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 text and visual presentations such as error listings, tooltips, and source code underlined with red squiggles. The design of human-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 developers 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 at 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 interpreting error messages can be explained by framing error messages as insufficient rational reconstructions in both visual and text 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 involving 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 comparative evaluation between two Java compilers and investigated their error messages through Toulmin's model of argument, a form of rational reconstruction. For generalizability, we additionally conducted a study to analyze confusing error messages on Stack Overflow against human-authored reconstructions of those error messages. We also analyzed these answers using Toulmin's model of argument. We 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 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.

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 External Member

Emerson Murphy-Hill
Chair of Advisory Committee

DEDICATION

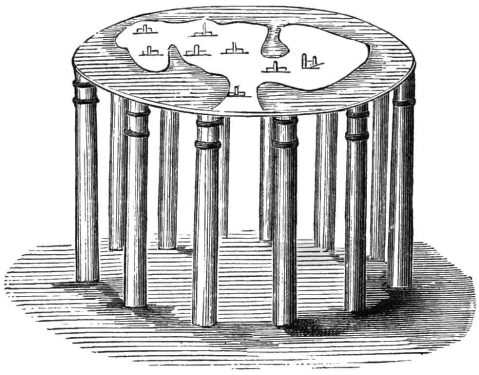
In memory of Mark Lusher (1972–2015).
Then you will shine among them like stars in the sky.

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 2009. 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 software 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

This dissertation would not have been possible without the love and unwavering support of my family. I am grateful to my wife Laurel, my son Joshua, and my daughter Emily.

For their academic guidance, thanks go to my committee members: Dr. Emerson Murphy-Hill, Dr. Chris Parnin, Dr. James Lester, Dr. Jing Feng, and Dr. Shriram Krishnamurthi. I also thank Dr. David Roberts for his advice and encouragement.

This material is based upon work supported by the National Science Foundation under grant numbers 1217700, 1318323, 1559593, and 1714538. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of the National Science Foundation. ABB, Amazon, Google, and Microsoft also provided funding and support for this material.

There, but for the grace of God, go I.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

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 interpreting error messages can be explained by framing error messages as insufficient rational reconstructions in both visual and text 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 [36, 53, 137, 138, 332]. Many of the problems that program analysis tools detect are also subtle to spot through manual inspection [15, 34, 177, 271]. Unfortunately, research has found that the output program of analysis tools—error messages—are confusing, unconstructive, misleading, or incomprehensible [50, 55, 83, 356, 381]. As a result, developers spend unnecessary effort comprehending and resolving the defects identified by the tools or simply abandon otherwise useful tools because they can't understand the error messages [65, 190].

What makes these error messages so confusing for developers?

2.2 Examples

Program analysis tools communicate these problems to developers through error messages, using various text and visual presentations. But rather than attempt to characterize precisely what an error message is (we'll do that in Chapter 3), let's take a tour of five concrete examples to explore different facets for why error messages might be difficult for developers to comprehend. Examples in Sections 2.2.1 to 2.2.4 explore text-based error messages, as found in console or terminal environments. Section 2.2.5 shifts direction and tailors text-based error messages to visual, integrated development environments (IDEs)—such as Visual Studio Code [260]. In particular, this example articulates how the appropriateness of an error message depends not only on the message content but also in how the error message is situated within the programming environment.

2.2.1 Portable C Compiler

We'll start with the PCC [193] compiler from 1979, for which a developer attempts to write 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 its simplicity, 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 that the error should actually be reported on 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 it is a 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 contemporary program analysis tools.

2.2.2 GNU Compiler Collection

Let's now look at GCC [129], 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 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, for example, by indicating that the problem is in the function `main`.

Still, the cause of the error message is disorienting from the developers' perspective, because the tool emits a location that *follows* the problem, rather than the location that immediately *precedes* it [140]. The easiest way to illustrate this concern is to contrast GCC with a compiler that does this correctly, at least in this case. Here's the output from Clang [235], part of LLVM:

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

The problem is now apparent, and the error message also matches the way developers would think about the problem, rather than what would be more convenient for the toolsmith—that is, the designer of the program analysis tool—to generate [250]. In other words, "I'm missing a semicolon at the end of Line 4" is a more direct explanation than "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 thoroughly disconcerting [43]. The reasoning of the compiler, strictly applied, can lead to idiosyncratic but nevertheless conformant fixes like this one:

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

2.2.3 Oracle Java Compiler (OpenJDK)

Here's a source listing in Java:

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

The source listing results in the following error message from the OpenJDK [286] compiler:[1]

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

[1]This error message was reported as bug JDK-4071337, "misleading error message when superclass constructor has throws clause." See: https://bugs.java.com/bugdatabase/view_bug.do?bug_id=4278961

Let's construct a rationale for what could cause this error message. First, we note that the error message involves a default constructor, but no such constructor explicitly appears in the `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:

```
1  class Toy {
2      Toy() throws Exception { }
3  }
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  }
```

Our last attempt is accepted by the compiler, and now we must retrospectively find reasons for why. For instance, why did we have to explicitly create a default constructor? 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) [139]:

> 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's now look at a source listing in C++, named `ptrcopy.cpp`, in which we'll use the Clang compiler from LLVM [235]:

```
#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 `foo` to the contents of `bar`. But attempting to do this generates an unwieldy 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> > *,
```

```
13        std::vector<std::unique_ptr<int, std::default_delete<int> >,
          ↪  std::allocator<std::unique_ptr<int, std::default_delete<int> >
          ↪  > > >, std::unique_ptr<int,
14        std::default_delete<int> > *>' requested here
15          __uninit_copy(__first, __last, __result);
16                        ^
17  /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
18        '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 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.h` (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 program 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.[2] First, it pinpoints a location that the developer actually 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. Third, even with this loss of precision, it's obvious what the problem is: `unique_ptr` is not copyable. 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

Thus far, we've only examined text-based error messages. However, in this section we'll argue that whether an error message is appropriate depends not only on the message content but also on how the error message is situated within the programming environment—for example, in a terminal or in an IDE. That is, sometimes the "medium is the message" [255], and it is the medium that shapes the way in which we should present error messages to the developer.

[2] A Stack Overflow answer 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`.

Let's look at an example of this through ESLint [420], a pluggable bug-finding utility for JavaScript. By pluggable, we mean: 1) developers can add custom bug-finding rules to extend the capability of the tool, and 2) the tool is intended to be bundled and used within other workflows, such as integrated development environments and build systems. Thus, the tool supports multiple output formats, or mediums, for presentation.

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 to emit the following error message:

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

We can use a template literal[3] to resolve the error message:

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

In contrast to the error messages from OpenJDK, GCC, and LLVM, the error message from ESLint almost seems like a regression to an earlier era of compilers! Unlike either of these tools, 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 relatively simple improvements to this error message to expand on the suggested `prefer-template` fix and add a rationale for the error:

```
× [eslint] Unexpected string concatenation (2, 13)
           Fix:  Use template literals instead of concatenation.
                 (re-run with --fix to automatically fix this problem)
           Why?  Template literals give you a readable, concise syntax
                 with proper newlines and string interpolation features.
                 (see: airbnb.io/javascript/#es6-template-literals)
```

[3]In prior editions of the ES2015 specification, template literals were called template strings. See: `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Template_literals`.

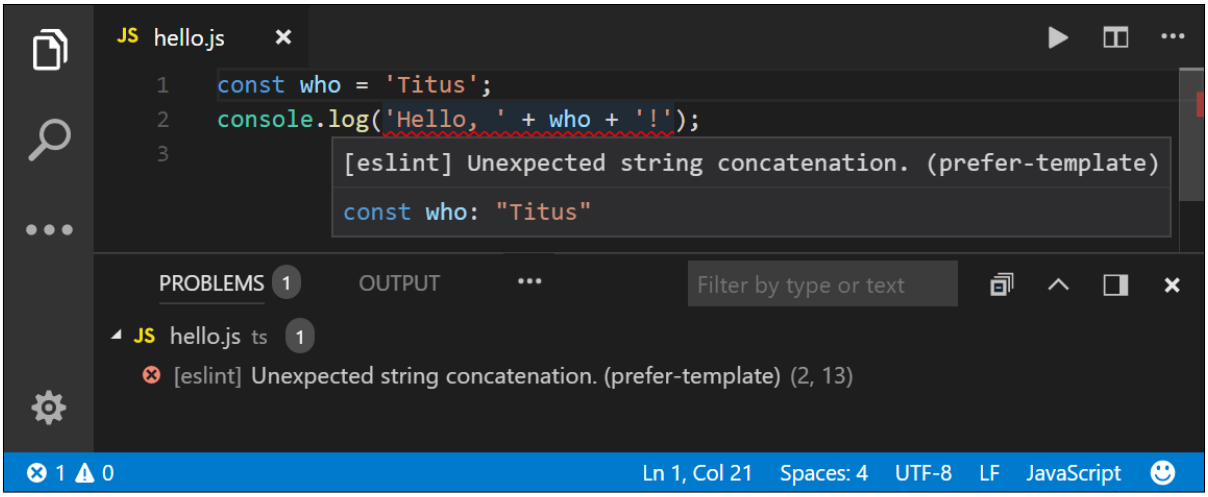
The above error message replaces the cryptic `prefer-template` key with natural language. Incidentally, this particular fix is trivially corrected by ESLint, and so the message also suggests the availability of this automated repair. Then, given the rationale for the error, we see that the reason ESLint recommends this change is because the Airbnb JavaScript standard considers template literals to be more readable. An extended explanation is offered through a link, and gives examples of converting string concatenation operations to template literals.

Although ESLint supports console output as a last resort, ESLint is really intended to be used as a program analysis building block to be incorporated within other programming and build environments. To illustrate this, let's consider again 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. Also, the developer can double-click the error message in the problems pane to directly navigate to Line 2 of `hello.js`. Given the single-line affordances in the problems pane, it now makes sense to present a terse error message 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). Therefore, providing a contextual code snippet within the error message itself would be redundant, given that we expect the errors to be presented in the IDE.

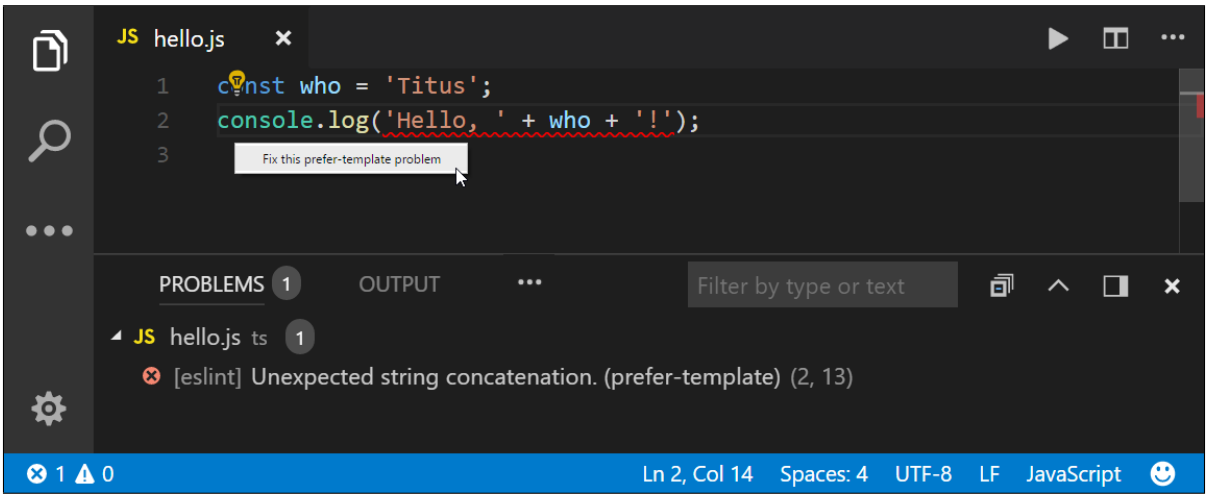
2.3 Objective, Significance, and Scope

Given the problematic error messages we detailed in the previous section, how can we rigorously explain the difficulties developers encounter with these messages? Importantly, how can we go about improving error messages through principled approaches—so they are more useful to developers?

The *objective* of this dissertation is to evaluate a novel framework that applies *rational reconstruction* theory to error messages as a unifying explanation for why error messages are difficult for developers to comprehend. We formalize this theoretical framework in the next section (Section 2.4). The *significance* of this research



(a) ESLint identifies a problem.



(b) ESLint suggests a fix for the problem.

Figure 2.1 For the 'unexpected string concatenation' error, the bare ESLint error message is appropriate when the message is presented in the Visual Studio Code IDE. Other affordances, such as the light bulb icon, tooltip overlay, and red wavy underline, provide additional context to the developer.

is that it demonstrates that error messages as rational reconstructions are more useful to developers than baseline error messages, because rational reconstructions align with the way in which they strategize and reason about problems in their code under the presence of errors. Furthermore, the design of human-friendly error messages is important because error messages are the primary communication channel through which tools provide feedback to developers. Consequently, the research in this dissertation advances a systematic, theoretical lens to investigate error messages in program analysis tools, and embodies six postulates that scaffold it:

Postulate I—Error messages are routine. The selected examples are everyday messages that confound developers, taken from program analysis tools used in industry. Generally, the types of error messages developer encounter don't require them to have knowledge of esoteric language concepts, nor do the error messages arise from extraordinary circumstances [342].

Postulate II—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 [65, 190]. But none of the selected examples were false positives, and all of 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.

Postulate III—Developers are intermediate-experts. This isn't just a problem for novices. The persona we assumed in the examples, and study in this dissertation, are developers who are intermediate-experts [348] in the programming ecosystem they use. That is, they are not completely new to programming (naive), nor are they novices who are in the early-stages of learning the programming language or their program analysis tools.

Postulate IV—There isn't only one 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 [18]. 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 intended to implement. Good automatic fixes are useful as accelerators for developers, but they don't remove the need to understand the error message.

Postulate V—Error messages present 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 reconstruct a chain of reasons to identify the cause of the error.

Postulate VI—Error messages are rational, but only from the perspective of the compiler. Let's turn again to the source code listing for the "Hello, world!" example from Section 2.2.2, but this time we will reformat the listing like so:

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

Some compilers remove insignificant whitespace, or trivia, because they are unnecessary in the program analysis pipeline; this results in a program that appears to the tool essentially like the above. From the perspective of the compiler, the error message `expected ';' before '}' token` is now perfectly rational if we reorient ourselves to the perspective of how the compiler thinks about the situation. But developers shouldn't have to know the internal mechanics of program analysis tools to understand their error messages.

The first three postulates eliminate specific factors as being the cause of developer difficulties with error messages: developers experience difficulties in cases when the error messages are routine, when they aren't false positives, and even when the developers are comfortable with their programming languages and tools. The fourth

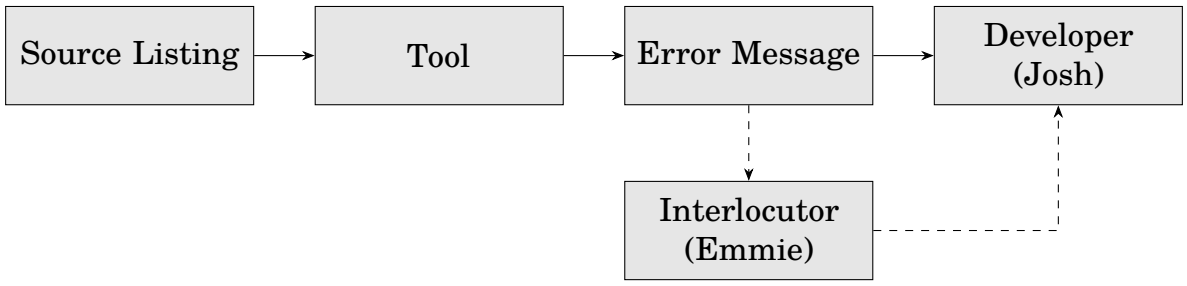


Figure 2.2 The theoretical framework of rational reconstruction.

postulate suggests that error messages are useful to developers for verification, even when tools suggest automatic fixes. The first four postulates, then, *scope* the dissertation. But it is the fifth and sixth postulates that motivate a theory to investigate: these two postulates conjecture a misalignment between the way program analysis tools present error messages to developers and the way in which developers attempt to comprehend the problem. In the next section, we formalize these misalignments through the theoretical framework of rational reconstruction.

2.4 Theoretical Framework

Example. Consider a scenario in which a developer, Josh, encounters a confusing error message. He isn't able to comprehend the error message, so he turns to a colleague, Emmie, 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 diagram this scenario as in Figure 2.2, with Emmie as the interlocutor—a participant in the discourse.

What did Emmie say 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 [248]. 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, Josh must reconstruct the rationale for the error message himself and come to the same conclusion as the conclusion of the error message.

The process of identifying these rationales is rational reconstruction. Within this theory, rationales come in two forms [58, 248, 253]: 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 computers as social actors suggests that it is: people treat computers and respond to computers as if they were real people [275]. And subsequent research has argued that it is often valuable for computational agents to mimic how people behave in human-human interactions and approximate them in human-computer interactions [31, 106, 262].

2. **Rational reconstructions are not historical reconstructions.** They admit orthogonal explanations of the problem, and the research implication is that 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 explanation to Josh for why the error message is emitted. In Chapter 9, 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 [290] proposes 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 software design documentation 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." A review of existing design guidelines in Section 3.5 demonstrates how rational reconstructions are a fruitful direction for error message investigation.

2.5 Research Paradigm

The research paradigm governs the philosophical assumptions and beliefs for how we conduct research and interpret research findings [76]. In this dissertation, I employ a pragmatic research paradigm, which I describe through the four basic questions of the research process [78]:

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 [264]. 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 [104]. Because of the lack of epistemological commitment, the output of pragmatic research is 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 [192].

2.5.3 Methodology

I use mixed-methods in my studies, applying both quantitative and qualitative methodologies, 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 6, I conducted a usability study of error messages within the Eclipse IDE, 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 7, 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 [347] to assess developer comprehension. In Chapter 8, I conducted a comparative evaluation between two Java compilers to understand developer preferences for different structures of error messages. I qualitatively investigated Stack Overflow posts related to error messages, and analyzed these posts through the lens of argumentation theory—a form of rational reconstruction.

2.6 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 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 published work that also appears in this dissertation:

- Chris Parnin and Emerson Murphy-Hill are co-authors for my scoping review (Chapter 4). Through various exchanges, Parnin and I discussed the motivation and framing for the introduction as interpreting 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 6). 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 library 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 6.1 and Figure 6.2). Smith and I pair-wrote the introduction and motivating example (Section 6.1) 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 6.2), 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 for area of interest detection in video data.

- Kevin Lubick, Samuel Christie, and Emerson Murphy-Hill are co-authors on my think-aloud study on how developers visualize error messages (Chapter 7). 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 7.1). 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, Emerson Murphy-Hill, and Chris Parnin are co-authors on my study investigating Stack Overflow questions and answers on error messages (Chapter 8). Ford and I had many early discussions on how to classify Stack Overflow posts before arriving at argumentation theory. In the paper, Ford wrote the query to compute Table 8.2. 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.

2.7 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 interpreting error messages can be explained by framing error messages as insufficient rational reconstructions in both visual and text presentations.

To defend the claim that "difficulties interpreting error messages produced by program analysis tools are a significant predictor of developers' inabilities to resolve defects," I offer evidence through the study in Chapter 6. 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 interpreting error messages can be explained by framing error messages as insufficient rational reconstructions in both visual and text presentations," I offer evidence through two studies. For visual presentations, this evidence is offered through the study in Chapter 7, in which we compare *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 text presentations, I offer evidence through the study in Chapter 8. The study finds 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.

2.8 How to Read the Dissertation

This dissertation is organized as self-contained chapters that together support the thesis (Chapter 1) and tell a coherent story about error messages as rational reconstructions. Figure 2.3 on page 24 provides a roadmap of the dissertation.

Chapter 2 (you're reading this now!) identifies and provides context for the research problem, articulates the objective, significance, and scope of the work, and proposes the theoretical framework of rational reconstruction through which I investigate error messages.

Following this chapter are three literature reviews which survey the literature through different perspectives. The state-of-the-art review presented in Chapter 3 familiarizes the reader to the research area of error messages, framed in terms of contemporary program analysis tools that developers actually use. The scoping review in Chapter 4 identifies rational reconstruction as a problem of alignment, and contributes a taxonomy of other dimensions that are important to the design of human-friendly error messages. The chapter may be of particular interest to readers who desire to bridge the research communities of human-computer interaction and programming languages. The mapping review in Chapter 5 defends the novelty of the thesis; the chapter demonstrates how the work in this dissertation fits conceptually with neighboring research disciplines, such as artificial intelligence and human factors.

Chapters 6 to 8 describe the three principal studies that provide evidence to support the thesis statement; each of these studies considers a different facet of rational reconstruction. Chapter 6 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 7 studies rational reconstructions as visual explanations for defects in source code. Chapter 8 studies rational reconstructions as text explanations, through two evaluations: a comparative evaluation of error messages, and an empirical evaluation of Stack Overflow that identifies situations when developers find error messages from program analysis to be insufficient.

In Chapter 9, we revisit the key contributions of this dissertation. The chapter interprets the findings from the studies and explicates them as design guidelines. These guidelines are operationalized and evaluated through a prototype compiler implemented in TypeScript.

The appendices provide supplemental material that may be of interest to the reader, but this material is not necessary to support the central claims of the thesis. Appendices A to C compile the materials from our user studies. Appendix D synthesizes existing guidelines for error messages from the research literature. Appendix E describes the implementation details of the TypeScript prototype compiler.

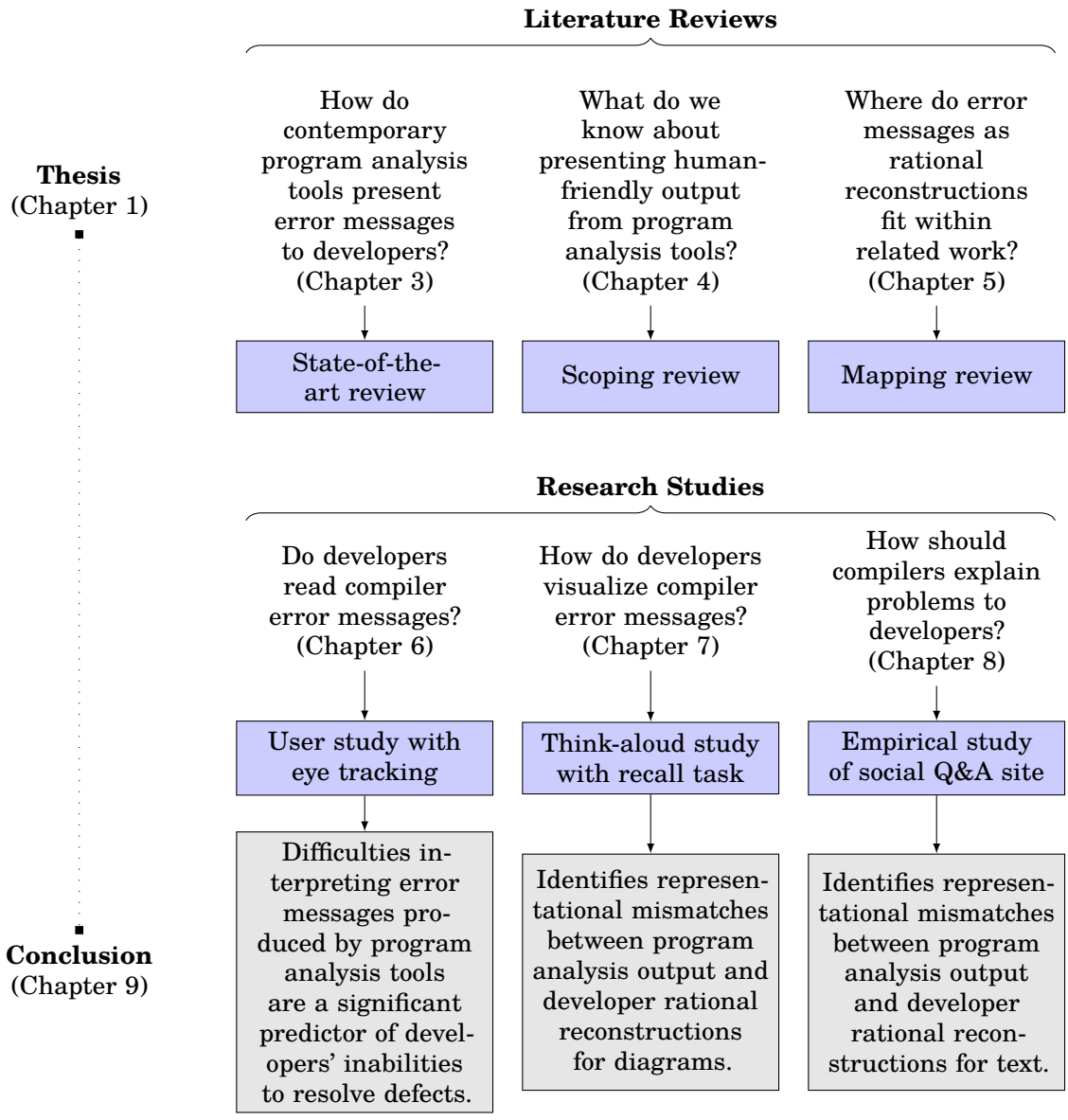


Figure 2.3 A roadmap of the dissertation. The dissertation is organized as self-contained (at least to the extent reasonably possible), modular chapters. Three literature reviews offer the reader different perspectives on rational reconstruction; these chapters may be read in any order. Each of the three research studies defends one of the claims in the thesis. Although the sequence of the research studies follows the sequence of claims in the thesis statement, these chapters may also be read in any order.

3 | How Do Contemporary Program Analysis Tools Present Error Messages to Developers?

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

This chapter presents a state-of-the-art review on program analysis tools, with the goal of familiarizing readers to current matters on how contemporary program analysis tools construct and present error messages to developers. The chapter frames these error messages through our theoretical framework of rational reconstruction, emphasizing how presentation design choices support or do not support developers. As a state-of-the-art review, the literature surveyed within this chapter is presented through the context of program analysis tools that developers actually use in practice.

Section 3.1 offers a general overview of program analysis tools. The overview articulates why traditional taxonomies of static and dynamic analysis are unsuitable lenses through which to investigate rational reconstruction, and proposes a presentation-centric organization of error messages as an alternative. Sections 3.2 and 3.3 characterize the design space of text and visual representations, respectively. Section 3.4 summarizes the existing literature on empirical distributions of error

messages from program analysis tools; this allows researchers as well as practitioners to prioritize improvements in error messages to problems that developers actually encounter. Section 3.5 chronologically reviews design guidelines towards presenting human-friendly error messages.

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 [36]. This dissertation focuses on a particular class of program analysis tools, source code analysis, that identifies defects within source code [36]:

> 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 [56, 112, 137, 332].

The first dimension is *when* the program analysis is performed. In static 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 [138]). In dynamic analysis, tools instrument or otherwise inspect the runtime and examine the flow of execution to identify defects [56] (see survey on dynamic analysis tools, also by Gosain and Sharma [137]). Static and dynamic analysis can also be applied synergistically to

strengthen program analysis [112]. 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 [79].

The second dimension is *where* the program analysis is performed. For example, FindBugs is a standalone static analysis tool for Java that *supplements* the error messages provided by the Java compiler [15]. In contrast, LLVM *embeds* program analysis tools directly within the compilation pipeline; Lattner and Adve [215] demonstrate that this type of "lifelong program analysis," such as interprocedural static leak detection, can enable identification of certain defects that would not be possible to find otherwise.

Unfortunately, neither of these implementation dimensions is satisfactory for understanding developer difficulties within the theoretical framework of rational construction (Section 2.4). In rational reconstruction, the implementation detail of whether the information for the problem is obtained using static, dynamic, or hybrid analysis is orthogonal to the appropriate presentation of the error message. For example, though both LLVM (static) and Valgrind [386] (dynamic) can report uninitialized variables, rational reconstruction suggests that the type of analysis should not dictate the appropriate error message to present to the developer.

3.2 Text Representations of Program Analysis

What do contemporary error messages look like when presented to developers through text interfaces, such as consoles and terminals? And how do program analysis tools construct these error messages?

In this section, we'll describe four categories of text representations. The first two representations, template diagnostics (Section 3.2.1) and extended explanations (Section 3.2.2), are primarily human-authored. In contrast to these human-authored messages, the next two representations, type errors (Section 3.2.3) and examples and counterexamples (Section 3.2.4), are computationally-constructed using various formal methods.

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 indicating 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 first bootstrap our investigation of this error message scheme using the following three source code listings to induce an error during program analysis:

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() {
```

```
7       Brick b;
8       b.m(1, 2);
9       return 0;
10  }
```

In all three listings—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+).

