How Should Compilers Explain Problems to Developers?

Titus Barik
Microsoft
Redmond, WA, USA
titus.barik@microsoft.com

Denae Ford
NC State University
Raleigh, NC, USA
dford3@ncsu.edu

Emerson Murphy-Hill
NC State University
Raleigh, NC, USA
emerson@csc.ncsu.edu

Chris Parnin
NC State University
Raleigh, NC, USA
cjparnin@ncsu.edu

ABSTRACT

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

Our findings suggest that, given a pair of error messages, developers significantly prefer the error message that employs proper argument structure over a deficient argument structure when neither offers a resolution—but will accept a deficient argument structure if it provides a resolution to the problem. Human-authored explanations on Stack Overflow converge to one of the three argument structures: those that provide a resolution to the error, simple arguments, and extended arguments that provide additional evidence for the problem. Finally, we contribute three practical design principles to inform the design and evaluation of compiler error messages.

CCS CONCEPTS

• **Human-centered computing → Empirical studies in HCI**; • **Software and its engineering → Integrated and visual development environments**;

KEYWORDS

communication theory, compilers, debugging, error messages, explanations, Stack Overflow

ACM Reference Format:
Titus Barik, Denae Ford, Emerson Murphy-Hill, and Chris Parnin. 2018. How Should Compilers Explain Problems to Developers?. In *Proceedings of the 26th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '18), November 4–9, 2018, Lake Buena Vista, FL, USA.* ACM, New York, NY, USA, 11 pages.
https://doi.org/10.1145/3236024.3236040

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ESEC/FSE '18, November 4–9, 2018, Lake Buena Vista, FL, USA
© 2018 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-5573-5/18/11…$15.00
https://doi.org/10.1145/3236024.3236040

1 INTRODUCTION

Compilers primarily give feedback about problems to developers through the use of error messages.[1] Despite the intended utility of error messages, researchers and practitioners alike have described their output as "cryptic" [44], "difficult to resolve" [44], "not very helpful" [48], "appalling" [5], "unnatural" [6], and "basically impenetrable" [40].

While poor error messages are paralyzing for novices, even experienced developers have substantial difficulties when comprehending and resolving them. A study conducted at Google found that nearly 30% of builds fail due to a compiler error, and that the median resolution time for each error is 12 minutes [38]. Surprisingly, the costly errors that developers make are rather mundane, relating to basic issues such as dependencies, type mismatches, syntax, and semantic errors. Barik et al. [2] conducted an eye-tracking study with developers and found that they spent up to 25% of their task time on reading error messages. In addition, developers in a study by Johnson et al. [19] reported that error messages were often not useful because they did not adequately *explain* the problem.

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

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

and the resulting error message from the OpenJDK compiler:

