /\ (region(b.base) <= 0) /\ linked(Malloc_0).
  (* Pre-condition *)
  Have: valid_rw(Malloc_0, a, 1) /\ valid_rw(Malloc_0, b, 1).
  (* Assertion 'rte,mem_access' *)
  Have: valid_rd(Malloc_0, a, 1).
}
Prove: valid_rd(Malloc_0, b, 1).
Prover Alt-Ergo returns Valid (Qed:7ms) (7ms) (13)

------------------------------------------------------------

Goal Assigns (file swap.c, line 5) in 'swap' (1/2):
Effect at line 9
Prove: true.
Prover Qed returns Valid (0.61ms)

------------------------------------------------------------

Goal Assigns (file swap.c, line 5) in 'swap' (2/2):
Effect at line 10
Prove: true.
Prover Qed returns Valid (0.58ms)

------------------------------------------------------------
```

G | Rational TypeScript

The following source code listing implements the duplicate function implementation error, `TS2393`, by extending the TypeScript Compiler API:

```
import * as ts from "byots";
import chalk from "chalk";
import minimist from "minimist";

const log = console.log;

interface IErrorReconstruction {
    diagnosticCode: string;
    fileName: string;
    name: string;
    lineAndCharacters: ts.LineAndCharacter[];
    lengths: number[];
    snippets: string[];
    positions: number[];
}

function getUntilNewLine(s: string): string {
    return s.slice(0, s.indexOf("\n"));
}

function compile(fileNames: string[], options: ts.CompilerOptions,
                 displayType?: string): void {
    const program = ts.createProgram(fileNames, options);
    // const emitResult = program.emit();

    const allDiagnostics = ts.getPreEmitDiagnostics(program);

    const duplicateFunctionError: IErrorReconstruction = {
        diagnosticCode: "TS2393",
        fileName: "",
        lengths: [],
        lineAndCharacters: [],
        name: "",
```

```
34          positions: [],
35          snippets: [],
36      };
37
38      allDiagnostics.forEach((diagnostic: ts.Diagnostic) => {
39          // Code 2393 = Duplicate function implementation.
40          if (diagnostic.file && diagnostic.code === 2393) {
41              const token = ts.getTokenAtPosition(diagnostic.file,
                ↪  diagnostic.start!, true);
42              const functionName = (token as any).escapedText;
43
44              if (duplicateFunctionError.name === "") {
45                  duplicateFunctionError.name = functionName;
46              } else if (duplicateFunctionError.name !== functionName) {
47                  // There are multiple duplicate function problems.
48                  // Skip this one for now.
49                  return;
50              }
51
52              const position = diagnostic.start!;
53              const diagnosticStart =
54                  diagnostic.file.getLineAndCharacterOfPosition(
55                      diagnostic.start!);
56
57              const startLinePosition =
58                  ts.getPositionOfLineAndCharacter(
59                      diagnostic.file,
60                      diagnosticStart.line, 0);
61              const theLine = getUntilNewLine(
62                  diagnostic.file.text.slice(startLinePosition));
63
64              duplicateFunctionError.fileName = diagnostic.file.fileName;
65              duplicateFunctionError.lineAndCharacters.push(diagnosticStart);
66              duplicateFunctionError.lengths.push(diagnostic.length!);
67              duplicateFunctionError.positions.push(position);
68              duplicateFunctionError.snippets.push(theLine);
69          }
70      });
71
72      if (duplicateFunctionError.name) {
73          if (displayType === "text") {
74              // displayText(duplicateFunctionError);
75          } else {
76              displayDiagram(duplicateFunctionError);
77          }
78      }
79  }
80
81  function displayDiagram(e: IErrorReconstruction): void {
```

```
    const first = e.lineAndCharacters[0];
    const second = e.lineAndCharacters[1];

    // Lines are zero-indexed, but presented to user as one-indexed.
    const firstLineLength = (first.line + 1).toString().length;
    const secondLineLength = (second.line + 1).toString().length;
    const linePad = secondLineLength + 1;

    log(chalk.redBright(`error[${e.diagnosticCode}]`) +
        chalk.whiteBright(`: duplicate implementation of function
        ↪  \`${e.name}\``));

    log(`${" ".repeat(linePad - 1)}` +
        `${chalk.blueBright("--> ")}` + chalk.white() +
        `${e.fileName}:${second.line + 1}:${second.character + 1}`);

    log(chalk.blueBright(`${" ".repeat(linePad)}|`));

    log(chalk.blueBright(`${first.line + 1}${" ".repeat(linePad -
    ↪  firstLineLength)}`) +
        chalk.redBright(`×`) +
        ` ${e.snippets[0]}`);

    log(chalk.blueBright(`${" ".repeat(linePad)}| ${"
    ↪  ".repeat(first.character)}`)
        + chalk.blueBright(`${"-".repeat(e.lengths[0])} previous
        ↪  implementation of \`foo\` here`));

    if (second.line === first.line + 1) {
        log(chalk.blueBright(`${" ".repeat(linePad)}|`));
    } else if (second.line === first.line + 2) {
        log(chalk.blueBright(`${" ".repeat(linePad)}|`));
    } else {
        log(chalk.blueBright("..."));
    }

    log(chalk.blueBright(`${second.line + 1} `) + chalk.redBright(`× `) +
    ↪  `${e.snippets[1]}`);

    log(chalk.blueBright(`${" ".repeat(linePad)}| `) +
        chalk.redBright(`${"
        ↪  ".repeat(first.character)}${"~".repeat(e.lengths[1])}`) +
        chalk.redBright(` \`${e.name}\` reimplemented here`));

    log(chalk.blueBright(`${" ".repeat(linePad)}|`));
    log(chalk.blueBright(`${" ".repeat(linePad)}=`) +
        chalk.whiteBright(` hint: `) +
```

```
123         `\`${e.name}\` must be implemented only once within the same
            ↪ namespace`);
124
125     log(chalk.blueBright(`${" ".repeat(linePad)}=`) +
126         chalk.whiteBright(` hint: `) +
127         `To overload a function, see
            ↪ https://www.typescriptlang.org/docs/handbook/functions.html`);
128 }
129
130 const argv = minimist(process.argv.slice(2), {string: "display"});
131
132 compile(argv._, {
133     module: ts.ModuleKind.CommonJS,
134     noEmitOnError: true,
135     noImplicitAny: true,
136     target: ts.ScriptTarget.ES5,
137 }, argv.display);
```

The above implementation indirectly loads the `typescript` package through `byots`. `byots` is a "Bring your own TypeScript" package for NPM that exposes many of the normally-internal APIs and makes them available for the toolsmith. In particular, our implementation relies on the `getTokenAtPosition` to retrieve the corresponding AST Node at a specified source code location. However, without `byots`, this useful function is not available.

We iterate over the `allDiagnostics` collection (Line 38) to obtain metadata about all `TS2393` errors. To keep the source code listing to a manageable length, the implementation silently discards all other errors. The algorithm also makes the assumption that there are only two duplicate functions, within a single file. There is some clerical effort in this function to reconstruct the linear positions within the source code representation and translate them to line and character positions suitable for the error messages.

The function `displayDiagram` renders the error message metadata to the console. Again, most of this implementation is simple clerical effort to correctly align the duplicate functions, to render the corresponding line numbers, and to position the explanatory text. The `chalk` package enables ANSI color support, at the expense of making the source code a bit more noisy.

H | Sudoku Puzzle

	2			3		9		7
	1							
4		7				2		8
		5	2				9	
			1	8		7		
	4				3			
				6			7	1
	7							
9		3		2		6		5