We then input these listings into assorted compilers for that programming language to obtain error messages. For Java error messages, we use OpenJDK [286] and the Eclipse Batch Compiler [105]. For C#, we use Roslyn [259] and Mono [257]. For C++, we use LLVM/Clang [235] and GCC [129]. The choice of compilers illustrates the diversity in how toolsmiths choose to present error messages for what is conceptually the same problem. Applying the source code to their corresponding tools yields the following error messages during compilation:

1 OpenJDK (Java):

```
1  Brick.java:6: error: reference to m is ambiguous
2        m(1, 2);
3        ^
4    both method m(int,double) in Brick and method m(double,int) in Brick
       ↪  match
5  1 error
```

2 Eclipse (Java)

```
1  ----------
2  Brick.java (at line 6)
3          m(1, 2);
4          ^
5  The method m(int, double) is ambiguous for the type Brick
6  ----------
7  1 problem (1 error)
```

3 Roslyn (C#)

```
1  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 of 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 (`8:13`) while LLVM indicates the problem at the name, `b.m` (`8:7`). There are other small variations in how the line and column 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 or property. Finally, GCC and LLVM colorize the output of the error messages for expressiveness; the LLVM toolsmiths reasonably argue that colors make it easier to distinguish the different elements of the error message [233].

How do program analysis tools construct these error messages? The messages I have just described are output from a *template* diagnostic [267].[1] 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 the file `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, now evident in the presented 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 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" [210]. A syntax error occurs when program analysis encounters an unexpected token, for example, from unintentionally misplaced

[1] Moulton and Muller [267] (1967) called these skeletal messages, but this term seems to have fallen out of favor.

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 will sometimes report the location of the syntax problem far removed from the actual cause of the syntax error [55]. For example, consider the following Racket [302] 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 in 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 are many research efforts to improve syntax errors—extensively covered in various literature reviews [42, 90, 91]—that historically have occurred in tandem with the construction of even the earliest compilers in the 1950s. Hammond and Rayward-Smith [150] describe some of the early syntactic error recovery and repair schemes, and characterize 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, the authors report that it results in cryptic and unhelpful error messages because of the lack of information available at the time of the error message; the approach also leads to spurious error messages. Subsequent research 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 that mathematically define notions of minimizing error distance, Campbell, Hindle, and Amaral [55] motivate a natural language model to improve error reporting by arguing that humans read code as they read natural language; follow-up work by Santos, Campbell, Patel, and colleagues [337] offer 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 toolsmiths to author more useful error messages. For example, Jeffery [181] 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 parser to produce the error message at the appropriate point. Pottier [308] improves upon this approach by enabling the parser to automatically build a collection of erroneous statements, rather than having toolsmiths provide examples manually. Though useful, both approaches still require toolsmiths to manually write 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 message presentations from Section 3.2.1. Johnson, Song, Murphy-Hill, and colleagues [190] reported that concise error messages in program analysis tools do not provide enough information. However, the authors also reported that developers did not want to sift through volumes of 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 [268], and allows them to selectively investigate unfamiliar error messages with a longer, more verbose explanation [384].

To illustrate how extended explanations are useful in practice, let's look at a simplified build pipeline using Bazel [135]—an open source release of the build system used internally at Google. Bazel includes a program analysis tool called Error Prone [134] that identifies defects in Java code. The build pipeline is guided by a `BUILD` file which specifies the targets to the build system:

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

In this scenario, we have only one build target named `shortset`, and this build target needs to compile only a single Java file, `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 Bazel results in the following output, which includes the 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 is also relatively terse. 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 able to detect this problem.[2] 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 for COBOL program debugging by Litecky [231]. In Litecky's implementation, 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 that included advice for how to address the problem (for a review of expert systems, see Section 5.3).

A limitation of the approach used by Error Prone is that the extended explanation is detached from 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 example given in the extended explanation, they may still not understand how the problem applies to their own code.

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

To mitigate this concern, the Dotty compiler offers a `--explain` flag that can provide an explanation that is contextual to the code that the developer has actually written [268]. For instance, consider the following Dotty code snippet (the Dotty language is a superset of Scala):

```
try {
  foo()
}
```

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 [384] and Elm [80] 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:

```
Explanation
===========
A try expression should be followed by some mechanism to handle any
exceptions thrown. Typically a catch expression follows the try and
pattern matches on any expected exceptions. For example:

import scala.util.control.NonFatal

try {
  foo()
} catch {
  case NonFatal(e) => ???
}

It is also possible to follow a try immediately by a finally - letting
```

```
16 the exception propagate - but still allowing for some clean up in
17 finally:
18
19 try {
20   foo()
21 } finally {
22   // perform your cleanup here!
23 }
24
25 It is recommended to use the NonFatal extractor to catch all exceptions
26 as it 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 20). The current implementation of this in Dotty is fairly rudimentary, and only slightly 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, for example, manipulating 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 research tools for generating context-relevant, extended explanations. The implementation of Tutoron by Head, Appachu, Hearst, and colleagues [159] detects potentially explainable code in a web page, parses it, and generates in-situ natural language explanations and demonstrations of the 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 [156]. The tools Prompter [307], Seahawk [306], and SurfClipse [316] automatically retrieve pertinent Stack Overflow explanations and present them within the IDE. The availability of these tools also suggests that developers find human-authored explanations on Stack Overflow to be useful; we investigate Stack Overflow explanations through an empirical study in Chapter 8.

3.2.3 Output as Type Errors

Rather than assembling error messages from a catalog of hand-authored strings, program analysis tools can leverage computational techniques to automatically construct explanations in error messages.

In this section, we describe type systems, one such form of program analysis tool for automatically constructing explanations. To explain how type systems operate, let's consider some simple Haskell expressions (evaluated in GHC [157]), prefixed with a `:t` command to tell 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 function returns a new list by taking a list and applying a function to every element in that list. Here's a canonical implementation of `map`:

```
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 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 to infer that the type is a character or boolean through *type inference*: a process of reconstructing missing type information through how it is used in the program [35, 82, 103, 298]. 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 toolsmiths, 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 [369]. Worse, relying on type inference as the sole mechanism for inducing error messages can lead to cryptic and counterintuitive error messages, even for simple problems. Consider the following example:

```
print 5
```

This program, of course, 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 ()
```

This Haskell error is pedantically correct, but framing this error message in terms of the type system makes it indecipherable. The problem becomes self-evident only if we realize that '-' is itself a function, masquerading as a unary negation:

```
print (- 5)
```

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

Error messages such as the above are endemic to programming languages that rely on type systems as the vehicle through which to present errors. The problems of inscrutable type errors are well known, and the impact of these cryptic type error messages is articulated by Charguéraud [59]. 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 message, the type checker would still need to make a decision on which of the many possible locations to actually report. Fourth and finally, type inference algorithms suffer from well-known biases—such as left-right bias [251]—in which the type checker will systematically report type conflicts near the end of 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 approaches that make type errors more helpful for developers. One approach is to selectively replace or supplement automated type inference diagnostics with human-authored error messages for common situations. Helium [162], a user-friendly compiler for learning Haskell, employs this approach: it uses a wide range of heuristics to present human-authored hints that supplement the type inference machinery. Similarly, the approached employed by Objective Caml "patches" the compiler [59]: the compiler considers the first top-level definition that fails to type-check and attempts to match the failure against a set of carefully-crafted secondary heuristics. These heuristics identify if the tool can present an alternative,

human-authored error message. If so, the alternative 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 rational reconstruction of the original type error.

To improve type error messages, three studies have applied statistical, machine learning, and search-based approaches. Zhang, Myers, Vytiniotis, and colleagues [421] apply Bayesian reasoning to type systems; the Bayesian model incorporates knowledge regarding common developer mistakes to more accurately pinpoint the location of the true error. Wu, Campora, and Chen [412] employ 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 suggest fixes that would resolve the ill-typed program. Lerner, Flower, Grossman, and colleagues [220] 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.

Other approaches apply formal methods to improve error messages. McAdam [250] uses a fix-oriented approach to error messages, based on *modulo isomorphism*, when type inference fails. The resulting error message is then presented in the form of a suggested change. For example, in 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
```

Instead, McAdam reports the following error message:

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

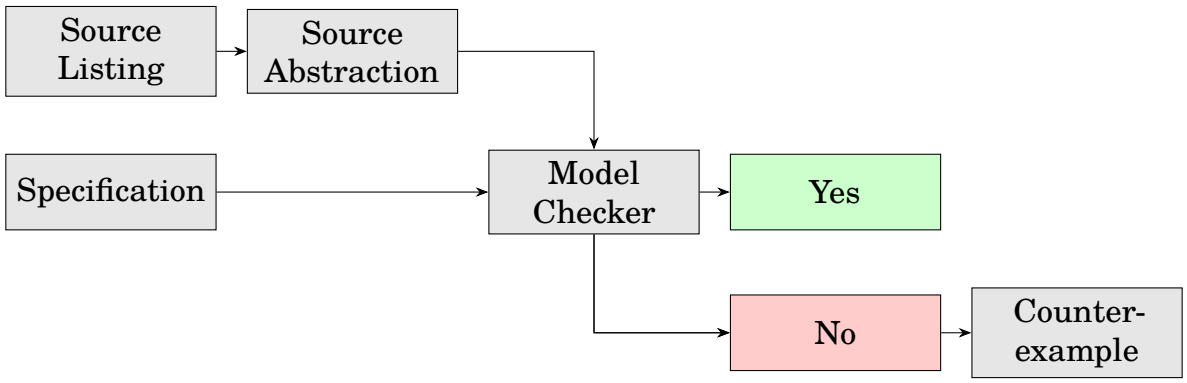


Figure 3.1 A typical model checking pipeline. The source listing is transformed into a source abstraction suitable for verification by the model checker, along with a specification describing some property for how the software should behave. Given these two inputs, the model checker can identify a violation of the property. If a violation exists, the model checker returns a counterexample of how the property can be violated.

Additional formal method approaches to improving type error messages include adapting and leveraging type constraint procedures during program analysis to aid error message construction [44, 237], using data flow reasoning to explain type errors [127], incorporating various modifications to the underlying type inference algorithm to provide a record of the specific reasoning steps used during inference [26, 103, 110, 372, 396], enabling interactive approaches to querying types [357, 371], and implementing algorithmic techniques to pinpoint the actual location of the problem [61, 160, 161, 251, 315, 341, 393, 416].

3.2.4 Output as Examples and Counterexamples

Software model checkers are another form of automated reasoning approach for constructing error messages; they do so by formally verifying some *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 or as a component of an error message explanation.

There are a variety of model checking tools for the domain of software engineering, including SLAM [17], SPIN [170], BLAST [165], Bandera [74], and Java PathFinder [390] (for extensive literature reviews on model checkers, see Mosleh, Alhussein, Baba, and colleagues [266] and Jhala and Majumdar [184]).

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.[3] 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`, `b = 0`, and `c = 1` at Line 7. Of course, the CBMC program analysis tool also detects that the source listing violates the specification, and emits the following snippet:

```
State 17 file file.c line 1 thread 0
------------------------------------------------
```

[3]The term *model* is ambiguous and has multiple meanings 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.

```
  INPUT a: 124955 (00000000000000011110100000011011)

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

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 type errors in Section 3.2.3, the presented counterexample reflects what was found through the execution of the internal algorithm, and not what is necessarily the best example to present to the developer.

This has led to principled approaches to presenting output in model checking tools. For example, tools like Aluminum [278] and Razor [333] provide a *minimal* example to the user: an example that contains 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 [277] allows the developer to interrogate the model checker about the counterexamples it provided, through "why?" and "why not?" questions.

Several approaches use multiple counterexamples to facilitate understanding, or supplement the counterexamples with additional information. Groce and Visser [145] argue that although model checking is effective at finding subtle errors, these errors can be difficult to understand from only a single counterexample. They

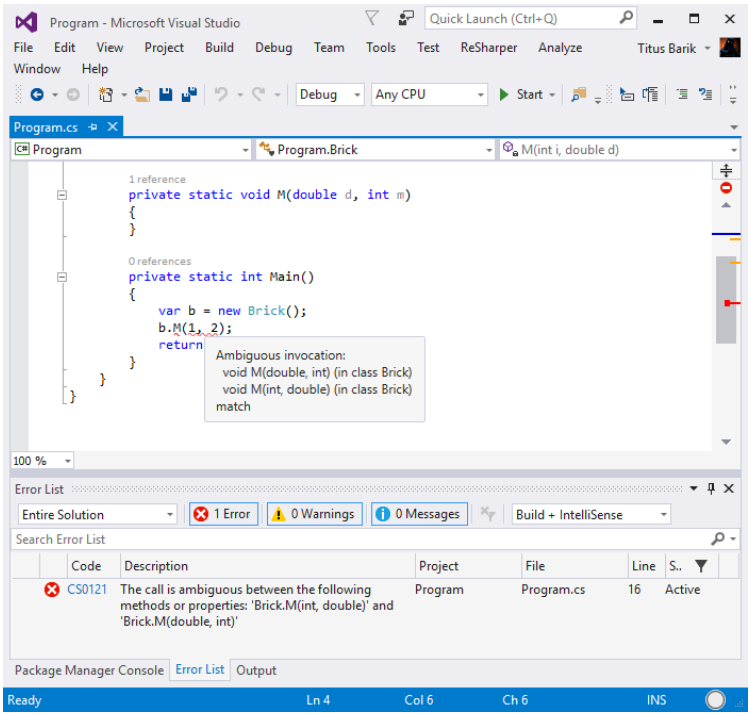
propose an automated technique for finding multiple counterexamples for an error, as well as traces that do not produce an error. Next, they analyze these executions to produce a more *succinct* description of the key elements of the error. Similarly, Ball, Naik, Rajamani, and colleagues [16] present 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 [397] propose a technique in which developers are able to zoom into potential software defects by analyzing a single, concrete counterexample. Gurfinkel and Chechik [147] 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 counterexamples better.

Finally, Hoskote, Kam, Ho, and colleagues [174] 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.

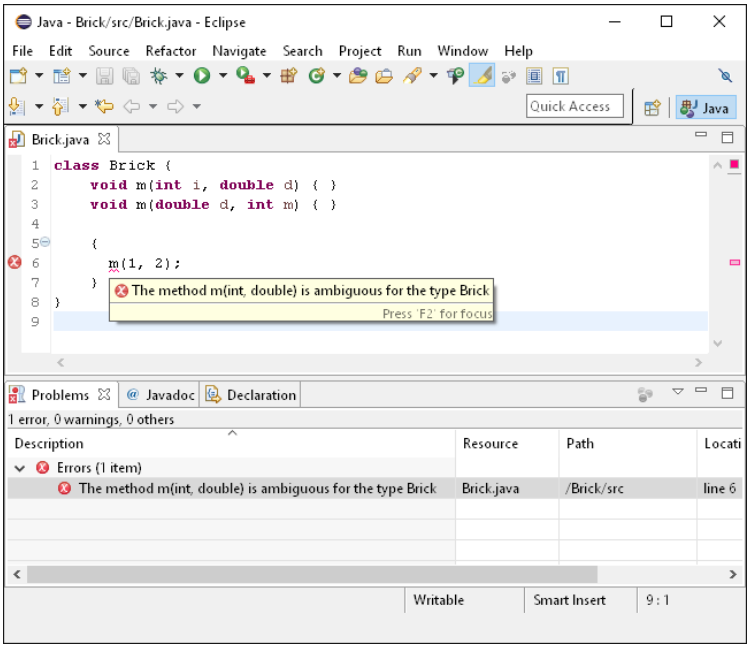
3.3 Visual Representations of Program Analysis

Thus far, we have presented error messages predominantly through terminal, text-only interfaces. In addition to terminals, many software developers utilize modern integrated development environments (IDE) such as Eclipse [105], Visual Studio [259], and IntelliJ [183], as a core part of their development process [367]. 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) are unbundled and rendered 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 a 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 can be searched, sorted, and filtered. The diagnostic object components, such as the description, file, and line number, are presented as indi-

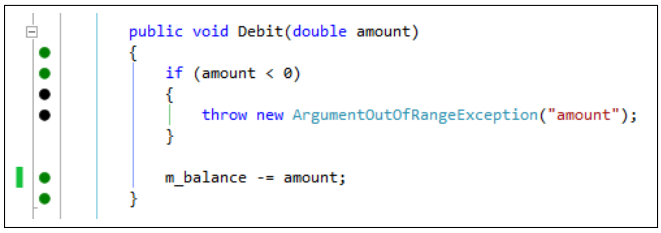


(a) Visual Studio

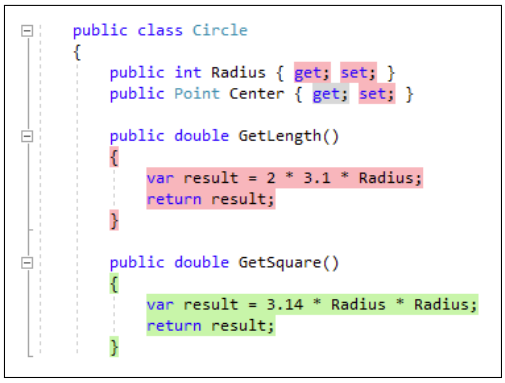


(b) Eclipse

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 integrate 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.

vidual columns in the table. Within the source code editor, a red wavy underline is presented under the corresponding source code for each 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 extensions such as NCrunch [276] and dotCover [182], leverage margin indicators as well as source code highlighting, to indicate unit test coverage (Figure 3.3).

The Eclipse IDE in Figure 3.2b uses similar affordances to Visual Studio, and other popular IDEs, such as Atom [131], Xcode [12], and Sublime Text [360], 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 [270] displays diagrammatic control flow and data flow information and overlays 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 7 is influenced by refactoring annotations. MrSpidey [120] is designed to

be a user-friendly interactive static debugger for Scheme; 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 [206].

The Rust Enhanced [331] extension for Sublime Text inlines error message information through *phantoms*. Phantoms are like tooltips, except that the information is directly embedded within the text view and remains persistent. Lieber, Brandt, and Miller [227] found that developers adopt unique debugging strategies when always-on information is available.

Representations of error messages also present information alongside the source editor. Risley and Smedley [322] 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, called a petal, corresponds to a category of errors [269]. 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 [187].

3.4 Errors Developers Make

A typical compiler, such as for Java or C#, recognizes several thousand possible program analysis errors. 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, Sadowski, Elbaum, and colleagues [342] conducted an empirical case study at Google of 26.6 million Java and C++ builds produced over nine months by thousands of developers. 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. 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, such as students using Java in the BlueJ IDE [208], the situation is even worse. Through telemetry of over 37 million compilation events, Altadmri and Brown [6] identified that nearly 48% of all compilations fail due to a compiler error. Similar to the errors made by expert developers at Google, novices too produced primarily syntax errors, type errors, and other semantic errors. For some reason, it appears that experience alone isn't making these errors go away.

Using a Python corpus of 1.6 million code submissions, of which 640,000 resulted in an error (approximately 40%), and re-examining the BlueJ dataset, Pritchard [310] model-fit the distribution of these error messages and found that they empirically resemble a Zipf-Mandelbrot distribution. Such power-law distributions have a small set of values that dominate the distribution, followed by a long tail that rapidly diminishes.

Other researchers have also discovered power-law distributions for error messages. In an early study by Litecky and Davis [230] in 1976, the authors empirically obtained and catalogued COBOL error diagnostics to identify a Pareto distribution: 20% percent of error types accounted for 80% of the total error frequency. In more recent work, a study by Jackson, Cobb, and Carver [178] found that the top ten errors represent nearly 52% of all errors, and that the top twenty represent 62.5% of all errors. And 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. Although Seo, Sadowski, Elbaum, and colleagues [342] did not model-fit the distributions (their paper, Figure 7), a visual inspection of Java and C++ suggests that a similar power-law effect is present.

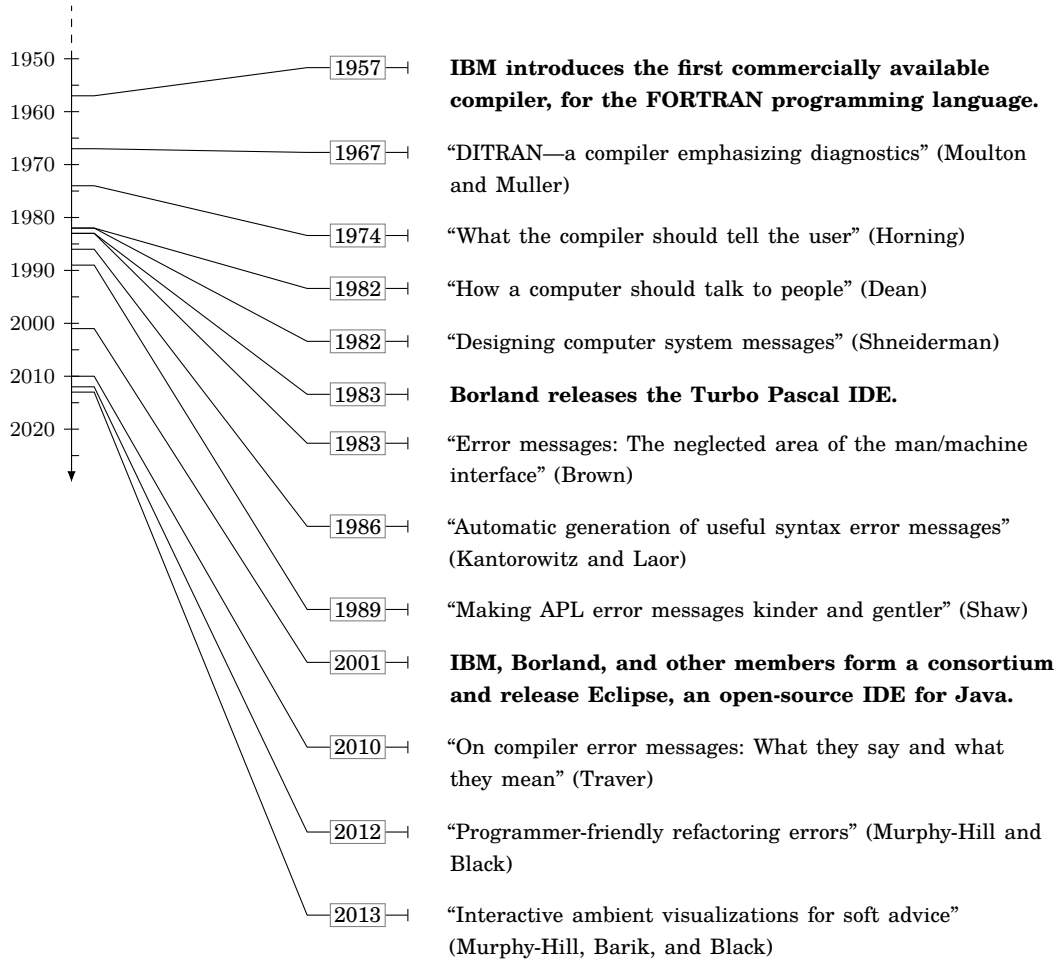


Figure 3.4 History of design guidelines for error messages in program analysis tools. In tandem with design guidelines, significant shifts in industry tools are indicated in **bold**.

The triangulation of these multiple data sources indicates two consistent features about errors across programming languages. First, the dominant errors, both in terms of cost and frequency, are relatively consistent irrespective of developer experience. This is interesting because it suggests that improvements to these error messages are likely to benefit a spectrum of developers. Second, the power-law distribution suggests that addressing even a small number of dominant errors would make program analysis tools more usable.

3.5 Design Guidelines for Error Messages

Researchers have contributed a lengthy history of design guidelines—rules that toolsmiths can apply to increase the usability of the tool [100]—to improving error messages in program analysis (Figure 3.4 on the previous page). In this section, we present the guidelines here narratively in terms of their influence towards rational reconstruction (having said that, an unadulterated inventory of the guidelines is also available in Appendix D).

The earliest guidelines by Moulton and Muller [267] in 1967 offer recommendations that today would seem astonishingly self-evident. For example, they recommend that all error messages be presented in terms of the source language: concretely, that error messages should provide the name of the variable actually used in the source code rather than the variable's memory address. There is also some consideration given to the structure: that ideally, error messages should suggest a correction.

Horning [172] echoes similar guidelines to Moulton and Muller; namely, that good error messages are source-oriented. Moreover, readable error messages should describe the symptom of the problem. Horning introduces some nascent ideas towards relationality: he notes that if an inconsistency involves information from some other part of the program—for example, a declaration—then that information should be displayed, even if only as a location. Happily, Horning contributes the notion that it is important to think of the interaction between developers and tools as a dialogue.

Less happily, both the guidelines by Moulton and Muller [267] and Horning [172] consider the extent of this dialogue, from tools to developers, to be in the form of substitutional fixes: "expected *this*, found *that*." These types of error messages are useful and certainly have their place (as we'll see in Chapter 8), but they aren't actually rational reconstructions. Nevertheless, we can consider these guidelines as primordial building blocks towards thinking about rational reconstruction. Dean [89] extends this thinking further by introducing an assortment of human-centric guidelines, from various inspirations in psychology. In particular, relevant to rational reconstruction are guidelines for presenting messages in a conversational

language, such as in everyday situations, and guidelines suggesting that having to re-read error messages is a signal that they are confusing (we'll revisit this in Chapter 6). As with earlier guidelines, Dean also notes the importance of showing relationships.

Shneiderman [349] adds little more in terms of new guidelines ("address the problem in the user's terms") for rational reconstruction; however, this work is influential because it contributes a controlled experiment in COBOL for different error message presentations. The study found that applying these design guidelines to error messages led participants to successfully resolve defects more often than the baseline messages.

In 1983, the Borland Turbo Pascal integrated development environment was released to the development community. This IDE (and others from the same era) introduced a host of a novel affordances, including multi-window environments, which enabled developers to interact—for example—with both a source code window and an error output window simultaneously. Responding to the affordances brought about by the IDE, Brown [50] advocated that these "modern" systems also presented opportunities to significantly improve error messages. His guidelines leverage these visual affordances: using color to identify offending symbols on the source code, providing a contextual window that displays several lines of source code before and after the error message, and providing some visual markers to show the developer where in the source code the error has occurred. In isolation, these guidelines aren't rational reconstructions; however, it's easy to see how highlighting and visual markers can allow rational reconstructions to more expressively communicate their explanations.

Around the same time as Brown [50], Kantorowitz and Laor [197] probe at some initial directions towards encoding explanation using visual presentations for syntax errors. For example, during a syntax error, they use '|' to indicate the first symbol that does not match the syntax of the language, a '.' to indicate where the compiler has resumed analysis, and a short message suggesting other tokens to substitute (',' or ';'):

```
const n=5: b:=4;
         |-----. { , ; } EXPECTED
```

Despite the improved visual presentation, these error messages are still fundamentally of the found-expected form. It is not until guidelines by Shaw [346] that we diverge from this form of error presentation. Specifically, Shaw introduces an essential contribution in rational reconstruction: that error messages should provide a reason for the error. Rather than returning (in APL), for example:

```
Length Error
       2 3 + 4 5 6
```

they propose the compiler instead emit:

```
Shape Error - Length of corresponding axes must be equal.
       2 3 + 4 5 6
       2 | 3
       ^   ^
```

The second error message gives a reason for why the length (shape) is invalid. As in Kantorowitz and Laor [197], the error also uses some visual representations, through '^' and '|'.

After Shaw, there's a roughly twenty-year gap in contributions to design guidelines, despite another substantial improvement to integrated development environments around the mid-1990s: the emergence of actual graphical user interfaces, versus text-based user interfaces, that could exploit high-resolution displays. Indeed, it's not until Traver [381] in 2010 that design guidelines are revisited. And even here, Traver presents a modernization of existing guidelines—namely, Horning [172]—and explains these guidelines in terms of contemporary C++ compilers (Table 1, their paper). I do have some intuitions about why this gap exists in the literature—however, explaining it isn't necessary to support the thesis.

Two additional design guidelines by Murphy-Hill and Black [270] and Murphy-Hill, Barik, and Black [269] (and described prior in Section 3.3) both contribute elements of rational reconstruction through presenting visual annotations—such as boxes and arrows—adorned directly on the source code that the developer actually edits. Both design guidelines present several useful recommendations, but of these the guideline of relationality is closest to ideas within rational reconstruction: the guidelines advocate that violations often arise from multiple program elements in the code, and that visualizations should reveal these relationships. In other words, the visualization should present a trace-explanation form of rational reconstruction.

In Section 9.2, I contribute design guidelines derived from the studies on rational reconstruction in this dissertation, and describe how my guidelines extend and differ from existing guidelines for error messages.

3.6 Conclusions

This state-of-the-art review described a design space of text and visual representations for error messages, through the lens of contemporary program analysis tools. A survey of distributions for error messages that developers actually make revealed that improvements to even a relatively small number of error messages could substantially benefit developers. The review also identified several points within this design space through which we can apply rational reconstruction to advance the state-of-the-art.

The studies in this dissertation address two such points. First, towards advancing our understanding of visual representations, we investigate box-and-arrow representations as a form of rational reconstruction to explain error messages (Chapter 7). Second, towards improving text explanations in template diagnostic error messages, we investigate a rational reconstruction model to improve the structure and content of these error messages (Chapter 8). There remain several questions about how to support computationally-constructed explanations, such as in type inference and model checking; these topics are also important but beyond the scope of the dissertation.