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

Although the location of the message is reasonable, intuitively, this is a poor explanation. The problem isn't just that the variable `x` is being assigned 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?

```
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 5.1).

[1] Modern compilers allow developers to turn *warning* messages into error messages, for example, by applying a `-Werror` flag. Thus, in this paper we treat warning messages as identical to error messages.

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.

If compiler error messages are framed as explanations, then it follows that we can *apply theories of explanation to understand why some error messages are more effective than others*. To that end, this paper applies Toulmin's model of argumentation (Section 2)—a theory for the structure and content of messages in everyday discourse—to the design and evaluation of compiler error messages.

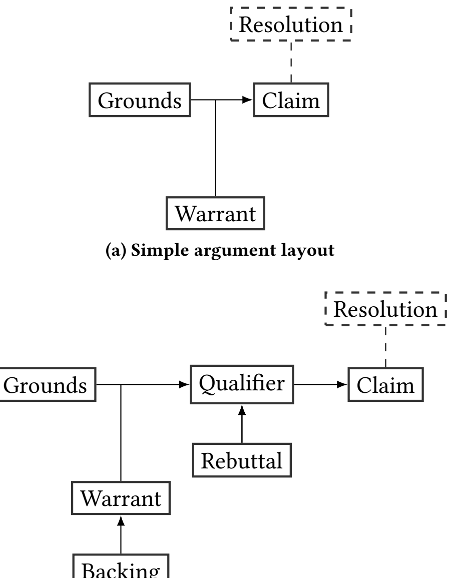
To understand if developers find explanatory error messages helpful, we conducted a comparative study between two compilers for the same programming language, and had experienced developers within a large software company indicate which message they would prefer in their compiler. Then, to understand why some error messages produced by compilers are less helpful than others, we conducted an empirical study through a popular question-and-answer site, Stack Overflow.[2] From Stack Overflow, we extracted 210 question-answer pairs posted by developers about compiler error messages, across seven different programming languages. For every question-answer pair, we qualitatively coded the compiler error message found within the question, and the accepted human-authored answer, through the theoretical model of argumentation. We characterized these question-answer pairs both in terms of the *structure* and *content* of their explanation. From this analysis, we can better understand the structure and content that compilers should use in explanations to developers.

The results of our studies provide support for presenting compiler error messages to developers as explanations. We find that: *1)* developers, when shown a pair of error messages, prefer the error message with a proper argument structure over the message with a deficient argument structure, but will prefer the deficient argument if it provides a *resolution* to the problem (Section 5.1); and *2)* human-authored explanations converge to argument structures that offer a simple resolution, or to structures with proper arguments (Section 5.2). They do so using a catalog of content within the structure (Section 5.3). From these results, we contribute three design principles for compiler authors to inform the design and evaluation of error messages (Section 8).

2 BACKGROUND ON EXPLANATIONS

Arguments are a form of justification-explanation in which reasons are used as evidence to support a conclusion [47]. Argumentation theory provides a lens through which we can evaluate the effectiveness of arguments [33, 46]. 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 [43]. Specifically, Toulmin's model of argument is a *macrostructure* model.

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



(a) Simple argument layout

(b) Extended argument layout

Figure 1: Toulmin's model of argument for (a) simple argument layout and (b) extended argument layout. Extended arguments add a rebuttal, backing, or qualifier to a simple argument.

Macrostructure examines how components combine to support the larger argument; within this macrostructure, *microstructure* examines the phrasing and composition of the "sentence-level" statements. For clarity, we will refer to macrostructure simply as *structure* (or layout) and microstructure as *content*.

In a simple argument layout (Figure 1a), 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 (with optional resolution), 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 within an argument (Figure 1b). In addition to the simple argument 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

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

needed to support the warrant. If *any* of these additional components are used in the argument, the argument becomes an extended argument. An example of how a compiler error message is mapped to an argument structure is illustrated in Figure 2; in this example, the error message is an extended argument because it has a backing.

3 METHODOLOGY

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 given a pair of messages where one of the messages provides a proper argument and the other message provides a deficient argument, developers should prefer the message with the proper argument. If this is not the case for the pair, then perhaps developers prefer the error message presentation because of 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 [19, 32]), 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.

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

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

Table 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".`

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 [8]. Though now discontinued, Jikes has been lauded by the developer community for giving "better error messages than the JDK compiler" [10].