This chapter presented a chronology of existing design guidelines, derived from the state of program analysis tools for that time period. In continuing this tradition, I too synthesize the research conducted within this dissertation and contribute a set of design guidelines for contemporary program analysis tools (Section 9.2).

In the next chapter, we conduct a scoping review. The scoping review positions the studies on rational reconstruction in this dissertation within a broader taxonomy of programming language research for error messages.

4 | What Do We Know About Presenting Human-Friendly Output from Program Analysis Tools?

Always take a banana to a party.

The Doctor

Rational reconstructions are only one of many research approaches towards improving the presentation of error messages.[1] Thus, to position the work in this dissertation within a broader taxonomy of programming language research for error messages, we conducted an interim scoping review of the proceedings from Programming Language Design and Implementation (PLDI), from 1988–2017. 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 their *program analysis output* as the primary investigation.

Notably, 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, with the longer-term goal of bridging HCI research with program analysis tools. Therefore, the contributions of this scoping review are:

[1] Significant portions of this chapter were previously published as 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 *Evaluation and Usability of Programming Languages and Tools (PLATEAU)*, 2017.

- 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 developers, instantiated as a taxonomy (Section 4.2). Our taxonomy is agnostic to a particular mode of output, such as text or graphics.

4.1 Methodology

4.1.1 What is a Scoping Review?

In this study, we conduct a *scoping review*—a reduced form of the traditional systematic literature review [13, 203]. 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 [340]. 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 chapter.

4.1.2 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 developer 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 developer. 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 (Section 4.2). For analysis, we opted for a narrative-commentary approach [141] in which we summarized the contributions of each of the papers with respect to human-friendly presentations.

4.1.3 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 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.

4.2 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 developers. Intentionally, we labeled the taxonomy features such that they do not commit to a particular text or visual affordance. For example, in a text interface, the feature of ranking (Section 4.2.7) 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 developer would select their desired option.

The identified taxonomy of presentation features are summarized in Table 4.1. 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.

4.2.1 Alignment

In alignment, program analysis output is presented in a representation that is already familiar to the developer.

Within this feature, Pombrio and Krishnamurthi [304] 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 developer. Pombrio and Krishnamurthi [304] introduce a process of *resugaring* to allow computation

Table 4.1 Taxonomy of Presentation

Feature	Description
Alignment (Section 4.2.1)	Presents analysis output in a representation that is already familiar to the developer.
Clustering and Classification (Section 4.2.2)	Organizes or separates information in a way that reduces cognitive burden.
Comparing (Section 4.2.3)	Facilitates comparisons when the developer has a need to examine or understand differences between two or more versions of their code.
Example (Section 4.2.4)	Examples and counterexamples are forms of output that provide evidence for why a situation can occur or how a situation can be violated.
Interactivity (Section 4.2.5)	The developer can interact with tools either before the output is produced, or collaboratively interact with multiple iterations of output, to arrive at a solution.
Localizing (Section 4.2.6)	Presents and pinpoints the relevant program locations for an error.
Ranking (Section 4.2.7)	Orders the output of the analysis in a systematic way.
Reducing (Section 4.2.8)	Reduces or filters a large design space of allowable analysis output, using some systematic rule, before this output is presented to the developer.
Tracing (Section 4.2.9)	Applies a form of slicing that involves flows of information, and supports understanding how information propagates across source code.

reductions in terms of the surface syntax. With similar aims, the AutoCorres tool uses a technique of *specification abstraction*, to present developers with a representation of the program at a human-readable abstraction while additionally producing a formal refinement of the final presentation [143].

Notions of natural language and readability find their place in several PLDI papers. Qiu, Garg, Ştefănescu, and colleagues [314] 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 [359] 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* [185]. 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 developers 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 developer to draw incorrect conclusions as a result of internal optimizations by the compiler. Earlier work by Brooks, Hansen, and Simmons [47] and Coutant, Meloy, and Ruscetta [75] also provide techniques that allow developers to reason about optimized code through mapping the state of the optimized execution back to the original source. For example, Brooks, Hansen, and Simmons [47] use highlighting, boxing, reverse video, grey-scale shading, and underlining to animate and convey runtime program behavior, overlaid on the original source code, to the developer.

4.2.2 Clustering and Classification

Clustering and classification output aims to organize or separate information in a way that reduces the cognitive burden for developers. For example, Narayanasamy, Wang, Tigani, and colleagues [273] 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 developer with a reproducible scenario of the data race to help the developer understand how it manifests.

Liblit, Naik, Zheng, and colleagues [226] 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 [225]. Ha, Rossbach, Davis, and colleagues [148] 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.

4.2.3 Comparing

Comparisons occur in program analysis tools when the developer has a need to examine or understand differences between two or more versions of their code. Within this feature, Hoffman, Eugster, and Jagannathan [168] 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 [173] identifies both semantic and text differences between two versions of a program, in contrast to traditional diff-tools that treat source as plain text.

4.2.4 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 [256]. 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 [422] 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 [320]; the underlying machinery is based on *generic fixpoint computations* which invokes a *modular reducer*.

Padon, McMillan, Panda, and colleagues [287] 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 [287] 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. Nguyễn and Van Horn implement a tool in Racket to generate counterexamples for erroneous modules and Isradisaikul and Myers [176] 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 developers implement concurrent data structures; it uses a *counter example guided inductive synthesis algorithm* (CEGIS) to converge to a solution within a handful of iterations [362]. 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 [220] 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 [63] 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.

4.2.5 Interactivity

We identified several papers whose tools support interactivity. That is, the developer 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 developer 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 text selections [223]. As the developer adds production rules to the grammar, Parsify uses colored regions overlaid on the examples to convey progress to the developer.

Live programming is a user interface capability that allows a developer to edit code and immediately see the effect of the code changes. Burckhardt, Fahndrich, Halleux, and colleagues [51] 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 developer 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 developers 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 [415]. The tool allows an *iterative* process through three *tiers* which assist developers at different levels of abstraction, from *transactions* at the highest-level tier to *lifetime relationships* at the lowest level tier.

Chameleon assists developers in choosing an abstract collection implementation in their algorithm [343]. 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 developers. 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 developer 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 developer can then—through applying predicates—provide feedback on whether the given analysis result is desirable.

Finally, MrSpidey is designed to be a user-friendly debugger for Scheme [120]; 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 developer through a value flow browser which overlays arrows over the program text. The developer can interactively expose portions of the value graph.

4.2.6 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 [421] 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 [195]. 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 [41].

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 [366] propose a technique called *thin slicing* which helps developers better identify bugs because it identifies more relevant lines of code than traditional slicing. Analogous to thin slicing, Zhang, Gupta, and Gupta [423] 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.

4.2.7 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" [62]. Chen, Groce, Zhang, and colleagues [62] 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 developer [38].

Coppa, Demetrescu, and Finocchi [73] present a profiling methodology and toolkit for helping developers 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 developer should spend effort parallelizing; the tool identifies these regions through a *hierarchical critical path analysis* and presents to the developer an ordered (by speedup) parallelism plan as a list of files and lines to modify [126].

Perelman, Gulwani, Ball, and colleagues [296] provide ranked expressions for completions in API libraries through a language of *partial expressions*, which allows the developer to leave "holes" for the parts they do not know.

4.2.8 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 [236]. Specifically, VMV is designed for scenarios in which developers desire to fix *new* defects introduced since a previous release.

4.2.9 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 [283] present a system that answers control-flow queries posed by developers 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 developers to interactively explore *information flows*—through *program dependence graphs* within their applications—and investigate counterexamples [186]. Taint analysis is another information-flow analysis that establishes whether values from unstructured parameters may flow into security-sensitive operations [383]; 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 [329], use data-flow analysis techniques to track errors as they propagate through file system code.

To support algorithmic debugging, Faddegon and Chitil [114] developed a library in Haskell, that, after annotating suspected functions, presents a detailed *computational tree*. Computational trees are essentially a trace to help developers 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 [14]; it uses *first-fault diagnosis* to discover what went wrong the *first* time the fault is encountered.

MemSAT helps developers 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 [378].

The Merlin security analysis tool infers *information flows* in a program to identify security vulnerabilities, such as cross-site scripting and SQL inject attacks [232]. Internally, the inference is based on modeling a *data propagation graph* using *probabilistic constraints*.

4.3 Conclusions

In this chapter, we conducted a scoping review of PLDI from the period 1988–2017. We identified and cataloged papers for program analysis tools that discussed or made contributions to the presentation of output intended for developers.

The resulting taxonomy of presentation clarifies the contributions of rational reconstruction theory and positions those contributions with respect to broader research efforts towards improving program analysis output. Concretely, the taxonomy suggests that the research on rational reconstruction conducted in this dissertation most closely advances research efforts in alignment and tracing features of the taxonomy, for example, through justification-explanations and trace-explanations, respectively (Section 2.4). Importantly, the scoping review underscores taxonomy features to which rational reconstruction does not noticeably contribute. For instance, rational reconstruction does not make contributions towards clustering and classification, comparing, examples and counterexamples, localizing, ranking, or reducing. Nevertheless, rational reconstruction can leverage advances in these taxonomy features when formulating explanations.

In the next chapter, we present a mapping review. The mapping review is comparable to a scoping review in that it also positions my research on rational reconstruction within a broader context. However, whereas the scoping review connects rational reconstruction to the *single* discipline of programming language research, the mapping review positions rational reconstruction conceptually with *multiple* neighboring research disciplines, such as artificial intelligence and human factors.

5 | Where Do Error Messages as Rational Reconstructions Fit Within 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 of related research in computer science and other disciplines? The mapping review conducted within this chapter delineates collections of research pertinent to the design of error messages, in strongly interrelated, neighboring research disciplines that include program comprehension (Section 5.1), human factors in error and warning design (Section 5.2), expert systems (Section 5.3), structure editors (Section 5.4), and error messages for novices (Section 5.5). Through establishing linkages between rational reconstruction and concepts within these collections, I defend the novelty of my thesis.

5.1 Program Comprehension in Debugging

Program comprehension is a cognitively-demanding activity, coordinating and competing with a number of parallel mental processes that include language [354], learning [317], attention [406], problem-solving [207], and memory [292]. 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 theories can support, complement, and integrate with respect to the theory of rational reconstruction to inform the design of error messages. In this section, I review three established theories of program comprehension during debugging and maintenance activities: plans (Section 5.1.1), beacons (Section 5.1.2), and information foraging theory (Section 5.1.3). I conclude with an explanation of how rational reconstruction fits with these theories (Section 5.1.4).

5.1.1 Plans

Shneiderman and Mayer [350] 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, called *plans*. When executing these plans results in an error message—a plan failure—the authors proposed that developers attempt to resolve the error in one of two cognitive contexts.

In the first context, the developer has a correct plan for what the program is supposed to do, but has at some point introduced a mistake in representing that idea in the source code in a way that has resulted in an error message. That is, the developer has incorrectly transformed *internal semantics* to *program statements*. Shneiderman and Mayer argued that developers can usually recover from such errors once they figure out the differences between what they intended to write and what they actually wrote.

In the second context, the developer has incorrectly transformed the *problem solution* to the *internal semantics* of the language, resulting in an error message. This second plan failure is more insidious: although the error message can appear to be the result of incorrectly translating internal semantics to program statements, the true cause is actually due to a misconception about how the internal semantics of the language actually work. In this context, what the developer intended to write is actually erroneous. Often, this second context requires the developer to completely reevaluate their programming strategy [350].

Other researchers have also considered debugging as the construction or execution of programming plans [49, 221, 323, 363, 364, 370, 392]. The planning theories have several differences: they are sometimes top-down comprehension models [48, 363] (that is, developers begin with a hypothesis about the program, then decompose

parts of the code to verify it), bottom-up comprehension models [222, 295] (that is, developers gather smaller chunks of code and piece them together to understand the program), or integrated models [249] (that is, developers interleave top-down and bottom-up planning). Nevertheless, there are many commonalities among them: namely, they are all cognitive processes that construct a mapping between the source code and a cognitive representation of the problem, and program understanding involves reconstructing some or all of these mappings.

5.1.2 Beacons

But what is the link between source code and mapping during planning and reconstruction? One theory is that developers search for beacons [48, 77, 406]—lines of code that serve as indicators of a particular structure or operation; developers use the presence or absence of these beacons to confirm or reject hypotheses about errors. To search for these beacons, developers employ a process called program slicing, in which developers break apart large programs into smaller, coherent pieces; these pieces need not be textually contiguous [401]. 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 [355] confirmed lower activation of brain areas during comprehension based on semantic cues, confirming that beacons ease comprehension.

5.1.3 Information Foraging Theory

An alternative theory for program comprehension that contrasts with the classical information processing theories thus far is information foraging theory [218]. Information foraging theory explains and predicts how developers navigate source code through an information-seeking model driven by cost minimization [122].

Specifically, the theory proposes a parsimonious predator-prey model in which a *predator* (for example, a developer who is debugging an error message) seeks one or more *prey* (for example, pertinent information useful to understanding and resolving the error). To do so, the predator (developer) would follow various *cues* (for example, affordances within the environment such as red wavy underlines) to

identify information *patches* (for example, a method in a source code file). Information foraging theory suggests that the predator (developer) decides which cues to follow or not follow based on *information scents*—essentially, a perceived likelihood measure that following the cue would successfully lead them to discovering prey (pertinent information). Thus, we can think of information foraging theory as a greedy search: at every decision point, the developer chooses the interaction in the environment that they estimate would maximize information gain while minimizing the cost to seek.

A salient property of information foraging theory is that the *environment* alone explains the developer's behavior. In other words, in contrast to classical information processing theories, information foraging theory does not require knowledge of what is inside the developer's mind [218].

5.1.4 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, and orthogonal to the ecological theory of information foraging, rational reconstruction can be used to support both cognitive and ecological theories. For instance, rational reconstructions 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. And within information foraging theory, rational reconstruction may act as a navigational aid that influences the developer's self-estimates for what information to seek.

5.2 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 [418], medication [116, 265], cigarettes and alcohol labels [30, 217], and on signs at beaches [246]. There are several noteworthy literature reviews in human factors which cover these and other applications for warnings and errors [92, 107, 216, 219, 293, 327, 408].

But are theories of warning design [107] for *physical objects* applicable to the design of error messages in *digital systems* like program analysis tools? Research suggests that it is [152, 299]: Pieterse and Gelderblom [299], for example, successfully applied warning design theory—originally studied in the context of the physical world—to the design of error messages in digital systems like online Internet banking. As part of their study, Pieterse and Gelderblom [299] adopted guidelines from warning design theory, and conducted a heuristic evaluation in which experts used these guidelines to effectively identify problems with error messages.

Mapping findings from warning design theory in the physical world to digital systems provide additional lenses 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 [71]. 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.

Warning design theory can also inform the design space of error messages as a medium (for example, as we've seen in Section 2.2.5). These dimensions include color [204], size [23, 45], location [217, 410], language of the warning text [2, 102, 152, 409], pictorials or icons [86, 180, 375], expertise [46], habituation [154, 202], and social influence [96, 108, 124]. These dimensions 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 how they would prioritize multiple errors.

Nevertheless, warning design theory does not obviate the need for rational reconstruction, because warning messages (at least, in warning design theory) aren't about *explanation*; rather, they are about presenting information in a way that allows people to make judgments about the level of *risk* they are willing to accept or not accept [216]. In short, although warning design and rational reconstruction have several commonalities, the two theories diverge in that they have different aims.

5.3 Expert Systems

Expert systems [228] are computer programs that reconstruct the expertise and reasoning capabilities of qualified specialists within limited domains [312], for emulating human expertise in well-defined problem domains [321], and for computationally representing task-specific human-knowledge of experts to systems [224]. One of the key goals of expert systems research is for systems to be able to explain their reasoning to users [312]. Some early research even suggested that program analysis tools could be investigated as expert systems [40, 151, 194, 231, 335]. Hence, the theory of rational reconstruction proposed in this dissertation can be seen as a natural extension to expert systems research.

Although first-generation expert systems in the 1970s, such as DENDRAL [115] and MACSYMA [245], focused primarily on performance and problem-solving—and not explanation—second-generation systems, notably MYCIN [66, 351], introduced the distinguishing feature of explanation facilities as part of the problem-solving process [88].

MYCIN became a model for succeeding systems [312], and the development of expert systems became a forcing function in artificial intelligence research for computational explanations and problem-solving [368]. 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 unable to explain the result? …I think not." And Chandrasekaran and Swartout [57] observed that "explanation of a knowledge system's conclusions can be as important as the conclusions themselves".

In addition to the technical contributions of expert systems, people find their explanations to be useful across several dimensions [144]. An evaluation by Ye and Johnson [417] 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 causal argument or rationale for each step taken by the expert system (justification), and 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 user attitudes towards the system. Lim, Dey, and Avrahami [229] found that explanations describing why a system behaved a certain way resulted in better understanding and stronger feelings trust. Wick and Thompson [405] proposed a computational model of reconstructive explanations for expert systems: that effective explanations often need to substantially reorganize the internal line of reasoning and decouple it from the line of explanation that is ultimately presented to the user. However, the authors did not actually evaluate these explanations.

Though both novices and experts find explanations to be useful, they use the explanations in different ways. Mao and Benbasat [242] showed that people requested explanations 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 [118] proposed that systems should not attempt to generate a perfect, one-shot explanation: instead, systems should initially provide a 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 [25, 171, 188, 254, 261, 288].

Standalone expert systems eventually declined in popularity [10]; nevertheless, their ideas were embedded in specialized contexts such as recommendation systems [289, 325, 377], and as intelligent tutoring systems [388]. The prolific rise and fall of expert systems brought to attention three critical weaknesses, particularly with respect to explanation [191]:

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 gave 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 also for comprehensible error messages in program analysis tools. It is precisely these first two points 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 contemporary program analysis tools be investigated as a form of expert system? If so, would explanations be as useful as they appear to be for classical 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 ideas from expert systems research to inform the implementation of program analysis tools.

5.4 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, these editors 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 [376] 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 [238], Unison [64], and isomor$f$ [199].

There are some arguments in favor of structure editing approaches, beyond preventing syntax errors [213, 344]. First, if developers think in terms of tree structures in their own mind, editors should support this mode of thinking [133, 201]. 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 [391]. Third, proponents of structure editing argue that antiquated text-based representations are inefficient ways of composing programs [239].

Unfortunately, the arguments in favor have not held up to scrutiny [387, 398]. Critics of structure editors have described them as restrictive, inflexible, and inefficient [201]. Thinking in terms of tree structures turns out not to be natural at all [167]: complex tasks require substantial experience with the underlying abstract tree structure, and additional developer experience doesn't lead 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 [373]. 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 [284]. Conventional structured editors forbid such common workflows [380].

Lightweight 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 [400], intelligent copying-and-pasting of code fragments [394], auto-completion [240], and refactoring are examples of tools whose usage can reduce, but not entirely prevent, the introduction of syntax errors [130]. But allowing for syntax errors to arise brings us right back to this dissertation: there remains a need to support comprehensible error messages. And even theoretically-perfect structure editors do not prevent other categories of errors, such as semantic errors, that developers introduce in source code.

5.5 Error Messages for Novices

Despite cognitive differences in the way novices and experts problem-solve during debugging (Section 5.1), examining literature for novices can provide insights into techniques (Section 5.5.1) and perspectives (Section 5.5.2) on the design of rational reconstructions 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 [24, 70, 87, 113, 117, 125, 158, 169, 198, 267, 365, 403, 404, 419]. In this section, I focus on literature applicable to modern programming environments as investigated within this dissertation.

5.5.1 Mini-Languages: The Rule of Least Power

In programming language design, the "rule of least power" suggests that one should choose the *least* powerful language suitable for a given purpose [33]. Such languages, Berners-Lee and Mendelsohn [33] argue, are usually easier for both program analysis tools and humans to reason about. One way to apply this rule is by designing a "mini-language": taking an expressive programming language (for example, C++), and adopting only a subset of its language concepts (for example, C++, but disallowing exception handling).

Disallowing programming concepts can result in more accurate rational reconstructions, particularly when there would otherwise be multiple plausible explanations for a problem [158]. 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 is unlikely to be known to or used by a novice—then it's reasonable to assume that the novice actually intended to simply perform a greater-than operation: `x > m`. In the latter case, the problem is likely a syntax error, and the program analysis tool can present the more appropriate rational reconstruction.

There are several mini-languages that apply the rule of least power to improve error messages. MiniJava—a teaching-oriented implementation of Java—removes much of the standard library, eliminates inner classes, prohibits the do-while statement, and simplifies some other language constructs [324]. Language levels in DrRacket provide full-fledged languages, staged into multiple sub-languages that track programming concepts that students have learned in their course [244]. Helium, a variation of Haskell, aims to provide higher-quality error messages tailored for students [162]. Because Helium removes type classes from the language, programs written in Helium are not generally compatible with Haskell.

Prohibiting expressive concepts in programming languages is sometimes necessary to do because the resulting error messages are otherwise unusable, even for experts. For example, the Google Style Guide explicitly prohibits certain language constructs, instead suggesting alternative and sometimes less expressive implementations [136]. The guide disallows the use of preprocessor macros entirely, because program analysis tools are ill-equipped to present adequate error feedback in their presence: "every error message from the compiler when developers incorrectly use that interface now must explain how the macros formed the interface" [136]. Similarly, template programming—despite being incredibly useful [1]—is also strongly

discouraged because it leads to poor compile-time messages, even for simple interfaces [136]. Because these language restrictions are a consequence of poor error messages, the motivation for investing in rational reconstruction is clear: if better error messages were available, developers could more freely use these expressive constructs in their code.

5.5.2 Enhancing Compiler Error Messages

Although there are substantive differences between novices and experts in debugging [146, 252, 389], 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 learned in one programming language to another [338, 413], or when contributing to unfamiliar code [209, 301].

Unfortunately, there is little consensus on what helps novices with error messages, even though researchers typically agree that error messages from industrial program analysis tools are inadequate for novices [125, 281, 294, 411, 414]. An extensive literature review by Qian and Lehman [313] offers a defense for this position, through examining barriers that novices have with syntactic knowledge, conceptual knowledge, and strategic knowledge. In short, 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 is intended for learning, whereas feedback for professional developers is intended to help them productively identify and resolve the issue [352].

Enhanced error messages, then, are the revisions made to existing error messages in order to make them more appropriate for novices. For example, Denny, Luxton-Reilly, and Carpenter [93] implemented a 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 they attempted to address 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 [297] 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 to the error messages and some hints for what to check. Pettit, Homer, and Gee [297] found no measurable benefit to students: they also suggested that perhaps students don't attentively read the error messages, although students overwhelmingly self-reported reading them. Nienaltowski, Pedroni, and Meyer [281] 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 executed study by Prather, Pettit, McMurry, and colleagues [309] found that enhanced compiler error messages did not quantitatively show a substantial 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. These papers have interesting ideas, but their studies have either limited or no evaluation, yield inconclusive results, or do not offer a principled rationale for how the error messages were improved [43, 70, 113, 123, 125, 175, 214, 339, 399, 403, 414].

Two notable exceptions are studies by Becker, Glanville, Iwashima, and colleagues [27] and Marceau, Fisler, and Krishnamurthi [244]. In contrast to studies that did not identify any significant improvements with enhanced error messages [93, 297], 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 [244] 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—despite the considerable effort

the development team invested into the design of error messages—that the team lacked a clear model of feedback to systematically guide the error message design process. They have conducted subsequent studies to develop a rubric for assessing the performance of error messages [243], and these rubrics have been applied as formal processes for assessing and evaluating error reports, such as in Pyret [411].

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 are not all that successful. Even when the error messages perform better than baseline errors, we gain few insights into why those messages are better. Consequently, theories such as rational reconstruction—although investigated for expert developers within this dissertation—could also be applied to help novice developers comprehend error messages (Section 9.4).

5.6 Conclusions

This chapter presented a mapping review relating rational reconstruction to strongly connected, neighboring research collections. By forcing this close association, I demonstrated that the theoretical framework advanced in this dissertation (Section 2.4) is often conceptually similar—yet never completely fits—to any another neighboring collection. In doing so, I defended the novelty of the thesis.

Having established the novelty of the thesis, we can now contribute the necessary evidence to support it. Accordingly, the following three chapters (Chapters 6 to 8) present research studies that tackle the claims of the thesis. For convenience, we reiterate the thesis statement below:

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

6 | Do Developers Read Compiler Error Messages?

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

The Doctor

If we invested significant effort towards improving error messages, would these investments actually make a difference?[1] How do developers use error messages to comprehend and resolve problems in the code? How difficult is it to read these error messages? Do developers—whether novices or experts—even read them at all?

In this chapter, we investigate open questions [93, 243, 270, 342] relating to how developers perceive and comprehend error messages through the various ways in which they are presented in integrated development environments. To understand if and how developers read error messages, we conducted an eye tracking study with 56 developers, recruited from undergraduate and graduate software engineering courses at our university, and observed as they resolved common, yet problematic error message defects in their 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, and motivate the theoretical framework of rational reconstruction that drives the research agenda within this dissertation.

Specifically, our study finds that:

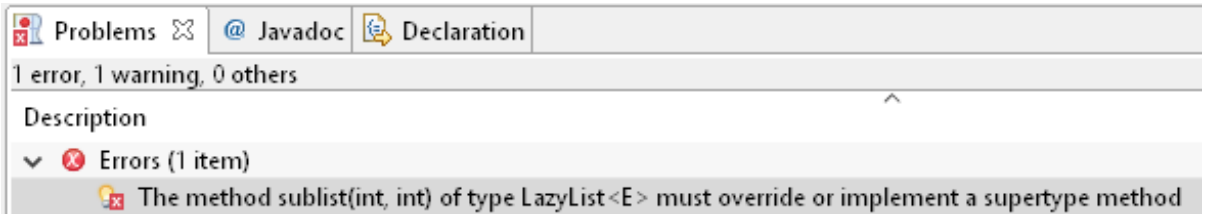
[1] Significant portions of this chapter were previously published as T. Barik, J. Smith, K. Lubick, and colleagues, "Do developers read compiler error messages?" In *International Conference on Software Engineering (ICSE)*, 2017, pp. 575–585.

- 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.

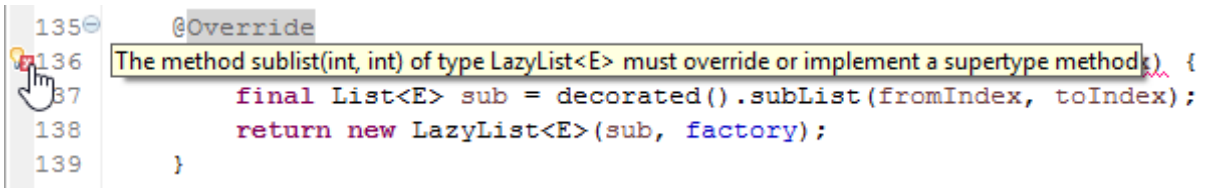
6.1 Motivating Example

Let's consider how error messages can become costly for developers to resolve, through precisely observing where and how developers look at information in their IDE. We'll do so through 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:

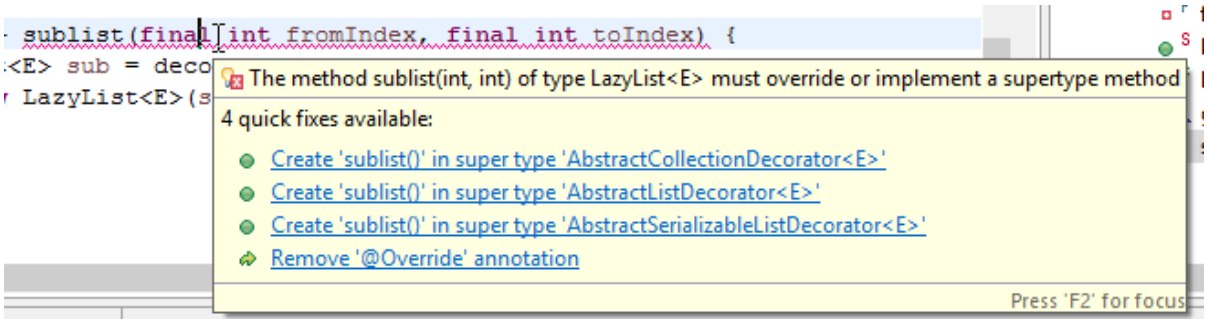


He silently reads the message about the `sublist` method, and then double-clicks the error in the problems pane. This redirects the IDE to the source editor, and Barry confirms that the error is related to the code that he just added. In the margin of the source editor, he now notices a light bulb icon,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

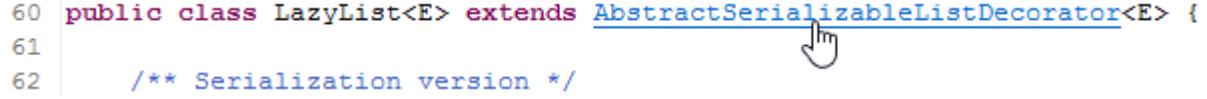


Unfortunately, the popup is less helpful than he expected because it repeats the message he has already seen in the problems pane. Moreover, the error popup text is obscuring the method signature which is where he believes the problem is actually located.

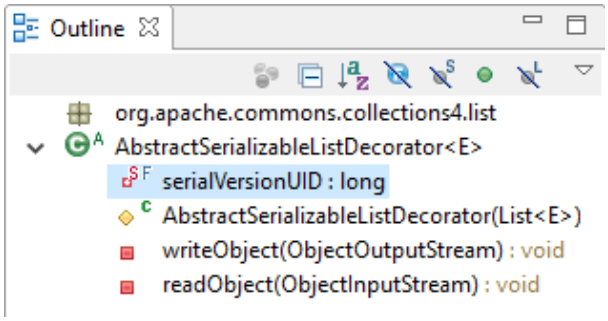
Next, he notices the red wavy underline, which in Eclipse indicates the presence of an error. Barry hovers over the underline, releaving a *Quick Fix popup*. Unlike the error popup, the Quick Fix popup provides possible "fixes" that change the source code, in addition to the error message. He spends 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?

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

6.2 Methodology

6.2.1 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 [342], where costly is defined as frequency of the error times the median resolution time. Additionally, the results of this research question provide descriptive statistics to identify if some categories of defects are more difficult to resolve than others. Identifying which categories of defects are more difficult to resolve can help pinpoint where rational reconstructions may be most beneficial to developers.

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. If this were indeed the case, then providing developers with rational reconstructions would be unlikely to substantially help developers; after all, they didn't even need the existing error message to resolve their problem.

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. Thus, we want to understand the extent to which poor error messages are harmful to the developer. If the difficulty of resolving compiler errors is attributable to the error message, then there is strong justification for pursuing research in rational reconstruction.

Table 6.1 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 [342]. The Eclipse version of this message is shown to the participants.

6.2.2 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 [353]. 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 chapter 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 [342]. 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 6.1). 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.

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 6.1) 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 [128] 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 [105] 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 [183] and Visual Studio [259]. 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.

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

6.2.3 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 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 an error messages 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

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

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.

6.2.4 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 [285] 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 [361]. 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 [318] 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.

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

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

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

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

At this point, we have computationally usable templates that represent meaningful areas of interest, and we need to identify where these templates occur within video frames. To do this, we used a template matching algorithm provided by OpenCV. This algorithm essentially takes a given template, and slides it over the entire frame. For every overlap, the algorithm computes a normalized score between 0.0 and 1.0, where 0.0 represents the least likely match, and 1.0 represents a perfect match. Through trial-and-error, we found that a threshold of 0.95 yields accurate detection of AOIs. Because this is a computationally demanding operation, we down-sampled both the templates and the video frames to 50% of their original

size to reduce the number of pixels needing to be processed at each frame. We then up-scaled the identified pixel locations to map them to the original video locations. The output of this procedure is a data file containing the identified AOIs for each video frame and the bounding box of that AOI.

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

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

6.3 Analysis

6.3.1 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.

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

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 [407]. 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.

6.3.2 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 [149]. Directly questioning the participant at the end of each task can introduce social-desirability and prime the participants' behavior for subsequent tasks [402], 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 [155]. 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 [319]. 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 [319]) 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 [196].

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.

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

Although Seo and colleagues identified a distribution of costly error messages [342], 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 [179].

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 [272]. 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 [189]. 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.

Table 6.2 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.

6.4 Results

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

Table 6.2 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.

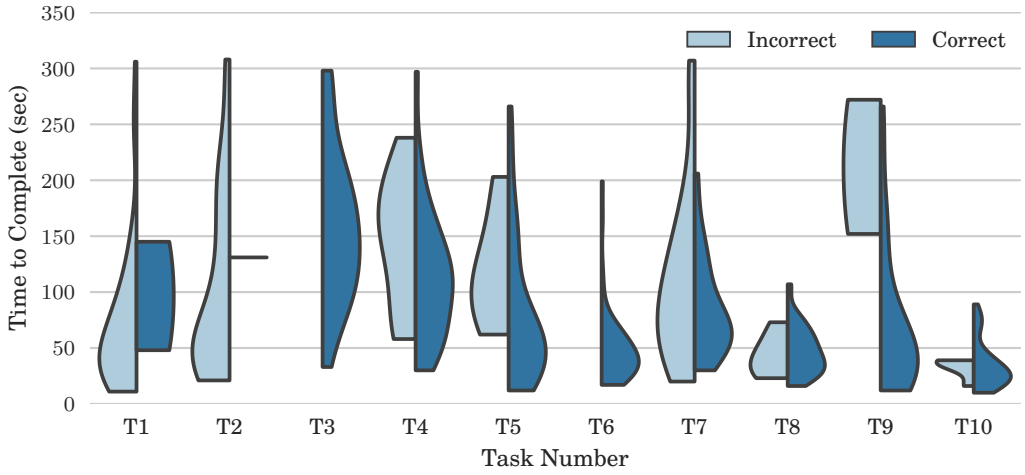


Figure 6.1 The time required for developer to commit to a solution that is correct or incorrect. Nearly all tasks (exceptions, T8 and T10) have high variance in resolution time to arrive, irrespective of correctness.

The n_{bins} columns in Table 6.2 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 6.1). Recall that the correct solution to this problem is to rename the `sublist` method to `subList`. Participants provided two incorrect solutions to this problem. They either removed the `@Override` annotation, which suppresses the error but does not resolve the defect, or they created a stub `sublist` method in the parent class, without recognizing the existing similarly-named method. Across all tasks, participants converged to a small set of incorrect solutions.

Figure 6.1 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 [342]. For some tasks, however, such as T8 and T10, most participants were able to correctly resolve the defect, and

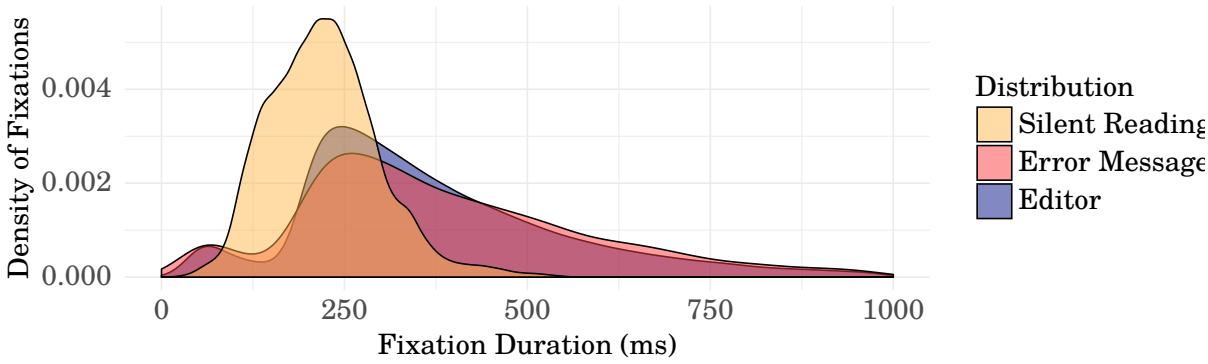


Figure 6.2 Comparison of fixation time distributions for silent reading of English passages, reading source in the editor, and reading of error messages.

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 [342], it is likely these errors are costly because they are arise frequently as a nuisance for developers, not because they are particularly difficult to resolve.

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

6.4.2 RQ2: Do developers read error messages?

Figure 6.2 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) [319]. Thus, reading source code is much more difficult than reading a general English passage, and marginally less difficult than the activity of typing while reading.

We compute the KL divergence between the source editor distribution and error message distribution ($D_{KL} = 0.059$), source editor distribution and silent reading distribution ($D_{KL} = 3.38$), and error message distribution and silent reading distribution ($D_{KL} = 2.37$). From the relatively small KL divergence between the source editor distribution and the error message distribution, we conclude that error message reading is comparable to source code reading ($u = 419$ms, $sd = 270$, $n_{\text{fix}} = 18573$), 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 6.3). 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.

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

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

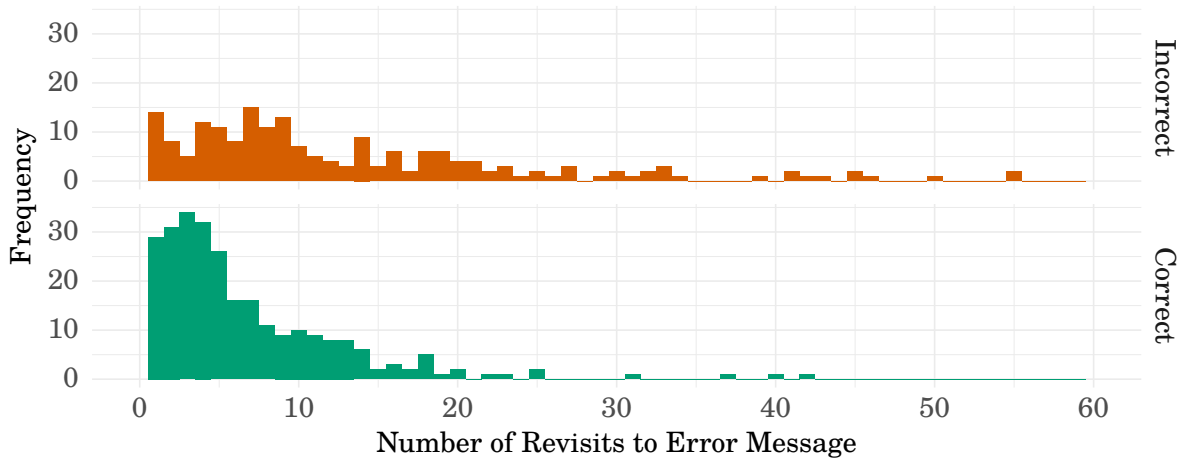
A Chi-squared test reveals a statistically significant difference between the number of revisits and the outcome of correctness over a distribution of tasks ($df = 1, G^2 = 60.9, p < .0001$). This suggests that the number of revisits, a proxy measure for reading difficulty [179], is a significant predictor 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.

Table 6.3 Participant Fixations to Areas of Interest

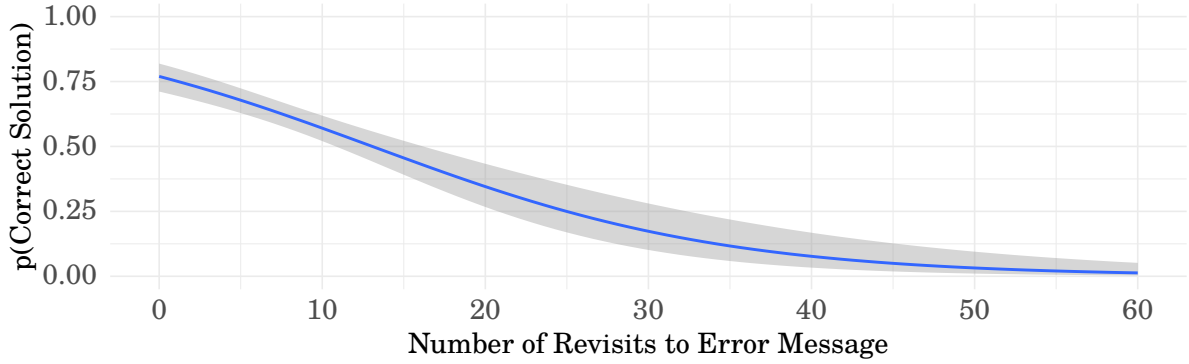
	Task									
Area of Interest	T1	T2	T3	T4	T5	T6	T7	T8	T9	T10
Source editor	66%	66%	74%	79%	68%	65%	78%	68%	80%	65%
Error areas	23%	23%	15%	14%	23%	25%	15%	17%	13%	20%
Navigation areas	10%	11%	11%	6%	9%	11%	7%	15%	7%	15%
Error Areas Breakdown[1]										
Error popup	1%	0.5%	—	0.4%	0.7%	2%	0.5%	1%	1%	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.

Figure 6.3 presents this statistical result more intuitively. From Figure 6.3a, 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 6.3b 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.



(a)



(b)

Figure 6.3 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.

6.5 Discussion

From the results of our research questions, we identify—using eye tracking—three problem areas in current development environments. These problems, if addressed, could improve rational reconstructions.

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 6.1, $n_{\text{bin}} = 1$).

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.

Toolsmiths 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.

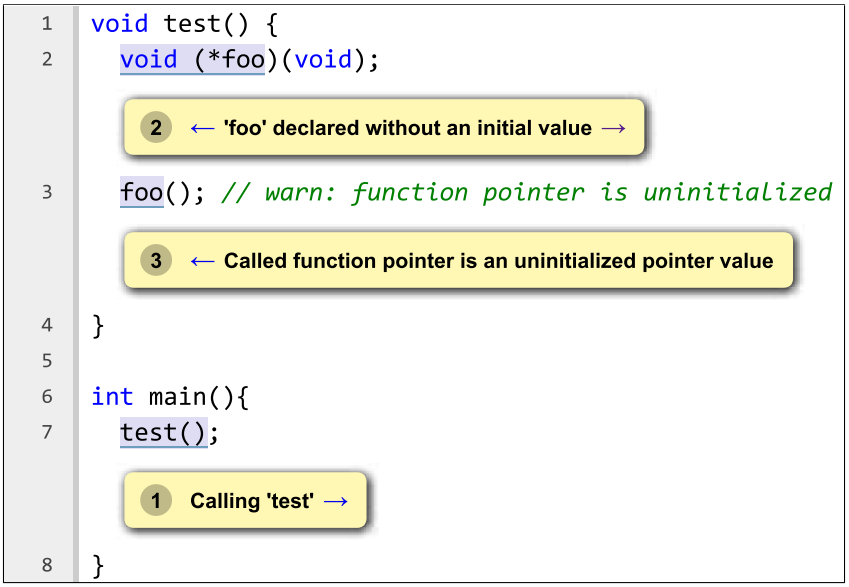


Figure 6.4 Emerging error reporting systems like LLVM scan-build provide stark contrast to those of conventional IDEs. Here, scan-build presents error messages for a potential memory leak as a sequence of steps alongside the source code to which the error applies.

To postulate why this might be, let's return to Table 6.1. Consider an error message as in T4, which reads:

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

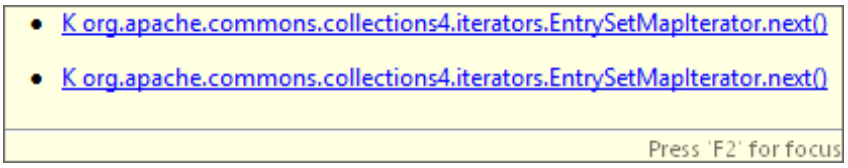
Even for relatively short error messages like this one, participants spent 14% of their time in the total task with fixations across essentially nine "words." Prior research in language cognition for bilingual sentence reading has found that switching languages is associated with a cognitive processing cost [263]. Similarly, one explanation for the apparent difficulty of reading error messages is that error messages consist of both natural language ("`is undefined for`") and code ("`Queue<E>`"), but are not entirely either. Consequently, developers must context-switch between two different modalities of reading to fully capture the information presented in an error message, leading to increased reading difficulty.

A second explanation for why error messages are comparable in difficulty to reading source code is because reading error messages requires the developer to also switch between the error message and the source code in order to understand the full context of the task. For example, a developer might read "`The method get()`"

and then suspend their reading of the error message to determine in the source editor the calling context of this method and figure out why and whether it should be called. In this case, error presentation approaches such as those found in LLVM scan-build [234] may prove beneficial to developers (Figure 6.4). 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 [18].

6.6 Limitations

Although we derived our errors based on frequency and difficulty of resolution from a prior Google study [342], 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 [310]. 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 [336]. 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 [72, 111].

6.7 Related Work in Eye Tracking

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, McMillan, McBurney, and colleagues [326] used eye tracking to understand how developers summarize code; the results of their experiment were then used to improve automatic code summarization. Romero, Cox, Boulay, and colleagues [328] used eye tracking to understand how developers found defects in source code under different representations of the source code, such as diagrams. Uwano, Nakamura, Monden, and colleagues [385] asked developers to perform code review tasks, during which participants had to locate defects in the code. The authors identified common scan patterns in subjects' eye movements. In a partial replication of Uwano's study, Sharif, Falcone, and Maletic [345] found differences between experts' and novices' scan patterns while locating defects. Bednarik and Tukiainen [28] found that repetitive patterns of visual attention were associated with lower performance. In our study, we also found that revisits to error message information is statistically significant with respect to the probability of correctness. Like other research attempting to find patterns in debugging activities, Hejmady and Narayanan [163] found visual pattern differences based on programming experience and familiarity with the IDE. Busjahn, Bednarik, Begel, and colleagues [52] were interested in how novices read source code; from eye tracking, they found that experts read code less linearly than novices. To justify investments in rational reconstruction, we are interested in when and how developers read error messages.

6.8 Conclusions

In this chapter, 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 not only 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 counts as a proxy for reading difficulty suggests that difficulties in solving a task can be attributed to difficulties in reading the error message. Thus, the evidence in this chapter supports the *first claim in my thesis*: difficulties interpreting error messages produced by program analysis tools are a significant predictor of developers' inabilities to resolve defects (Chapter 1).

If we further postulate that error messages as rational reconstructions are useful for developers, then could it be the case that developers have difficulties interpreting error messages because they are *insufficient* rational reconstructions? And would addressing these deficiencies make error messages more useful and comprehensible to developers?

In the next two chapters, we tackle these questions. In Chapter 7, we consider a visual form of rational reconstruction, called trace-explanation, and demonstrate how existing visual presentations in contemporary program analysis are insufficient rational reconstructions. In Chapter 8, we consider a text form of rational reconstruction, called justification-explanation, to demonstrate that contemporary text presentations are also insufficient rational reconstructions. Considering both trace-explanation and justification-explanation provide full coverage over the theoretical framework of rational reconstruction (Section 2.4).

7 | How Do Developers Visualize Compiler Error Messages?

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

Integrated development environments such as Eclipse, IntelliJ, and Visual Studio offer a number of visualizations to assist developers in more effectively identifying and comprehending program analysis error messages (Section 3.3).[1] For example, in addition to the text description of the error found in a console output or dedicated error window, these messages may include an indicator in one or more margins along with a red wavy underline overlaid on the source text; such visualizations diagrammatically indicate a relevant location of an error. Encouragingly, Ainsworth and Loizou [5] found that the use of diagrams promote the self-explanation effect significantly more than text alone.

Despite the visual affordances in contemporary IDEs, we postulate one of the reasons visual error messages remain confusing for developers is because they are insufficient rational reconstructions. Specifically, these visualizations do not support developers during *self-explanation*—essentially, the cognitive process by

[1] Significant portions of this chapter were previously published as T. Barik, K. Lubick, S. Christie, and colleagues, "How developers visualize compiler messages: A foundational approach to notification construction," in *IEEE Working Conference on Software Visualization (VISSOFT)*, 2014, pp. 87–96.

which humans self-generate explanations to themselves and to others in order to understand a situation [291]. Failures during the self-explanation process can result in a substantial loss of productivity because humans are imperfect and bounded in knowledge, attention, and expertise [205].

In this chapter, we argue that developers stand to significantly benefit when program analysis exposes its automated reasoning through trace-explanations (described in the theoretical framework in Section 2.4), a form of rational reconstruction in which error messages chain backwards to establish a line-of-reasoning to the causes of the error. We contend that program analysis tools can leverage these trace-explanation structures to generate expressive, *explanatory visualizations* that align with the way developers self-explain error messages. Our contributions in this chapter are:

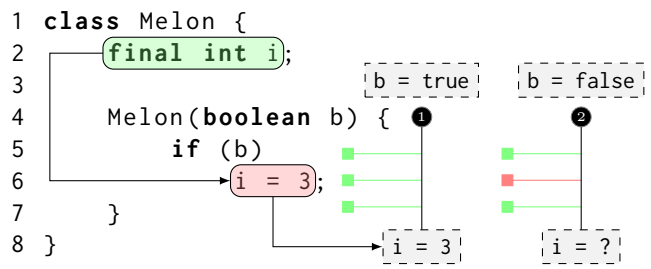
- A foundational set of composable visual annotations that aid developers in better comprehending error messages.
- An *explanation task* evaluation, using a set of paper mockups, which demonstrates that our explanatory visualizations yield more correct self-explanations than the baseline visualizations 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 messages.

7.1 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 7.1a, 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 7.1c).

```
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 7.1 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 text error message.

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 error messages.

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* [205]. 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 7.1b. 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`.

Table 7.1 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.

This hypothetical scenario illustrates why the dominant visualization paradigm is not sufficient in supporting the process of self-explanation. As we argue in this chapter, 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. As insufficient rational reconstructions, the output does not adequately support the developer during self-explanation.

7.2 Pilot Study

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

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

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

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

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 7.3. 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 7.1.

Table 7.2 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 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.

7.3 Explanatory Visualizations of Error Messages

We propose a set of eight visual annotations, which are summarized in Table 7.2. We now concretely describe these annotations using the motivational example from Figure 7.1b. 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 7.1a). In the example, this is `final int i`.

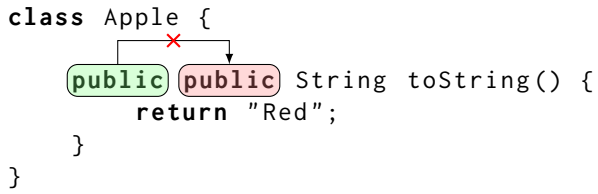
Continuing our example, the starting point is associated with a second point, `int i`, because this is where the potential assignment to the variable occurs. We indicate the starting point with `code` (red rectangle with rounded corners), and the association is indicated by (a directional arrow).

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

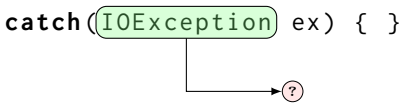
This composite annotation demonstrates that several basic annotations can be combined to create a new annotation for expressing a more complex concept. One of these components is the code coverage annotation. This annotation uses (green line) and (red line) to indicate whether or not a line is covered.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

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

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



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



7.4 Methodology

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

7.4.1 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 messages?

Unlike baseline visualizations, explanatory visualizations are rational reconstructions designed to expose the reasoning process of the compiler in a way that aligns with a developers' own self-explanation process.

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 messages. 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.

7.4.2 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%).

7.4.3 Selection Criteria for Mockups

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

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

We selected our compiler error examples from the OpenJDK diagnostics framework.[2] 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.

[2]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/`.

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 7.3 on the following page.

Table 7.3 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`

7.4.4 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. 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 7.2). 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[3] to render visualizations that we think are plausible for compilers to render if they expose the appropriate data structures to a visualization system.

7.4.5 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.

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

7.4.6 Experimental Procedure

7.4.6.1 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.

7.4.6.2 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 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.

7.4.6.3 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 messages, 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 7.3, 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 7.1a. The treatment group received paper mockups containing our explanatory visualization as in Figure 7.1b. Below the source code listing, all participants received the full error message text (Figure 7.1c). In the treatment group, we provided participants with a visual an-

notation legend (Table 7.2), 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 message. 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.

7.4.6.4 Cognitive Dimensions Survey

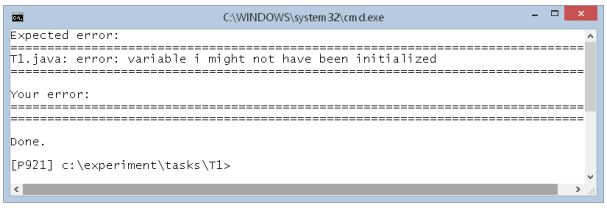
To evaluate aspects of visualizations that developers find useful in self-explanation (RQ3), participants completed a Cognitive Dimensions of Notations questionnaire (CD) [142], which we simplified for error messages. We chose this evaluation instrument over other usability instruments because the analysis is usable by non-specialists in HCI (in contrast with Nielson and Molich's heuristic evaluation [280]). 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:

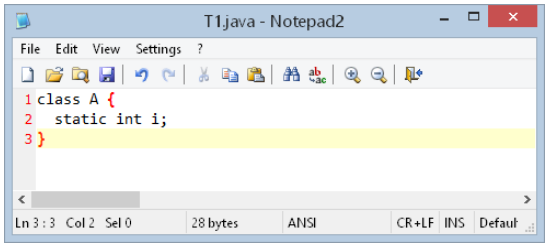
Consistency similar semantics are expressed in similar syntactic forms

Hidden dependencies important links between entities are not visible

Hard mental operations high demand on cognitive resources



(a) Command prompt.



(b) Minimal text editor.

Figure 7.2 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.

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.

7.4.6.5 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.

7.4.6.6 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 messages (RQ4). To evaluate this hypothesis, we asked participants to write source code listings on a computer from scratch in order to *generate* a provided compiler error.

Participants did so through the interface shown in Figure 7.2. We gave the participants a command prompt (Figure 7.2a) 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 7.2b). 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 7.2, 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 7.3, 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 [347]. Specifically, one component of his experiment involved non-programmers and programmers memorizing a proper `FORTRAN` program printed on paper through a line printer. He also printed a second program with shuffled lines. He found that non-programmers had similar performance in recall with both the proper and shuffled versions of the program, but that programmers had significantly better recall on the proper version of the program. Through the development of his cognitive syntantic/semantic model, he suggests that "performance on a recall task would be a good measure of program comprehension" because such a task cannot be accomplished by rote memorization, and instead requires "recognizing meaningful program structures enabling them to recode the syntax of the program into a higher level internal semantic structure" [347].

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

Figure 7.3 Explanation rating by group. The treatment group (T) provided significantly higher rated explanations than the control group (C).

7.5 Results

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

Our hypothesis was that having visual explanations for compiler error messages 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

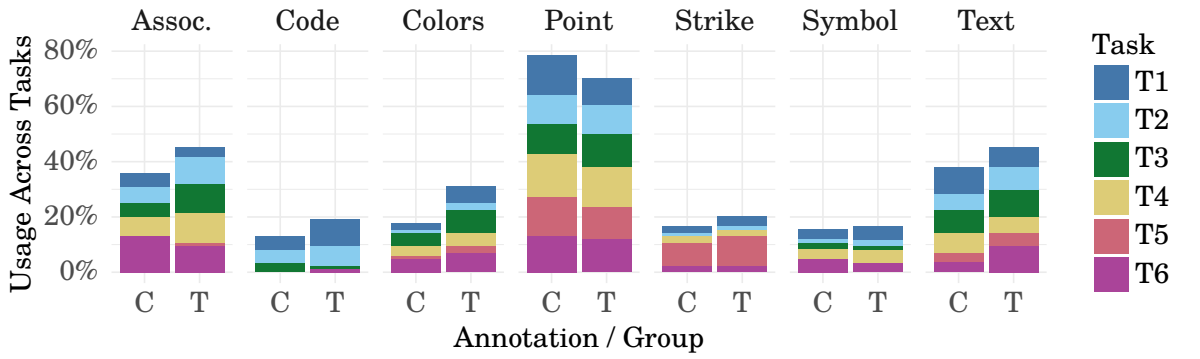


Figure 7.4 Annotations by group, filled with usage across tasks. The distribution of annotations used by the control (C) and treatment groups (T) were not identified as being significantly different, but the treatment group used annotations significantly more often.

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 7.3. 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 ($n = 168$, $df = 2$, $\chi^2 = 3.37$, $p = .19$).

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

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.

Table 7.4 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	

Consider for a moment the visualizations drawn by two participants in our study, shown in Figure 7.5. In Figure 7.5a, the control group participant receives a score of Fail, because he incorrectly self-explains that the problem must be due to not initializing the variable at its point of declaration. He then either ignores the conditional statement in which the constant value is re-assigned, or fails to notice that the variable is declared as `final`. In Figure 7.5b, 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 7.4 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 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 7.4 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 anno-

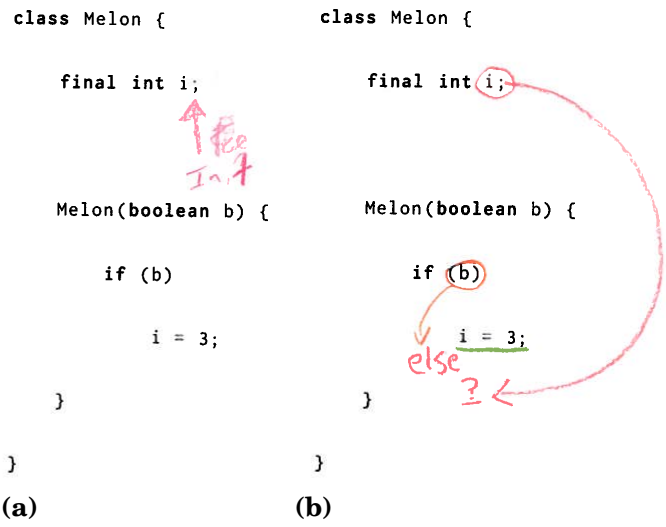


Figure 7.5 A contrast between visual explanations offered by (a) control group participant with explanation rating of Fail, and (b) treatment group participant with explanation rating of Excellent.

tations are intuitive even without priming the participants. Although Figure 7.4 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.

Table 7.5 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

7.5.3 RQ3: What aspects differentiate explanatory visualizations from baseline visualizations?

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 7.5 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.

7.5.4 RQ4: Do better self-explanations enable developers to construct better mental models of error messages?

Figure 7.6 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 7.5.1 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.

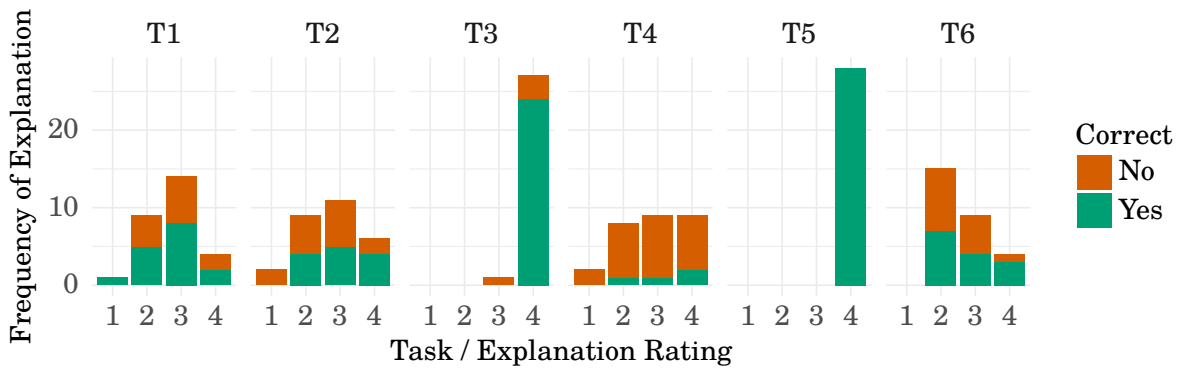


Figure 7.6 Task by explanation rating. Each of the six tasks are broken by explanation rating (1 = Fail, 2 = Poor, 3 = Good, 4 = Excellent) from the first phase of the experiment. For each explanation rating, the frequency of correct and incorrect recall tasks from the second phase of the experiment is indicated by filling in the bars. Higher rated explanations lead to significantly better recall correctness.

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 7.6, 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.

7.6 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 error messages. 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.

7.7 Conclusions

The experiments in this chapter demonstrate how explanatory visualizations—that is, trace-explanation forms of rational reconstruction—facilitate developer self-explanation. Through the errors messages in this study, we found that when such visualizations align with developer expectations, developers better comprehend error messages, use these visualizations more often in their own self-explanations, and construct better mental models of error messages. Crucially, we found that baseline visualizations fail to explicate causal relations that are necessary for trace-explanations. In contrast, our enhanced explanatory visualizations supported developers by revealing these relations through box-and-arrow representations overlaid on the source code—as well as through the occasional use of explanatory text and concrete examples. Thus, the evidence in this chapter supports the *second claim in my thesis*: difficulties interpreting error messages can be explained by framing error messages as insufficient rational reconstructions in visual presentations (Chapter 1).

In the next chapter, we consider rational reconstructions for text error messages. We investigate these text error messages through Toulmin's model of argument—an informal justification-explanation model used in everyday discourse.

8 | 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

In this chapter, we apply a justification-explanation form of rational reconstruction, Toulmin's model of argument (Section 8.2), to the design and evaluation of compiler error messages.[1] We define error messages that properly implement Toulmin's model in *structure* and *content* as *explanatory* error messages, and we consider error messages that do not properly implement this model to be insufficient rational reconstructions.

To understand if developers find explanatory error messages helpful, we conducted a comparative evaluation between two compilers for the same programming language, and had experienced developers within Microsoft 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.[2] From Stack Overflow, we extracted 210 question-answer pairs posted by developers about compiler

[1] Significant portions of this chapter were previously published as T. Barik, D. Ford, E. Murphy-Hill, and colleagues, "How should compilers explain problems to developers?" In *Foundations of Software Engineering (ESEC/FSE)*, 2018.

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

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 Toulmin's model of argument. We characterized these question-answer pairs both in terms of the *structure* and *content* of their explanation.

The results of our studies provide support for presenting compiler error messages to developers as explanations, and inform their design. 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 (Section 8.5.1).
- Human-authored explanations converge to argument structures that offer a simple resolution, or to structures with proper arguments (Section 8.5.2). They do so using a catalog of content within the structure (Section 8.5.3).

8.1 Motivating Example

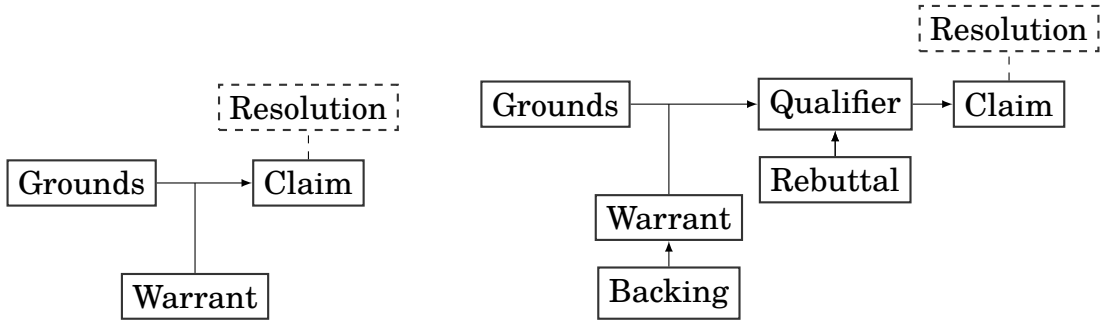
It isn't difficult to come up with instances of poor error message explanations, even for routine problems. Consider the following Java code snippet:

```
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 in a loop; this particular variable also 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?



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

Figure 8.1 A prototypical Toulmin's model of argument for (a) simple argument layout, and (b) extended argument layout. The possible need for auxiliary steps to convince the other party yields the extended argument layout.

```
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 8.5.1). Specifically, the second message not only indicates that there is a problem (`the blank variable "x" cannot be assigned`"), but 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.

8.2 Background on Explanations

Arguments are a form of justification-explanation, in which reasons are used as evidence to support a conclusion [395]. Argumentation theory provides a lens through which we can evaluate the effectiveness of arguments [109, 300]. Within argumentation theory, Toulmin's model of argument is one such informal reasoning

model. The model characterizes everyday arguments, or how arguments occur in practice through ordinary human dialogue [379]. Specifically, Toulmin's model of argument is a *macrostructure* model. Macrostructure examines how components combine to support the larger argument; 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 8.1a on the previous page), the first component is a *claim*—the assertion, view or judgment to be justified; resolutions are also a form of *claim*, though resolutions are optional in an argument layout. The second component, *grounds*, are data that provide evidence for this claim. The third component is a 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 these 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 8.1b on the preceding page). 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, a warrant may 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 of how a compiler error message is mapped to an argument structure is illustrated in Figure 8.2 on the next page; in this example, the error message is an extended argument because it has a backing.

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

8.3 Methodology

8.3.1 Research Questions

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

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 presentations based on other factors, such as the verbosity of the error message.

RQ2: How is the structure of explanations in Stack Overflow different from compiler error messages? If compiler error messages and Stack Overflow accepted answers use significantly different argument layout components, this would suggest that *structure* differences in argument play an important role in the confusion developers face with compiler errors. While some approaches to improving compiler error messages focus on the content (for example, "confusing wording" in the messages [190, 281]), 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 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.

8.3.2 Phase I: Study Design for Comparative Evaluation

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

Compiler selection rationale. We needed to compare two compilers which produced different error messages for the same problem in the code, preferably where one compiler produced error messages with better explanatory structure than the other. We selected the Jikes and OpenJDK compilers for this purpose. Jikes is a Java compiler created by IBM for professional use, with a primary design goal of high-quality explanations produced by the compiler [60]. 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 those which contained argument structure. To determine if an error message contained any elements of argument structure, we tagged each message using labels from Toulmin's model of argument: claim (and

resolution), grounds, warrant, qualifier, rebuttal, and backing. From this analysis, we found 30 error messages which used at least 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 8.1 on page 144) 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 before the study.

E1 *Deficient argument vs. simple argument.* Both OpenJDK and Jikes make a *claim* that the variable might be assigned 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 is 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 should be determined by the content of the claim, and 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 the 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 Microsoft to participate in this study. As we were primarily interested in professional software development, we selected our population from full-time software developers—excluding interns or roles such as testers or project managers. We invited 300 developers to participate in our study 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, as well as the source code listing that produced the two error messages. We randomized error message order. On average, participants took seven minutes to complete our study.

8.3.3 Phase II: Study Design for Stack Overflow

Research context. Previous research on Stack Overflow by Treude, Barzilay, and Storey [382] 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 [241]. Nasehi, Sillito, Maurer, and colleagues [274] inspected Stack Overflow questions-answer pairs to identify attributes of good code examples; in this study, we adopt a similar methodology. Stack Overflow provides an open-access API, Stack Exchange Data Explorer,[3] 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.

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

Table 8.1 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".`

Table 8.2 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	8275	1129	9404	61%

[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` or `compiler-warnings` that have accepted answers.

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 links to an associated *accepted answer*, which in this chapter 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 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 question. If the question did not contain a compiler error message, the question-answer pair was dropped from analysis.[4]

Sampling strategy. To obtain diversity across programming languages, we used 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 8.2). 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 were unambiguously "trolling," [153] such as through deliberately bogus questions.[5] The time required to manually categorize question-answer pairs has high variance, from 5-15 minutes, depending on the complexity of the pair. Thus, to balance breadth of languages and depth of error messages in each language—while still keeping 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 each compiler error message extracted from the Stack Overflow question and over the complete Stack Overflow accepted answer for that question. We tagged these using labels from Toulmin's 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.

[4]Quotations to Stack Overflow questions and answers are indicated as Q:*id* or A:*id*, and can be directly accessed through `https://www.stackoverflow.com/questions/:id`.

[5]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/`)

During the coding process, we employed the technique of *negotiated agreement* as a means to address the reliability of coding [54]. 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.

Validity of negotiated agreement. Though negotiated agreement is established in other disciplines, this qualitative coding technique has only recently been applied to software engineering research (notably, by Hilton, Nelson, Dig, and colleagues [166]). Thus, to assess the validity of negotiated agreement, we recruited two independent evaluators (IEV1/IEV2) to classify 20 random question-answer pairs (that is, 40 messages, or approximately 10% of the pairs). Evaluators were provided with the definitions of argument layout components (Table 8.4) as well as a diagram of proper and deficient argument layouts (Figure 8.3).

Evaluators classified messages in the question-answer pairs as either claim-only, claim-resolution, simple argument, or extended argument. We then calculated Cohen's Kappa between the evaluators and against our own negotiated agreement classifications: κ between IEV1 and IEV2 ($\kappa = 0.96$), IEV1 and negotiated agreement ($\kappa = 0.78$), and IEV2 and negotiated agreement ($\kappa = 0.71$). Cohen's κ supports the validity of negotiated agreement ($\kappa > 0.70$ is considered "substantial agreement" by Landis and Koch [212] , "good" by Bland and Altman [39], and "fair–good" by Fleiss, Levin, and Paik [121]). Follow-up revealed that evaluators were hesitant to apply the tag "backing" due to inexperience with Toulmin's model of argument. Therefore, cases of disagreement between independent evaluators can be explained by evaluators shifting their assessment of extended arguments to simple arguments.

8.4 Analysis

8.4.1 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 8.1), using the developer responses as the observed values for OpenJDK and Jikes. If we use a null hypothesis where both messages are equally acceptable, then the expected values would be split such that OpenJDK and Jikes receive roughly 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.

8.4.2 RQ2: How is the structure of explanations in Stack Overflow different from compiler error messages?

As the first step towards answering this research, we wanted to quantify whether the argument structure between compiler 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 [358] 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 corresponding accepted, human-authored answer. Essentially, the goal of this analysis is to identify if these two groups are statistically different.

However, since error messages 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 argument components, E:

$$E = \langle a_1, a_2, \ldots, a_n, r \rangle \tag{8.1}$$

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 the sets [358]. Next, we perform a permutation testing calculation, fully described in Simpson, Lyday, Hayasaka, and colleagues [358]. The explanation of this algorithm is fairly intricate, but the essential result of this computation is a Jaccard ratio and an empirically-obtained p-value.

To characterize what types of argument structures are found in accepted answers, we used quasi-statistics—essentially, a process of transforming qualitative data to simple counts—to aid in the interpretation of the Stack Overflow data [247]. 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 ignored 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 8.2).

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

To identify the *content* of arguments, that is, the techniques developers use within the argument 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 [334]. As a concrete example,

Table 8.3 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	71%
E5	.732	36	53%	32	47%

[1] * Indicates a statistically significant result.

the argument component of *backing* can be provided by pointing to a location or program element in the code (blame), through a code example that provides evidence for the problem, or through external resources, such as programming language specifications.

In addition to random sampling, we performed *purposive sampling*, or non-probabilistic sampling, on question-answer pairs to compose *memos* [37]. We wrote these memos to capture interesting exchanges or properties of the question-answer pairs, to promote depth and credibility, and to frame the posters' information needs and responses through their reported experiences. That is, the memos provide a *thick description* to contextualize the findings [305].

8.5 Results

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

From Table 8.3, 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 developers to prefer Jikes because the argument is presented 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.

8.5.2 RQ2: How is the structure of explanations in Stack Overflow different from compiler error messages?

The Jaccard ratio of the two groups is $R_j = 1.6441$, 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 messages from the questions and the Stack Overflow accepted answers are significantly different in terms of argument layout.

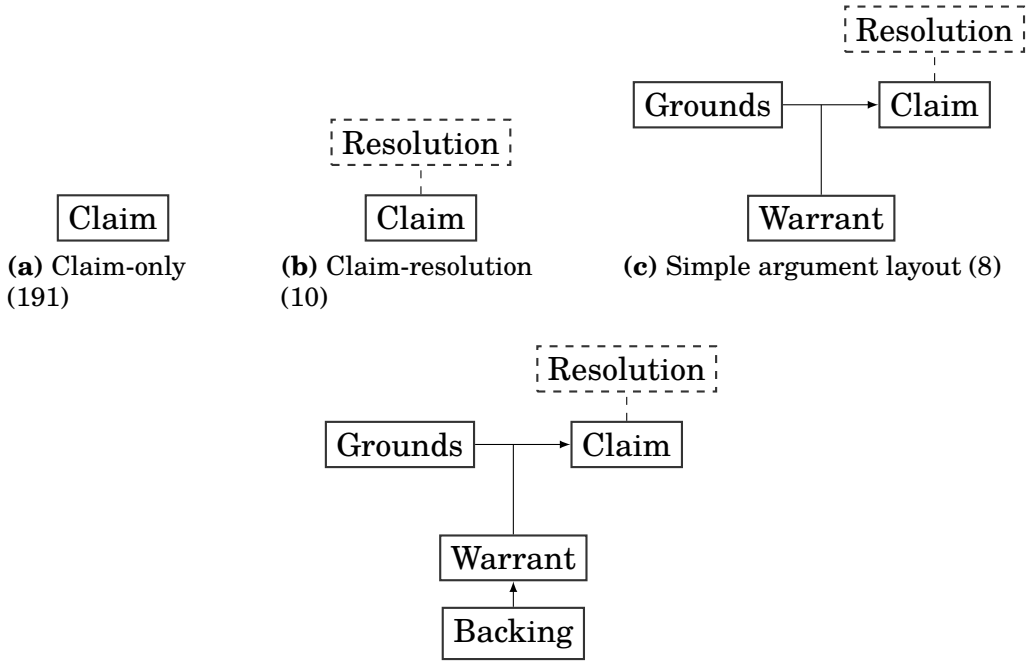
Because the questioner asked a question about the compiler error message, this indicates some confusion with the error messages they were presented with. Because the same questioner also marked the Stack Overflow answer as accepted, we can assume that the answer has resolved whatever confusion they had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

The resulting argument structures are found in Figures 8.3 and 8.4 on the following page, for compiler error messages and for Stack Overflow accepted answers, respectively. For each group, the argument layouts are ordered from most frequently observed to least frequently observed. In this quasi-statistical reporting, it becomes clear why the argument layout for compiler errors and Stack Overflow accepted answers were found to be significantly different: 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 somewhat inverted in argument layout frequency: the most frequent argument layout extends simple argument layout with backing, followed by claim-resolution and simple argument in roughly balanced frequencies. 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's model of argument, these answers satisfactorily resolved the confusion of the developer when the original compiler error message did not.

8.5.3 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 8.4 on page 154.



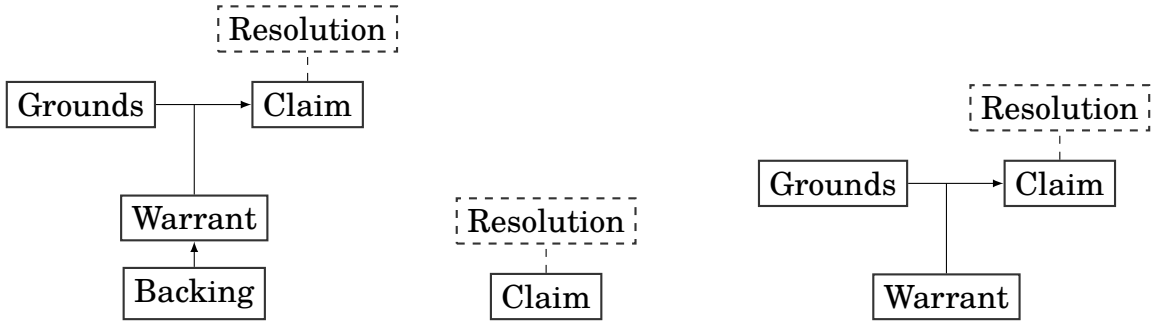
(a) Claim-only (191)

(b) Claim-resolution (10)

(c) Simple argument layout (8)

(d) With backing (1)

Figure 8.3 Identified argument layouts for compiler error messages (as found in Stack Overflow questions). Counts are indicated in parentheses.



(a) With backing (102)

(b) Claim-resolution (59)

(c) Simple argument layout (49)

Figure 8.4 Identified argument layouts for Stack Overflow accepted answers. Counts are indicated in parentheses.

Table 8.4 Argument Layout Components for Error Messages

Attribute	Description
Simple Argument Components	
Claim (Section 8.5.3.1)	The claim is the concluding assertion or judgment about a problem in the code.
Resolution (Section 8.5.3.2)	Resolutions suggest concrete actions to the source code to remediate the problem.
Grounds (Section 8.5.3.3)	Facts, rules, and evidence to support the claim.
Warrant (Section 8.5.3.4)	Bridging statements that connect the grounds to the claim. Provides justification for using the grounds to support the claim.
Extended Argument Components	
Backing (Section 8.5.3.5)	Additional evidence to support the warrant, if the warrant is not accepted.
Qualifier (Section 8.5.3.6)	This is the degree of belief for a claim, used to weaken the claim.
Rebuttal (Section 8.5.3.7)	Exceptions to the claim or other components of the argument.

8.5.3.1 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 connect their ground and warrant (A28880386). We did, however, encounter instances where developers explained error messages by first *rephrasing* it, 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. For example, the compiler error message:

```
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.”

8.5.3.2 Resolution

One way in which error messages can convince developers of an argument is to offer a solution to the argument that resolves their issue; this type of resolution is a second form of claim. Typically, Stack Overflow accepted answers provide these resolutions 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,
  Vector3f& vec_b) {
```

8.5.3.3 Grounds

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 a 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), and “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, in which 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 a buffer overflow."

8.5.3.4 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 implicitly identified [109]. During our analysis, we would insert implicit "since" or "because" phrases during reading of the error message on Stack Overflow answers to identify implicit warrants.

In some compilers, messages can bridge grounds with warrants through explicit concatenations, 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 need 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.

8.5.3.5 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 [109]. For presenting error messages, we found that while grounds were typically additional statements, backing was provided through the use of resources. These resources include code examples or code snippets (A2640738, A1811168), references to programming language specifications (A5005384), and occasionally, bug reports (A37830382) or tutorials (A2640738).

8.5.3.6 Qualifiers

Despite the usefulness of static analysis techniques for reporting compiler error messages to developers, many classes of analysis feedback are undecidable or computationally hard, which necessitate the use of unsound simplifications [211]. 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 were simply used as casual linguistic constructs (essentially, fillers) or if the answers 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 (Q5013194, Q36476599).

8.5.3.7 Rebuttal

We found few instances of rebuttals within Stack Overflow accepted answers, and one of the reasons we believe rebuttals to be relatively infrequent is that any disagreements between participants are primarily relegated to meta-discussions *attached* to the accepted answer in order to reach consensus, rather than being incorporated within the answer itself. Thus, we interpreted rebuttals liberally as statements in which an answer would retract some ground or resolution due to some 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."

8.6 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 developers prefer 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 difficulty or type of error message. Because the authors selected the explanations to present, we may have 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 eliminate it entirely.

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 [20]. Similarly, Flanagan, Flatt, Krishnamurthi, and colleagues [120]

use a diagrammatic representation on the source code to help developers understand code flow for an error. Other tools like Path Projection [200] and Theseus [227] 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 a particular aspect of developer experiences as developers 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* [303]. In analytic generalization, we generalize from individual statements within question-answer pairs to broader concepts or higher-order abstractions through the application of argumentation theory.

8.7 Conclusions

In this chapter, we conducted studies on error messages using Toulmin's model of argument, a justification-explanation form of rational reconstruction. We investigated error messages and classified them as either proper or deficient argument structures. The results of our comparative evaluation found 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. Intuitively, this makes sense: having a quick fix that would correctly solve the problem can essentially short-circuit the need for rational reconstruction. But when there is no

obvious fix available, developers rely on proper arguments to facilitate understanding of the problem. Similarly, the results of our Stack Overflow investigation found that human-authored explanations converge to argument structures that offer a simple resolution, or to structures with proper arguments. Notably, although many compiler error messages consist of insufficient, claim-only arguments—we don't find deficient arguments in the corresponding human-authored, accepted answers. Thus, the evidence in this chapter supports the *third and final claim in my thesis*: difficulties interpreting error messages can be explained by framing error messages as insufficient rational reconstructions in text presentations (Chapter 1).

In the next chapter, we revisit the key contributions of the dissertation, framed in terms of the thesis. We explicate design guidelines resulting from our research studies and operationalize them through a prototype compiler implemented in TypeScript. Opportunities for future work are discussed.

9 | Conclusions

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

9.1 Thesis Revisited

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 interpreting error messages can be explained by framing error messages as insufficient rational reconstructions in both visual and text presentations.

I defended the claims of the thesis statement through three studies. In the first study (Chapter 6), 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 7), 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 identified 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 argumentation theory, a form of rational reconstruction. The third study (Chapter 8) found 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 indicated significant preference for structures that either provided a simple resolution, or employed 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 9.2). 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 9.3). 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 9.4). This discussion provides a landscape for what I believe are impactful directions within the discipline of error message comprehension.

9.2 Design Guidelines

To generalize the studies in this dissertation to other programming languages and environments—and to make our findings accessible and actionable to practitioners [164, 282]—I synthesize five design guidelines derived from our studies.

My design guidelines extend the chronology of design guidelines from Section 3.5, and are closest conceptually to Horning [172] and Dean [89]. In particular, Horning [172] frames interactions with program analysis as a dialogue or conversation, and Dean [89] applies an assortment of ideas from human psychology to error messages. Like Horning [172], my design guidelines are also inspired by human dialogue, but formalized through theory rather than observation. In contrast to the sampling of

psychology theories by Dean [89], my design guidelines offer a unified lens for the construction and design of error messages. Other existing guidelines are similarly atheoretical [50, 267, 269, 270, 346, 349, 381].

Guideline I—Implement rational reconstructions for humans, not tools. This is in some ways less of a guideline and more of a philosophy about how to think about the design of error messages (Section 2.4). The studies in this dissertation demonstrate that developers reason about error messages differently than the underlying mechanism by which program analysis algorithms typically identify a problem in the source. As such, simply exposing internal program analysis processes to developers isn't generally useful (Chapters 6 to 8). Consequently, this guideline suggests that human-centered expressions of error messages should be rational reconstructions as a routine part of error message design.

Guideline II—Use code as the medium through which to situate error messages. The study in Chapter 6 applied revisits to error messages as a signal of comprehension difficulty. But revisits can also be applied as a usability heuristic to identify problems in the arrangement of information [119, 311]. As an example, we found that when error messages are presented as tooltips, they often occlude the corresponding source code that developers are trying to investigate.

This guideline suggests that toolsmiths should provide context for error messages framed through the primary artifact within which developers work—source code. In text environments, consider using affordances to display error message information in proximity to or even interleaved with the source code (Chapter 6). In visual environments, consider diagrammatic elements and other graphical affordances overlaid on the source code to explicate relationships. However, be wary of introducing unfamiliar notation: stick with basic diagrammatic techniques, such as lines, boxes, and arrows (Chapter 7).

Guideline III—Present rational reconstructions as coherent narratives of causes to symptoms. As with good explanations, error messages are easier to comprehend if there is a coherent narrative of causes to symptoms that logically explains why or how a problem has occurred. In terms of rational reconstruction,

these coherent narratives manifest through either trace-explanations (Chapter 7) or justification-explanations (Chapter 8). The sequence of explanation need not be elaborate: in text, a straightforward ground that connects to the claim—or visually, an arrow that reveals why certain program elements are connected—can be enough to help developers better comprehend an error message.

Guideline IV—Distinguish fixes from explanations. Fixes are immensely useful to developers, and so it isn't surprising that many of the existing guidelines recommend them when presenting error messages (Section 3.5). Nevertheless, fixes aren't actually rational reconstructions (Chapter 8), nor are they always appropriate. In Chapter 6 (see also studies by Barik, Song, Johnson, and colleagues [18]), we found that difficult error messages also tend to have context-sensitive resolutions. In such cases, generic fixes may remove the error message, but not actually resolve the root cause of the problem. Even when automatic fixes are ultimately satisfactory, the developer may still want to know why the suggested fix is appropriate to apply.

Guideline V—Give developers autonomy over error message presentation. Depending on their familiarity with the code and their expertise, developers may need more or less help in comprehending the problem. For example, Dean [89] notes that "different people want different amounts of information, and the same people want different amounts at different times." Feedback from our own study in Section 9.3 also suggests that developers will desire different amounts of feedback, even for the same underlying problem. Thus, consider supporting mechanisms to progressively elaborate error messages, and provide configuration options to allow developers to tailor the verbosity and type of information presented in the error message.

9.3 Toward Engineering a Compiler

9.3.1 Approach

To apply our theory into practice, I operationalized our design guidelines from the previous section into a production compiler. Specifically, I modified the Microsoft TypeScript compiler [258] to generate rational reconstructions for `TS2393`, a duplicate function implemention error. I assessed this prototype, called Rational TypeScript, 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 industry 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 easy to introspect and to extend the compiler. The TypeScript source code builds relatively quickly, on the order of seconds, and this rapid turnaround facilitates rapid iteration when developing the prototype. 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 implementation as the vehicle to investigate rational reconstruction 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 element. I wanted an error message that would require introspecting the compiler during the creation of the rational reconstruction, yet at the same time would not require extraordinary effort to surface the underlying compiler structures. To support a potential study, I also wanted an error message that used concepts likely to be familiar to even occasional TypeScript developers. The duplicate function implementation error satisfies all of these considerations. Moreover, variations of the error message were used in previous studies: T8 in Chapter 6, and `Brick` in Chapter 7.

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 what information the compiler should collect before we knew when (or if) we would even encounter an error. The second approach is retrospective: we let the compiler run its 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 introspection is then used to construct a rational reconstruction. The implementation details for how this process works is found in Appendix E.

9.3.2 Example: Duplicate Function Implementation Error

In this section, let's see how TypeScript and Rational TypeScript present a duplicate function implementation error (`TS2393`) to a developer. The following TypeScript file induces this error 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.
```

[1] If a 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 runtime type checks to guide the behavior of the function.

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 the error message back to their source 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 over the baseline TypeScript message. The key difference from the baseline TypeScript message is that Rational TypeScript collects the duplicate function implementations and presents them diagrammatically (using ASCII characters such as '`-->`') 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 message provides warrants for why the problem is a problem. The error message provides a pointer to supplemental documentation from the TypeScript handbook, for the use case in which the developer is intending to perform function overloading. Additionally, the error message uses colorized output to visually partition the different components of the error messages, such as the source code and explanatory text. For other presentation choices, such as placement of the error code, we adopted existing conventions from Rust UX guidelines for error messages [330].

Table 9.1 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.

9.3.3 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 Microsoft, across various levels of seniority (Table 9.1). 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 the build system, all participants used Visual Studio Code or Visual Studio to write the software, but compiled code using a command-line 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 for their feedback about the error messages, 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. *Limitations.* The focus group was conducted with professional developers; it's possible that a heuristic evaluation conducted by HCI experts would produce additional usability feedback beyond the feedback from the focus group.