Task selection. To select candidate error messages, the first author wrote 49 source code listings that induced an error message in both Jikes and OpenJDK. Although Jikes has 293 possible compiler errors, it is not possible to map all errors directly to OpenJDK because the compilers are not isomorphic. That is, source code accepted by one compiler may be rejected by the other.

From this set, the first author 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, the first author tagged each message using labels from Toulmin's model of argument: claim (and resolution), grounds, warrant, qualifier, rebuttal, and backing. 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 deliberately selected 5 OpenJDK and Jikes compiler error messages (Table 1) that address the same problem, but differ in argument structure. For each pair of error messages, we consulted with two compiler experts to form a hypothesis on how differences in argument structure would influence the results:

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 influenced by quality of the content, and not the argument structure. Thus, we expected participants to prefer Jikes.

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 a large software company 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 responses. The average reported experience of our participants was 6.3 years. 45 participants self-reported being proficient in Java, and 23 self-reported being proficient with some other OOP language (e.g, C#).

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 forced binary choice [12] for either the Jikes or OpenJDK version of the error message, alongside 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.

3.3 Phase II: Study Design for Stack Overflow

To answer RQ2 and RQ3, we conducted a study on Stack Overflow.

Research context. Previous research on Stack Overflow by Treude et al. [45] 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 [26]. Additionally, Stack Overflow provides an open-access API, through 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.

Data collection. We extracted all posts of type question or answer, tagged as "compiler-errors" or "compiler-warnings," yielding 11736 "compiler errors" and 1553 "compiler-warnings" (but not including "compiler errors", as some warnings are also tagged as errors). Because some systems allow developers to flag warnings as errors, we included these warnings in our set.

A subset of these 13289 questions links to an associated *accepted answer*, which in this paper we term *question-answer pairs.* An accepted answer is an answer marked by the original questioner as being satisfactory in resolving or addressing their original question.

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

Table 2: Compiler Errors and Warnings Count by Tag

	Question Count[1]				
Tag	Errors[2]	Warnings[3]	Total	% Acc.[4]	% Cov.[5]
C++	3508	421	3929	63%	40%
Java	2078	170	2248	55%	20%
C	1179	286	1465	61%	14%
C#	783	122	905	69%	10%
Obj. C	270	109	379	65%	4%
Swift	246	17	263	56%	2%
Python	211	4	215	53%	2%
Subtotals	8275	1129	9404	61%	92%
Totals	11736	1553	13289	58%	100%

[1] Questions may be counted more than once if they have multiple tags, for example, `C` and `C++`.
[2] Questions tagged as `compiler-errors`.
[3] Questions tagged with `compiler-warnings`, but not `compiler-errors`.
[4] Percentage of questions that have accepted answers.
[5] Coverage of tag over the 25 most popular languages from the Stack Overflow Developer Survey 2018.

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. 7741 (58%) of messages have accepted answers (Table 2).

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

Sampling strategy. To target diversity (rather than representativeness) in programming languages [29], we used stratified sampling across the top programming languages in our corpus, until we covered over 90% of the 25 most popular languages from the Stack Overflow Developer Survey 2018 [42]. This threshold was exceeded at Python (92%, Table 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," [15] such as through deliberately bogus questions.[4] 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.

[4]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/)

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.

During the coding process, we employed the technique of *negotiated agreement* as a means to address the reliability of coding [7]. 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 et al. [16]). 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 4) as well as a diagram of proper and deficient argument layouts (Figure 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 κ between the evaluators and against our negotiated agreement classifications: 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 [24], "good" by Bland and Altman [4], and "fair–good" by Fleiss et al. [14]). 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 extended arguments to simple arguments.

Supporting verifiability. If using a supported PDF reader, quotations from Stack Overflow are hyperlinked and can be clicked to take the reader to the corresponding Stack Overflow page.[5]

4 ANALYSIS

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 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.

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

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