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 reconstruction in the design and implementation of practical program analysis tools.

9.4 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 reconstruction in program analysis tools:

- **Benefits of wrong error messages.** This dissertation investigated the presentation 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. Our formative evaluation with Rational TypeScript, using principal and senior software developers as participants, revealed 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 [342] to inform our research designs, and Google has proprietary build processes that are not necessarily representative of other organizations. From what other sources might we 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 to 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 of 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 toolsmiths on the quality of their own messages? Do toolsmiths have realistic assumptions about the developers who use their tools? Understanding the process through which toolsmiths 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 toolsmiths currently do—the authorial burden of constructing a good error message is high. Would it be possible to build intelligent program analysis tools that 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.

9.5 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] D. Abrahams and A. Gurtovoy, *C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond*. Addison-Wesley, 2004 (see p. 78).

[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. 72).

[3] A.-R. Adl-Tabatabai and T. Gross, "Source-level debugging of scalar optimized code," in *Programming Language Design and Implementation (PLDI)*, 1996, pp. 33–43 (see p. 60).

[4] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*, 2nd ed. Addison-Wesley, 2007 (see p. 32).

[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. 110).

[6] A. Altadmri and N. C. Brown, "37 million compilations: Investigating novice programming mistakes in large-scale student data," in *ACM Technical Symposium on Computing Science Education (SIGCSE)*, 2015, pp. 522–527 (see p. 49).

[7] A. Altadmri, M. Kölling, and N. C. C. Brown, "The cost of syntax and how to avoid it: Text versus frame-based editing," in *International Conference on Computers, Software and Applications (COMPSAC)*, 2016, pp. 748–753 (see p. 76).

[8] B. de Alwis and G. Murphy, "Using visual momentum to explain disorientation in the Eclipse IDE," in *Visual Languages and Human-Centric Computing (VL/HCC)*, 2006, pp. 51–54 (see p. 106).

[9] G. Ammons, D. Mandelin, R. Bodík, and J. R. Larus, "Debugging temporal specifications with concept analysis," in *Programming Language Design and Implementation (PLDI)*, 2003, pp. 182–195 (see p. 61).

[10] C. Angeli, "Diagnostic expert systems: From expert's knowledge to real-time systems," in *Advanced Knowledge Based Systems: Model, Applications & Research*, 2010, ch. 4, pp. 50–73 (see p. 75).

[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 *Programming Language Design and Implementation (PLDI)*, 2009, pp. 38–49 (see p. 64).

[12] Apple. (2018). Xcode, [Online]. Available: `http://developer.apple.com` (see p. 47).

[13] H. Arksey and L. O'Malley, "Scoping studies: Towards a methodological framework," *International Journal of Social Research Methodology*, vol. 8, no. 1, pp. 19–32, 2005 (see p. 56).

[14] A. Ayers, R. Schooler, C. Metcalf, A. Agarwal, J. Rhee, and E. Witchel, "TraceBack: First fault diagnosis by reconstruction of distributed control flow," in *Programming Language Design and Implementation (PLDI)*, 2005, pp. 201–212 (see p. 66).

[15] 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 pp. 2, 27).

[16] 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 *Principles of Programming Languages (POPL)*, vol. 38, 2003, pp. 97–105 (see p. 45).

[17] T. Ball and S. K. Rajamani, "The SLAM project: Debugging system software via static analysis," in *Principles of Programming Languages (POPL)*, 2002, pp. 1–3 (see p. 43).

[18] 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 *International Conference on Software Maintenance and Evolution (ICSME)*, 2016, pp. 211–221 (see pp. 14, 106, 164).

[19] T. Barik, D. Ford, E. Murphy-Hill, and C. Parnin, "How should compilers explain problems to developers?" In *Foundations of Software Engineering (ESEC/FSE)*, 2018 (see p. 136).

[20] T. Barik, K. Lubick, S. Christie, and E. Murphy-Hill, "How developers visualize compiler messages: A foundational approach to notification construction," in *IEEE Working Conference on Software Visualization (VISSOFT)*, 2014, pp. 87–96 (see pp. 110, 158).

[21] 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 *Evaluation and Usability of Programming Languages and Tools (PLATEAU)*, 2017 (see p. 55).

[22] T. Barik, J. Smith, K. Lubick, E. Holmes, J. Feng, E. Murphy-Hill, and C. Parnin, "Do developers read compiler error messages?" In *International Conference on Software Engineering (ICSE)*, 2017, pp. 575–585 (see p. 82).

[23] 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. 72).

[24] M. Barr, S. Holden, D. Phillips, and T. Greening, "An exploration of novice programming errors in an object-oriented environment," in *Innovation and Technology in Computer Science Education (ITiCSE)*, 1999, pp. 42–46 (see p. 77).

[25] J. A. Bateman and C. Paris, "Phrasing a text in terms the user can understand," in *International Joint Conferences on Artificial Intelligence (IJCAI)*, 1989, pp. 1511–1517 (see p. 74).

[26] M. Beaven and R. Stansifer, "Explaining type errors in polymorphic languages," *ACM Letters on Programming Languages and Systems (LOPLAS)*, vol. 2, no. 1-4, pp. 17–30, 1993 (see p. 42).

[27] B. A. Becker, G. Glanville, R. Iwashima, C. McDonnell, K. Goslin, and C. Mooney, "Effective compiler error message enhancement for novice programming students," *Computer Science Education*, vol. 26, no. 2-3, pp. 148–175, 2016 (see p. 80).

[28] R. Bednarik and M. Tukiainen, "Temporal eye-tracking data: Evolution of debugging strategies with multiple representations," in *Eye Tracking Research & Applications (ETRA)*, 2008, pp. 99–102 (see p. 108).

[29] I. Beer, S. Ben-David, H. Chockler, A. Orni, and R. Trefler, "Explaining counterexamples using causality," in *Computer Aided Verification (CAV)*, 2009, pp. 94–108 (see p. 44).

[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. 71).

[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. 16).

[32] T. Berger, M. Völter, H. P. Jensen, T. Dangprasert, and J. Siegmund, “Efficiency of projectional editing: A controlled experiment,” in *Foundations of Software Engineering (FSE)*, 2016, pp. 763–774 (see p. 76).

[33] T. Berners-Lee and N. Mendelsohn. (2006). The rule of least power, [Online]. Available: `https://www.w3.org/2001/tag/doc/leastPower.html` (see p. 77).

[34] A. Bessey, D. Engler, K. Block, B. Chelf, A. Chou, B. Fulton, S. Hallem, C. Henri-Gros, A. Kamsky, and S. McPeak, “A few billion lines of code later: Using static analysis to find bugs in the real world,” *Communications of the ACM*, vol. 53, no. 2, pp. 66–75, 2010 (see p. 2).

[35] G. Bierman, M. Abadi, and M. Torgersen, “Understanding TypeScript,” in *European Conference on Object-Oriented Programming (ECOOP)*, 2014, pp. 257–281 (see p. 39).

[36] D. Binkley, “Source code analysis: A road map,” in *Future of Software Engineering (FOSE)*, 2007, pp. 104–119 (see pp. 2, 26).

[37] 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. 150).

[38] S. Blackshear and S. K. Lahiri, “Almost-correct specifications: A modular semantic framework for assigning confidence to warnings,” in *Programming Language Design and Implementation (PLDI)*, 2013, pp. 209–218 (see p. 65).

[39] J. M. Bland and D. G. Altman, “Statistical methods for assessing agreement between two methods of clinical measurement,” *The Lancet*, vol. 327, no. 8476, pp. 307–310, 1986 (see p. 147).

[40] D. G. Bobrow, S. Mittal, and M. J. Stefik, “Expert systems: Perils and promise,” *Communications of the ACM*, vol. 29, no. 9, pp. 880–894, 1986 (see p. 73).

[41] M. D. Bond, G. Z. Baker, and S. Z. Guyer, “Breadcrumbs: Efficient context sensitivity for dynamic bug detection analyses,” in *Programming Language Design and Implementation (PLDI)*, 2010, pp. 13–24 (see p. 64).

[42] P. N. van den Bosch, “A bibliography on syntax error handling in context free languages,” *ACM SIGPLAN Notices*, vol. 27, no. 4, pp. 77–86, 1992 (see p. 32).

[43] 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 pp. 5, 80).

[44] 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. 42).

[45] 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. 72).

[46] 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. 72).

[47] G. Brooks, G. J. Hansen, and S. Simmons, “A new approach to debugging optimized code,” in *Programming Language Design and Implementation (PLDI)*, 1992, pp. 1–11 (see p. 60).

[48] 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 pp. 69, 70).

[49] ——, “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. 69).

[50] P. J. Brown, “Error messages: The neglected area of the man/machine interface,” *Communications of the ACM*, vol. 26, no. 4, pp. 246–249, 1983 (see pp. 2, 50, 52, 163, 248).

[51] 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 *Programming Language Design and Implementation (PLDI)*, 2013, pp. 95–104 (see p. 63).

[52] 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 *International Conference on Program Comprehension (ICPC)*, 2015, pp. 255–265 (see p. 108).

[53] C. Calcagno and D. Distefano, "Infer: An automatic program verifier for memory safety of C programs," in *NASA Formal Methods (NFM)*, 2011, pp. 459–465 (see p. 2).

[54] J. L. Campbell, C. Quincy, J. Osserman, and O. K. Pedersen, "Coding in-depth semistructured interviews: Problems of unitization and intercoder reliability and agreement," *Sociological Methods & Research*, vol. 42, no. 3, pp. 294–320, 2013 (see p. 147).

[55] J. C. Campbell, A. Hindle, and J. N. Amaral, "Syntax errors just aren't natural: Improving error reporting with language models," in *Mining Software Repositories (MSR)*, 2014, pp. 252–261 (see pp. 2, 32, 33).

[56] E. Chailloux, P. Manoury, and B. Pagano, "Program analysis tools," in *Developing Applications with Objective Caml*, O'Reilly, 2000 (see p. 26).

[57] 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. 73).

[58] B. Chandrasekaran, M. C. Tanner, and J. R. Josephson, "Explaining control strategies in problem solving," *IEEE Expert*, vol. 4, no. 1, pp. 9–15, 19–24, 1989 (see p. 16).

[59] A. Charguéraud, "Improving type error messages in OCaml," in *ML Family/OCaml Workshops*, 2014, pp. 80–97 (see p. 40).

[60] P. Charles and D. Shields. (1998). Frequently asked questions about Jikes, [Online]. Available: `https://www.cc.gatech.edu/data_files/public/doc/jikesfaq.html` (see p. 141).

[61] S. Chen, M. Erwig, and K. Smeltzer, "Let's hear both sides: On combining type-error reporting tools," in *Visual Languages and Human-Centric Computing (VL/HCC)*, 2014, pp. 145–152 (see p. 42).

[62] Y. Chen, A. Groce, C. Zhang, W.-K. Wong, X. Fern, E. Eide, and J. Regehr, "Taming compiler fuzzers," in *Programming Language Design and Implementation (PLDI)*, 2013, pp. 197–208 (see p. 65).

[63] S. Cherem, L. Princehouse, and R. Rugina, “Practical memory leak detection using guarded value-flow analysis,” in *Programming Language Design and Implementation (PLDI)*, 2007, pp. 480–491 (see p. 63).

[64] P. Chiusano. (2017). Unison: Next-generation programming platform, [Online]. Available: `http://unisonweb.org/` (see p. 76).

[65] M. Christakis and C. Bird, “What developers want and need from program analysis: An empirical study,” in *Automated Software Engineering (ASE)*, 2016, pp. 332–343 (see pp. 2, 13).

[66] 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. 73).

[67] E. M. Clarke, “The birth of model checking,” in *25 Years of Model Checking: History, Achievements, Perspectives*, Springer, 2008, pp. 1–26 (see p. 42).

[68] E. M. Clarke, O. Grumberg, and D. A. Peleg, *Model Checking*. The MIT Press, 1999 (see p. 42).

[69] E. Clarke, D. Kroening, and F. Lerda, “A tool for checking ANSI-C programs,” in *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2004, pp. 168–176 (see p. 43).

[70] R. W. Conway and T. R. Wilcox, “Design and implementation of a diagnostic compiler for PL/I,” *Communications of the ACM*, vol. 16, no. 3, pp. 169–179, 1973 (see pp. 77, 80).

[71] 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. 72).

[72] L. Cooke and E. Cuddihy, “Using eye tracking to address limitations in think-aloud protocol,” in *International Professional Communication Conference (IPCC)*, 2005, pp. 653–658 (see p. 108).

[73] E. Coppa, C. Demetrescu, and I. Finocchi, “Input-sensitive profiling,” in *Programming Language Design and Implementation (PLDI)*, 2012, pp. 89–98 (see p. 65).

[74] J. C. Corbett, M. B. Dwyer, J. Hatcliff, and Robby, “Bandera: A source-level interface for model checking Java programs,” in *International Conference on Software Engineering (ICSE)*, 2000, pp. 762–765 (see p. 43).

[75] D. S. Coutant, S. Meloy, and M. Ruscetta, “Doc: A practical approach to source-level debugging of globally optimized code,” in *Programming Language Design and Implementation (PLDI)*, 1988, pp. 125–134 (see p. 60).

[76] J. W. Creswell, *Research Design: Qualitative, Quantitative, and Mixed Methods Approaches*, 4th ed. SAGE Publications, 2014 (see p. 17).

[77] M. E. Crosby, J. Scholtz, and S. Wiedenbeck, “The roles beacons play in comprehension for novice and expert programmers,” in *Psychology of Programming Interest Group (PPIG)*, 2002, pp. 58–73 (see p. 70).

[78] M. Crotty, *The Foundations of Social Research: Meaning and Perspective in the Research Process*. SAGE Publications, 1998 (see p. 17).

[79] C. Csallner and Y. Smaragdakis, “Check ’n’ Crash: Combining static checking and testing,” in *International Conference on Software Engineering (ICSE)*, 2005, pp. 422–431 (see p. 27).

[80] E. Czaplicki. (2015). Compiler errors for humans, [Online]. Available: `http://elm-lang.org/blog/compiler-errors-for-humans` (see p. 36).

[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. 42).

[82] L. Damas and R. Milner, “Principal type-schemes for functional programs,” in *Principles of Programming Languages (POPL)*, 1982, pp. 207–212 (see p. 39).

[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 (SEFM)*, 2017, pp. 168–184 (see pp. 2, 44).

[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. 45).

[85] I. F. Darwin, *Java Cookbook*. O’Reilly, 2004 (see p. 141).

[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. 72).

[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. 77).

[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. 73).

[89] M. Dean, “How a computer should talk to people,” *IBM Systems Journal*, vol. 21, no. 4, pp. 424–453, 1982 (see pp. 17, 50–52, 162–164, 247).

[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. 32).

[91] ——, “LR techniques for handling syntax errors,” *Computer Languages*, vol. 24, no. 2, pp. 73–98, 1998 (see p. 32).

[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. 71).

[93] P. Denny, A. Luxton-Reilly, and D. Carpenter, “Enhancing syntax error messages appears ineffectual,” in *Innovation and Technology in Computer Science Education (ITiCSE)*, 2014, pp. 273–278 (see pp. 79, 80, 82).

[94] P. Denny, A. Luxton-Reilly, E. Tempero, and J. Hendrickx, “Understanding the syntax barrier for novices,” in *Innovation and Technology in Computer Science Education (ITiCSE)*, 2011, pp. 208–212 (see p. 49).

[95] N. K. Denzin, “Moments, mixed methods, and paradigm dialogs,” *Qualitative Inquiry*, vol. 16, no. 6, pp. 419–427, 2010 (see p. 18).

[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. 72).

[97] L. Diekmann and L. Tratt, “Eco: A language composition editor,” in *Software Language Engineering (SLE)*, 2014, pp. 82–101 (see p. 76).

[98] I. Dillig, T. Dillig, and A. Aiken, “Automated error diagnosis using abductive inference,” in *Programming Language Design and Implementation (PLDI)*, 2012, pp. 181–192 (see p. 63).

[99] D. von Dincklage and A. Diwan, "Explaining failures of program analyses," in *Programming Language Design and Implementation (PLDI)*, 2008, pp. 260–269 (see p. 64).

[100] A. Dix, J. Finlay, G. D. Abowd, and R. Beale, *Human-Computer Interaction*, 3rd ed. Prentice-Hall, 2004 (see p. 51).

[101] V. Donzeau-Gouge, G. Huet, G. Kahn, and B. Lang, "Programming environments based on structured editors: The MENTOR experience," French Institute for Research in Computer Science and Automation, Tech. Rep., 1980 (see p. 76).

[102] 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. Supplement 1, pp. 3633–3640, 2012 (see p. 72).

[103] D. Duggan and F. Bent, "Explaining type inference," *Science of Computer Programming*, vol. 27, no. 1, pp. 37–83, 1996 (see pp. 39, 42).

[104] S. Easterbrook, J. Singer, M.-A. Storey, and D. Damian, "Selecting empirical methods for software engineering research," in *Guide to Advanced Empirical Software Engineering*, 2008, ch. 11, pp. 285–311 (see p. 18).

[105] Eclipse Foundation. (2014). Eclipse (Luna), [Online]. Available: `http://www.eclipse.org` (see pp. 29, 45, 90).

[106] J. Edlund, J. Gustafson, M. Heldner, and A. Hjalmarsson, "Towards human-like spoken dialogue systems," *Speech Communication*, vol. 50, no. 8–9, pp. 630–645, 2008 (see p. 16).

[107] J. Edworthy and A. Adams, *Warning Design: A Research Prospective*. Taylor & Francis, 1996 (see pp. 71, 72).

[108] 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. 72).

[109] 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*. Springer, 2014 (see pp. 138, 156, 157).

[110] N. El Boustani and J. Hage, “Improving type error messages for generic Java,” in *Partial Evaluation and Program Manipulation (PEPM)*, 2009, pp. 131–140 (see p. 42).

[111] F. Elbabour, O. Alhadreti, and P. Mayhew, “Eye tracking in retrospective think-aloud usability testing: Is there added value?” *Journal of Usability Studies*, vol. 12, no. 3, pp. 95–110, 2017 (see p. 108).

[112] M. D. Ernst, “Static and dynamic analysis: Synergy and duality,” in *Workshop on Dynamic Analysis (WODA)*, 2003, pp. 25–28 (see pp. 26, 27).

[113] G. Evangelidis, V. Dagdilelis, M. Satratzemi, and V. Efopoulos, “X-Compiler: Yet another integrated novice programming environment,” in *International Conference on Advanced Learning Technologies (ICALT)*, 2001, pp. 166–169 (see pp. 77, 80).

[114] M. Faddegon and O. Chitil, “Lightweight computation tree tracing for lazy functional languages,” in *Programming Language Design and Implementation (PLDI)*, 2016, pp. 114–128 (see p. 66).

[115] E. A. Feigenbaum, B. G. Buchanan, and J. Lederberg, “On generality and problem solving: A case study using the DENDRAL program,” Stanford University Computer Science Department, Tech. Rep., 1970 (see p. 73).

[116] 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. 71).

[117] 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)*, Springer, 1997, pp. 369–388 (see p. 77).

[118] G. Fischer, T. Mastaglio, B. Reeves, and J. Rieman, “Minimalist explanations in knowledge-based systems,” in *Hawaii International Conference on System Sciences (HICSS)*, vol. 3, 1990, pp. 309–317 (see p. 74).

[119] P. M. Fitts, R. E. Jones, and J. L. Milton, “Eye movements of aircraft pilots during instrument-landing approaches,” *Aeronautical Engineering Review*, vol. 9, no. 2, pp. 23–29, 1950 (see p. 163).

[120] C. Flanagan, M. Flatt, S. Krishnamurthi, S. Weirich, and M. Felleisen, “Catching bugs in the web of program invariants,” in *Programming Language Design and Implementation (PLDI)*, 1996, pp. 23–32 (see pp. 47, 64, 158).

[121] J. L. Fleiss, B. Levin, and M. C. Paik, *Statistical Methods for Rates and Proportions*. John Wiley & Sons, 2013 (see p. 147).

[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, 14:1–14:41, 2013 (see p. 70).

[123] T. Flowers, C. A. Carver, and J. Jackson, "Empowering students and building confidence in novice programmers through Gauntlet," in *Frontiers in Education (FIE)*, vol. 1, 2004, T3H/10–T3H/13 (see p. 80).

[124] B. J. Fogg, *Persuasive Technology: Using Computers to Change What We Think and Do*. Morgan Kaufmann, 2002 (see p. 72).

[125] S. N. Freund and E. S. Roberts, "Thetis: An ANSI C programming environment designed for introductory use," in *ACM Technical Symposium on Computer Science Education (SIGCSE)*, 1996, pp. 300–304 (see pp. 77, 79, 80).

[126] S. Garcia, D. Jeon, C. M. Louie, and M. B. Taylor, "Kremlin: Rethinking and rebooting gprof for the multicore age," in *Programming Language Design and Implementation (PLDI)*, 2011, pp. 458–469 (see p. 65).

[127] H. Gast, "Explaining ML type errors by data flows," in *Implementation and Application of Functional Languages (IFL)*, 2005, pp. 72–89 (see p. 42).

[128] Gazepoint. (2018). Gazepoint, [Online]. Available: `http://www.gazept.com` (see p. 90).

[129] GCC Project. (2018). GNU Compiler Collection (GCC 7.3.0), [Online]. Available: `http://gcc.gnu.org/` (see pp. 4, 29).

[130] X. Ge and E. Murphy-Hill, "Manual refactoring changes with automated refactoring validation," in *International Conference on Software Engineering (ICSE)*, 2014, pp. 1095–1105 (see p. 77).

[131] GitHub. (2018). Atom, [Online]. Available: `http://atom.io` (see p. 47).

[132] L. M. Given, Ed., *The SAGE Encyclopedia of Qualitative Research Methods*. SAGE Publications, 2008, vol. 1–2 (see p. 17).

[133] A. Gomolka and B. Humm, “Structure editors: Old hat or future vision?” In *Evaluation of Novel Approaches to Software Engineering (ENAS)*, 2013, pp. 82–97 (see p. 76).

[134] Google. (2017). Error Prone, [Online]. Available: `http://errorprone.info/` (see p. 34).

[135] ——, (2018). Bazel: A fast, scalable, multi-language and extensible build system, [Online]. Available: `https://bazel.build/` (see p. 34).

[136] ——, (2018). Google C++ style guide, [Online]. Available: `https://google.github.io/styleguide/cppguide.html` (see pp. 78, 79).

[137] A. Gosain and G. Sharma, “A survey of dynamic program analysis techniques and tools,” in *Frontiers of Intelligent Computing: Theory and Applications (FICTA)*. 2015, pp. 113–122 (see pp. 2, 26).

[138] ——, “Static analysis: A survey of techniques and tools,” in *International Conference on Intelligent Computing and Applications (ICICA)*, 2015, pp. 581–591 (see pp. 2, 26).

[139] J. Gosling, B. Joy, G. Steele, G. Bracha, A. Buckley, and D. Smith. (2017). Java language and virtual machine specifications, [Online]. Available: `http://docs.oracle.com/javase/specs` (see p. 6).

[140] B. Gough, “Common error messages,” in *An Introduction to GCC*. Network Theory, 2005, ch. 13 (see p. 4).

[141] M. J. Grant and A. Booth, “A typology of reviews: An analysis of 14 review types and associated methodologies,” *Health Information and Libraries Journal*, vol. 26, no. 2, pp. 91–108, 2009 (see p. 57).

[142] 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. 125).

[143] D. Greenaway, J. Lim, J. Andronick, and G. Klein, “Don’t sweat the small stuff: Formal verification of C code without the pain,” in *Programming Language Design and Implementation (PLDI)*, 2014, pp. 429–439 (see p. 60).

[144] 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. 74).

[145] A. Groce and W. Visser, “What went wrong: Explaining counterexamples,” in *SPIN Workshop on Model Checking of Software (SPIN)*, Berlin, Heidelberg, 2003, pp. 121–136 (see p. 44).

[146] L. Gugerty and G. Olson, “Debugging by skilled and novice programmers,” in *Human Factors in Computing Systems (CHI)*, 1986, pp. 171–174 (see p. 79).

[147] A. Gurfinkel and M. Chechik, “Proof-like counter-examples,” in *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2003, pp. 160–175 (see p. 45).

[148] 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 *Programming Language Design and Implementation (PLDI)*, 2007, pp. 101–111 (see p. 61).

[149] 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. 95).

[150] 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. 32).

[151] M. T. Harandi and J. Q. Ning, “PAT: A knowledge-based program analysis tool,” in *International Conference on Software Maintenance (ICSM)*, 1988, pp. 312–318 (see p. 73).

[152] 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 (FC)*, 2013, pp. 94–111 (see p. 72).

[153] C. Hardaker, “Trolling in asynchronous computer-mediated communication: From user discussions to academic definitions,” *Journal of Politeness Research*, vol. 6, no. 2, pp. 215–242, 2010 (see p. 146).

[154] 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. 72).

[155] L. R. Harris and M. Jenkin, “Vision and attention,” in *Vision and Attention*, Springer, 2001, ch. 1, pp. 1–17 (see p. 95).

[156] B. Hartmann, D. MacDougall, J. Brandt, and S. R. Klemmer, "What would other programmers do? Suggesting solutions to error messages," in *Human Factors in Computing Systems (CHI)*, 2010, pp. 1019–1028 (see p. 38).

[157] Haskell Project. (2017). Glasgow Haskell Compiler (GHC 8.0.2), [Online]. Available: `http://www.haskell.org` (see p. 38).

[158] R. W. Hasker, "HiC: A C++ compiler for CS1," *Journal of Computing Sciences in Colleges*, vol. 18, no. 1, pp. 56–64, 2002 (see pp. 77, 78).

[159] A. Head, C. Appachu, M. A. Hearst, and B. Hartmann, "Tutorons: Generating context-relevant, on-demand explanations and demonstrations of online code," in *Visual Languages and Human-Centric Computing (VL/HCC)*, 2015, pp. 3–12 (see p. 37).

[160] B. Heeren, "Top quality type error messages," PhD thesis, Utrecht University, 2005 (see p. 42).

[161] B. Heeren, J. Hage, and S. D. Swierstra, "Scripting the type inference process," in *International Conference on Functional Programming (ICFP)*, 2003, pp. 3–13 (see p. 42).

[162] B. Heeren, D. Leijen, and A. van IJzendoorn, "Helium, for learning Haskell," in *Workshop on Haskell*, 2003, pp. 62–71 (see pp. 40, 78).

[163] P. Hejmady and N. H. Narayanan, "Visual attention patterns during program debugging with an IDE," in *Eye Tracking Research and Applications (ETRA)*, 2012, pp. 197–200 (see p. 108).

[164] S. Henninger, K. Haynes, and M. W. Reith, "A framework for developing experience-based usability guidelines," in *Designing Interactive Systems (DIS)*, 1995, pp. 43–53 (see p. 162).

[165] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre, "Software verification with BLAST," in *SPIN Workshop on Model Checking of Software (SPIN)*, Berlin, Heidelberg, 2003, pp. 235–239 (see p. 43).

[166] M. Hilton, N. Nelson, D. Dig, T. Tunnell, and D. Marinov, "Continuous integration (CI) needs and wishes for developers of proprietary code," Oregon State University, Tech. Rep., 2016 (see p. 147).

[167] A. Hindle, E. T. Barr, Z. Su, M. Gabel, and P. Devanbu, "On the naturalness of software," in *International Conference on Software Engineering (ICSE)*, 2012, pp. 837–847 (see p. 76).

[168] K. J. Hoffman, P. Eugster, and S. Jagannathan, “Semantics-aware trace analysis,” in *Programming Language Design and Implementation (PLDI)*, 2009, pp. 453–464 (see p. 61).

[169] R. C. Holt, D. B. Wortman, D. T. Barnard, and J. R. Cordy, “SP/k: A system for teaching computer programming,” *Communications of the ACM*, vol. 20, no. 5, pp. 301–309, 1977 (see p. 77).

[170] G. J. Holzmann, “The model checker SPIN,” *IEEE Transactions on Software Engineering*, vol. 23, no. 5, pp. 279–295, 1997 (see p. 43).

[171] 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. 74).

[172] J. J. Horning, “What the compiler should tell the user,” in *Compiler Construction: An Advanced Course*, Springer, 1974, pp. 525–548 (see pp. 50, 51, 53, 162, 246).

[173] S. Horwitz, “Identifying the semantic and textual differences between two versions of a program,” in *Programming Language Design and Implementation (PLDI)*, 1990, pp. 234–245 (see p. 61).

[174] Y. Hoskote, T. Kam, P.-H. Ho, and X. Zhao, “Coverage estimation for symbolic model checking,” in *Design Automation Conference (DAC)*, 1999, pp. 300–305 (see p. 45).

[175] M. Hristova, A. Misra, M. Rutter, and R. Mercuri, “Identifying and correcting Java programming errors for introductory computer science students,” in *ACM Technical Symposium on Computer Science Education (SIGCSE)*, 2003, pp. 153–156 (see p. 80).

[176] C. Isradisaikul and A. C. Myers, “Finding counterexamples from parsing conflicts,” in *Programming Language Design and Implementation (PLDI)*, 2015, pp. 555–564 (see p. 62).

[177] D. Jackson and M. Vaziri, “Finding bugs with a constraint solver,” in *International Symposium on Software Testing and Analysis (ISSTA)*, 2000, pp. 14–25 (see p. 2).

[178] J. Jackson, M. Cobb, and C. Carver, “Identifying top Java errors for novice programmers,” in *Frontiers in Education (FIE)*, 2005, pp. 24–27 (see p. 49).

[179] J. Z. Jacobson and P. Dodwell, "Saccadic eye movements during reading," *Brain and Language*, vol. 8, no. 3, pp. 303–314, 1979 (see pp. 97, 101).

[180] 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. 72).

[181] 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 p. 33).

[182] JetBrains. (2018). dotCover, [Online]. Available: `http://www.jetbrains.com` (see p. 47).

[183] ——, (2018). IntelliJ, [Online]. Available: `https://www.jetbrains.com` (see pp. 45, 90).

[184] R. Jhala and R. Majumdar, "Software model checking," *ACM Computing Surveys*, vol. 41, no. 4, 21:1–21:54, 2009 (see p. 43).

[185] G. Jin, L. Song, W. Zhang, S. Lu, and B. Liblit, "Automated atomicity-violation fixing," in *Programming Language Design and Implementation (PLDI)*, 2011, pp. 389–400 (see p. 60).

[186] A. Johnson, L. Waye, S. Moore, and S. Chong, "Exploring and enforcing security guarantees via program dependence graphs," in *Programming Language Design and Implementation (PLDI)*, 2015, pp. 291–302 (see p. 66).

[187] B. Johnson, "A tool (mis)communication theory and adaptive approach for supporting developer tool use," PhD thesis, North Carolina State University, 2017 (see p. 48).

[188] B. Johnson, R. Pandita, E. Murphy-Hill, and S. Heckman, "Bespoke tools: Adapted to the concepts developers know," in *Foundations of Software Engineering (ESEC/FSE)*, 2015, pp. 878–881 (see p. 74).

[189] 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 *Foundations of Software Engineering (FSE)*, 2016, pp. 73–84 (see p. 97).

[190] B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge, "Why don't software developers use static analysis tools to find bugs?" In *International Conference on Software Engineering (ICSE)*, 2013, pp. 672–681 (see pp. 2, 13, 33, 140).

[191] H. Johnson and P. Johnson, “Explanation facilities and interactive systems,” in *Intelligent User Interfaces (IUI)*, 1993, pp. 159–166 (see p. 75).

[192] 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. 18).

[193] S. C. Johnson. (2018). Portable C Compiler (PCC 1.2.0), [Online]. Available: `http://pcc.ludd.ltu.se/` (see p. 3).

[194] W. L. Johnson and E. Soloway, “PROUST: Knowledge-based program understanding,” *Transactions on Software Engineering*, vol. SE-11, no. 3, pp. 267–275, 1985 (see p. 73).

[195] M. Jose and R. Majumdar, “Cause clue clauses: Error localization using maximum satisfiability,” in *Programming Language Design and Implementation (PLDI)*, 2011, pp. 437–446 (see p. 64).

[196] J. M. Joyce, “Kullback-Leibler divergence,” in *International Encyclopedia of Statistical Science*. Springer, 2011, pp. 720–722 (see p. 96).

[197] 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. 50, 52, 53, 248).

[198] C. Kelleher and R. Pausch, “Lowering the barriers to programming: A taxonomy of programming environments and languages for novice programmers,” *ACM Computing Surveys*, vol. 37, no. 2, pp. 83–137, 2005 (see p. 77).

[199] A. Kent and B. Kent. (2018). Isomof, [Online]. Available: `https://isomorf.io/` (see p. 76).

[200] Y. P. Khoo, J. S. Foster, M. Hicks, and V. Sazawal, “Path projection for user-centered static analysis tools,” in *Program Analysis for Software Tools and Engineering (PASTE)*, 2008, pp. 57–63 (see p. 159).

[201] A. A. Khwaja and J. E. Urban, “Syntax-directed editing environments: Issues and features,” in *ACM Symposium on Applied Computing (SAC)*, 1993, pp. 230–237 (see p. 76).

[202] 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. 72).

[203] B. Kitchenham, “Procedures for performing systematic reviews,” Keele University and National ICT Australia, Tech. Rep., 2004 (see p. 56).

[204] 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. 72).

[205] 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. 1–2, pp. 41–84, 2005 (see pp. 111, 113).

[206] ——, “Finding causes of program output with the Java Whyline,” in *Human Factors in Computing Systems (CHI)*, 2009, pp. 1569–1578 (see p. 48).

[207] J. Koenemann and S. P. Robertson, “Expert problem solving strategies for program comprehension,” in *Human Factors in Computing Systems (CHI)*, 1991, pp. 125–130 (see p. 68).

[208] M. Kölling, B. Quig, A. Patterson, and J. Rosenberg, “The BlueJ system and its pedagogy,” *Computer Science Education*, vol. 13, no. 4, pp. 249–268, 2003 (see p. 49).

[209] N. Kulkarni and V. Varma, “Supporting comprehension of unfamiliar programs by modeling an expert’s perception,” in *Realizing Artificial Intelligence Synergies in Software Engineering (RAISE)*, 2014, pp. 19–24 (see p. 79).

[210] S. K. Kummerfeld and J. Kay, “The neglected battle fields of syntax errors,” in *Australasian Conference on Computing Education (ACE)*, 2003, pp. 105–111 (see p. 31).

[211] 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. 157).

[212] J. R. Landis and G. G. Koch, “The measurement of observer agreement for categorical data,” *Biometrics*, vol. 33, no. 1, pp. 159–174, 1977 (see p. 147).

[213] B. Lang, “On the usefulness of syntax directed editors,” in *Advanced Programming Environments*, 1986, pp. 47–51 (see p. 76).

[214] B. Lang, “Teaching new programmers: A Java tool set as a student teaching aid,” in *Principles and Practice of Programming (PPPJ)*, 2002, pp. 95–100 (see p. 80).

[215] C. Lattner and V. Adve, “LLVM: A compilation framework for lifelong program analysis & transformation,” in *Code Generation and Optimization (CGO)*, 2004, pp. 75–86 (see p. 27).

[216] 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 pp. 71, 73).

[217] 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 pp. 71, 72).

[218] 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. 70, 71).

[219] M. R. Lehto and J. M. Miller, *Warnings: Fundamentals, Design, and Evaluation Methodologies*. Fuller Technical Publications, 1986 (see p. 71).

[220] B. S. Lerner, M. Flower, D. Grossman, and C. Chambers, “Searching for type-error messages,” in *Programming Language Design and Implementation (PLDI)*, 2007, pp. 425–434 (see pp. 41, 62).

[221] S. Letovsky and E. Soloway, “Delocalized plans and program comprehension,” *IEEE Software*, vol. 3, no. 3, pp. 41–49, 1986 (see p. 69).

[222] S. Letovsky, “Cognitive processes in program comprehension,” *Journal of Systems and Software*, vol. 7, no. 4, pp. 325–339, 1987 (see p. 70).

[223] A. Leung, J. Sarracino, and S. Lerner, “Interactive parser synthesis by example,” in *Programming Language Design and Implementation (PLDI)*, 2015, pp. 565–574 (see p. 63).

[224] S.-H. 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. 73).

[225] B. Liblit, A. Aiken, A. X. Zheng, and M. I. Jordan, “Bug isolation via remote program sampling,” in *Programming Language Design and Implementation (PLDI)*, 2003, pp. 141–154 (see p. 61).

[226] B. Liblit, M. Naik, A. X. Zheng, A. Aiken, and M. I. Jordan, "Scalable statistical bug isolation," in *Programming Language Design and Implementation (PLDI)*, 2005, pp. 15–26 (see p. 61).

[227] T. Lieber, J. R. Brandt, and R. C. Miller, "Addressing misconceptions about code with always-on programming visualizations," in *Human Factors in Computing Systems (CHI)*, 2014, pp. 2481–2490 (see pp. 48, 159).

[228] J. Liebowitz, Ed., *The Handbook of Applied Expert Systems*. CRC Press, 1997 (see p. 73).

[229] B. Y. Lim, A. K. Dey, and D. Avrahami, "Why and why not explanations improve the intelligibility of context-aware intelligent systems," in *Human Factors in Computing Systems (CHI)*, 2009, pp. 2119–2128 (see p. 74).

[230] C. R. Litecky and G. B. Davis, "A study of errors, error-proneness, and error diagnosis in Cobol," *Communications of the ACM*, vol. 19, no. 1, pp. 33–38, 1976 (see p. 49).

[231] C. Litecky, "An expert system for COBOL program debugging," *ACM SIGMIS Database: The DATABASE for Advances in Information Systems*, vol. 20, no. 1, pp. 1–6, 1989 (see pp. 35, 73).

[232] B. Livshits, A. V. Nori, S. K. Rajamani, and A. Banerjee, "Merlin: Specification inference for explicit information flow problems," in *Programming Language Design and Implementation (PLDI)*, 2009, pp. 75–86 (see p. 67).

[233] LLVM Project. (2017). Expressive diagnostics, [Online]. Available: `http://clang.llvm.org/diagnostics.html` (see p. 31).

[234] ——, (2018). Clang Static Analyzer, [Online]. Available: `http://clang-analyzer.llvm.org` (see p. 106).

[235] ——, (2018). LLVM Compiler Infrastructure (clang 6.0.0), [Online]. Available: `http://llvm.org/` (see pp. 4, 7, 29).

[236] F. Logozzo, S. K. Lahiri, M. Fähndrich, and S. Blackshear, "Verification modulo versions: Towards usable verification," in *Programming Language Design and Implementation (PLDI)*, 2014, pp. 294–304 (see p. 66).

[237] C. Loncaric, S. Chandra, C. Schlesinger, and M. Sridharan, "A practical framework for type inference error explanation," in *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2016, pp. 781–799 (see p. 42).

[238] E. Lotem and Y. Chuchem. (2016). Lamdu: Towards a new programming experience, [Online]. Available: http://www.lamdu.org/ (see p. 76).

[239] T. F. Lunney and R. H. Perrott, "Syntax-directed editing," *Software Engineering Journal*, vol. 3, no. 2, pp. 37–46, 1988 (see p. 76).

[240] M. Madsen, B. Livshits, and M. Fanning, "Practical static analysis of JavaScript applications in the presence of frameworks and libraries," in *Foundations of Software Engineering (ESEC/FSE)*, 2013, pp. 499–509 (see p. 77).

[241] L. Mamykina, B. Manoim, M. Mittal, G. Hripcsak, and B. Hartmann, "Design lessons from the fastest Q&A site in the west," in *Human Factors in Computing Systems (CHI)*, 2011, pp. 2857–2866 (see p. 143).

[242] 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. 74).

[243] G. Marceau, K. Fisler, and S. Krishnamurthi, "Measuring the effectiveness of error messages designed for novice programmers," in *ACM Technical Symposium on Computer Science Education (SIGCSE)*, 2011, pp. 499–504 (see pp. 81, 82).

[244] ——, "Mind your language: On novices' interactions with error messages," in *Onward!*, 2011, pp. 3–17 (see pp. 78, 80).

[245] W. A. Martin and R. J. Fateman, "The MACSYMA system," in *Symposium on Symbolic and Algebraic Manipulation (SYMSAC)*, 1971, pp. 59–75 (see p. 73).

[246] B. Matthews, R. Andronaco, and A. Adams, "Warning signs at beaches: Do they work?" *Safety Science*, vol. 62, pp. 312–318, 2014 (see p. 71).

[247] J. A. Maxwell, "Using numbers in qualitative research," *Qualitative Inquiry*, vol. 16, no. 6, pp. 475–482, 2010 (see p. 149).

[248] G. R. Mayes, "Argument-explanation complementarity and the structure of informal reasoning," *Informal Logic*, vol. 30, no. 1, pp. 92–111, 2010 (see p. 16).

[249] A. von Mayrhauser and A. M. Vans, "From program comprehension to tool requirements for an industrial environment," in *IEEE Workshop on Program Comprehension*, 1993, pp. 78–86 (see p. 70).

[250] B. McAdam, "How to repair type errors automatically," in *Trends in Functional Programming*, vol. 3, Intellect Books, 2002, ch. 8 (see pp. 5, 41).

[251] B. J. McAdam, "On the unification of substitutions in type inference," in *Implementation of Functional Languages (IFL '98)*, 1999, pp. 137–152 (see pp. 40, 42).

[252] 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. 79).

[253] M. W. McKeon, "On the rationale for distinguishing arguments from explanations," *Argumentation*, vol. 27, no. 3, pp. 283–303, 2013 (see p. 16).

[254] K. R. McKeown, M. Wish, and K. Matthews, "Tailoring explanations for the user," in *International Joint Conference on Artificial Intelligence (IJCAI)*, 1985, pp. 794–798 (see p. 74).

[255] M. McLuhan, *Understanding Media: The Extensions of Man*. The MIT Press, 1994 (see p. 9).

[256] D. Menendez and S. Nagarakatte, "Alive-Infer: Data-driven precondition inference for peephole optimizations in LLVM," in *Programming Language Design and Implementation (PLDI)*, 2017, pp. 49–63 (see p. 62).

[257] Microsoft. (2018). Mono (5.12.0), [Online]. Available: `http://www.mono-project.com` (see p. 29).

[258] ——, (2018). TypeScript (2.9), [Online]. Available: `http://www.typescriptlang.org` (see p. 165).

[259] ——, (2018). Visual Studio 2017, [Online]. Available: `http://www.visualstudio.com` (see pp. 29, 45, 90).

[260] ——, (2018). Visual Studio Code, [Online]. Available: `http://code.visualstudio.com` (see p. 3).

[261] 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. 74).

[262] J. D. Moore, "What makes human explanations effective?" In *Conference of the Cognitive Science Society*, 1993, pp. 131–136 (see p. 16).

[263] 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. 105).

[264] 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. 17).

[265] 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*, vol. 53B, no. 4, P240–P254, 1998 (see p. 71).

[266] 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*, Springer, 2016, pp. 279–294 (see p. 43).

[267] 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. 31, 50, 51, 77, 163, 246).

[268] F. Mulder. (2016). Awesome error messages for Dotty, [Online]. Available: `https://www.scala-lang.org/blog/2016/10/14/dotty-errors.html` (see pp. 33, 36).

[269] 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. 48, 50, 53, 163, 249).

[270] 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. 47, 50, 53, 82, 163, 249).

[271] M. Musuvathi, D. Y. W. Park, A. Chou, D. R. Engler, and D. L. Dill, "CMC: A pragmatic approach to model checking real code," in *Operating Systems Design and Implementation (OSDI)*, 2002, pp. 75–88 (see p. 2).

[272] 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. 97).

[273] S. Narayanasamy, Z. Wang, J. Tigani, A. Edwards, and B. Calder, “Automatically classifying benign and harmful data races using replay analysis,” in *Programming Language Design and Implementation (PLDI)*, 2007, pp. 22–31 (see p. 61).

[274] 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 *International Conference on Software Maintenance (ICSM)*, 2012, pp. 25–34 (see p. 143).

[275] C. Nass, J. Steuer, and E. R. Tauber, “Computers are social actors,” in *Human Factors in Computing Systems (CHI)*, 1994, pp. 72–78 (see p. 16).

[276] NCrunch Project. (2018). NCrunch, [Online]. Available: `http://www.ncrunch.net` (see p. 47).

[277] T. Nelson, N. Danas, D. J. Dougherty, and S. Krishnamurthi, “The power of ‘why’ and ‘why not’: Enriching scenario exploration with provenance,” in *Foundations of Software Engineering (ESEC/FSE)*, 2017, pp. 106–116 (see p. 44).

[278] T. Nelson, S. Saghafi, D. J. Dougherty, K. Fisler, and S. Krishnamurthi, “Aluminum: Principled scenario exploration through minimality,” in *International Conference on Software Engineering (ICSE)*, 2013, pp. 232–241 (see p. 44).

[279] P. C. Nguyễn and D. Van Horn, “Relatively complete counterexamples for higher-order programs,” in *Programming Language Design and Implementation (PLDI)*, 2015, pp. 446–456 (see p. 62).

[280] J. Nielsen and R. Molich, “Heuristic evaluation of user interfaces,” in *Human Factors in Computing Systems (CHI)*, 1990, pp. 249–256 (see p. 125).

[281] M.-H. Nienaltowski, M. Pedroni, and B. Meyer, “Compiler error messages: What can help novices?” In *ACM Technical Symposium on Computer Science Education (SIGCSE)*, 2008, pp. 168–172 (see pp. 79, 80, 140).

[282] D. A. Norman, “Design principles for human-computer interfaces,” in *Human Factors in Computing Systems (CHI)*, 1983, pp. 1–10 (see p. 162).

[283] P. Ohmann, A. Brooks, L. D’Antoni, and B. Liblit, “Control-flow recovery from partial failure reports,” in *Programming Language Design and Implementation (PLDI)*, 2017, pp. 390–405 (see p. 66).

[284] C. Omar, I. Voysey, M. Hilton, J. Sunshine, C. L. Goues, J. Aldrich, and M. A. Hammer, "Toward semantic foundations for program editors," in *Summit on Advances in Programming Languages (SNAPL)*, 2017, 11:1–11:12 (see p. 76).

[285] OpenCV Project. (2018). OpenCV, [Online]. Available: `http://opencv.org` (see p. 92).

[286] Oracle. (2018). OpenJDK 8, [Online]. Available: `http://java.sun.com` (see pp. 5, 29).

[287] O. Padon, K. L. McMillan, A. Panda, M. Sagiv, and S. Shoham, "Ivy: Safety verification by interactive generalization," in *Programming Language Design and Implementation (PLDI)*, 2016, pp. 614–630 (see p. 62).

[288] C. L. Paris, "Tailoring object descriptions to a user's level of expertise," *Computational Linguistics*, vol. 14, no. 3, pp. 64–78, 1988 (see p. 74).

[289] 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. 75).

[290] 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. 17).

[291] C. Parnin, "Subvocalization—toward hearing the inner thoughts of developers," in *International Conference on Program Comprehension (ICPC)*, 2011, pp. 197–200 (see p. 111).

[292] C. Parnin and S. Rugaber, "Programmer information needs after memory failure," in *International Conference on Program Comprehension (ICPC)*, 2012, pp. 123–132 (see p. 68).

[293] 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. 71).

[294] 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 *Innovation and Technology in Computer Science Education (ITiCSE)*, 2007, pp. 204–223 (see p. 79).

[295] N. Pennington, "Comprehension strategies in programming," in *Empirical Studies of Programmers*, 1987, pp. 100–113 (see p. 70).

[296] D. Perelman, S. Gulwani, T. Ball, and D. Grossman, “Type-directed completion of partial expressions,” in *Programming Language Design and Implementation (PLDI)*, 2012, pp. 275–286 (see p. 65).

[297] R. S. Pettit, J. Homer, and R. Gee, “Do enhanced compiler error messages help students? Results inconclusive,” in *ACM Technical Symposium on Computer Science Education (SIGCSE)*, 2017, pp. 465–470 (see p. 80).

[298] B. C. Pierce and D. N. Turner, “Local type inference,” *ACM Transactions on Programming Languages and Systems*, vol. 22, no. 1, pp. 1–44, 2000 (see p. 39).

[299] H. Pieterse and H. Gelderblom, “Guidelines for error message design,” *International Journal of Technology and Human Interaction*, vol. 14, no. 1, pp. 80–98, 2018 (see p. 72).

[300] N. Pinkwart, K. Ashley, C. Lynch, and V. Aleven, “Evaluating an intelligent tutoring system for making legal arguments with hypotheticals,” *International Journal of Artificial Intelligence in Education*, vol. 19, no. 4, pp. 401–424, 2009 (see p. 138).

[301] 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 *International Conference on Software Maintenance and Evolution (ICSME)*, 2015, pp. 11–20 (see p. 79).

[302] PLT. (2018). Racket, [Online]. Available: `https://racket-lang.org` (see pp. 32, 48).

[303] 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. 159).

[304] J. Pombrio and S. Krishnamurthi, “Resugaring: Lifting evaluation sequences through syntactic sugar,” in *Programming Language Design and Implementation (PLDI)*, 2014, pp. 361–371 (see p. 58).

[305] J. Ponterotto, “Brief note on the origins, evolution, and meaning of the qualitative research concept thick description,” *The Qualitative Report*, vol. 11, no. 3, pp. 538–549, 2006 (see p. 150).

[306] L. Ponzanelli, A. Bacchelli, and M. Lanza, “Seahawk: Stack Overflow in the IDE,” in *International Conference on Software Engineering (ICSE)*, 2013, pp. 1295–1298 (see p. 38).

[307] 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 *Mining Software Repositories (MSR)*, 2014, pp. 102–111 (see p. 38).

[308] F. Pottier, “Reachability and error diagnosis in LR(1) parsers,” in *International Conference on Compiler Construction (CC)*, 2016, pp. 88–98 (see p. 33).

[309] 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 *International Computing Education Research (ICER)*, 2017, pp. 74–82 (see p. 80).

[310] D. Pritchard, “Frequency distribution of error messages,” in *Evaluation and Usability of Programming Languages and Tools (PLATEAU)*, 2015, pp. 1–8 (see pp. 49, 107).

[311] R. W. Proctor and T. Van Zandt, *Human factors in simple and complex systems*, 3rd ed. CRC Press, 2018 (see p. 163).

[312] F. Puppe, *Systematic Introduction to Expert Systems: Knowledge Representations and Problem-Solving Methods*. Springer, 1993 (see p. 73).

[313] Y. Qian and J. Lehman, “Students’ misconceptions and other difficulties in introductory programming: A literature review,” *ACM Transactions on Computing Education*, vol. 18, no. 1, 1:1–1:24, 2017 (see p. 79).

[314] X. Qiu, P. Garg, A. Ştefănescu, and P. Madhusudan, “Natural proofs for structure, data, and separation,” in *Programming Language Design and Implementation (PLDI)*, 2013, pp. 231–242 (see p. 60).

[315] 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. 42).

[316] M. M. Rahman and C. K. Roy, “SurfClipse: Context-aware meta-search in the IDE,” in *International Conference on Software Maintenance and Evolution (ICSME)*, 2014, pp. 617–620 (see p. 38).

[317] V. Rajlich and N. Wilde, “The role of concepts in program comprehension,” in *International Workshop on Program Comprehension*, 2002, pp. 271–278 (see p. 68).

[318] J. W. Ratcliff and D. E. Metzener, “Pattern matching: The Gestalt approach,” *Dr. Dobbs Journal*, vol. 13, no. 7, p. 46, 1988 (see p. 92).

[319] 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. 96, 101).

[320] J. Regehr, Y. Chen, P. Cuoq, E. Eide, C. Ellison, and X. Yang, “Test-case reduction for C compiler bugs,” in *Programming Language Design and Implementation (PLDI)*, 2012, pp. 335–346 (see p. 62).

[321] G. Riley, “CLIPS: An expert system building tool,” in *National Technology Transfer Conference and Exposition*, vol. 2, 1991, pp. 149–158 (see p. 73).

[322] C. C. Risley and T. J. Smedley, “Visualization of compile time errors in a Java compatible visual language,” in *IEEE Symposium on Visual Languages*, 1998, pp. 22–29 (see p. 48).

[323] R. S. Rist, “Plans in programming: Definition, demonstration, and development,” in *Empirical Studies of Programmers*, vol. 1, 1986, ch. 3, pp. 28–47 (see p. 69).

[324] E. Roberts, “An overview of MiniJava,” in *ACM Technical Symposium on Computer Science Education (SIGCSE)*, 2001, pp. 1–5 (see p. 78).

[325] M. Robillard, R. Walker, and T. Zimmermann, “Recommendation systems for software engineering,” *IEEE Software*, vol. 27, no. 4, pp. 80–86, 2010 (see p. 75).

[326] 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 *International Conference on Software Engineering (ICSE)*, 2014, pp. 390–401 (see p. 108).

[327] 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. 71).

[328] 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 (Diagrams)*, 2002, pp. 221–235 (see p. 108).

[329] C. Rubio-González, H. S. Gunawi, B. Liblit, R. H. Arpaci-Dusseau, and A. C. Arpaci-Dusseau, "Error propagation analysis for file systems," in *Programming Language Design and Implementation (PLDI)*, 2009, pp. 270–280 (see p. 66).

[330] Rust Programming Language Project. (2018). Pre-RFC?: rustc UX Guidelines, [Online]. Available: `https://internals.rust-lang.org/t/pre-rfc-rustc-ux-guidelines/2419` (see p. 167).

[331] ——, (2018). Rust Enhanced, [Online]. Available: `https://github.com/rust-lang/rust-enhanced` (see p. 48).

[332] C. Sadowski, J. van Gogh, C. Jaspan, E. Soderberg, and C. Winter, "Tricorder: Building a program analysis ecosystem," in *International Conference on Software Engineering (ICSE)*, 2015, pp. 598–608 (see pp. 2, 26).

[333] S. Saghafi, R. Danas, and D. J. Dougherty, "Exploring theories with a model-finding assistant," in *International Conference on Automated Deduction (CADE)*, 2015, pp. 434–449 (see p. 44).

[334] J. Saldaña, *The Coding Manual for Qualitative Researchers*. SAGE Publications, 2009 (see p. 149).

[335] P. de le Salle and F. Ouabdesselam, "SIAM, expert system for software maintenance," in *IEE/BCS Conference: Software Engineering*, 1988, pp. 107–111 (see p. 73).

[336] I. Salman, A. T. Misirli, and N. Juristo, "Are students representatives of professionals in software engineering experiments?" In *International Conference on Software Engineering (ICSE)*, 2015, pp. 666–676 (see p. 107).

[337] 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 *Software Analysis, Evolution, and Reengineering (SANER)*, 2018, pp. 311–322 (see p. 33).

[338] J. Scholtz and S. Wiedenbeck, "Learning a new programming language: A model of the planning process," in *Hawaii International Conference on System Sciences (HICSS)*, vol. 2, 1991, pp. 3–12 (see p. 79).

[339] T. Schorsch, Tom, Schorsch, and Tom, “CAP: An automated self-assessment tool to check Pascal programs for syntax, logic and style errors,” in *ACM Technical Symposium on Computer science Education (SIGCSE)*, 1995, pp. 168–172 (see p. 80).

[340] H. J. Schünemann and L. Moja, “Reviews: Rapid! Rapid! Rapid! …and systematic,” *Systematic Reviews*, vol. 4, no. 1, 4:1–4:3, 2015 (see p. 56).

[341] E. L. Seidel, H. Sibghat, K. Chaudhuri, W. Weimer, and R. Jhala, “Learning to blame: Localizing novice type errors with data-driven diagnosis,” *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, vol. 1, 60:1–60:27, 2017 (see p. 42).

[342] H. Seo, C. Sadowski, S. Elbaum, E. Aftandilian, and R. Bowdidge, “Programmers’ build errors: A case study (at Google),” in *International Conference on Software Engineering (ICSE)*, 2014, pp. 724–734 (see pp. 13, 49, 82, 86, 88, 89, 96, 99, 100, 107, 170).

[343] O. Shacham, M. Vechev, and E. Yahav, “Chameleon: Adaptive selection of collections,” in *Programming Language Design and Implementation (PLDI)*, 2009, pp. 408–418 (see p. 64).

[344] U. Shani, “Should program editors not abandon text oriented commands?” *ACM SIGPLAN Notices*, vol. 18, no. 1, pp. 35–41, 1983 (see p. 76).

[345] B. Sharif, M. Falcone, and J. I. Maletic, “An eye-tracking study on the role of scan time in finding source code defects,” in *Eye Tracking Research and Applications (ETRA)*, 2012, pp. 381–384 (see p. 108).

[346] W. J. Shaw, “Making APL error messages kinder and gentler,” in *APL as a Tool of Thought (APL)*, 1989, pp. 320–324 (see pp. 50, 53, 163, 248).

[347] B. Shneiderman, “Measuring computer program quality and comprehension,” *International Journal of Man-Machine Studies*, vol. 9, no. 4, pp. 465–478, 1977 (see pp. 18, 127).

[348] B. Shneiderman, “Exploratory experiments in programmer behavior,” *International Journal of Computer & Information Sciences*, vol. 5, no. 2, pp. 123–143, 1976 (see p. 13).

[349] ——, “Designing computer system messages,” *Communications of the ACM*, vol. 25, no. 9, pp. 610–611, 1982 (see pp. 50, 52, 163, 247).

[350] 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. 69).

[351] E. Shortliffe, *Computer-Based Medical Consultations: MYCIN*. Elsevier, 1976 (see p. 73).

[352] V. J. Shute, "Focus on formative feedback," *Review of Educational Research*, vol. 78, no. 1, pp. 153–189, 2008 (see p. 79).

[353] J. Siegmund, "Framework for measuring program comprehension," PhD thesis, University of Magdeburg, 2012 (see p. 89).

[354] 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 *International Conference on Software Engineering (ICSE)*, 2014, pp. 378–389 (see p. 68).

[355] 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 *Foundations of Software Engineering (ESEC/FSE)*, 2017, pp. 140–150 (see p. 70).

[356] J. Siek and A. Lumsdaine, "Concept checking: Binding parametric polymorphism in C++," in *Workshop on C++ Template Programming*, 2000 (see p. 2).

[357] A. Simon, O. Chitil, and F. Huch, "Typeview: A tool for understanding type errors," in *Implementation of Functional Languages (IFL)*, 2000, pp. 63–69 (see p. 42).

[358] 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, 171:7–171:13, 2013 (see pp. 148, 149).

[359] R. Singh, S. Gulwani, and A. Solar-Lezama, "Automated feedback generation for introductory programming assignments," in *Programming Language Design and Implementation (PLDI)*, 2013, pp. 15–26 (see p. 60).

[360] J. Skinner and W. Bond. (2018). Sublime Text, [Online]. Available: `http://www.sublimetext.com` (see p. 47).

[361] R. Smith, “An overview of the Tesseract OCR engine,” in *International Conference on Document Analysis and Recognition (ICDAR)*, 2007, pp. 629–633 (see p. 92).

[362] A. Solar-Lezama, C. G. Jones, and R. Bodik, “Sketching concurrent data structures,” in *Programming Language Design and Implementation (PLDI)*, 2008, pp. 136–148 (see p. 62).

[363] E. Soloway, B. Adelson, and K. Ehrlich, “Knowledge and processes in the comprehension of computer programs,” in *The Nature of Expertise*, Taylor & Francis, 1988, ch. 4, pp. 129–152 (see p. 69).

[364] 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. 69).

[365] 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. 77).

[366] M. Sridharan, S. J. Fink, and R. Bodik, “Thin slicing,” in *Programming Language Design and Implementation (PLDI)*, 2007, pp. 112–122 (see p. 65).

[367] Stack Overflow. (2017). Stack Overflow developer survey results: 2017, [Online]. Available: `https://insights.stackoverflow.com/survey/2017` (see p. 45).

[368] 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. 73).

[369] J. Stolarek, “Understanding basic Haskell error messages,” *The Monad.Reader*, no. 20, pp. 21–41, 2012 (see p. 39).

[370] 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. 69).

[371] P. J. Stuckey, M. Sulzmann, and J. Wazny, “Interactive type debugging in Haskell,” in *Workshop on Haskell*, 2003, pp. 72–83 (see p. 42).

[372] ——, “Improving type error diagnosis,” in *Workshop on Haskell*, 2004, pp. 80–91 (see p. 42).

[373] B. Sufrin and O. De Moor, “Modeless structure editing,” in *Oxford-Microsoft Symposium in Celebration of the Work of Tony Hoare*, 1999, pp. 1–17 (see p. 76).

[374] B. A. Syed. (2018). Bring your own TypeScript (byots 2.9), [Online]. Available: `https://github.com/basarat/byots` (see p. 253).

[375] J. Tao, N. Yafeng, and Z. Lei, “Are the warning icons more attentional?” *Applied Ergonomics*, vol. 65, pp. 51–60, 2017 (see p. 72).

[376] T. Teitelbaum and T. Reps, “The Cornell Program Synthesizer: A syntax-directed programming environment,” *Communications of the ACM*, vol. 24, no. 9, pp. 563–573, 1981 (see p. 76).

[377] N. Tintarev and J. Masthoff, “A survey of explanations in recommender systems,” in *International Conference on Data Engineering (ICDE)*, 2007, pp. 801–810 (see p. 75).

[378] E. Torlak, M. Vaziri, and J. Dolby, “MemSAT: Checking axiomatic specifications of memory models,” in *Programming Language Design and Implementation (PLDI)*, 2010, pp. 341–350 (see p. 67).

[379] S. Toulmin, *The Uses of Argument*. Cambridge University Press, 2003 (see p. 139).

[380] L. Tratt. (2004). An editor for composed programs, [Online]. Available: `https://tratt.net/laurie/blog/entries/an_editor_for_composed_programs.html` (see p. 76).

[381] V. J. Traver, “On compiler error messages: What they say and what they mean,” *Advances in Human-Computer Interaction*, vol. 2010, 602570:1–602570:26, 2010 (see pp. 2, 50, 53, 163, 248).

[382] C. Treude, O. Barzilay, and M.-A. Storey, “How do programmers ask and answer questions on the web?” In *International Conference on Software Engineering (ICSE)*, 2011, pp. 804–807 (see p. 143).

[383] O. Tripp, M. Pistoia, S. J. Fink, M. Sridharan, and O. Weisman, “TAJ: Effective taint analysis of web applications,” in *Programming Language Design and Implementation (PLDI)*, 2009, pp. 87–97 (see p. 66).

[384] J. Turner. (2016). Shape of errors to come, [Online]. Available: `https://blog.rust-lang.org/2016/08/10/Shape-of-errors-to-come.html` (see pp. 33, 36).

[385] H. Uwano, M. Nakamura, A. Monden, and K.-i. Matsumoto, "Analyzing individual performance of source code review using reviewers' eye movement," in *Eye Tracking Research and Applications (ETRA)*, San Diego, California, 2006, pp. 133–140 (see p. 108).

[386] Valgrind Project. (2017). Valgrind (3.13.0), [Online]. Available: `http://valgrind.org` (see p. 27).

[387] M. L. Van De Vanter, "Practical language-based editing for software engineers," in *Software Engineering and Human-Computer Interaction (SE-HCI)*, 1995, pp. 251–267 (see p. 76).

[388] 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. 75).

[389] 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. 79).

[390] 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. 43).

[391] M. Voelter, J. Siegmund, T. Berger, and B. Kolb, "Towards user-friendly projectional editors," in *Software Language Engineering (SLE)*, 2014, pp. 41–61 (see p. 76).

[392] A. Von Mayrhauser and A. M. Vans, "Program comprehension during software maintenance and evolution," *Computer*, vol. 28, no. 8, pp. 44–55, 1995 (see p. 69).

[393] P. Wadler, "A complement to blame," in *Summit on Advances in Programming Languages (SNAPL 2015)*, vol. 32, Dagstuhl, Germany, 2015, pp. 309–320 (see p. 42).

[394] G. Wallace, R. Biddle, and E. Tempero, "Smarter cut-and-paste for programming text editors," in *Australasian User Interface Conference (AUIC)*, 2001, pp. 56–63 (see p. 77).

[395] D. Walton, "Explanations and arguments based on practical reasoning," in *International Joint Conferences on Artificial Intelligence (IJCAI)*, 2009, pp. 72–83 (see p. 138).

[396] M. Wand, “Finding the source of type errors,” in *Principles of Programming Languages (POPL)*, 1986, pp. 38–43 (see p. 42).

[397] C. Wang, Z. Yang, F. Ivančić, and A. Gupta, “Whodunit? Causal analysis for counterexamples,” in *Automated Technology for Verification and Analysis (ATVA0*, 2006, pp. 82–95 (see p. 45).

[398] R. C. Waters, “Program editors should not abandon text oriented commands,” *ACM SIGPLAN Notices*, vol. 17, no. 7, pp. 39–46, 1982 (see p. 76).

[399] 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 *International Conference on Web-Based Learning (ICWL)*, 2012, pp. 228–239 (see p. 80).

[400] G. Weber, “Code is not just text: Why our code editors are inadequate tools,” in *International Conference on the Art, Science and Engineering of Programming (Programming)*, 2017, 35:1–35:3 (see p. 77).

[401] M. Weiser, “Programmers use slices when debugging,” *Communications of the ACM*, vol. 25, no. 7, pp. 446–452, 1982 (see p. 70).

[402] D. T. Welsh and L. D. Ordóñez, “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. 95).

[403] J. Whittle, A. Bundy, R. Boulton, and H. Lowe, “An ML editor based on proofs-as-programs,” in *Automated Software Engineering (ASE)*, 1999, pp. 166–173 (see pp. 77, 80).

[404] J. Whittle, A. Bundy, and H. Lowe, “An editor for helping novices to learn Standard ML,” in *Programming Languages: Implementations, Logics, and Programs (PLILP)*, 1997, pp. 389–405 (see p. 77).

[405] M. R. Wick and W. B. Thompson, “Reconstructive expert system explanation,” *Artificial Intelligence*, vol. 54, no. 1, pp. 33–70, 1992 (see p. 74).

[406] S. Wiedenbeck, “Beacons in computer program comprehension,” *International Journal of Man-Machine Studies*, vol. 25, no. 6, pp. 697–709, 1986 (see pp. 68, 70).

[407] 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. 95).

[408] M. S. Wogalter and C. B. Mayhorn, “The future of risk communication: Technology-based warning systems,” in *Handbook of Warnings*, CRC Press, 2006, ch. 63, pp. 783–794 (see p. 71).

[409] 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. 72).

[410] 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. 72).

[411] J. Wrenn and S. Krishnamurthi, “Error messages are classifiers: A process to design and evaluate error messages,” in *Onward!*, 2017, pp. 134–147 (see pp. 79, 81).

[412] B. Wu, J. P. Campora, and S. Chen, “Learning user friendly type-error messages,” in *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, 2017, 106:1–106:29 (see p. 41).

[413] Q. Wu and J. R. Anderson, “Problem-solving transfer among programming languages,” Carnegie Mellon University, Tech. Rep., 1990 (see p. 79).

[414] S. Xinogalos, M. Satratzemi, and V. Dagdilelis, “An introduction to object-oriented programming with a didactic microworld: objectKarel,” *Computers & Education*, vol. 47, no. 2, pp. 148–171, 2006 (see pp. 79, 80).

[415] G. Xu, M. D. Bond, F. Qin, and A. Rountev, “LeakChaser: Helping programmers narrow down causes of memory leaks,” in *Programming Language Design and Implementation (PLDI)*, 2011, pp. 270–282 (see p. 63).

[416] J. Yang, “Explaining type errors by finding the source of a type conflict,” in *Trends in Functional Programming*, Intellect Books, 2000, ch. 7, pp. 58–66 (see p. 42).

[417] 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. 74).

[418] 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. 71).

[419] E. A. Youngs, “Human errors in programming,” *International Journal of Man-Machine Studies*, vol. 6, no. 3, pp. 361–376, 1974 (see p. 77).

[420] N. C. Zakas. (2018). ESLint 5, [Online]. Available: `http://eslint.org` (see p. 10).

[421] D. Zhang, A. C. Myers, D. Vytiniotis, and S. Peyton-Jones, “Diagnosing type errors with class,” in *Programming Language Design and Implementation (PLDI)*, 2015, pp. 12–21 (see pp. 41, 64).

[422] Q. Zhang, C. Sun, and Z. Su, “Skeletal program enumeration for rigorous compiler testing,” in *Programming Language Design and Implementation (PLDI)*, 2017, pp. 347–361 (see p. 62).

[423] X. Zhang, N. Gupta, and R. Gupta, “Pruning dynamic slices with confidence,” in *Programming Language Design and Implementation (PLDI)*, 2006, pp. 169–180 (see p. 65).

APPENDICES

A | Study Materials for "Do Developers Read Compiler Error Messages?" (Chapter 6)

A.1 Interview Protocol

A.1.1 Outline

1. Pre-participant steps.
2. Initial steps (namely, calibration).
3. Tasks (5 minutes x 10 tasks = 50 minutes).
4. Post-questionnaire.

A.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.

A.1.3 Arrival Steps

1. Have participant sign consent form.

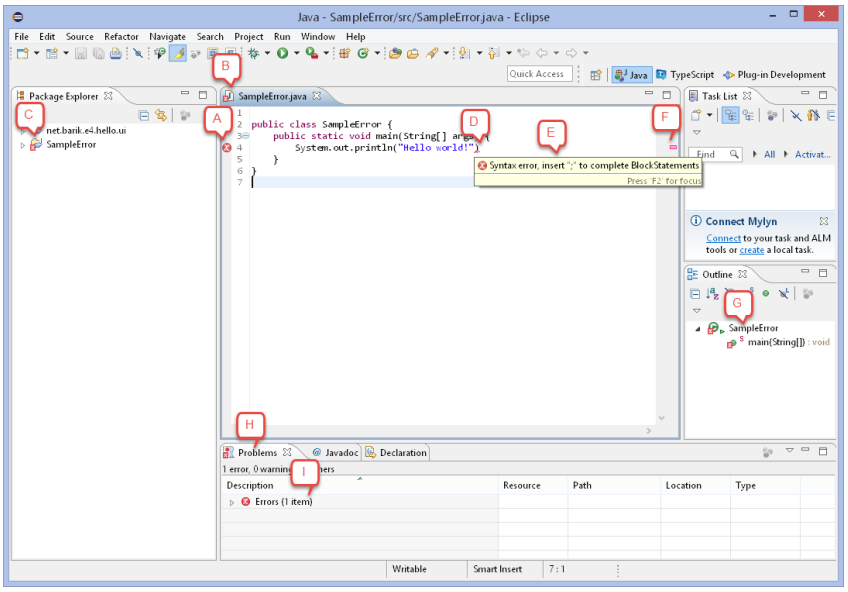


Figure A.1 Error messages sheet provided to participants to familiarize them with all notification sources in the Eclipse IDE.

2. Have participant sit in chair and get comfy. Hand them training picture and tell them these are all the places an error message 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.

A.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.

[Show error messages sheet to user] (Figure A.1)

Do you have any questions at this time?

A.1.5 Post-study Questionnaire

Give participant the questionnaire after they have completed the tasks.

A.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?

A.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`

A.2.1 Task 1: SUBLIST

A.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 A.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 A.2 Partial source listing for `LazyList.java` (IDE).

A.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. `/org/apache/commons/collections4/list`
Location. `line 136`
Type. `Java Problem`

A.2.2 Task 2: NODECACHE

A.2.2.1 Source Listings

```
@@ -107,4 +106,0 @@
-    public boolean isEmpty() {
-        return size() == 0;
-    }
-
```

Listing A.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 A.4 Partial source listing for `CursorableLinkedList.java` (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 A.5 Partial source listing for `NodeCachingLinkedList.java` (IDE).

A.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. `/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. /org/apache/commons/collections4/list
Location. line 42
Type. Java Problem

A.2.3 Task 3: `IMPORT`

A.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 A.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 A.7 Partial source listing for `AbstractLinkedMap.java` (IDE).

A.2.3.2 Error Messages (48 Items)

Description. `The import org.apache.commons.collections3 cannot be resolved`
Resource. `AbstractLinkedMap.java`
Path. `/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. /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. /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. /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. /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. /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.)

A.2.4 Task 4: QUEUEGET

A.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 A.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 A.9 Partial source listing for `PredicatedQueue.java` (IDE).

A.2.4.2 Error Message

Description. `The method get() is undefined for the type Queue<E>`
Resource. `PredicatedQueue.java`
Path. `/org/apache/commons/collections4/queue`
Location. `line 111`
Type. `Java Problem`

A.2.5 Task 5: SETADD

A.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 A.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 A.11 Partial source listing for `TransformedSet.java` (IDE).

A.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. `/org/apache/commons/collections4/set`
Location. `line 83`
Type. `Java Problem`

A.2.6 Task 6: KEYSETKV

A.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 A.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 A.13 Partial source listing for `FixedSizeMap.java` (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. `/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. `/org/apache/commons/collections4/map`
Location. `line 147`
Type. `Java Problem`

A.2.7 Task 7: CLAZZ

A.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 A.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 A.15 Partial source listing for `MultiValueMap.java` (IDE).

A.2.7.2 Error Message

Description. `Unhandled exception type InstantiationException`
Resource. `MultiValueMap.java`
Path. `/org/apache/commons/collections4/map`
Location. `line 552`
Type. `Java Problem`

A.2.8 Task 8: NEXT

A.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 A.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 A.17 Partial source listing for `EntrySetMapIterator.java` (IDE).

A.2.8.2 Error Message

Description. Duplicate method next() in type EntrySetMapIterator<K,V>
Resource. EntrySetMapIterator.java
Path. /org/apache/commons/collections4/iterators
Location. line 74
Type. Java Problem

Description. Duplicate method next() in type EntrySetMapIterator<K,V>
Resource. EntrySetMapIterator.java
Path. /org/apache/commons/collections4/iterators
Location. line 162
Type. Java Problem

A.2.9 Task 9: READOBJSTATIC

A.2.9.1 Source Listing

```
@@ -94,3 +94,3 @@
     @SuppressWarnings("unchecked")
-    private void readObject(final ObjectInputStream in)
+    private static void readObject(final ObjectInputStream in)
                 throws IOException, ClassNotFoundException {
```

Listing A.18 Injected fault in `UnmodifiableQueue.java`.

```
94      @SuppressWarnings("unchecked")
95      private static void readObject(final ObjectInputStream in)
96                  throws IOException, ClassNotFoundException {
97          in.defaultReadObject();
98          setCollection((Collection<E>) in.readObject());
99      }
```

Listing A.19 Partial source listing for `UnmodifiableQueue.java` (IDE).

A.2.9.2 Error Message

Description. `Cannot make a static reference to the non-static type E`
Resource. `UnmodifiableQueue.java`
Path. `/org/apache/commons/collections4/queue`
Location. `line 97`
Type. `Java Problem`

A.2.10 Task 10: SWITCH

A.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 A.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 A.21 Partial source listing for `Flat3Map.java` (IDE).

A.2.10.2 Error Message

Description. `Syntax error on token "default", : expected after this token`
Resource. `Flat3Map.java`
Path. `/org/apache/commons/collections4/map`
Location. `line 1242`
Type. `Java Problem`

A.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

B | Study Materials for "How Do Developers Visualize Compiler Error Messages?" (Chapter 7)

B.1 Interview Protocol

B.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.

B.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.

B.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.

B.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.

B.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.

B.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?

B.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
```

B.3 Visual Markings Cheat Sheet

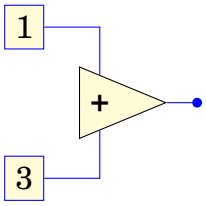
Symbol	Description
code	The starting location of the error.
code	Indicates issues related to the error.
	Arrows can be followed. They indicate the next relevant location to check.
❶	Enumerations are used to number items of potential interest, especially when the information doesn't fit within the source 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.

B.4 Dimensions Survey for All 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
```

B.5 Tasks

This section presents the six tasks. Participants were provided with a source listing and the compiler output for that listing. Participants in the control group received by the baseline version of the task with wavy underline annotations on the source; participants in the treatment group received explanatory annotations on the source code.

All participants received an annotation sheet containing the source code and compiler output. The source code did not have any preexisting annotations, and the participants were instructed to mark their own annotations using colored pencils. In addition, the annotation sheets contained the following 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

B.5.1 Task: Apple

B.5.1.1 Baseline

```
1 class Apple {
2     public public String toString() {
3       return "Red";
4     }
5 }
```

B.5.1.2 Explanatory

```
1    class Apple {
2        public public String toString() {
3            return "Red";
4        }
5    }
```

B.5.1.3 Compiler Output

```
Apple.java:2: error: repeated modifier
    public public String toString() {
           ^
1 error
```

B.5.2 Task: Brick

B.5.2.1 Baseline

```
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  }
```

B.5.2.2 Explanatory

```
1     class Brick {
2     ❶ void m(int i, double d) { }
3     ❷ void m(double d, int m) { }
4
5        {
6           m(1, 2);      ❶ m((int) 1, (double) 2);
7        }                ❷ m((double) 1, (int) 2);
8     }
```

B.5.2.3 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
```

B.5.3 Task: Kite

B.5.3.1 Baseline

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

B.5.3.2 Explanatory

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

B.5.3.3 Compiler Output

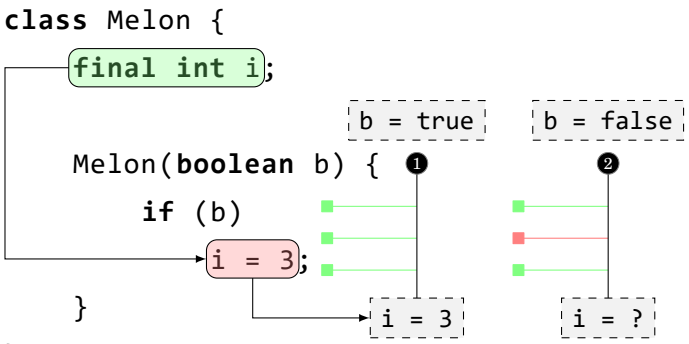
```
Kite.java:5: error: unreported exception Exception in default constructor
class Kite extends Toy {
^
1 error
```

B.5.4 Task: Melon

B.5.4.1 Baseline

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

B.5.4.2 Explanatory



B.5.4.3 Compiler Output

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

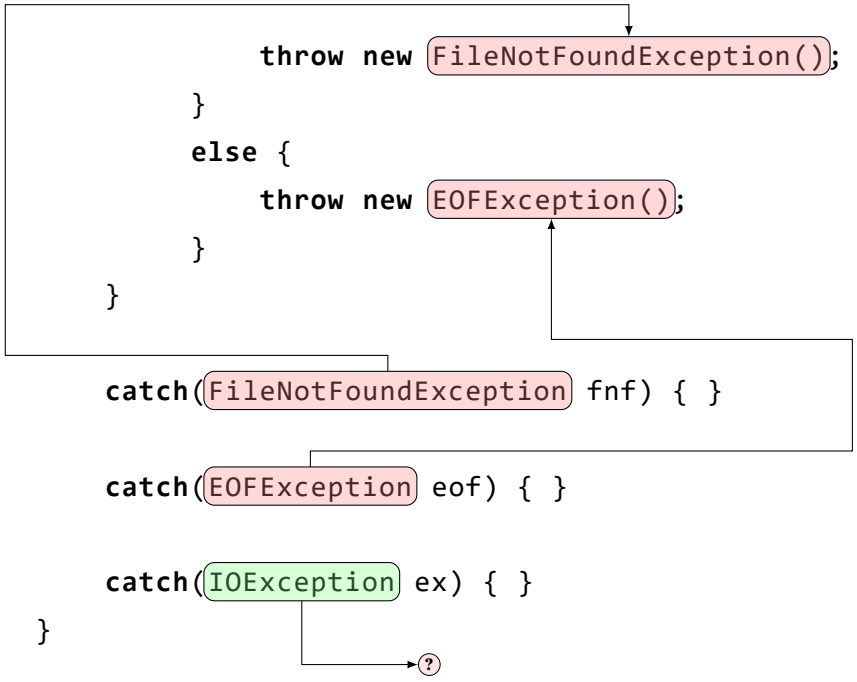
B.5.5 Task: Trumpet

B.5.5.1 Baseline

```
import java.io.*;

class Trumpet {

    void play() {
        try {
            if (true) {
                throw new FileNotFoundException();
            }
            else {
                throw new EOFException();
            }
        }
        catch(FileNotFoundException fnf) { }
        catch(EOFException eof) { }
        catch(IOException ex) { }
    }
}
```

B.5.5.2 Explanatory

```
 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    }
```



B.5.5.3 Compiler Output

```
Trumpet.java:16: warning: unreachable catch clause
        catch(IOException ex) { }
        ^
  thrown types FileNotFoundException,EOFException have already been caught
1 warning
```

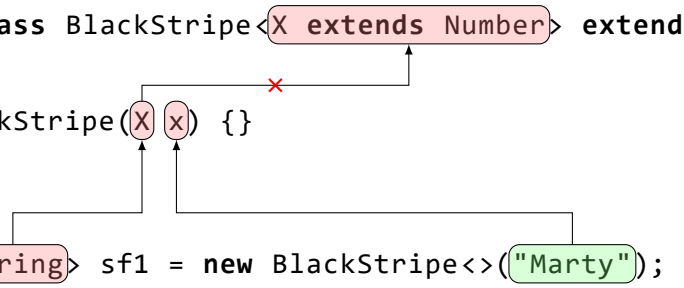
B.5.6 Task: Zebra

B.5.6.1 Baseline

```
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 }
```

B.5.6.2 Explanatory

```
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    }
```



B.5.6.3 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
```

C | Study Materials for “How Should Compilers Explain Problems to Developers?” (Chapter 8)

We conducted a comparative evaluation through an online survey instrument.

C.1 Survey

C.1.1 Demographic Information

How many years of professional programming experience do you have?

_____

How proficient are you with the following programming languages? (Not proficient, Somewhat proficient, Proficient, Very proficient)

```
C
C#
C++
Java
Python
Objective-C
Swift
```

The following questions will ask you to compare two error messages for a trivial code snippet that generates the error. There are five code snippets in total. We've removed the extraneous parts of the error message to allow you focus on the message text for the error. The error messages are for Java, but it's okay if you aren't an expert on Java programming.

(The errors and options are presented to the developer in randomized order.)

C.1.2 Error Message: E1

Given the following code snippet:

```
class Foo {
  Foo() {
    final int x;

    for (int i = 0; i < 5; ++i) {
      x = i;
    }
  }
}
```

Which error message would you prefer to see from your compiler?

1. `Variable x might be assigned in loop.`
2. `The blank final variable "x" cannot be assigned within the body of a loop that may execute more than once.`

C.1.3 Error Message: E2

Given the following code snippet:

```
class Foo {
    char variable;
    int varname;

    void foo() {
        System.out.println(varnam);
    }
}
```

Which error message would you prefer to see from your compiler?

1. `cannot find symbol`
   `symbol:   variable varnam`
   `location: class Foo`

2. No field named "varnam" was found in type "Foo". However, there is an accessible field "varname" whose name closely matches the name "varnam".

C.1.4 Error Message: E3

Given the following code snippet:

```
class Foo {
        static void f() {
        }

        void bar() {
                Foo f = new Foo();
                f.f();
        }
}
```

Which error message would you prefer to see from your compiler?

1. static method should be qualified by type name, Foo, instead of by an expression
2. 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.

C.1.5 Error Message: E4

Given the following code snippet:

```
class A {
  void remove(int x) { }

  class B {
    void remove() {
      remove(3);
    }
  }
}
```

Which error message would you prefer to see from your compiler?

1. 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
2. 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.

C.1.6 Error Message: E5

Given the following code snippet:

```
class A {
  class B {
    static String s;
  }
}
```

Which error message would you prefer to see from your compiler?

1. `Illegal static declaration in inner class A.B. Modifier 'static' is only allowed in constant variable declarations.`
2. `This static variable declaration is invalid, because it is not final, but is enclosed in an inner class, "B".`

C.1.7 Stack Overflow

Almost done. Just a few questions on how you use Stack Overflow.
How often do you turn to Stack Overflow for help when you encounter a confusing compiler error message?

```
( ) Never
( ) Rarely
( ) Sometimes
( ) Often
( ) Always or nearly always
```

How often do the Stack Overflow answers help you resolve the compiler error message?

```
( ) Never
( ) Rarely
( ) Sometimes
( ) Often
( ) Always or nearly always
```

D | Guidelines for Designing Error Messages

Table D.1 Summary of Design Guidelines for 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 Specific Localize the problem Complete Readable Restrained and polite

Reference	Guidelines
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.
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

Reference	Guidelines
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 The errant data are identified Corrective action is prescribed
Traver (2010)	Clarity & brevity Specificity Context-insensitivity Locality Proper phrasing: Positive tone Constructive guidance Programmer language

Reference	Guidelines
Murphy-Hill and Black (2012)	Expressiveness Locatability Completeness Estimability Relationality Perceptibility Distinguishability
Murphy-Hill, Barik, and Black (2013)	Restraint Relationality Partiality Nondistracting Estimability Availability Unobtrusiveness Context-sensitivity Lucidity

E | Rational TypeScript

The following source code listing implements the duplicate function implementation error, TS2393, by extending the TypeScript Compiler API:

```
import 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: "",
        positions: [],
        snippets: [],
    };
```

```
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          displayDiagram(duplicateFunctionError);
74      }
75  }
76
77  function displayDiagram(e: IErrorReconstruction): void {
78      const first = e.lineAndCharacters[0];
79      const second = e.lineAndCharacters[1];
80
81      // Lines are zero-indexed, but presented to user as one-indexed.
82      const firstLineLength = (first.line + 1).toString().length;
83      const secondLineLength = (second.line + 1).toString().length;
84      const linePad = secondLineLength + 1;
85
```

```
    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)}| ${"
    ↪  ".repeat(first.character)}`)
        + chalk.redBright(`${"-".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: `) +
        `\`${e.name}\` must be implemented only once within the same
        ↪  namespace`);

    log(chalk.blueBright(`${" ".repeat(linePad)}=`) +
        chalk.whiteBright(` hint: `) +
        `To overload a function, see https://www.typescriptlang
        ↪  .org/docs/handbook/functions.html`);
}

const argv = minimist(process.argv.slice(2), { string: "display" });

compile(argv._, {
```

```
129       module: ts.ModuleKind.CommonJS,
130       noEmitOnError: true,
131       noImplicitAny: true,
132       target: ts.ScriptTarget.ES5,
133   }, argv.display);
```

The above implementation indirectly loads the `typescript` package through `byots` [374]. `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.