To answer this research question, we applied a statistical permutation testing approach by Simpson et al. [41] that tests if pairs of compiler error messages between the group of Stack Overflow questions and the group of corresponding Stack Overflow accepted human-authored answers are significantly non-overlapping in argument structure. To translate these compiler errors and answers into a form suitable for statistical analysis, we represented them as an ordered set of argument components, E:

$$E = \langle a_1, a_2, \ldots, a_n \rangle \quad (1)$$

where $a_1, a_2, \ldots, a_n$ are labels for the seven argument components, such as grounds, warrants, backing, and so on. 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, the Jaccard index quantifies the similarity between the two sets; essentially, this index is the intersection over the union of the sets, ranging from 0 (no overlap between the sets) to 1 (perfect overlap between the sets).

Next, we perform the permutation testing procedure (PNF-J) fully described in Simpson et al. [41]. Essentially, the procedure computes the Jaccard index between every pair of error messages, within and between the two groups. The test statistic is the mean Jaccard ratio, defined as the mean Jaccard index *within* groups divided by the mean Jaccard index *between* groups. This statistic is recomputed after permuting the group labels such that the grouping is essentially at random, and this computation is repeatedly performed over some large number of trials. A p-value is obtained from the number of random trials that yield a larger statistic than the true grouping. If the p-value is small ($\alpha = 0.05$), then the two groups are significantly non-overlapping.

To characterize what types of argument structures are found in accepted answers, we used quasi-statistics [27]—essentially, a process of transforming qualitative data to simple counts—to aid in the interpretation of the Stack Overflow data. 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 2).

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

To further characterize the content, we performed an additional *purposive sampling*, or non-probabilistic sampling, on question-answer pairs from the entire Stack Overflow dataset and composed *memos* [3]. These memos, or author annotations on question-answer pairs, capture interesting exchanges or properties of the question-answer pairs, promote depth and credibility, and frame the posters' information needs and responses through their reported experiences. That is, the memos provide a *thick description* to contextualize the findings [35].

Table 3: OpenJDK and Jikes Error Message Preferences

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

[1] * indicates a statistically significant result.

5 RESULTS

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

From Table 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.

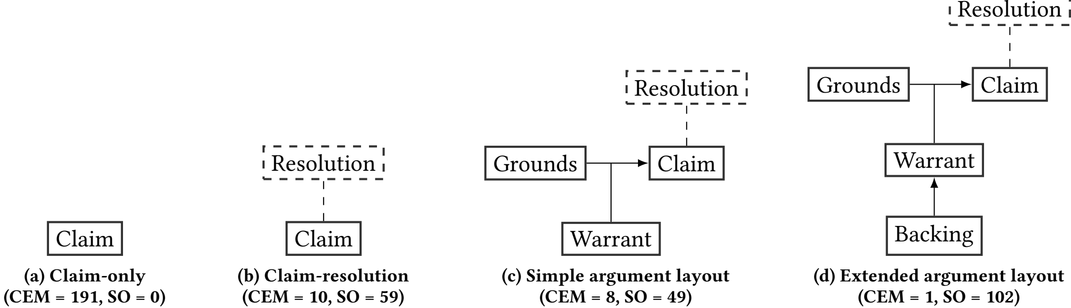
E1 *Deficient argument vs. simple argument.* As we expected, developers significantly preferred the simple argument from Jikes to the deficient argument in OpenJDK.

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.



(a) Claim-only (CEM = 191, SO = 0)

(b) Claim-resolution (CEM = 10, SO = 59)

(c) Simple argument layout (CEM = 8, SO = 49)

(d) Extended argument layout (CEM = 1, SO = 102)

Figure 3: Identified argument layouts for compiler error messages (CEM) and their associated Stack Overflow accepted answers (SO). Counts are indicated in parentheses. Problematic compiler error messages predominantly present a claim with no additional information, and occasionally present a resolution. In contrast, Stack Overflow accepted answers use proper extended argument and simple argument layouts, as well as the claim-resolution layout.

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

Permutation testing identified that the compiler error messages from the questions and the Stack Overflow accepted answers are significantly non-overlapping in terms of their argument structure (Jaccard ratio $R_j = 1.6$, for repeated iterations, $p = 0.008 \pm 0.001$).

Because the questioner asked a question about the compiler error message, this indicates some confusion with the error message 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 layout between the compiler error message and the accepted answer is significantly different, we can conclude that differences in the argument layout structure contributed to the acceptance of the Stack Overflow answer.

The identified argument layouts are found in Figure 3, for compiler error messages and for Stack Overflow accepted answers. In this quasi-statistical reporting, it becomes clear why the argument layouts for compiler errors and Stack Overflow accepted answers were found to be significantly different: problematic 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, the most frequent argument layout in Stack Overflow accepted answers are extended arguments (d), followed by claim-resolution (b) and simple argument (c) in comparable 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.

Table 4: Argument Layout Components for Error Messages

Attribute	Description
Simple Argument Components	
CLAIM (Section 5.3.1)	The claim is the concluding assertion or judgment about a problem in the code.
RESOLUTION (Section 5.3.2)	Resolutions suggest concrete actions to the source code to remediate the problem.
GROUNDS (Section 5.3.3)	Facts, rules, and evidence to support the claim.
WARRANT (Section 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 5.3.5)	Additional evidence to support the warrant, if the warrant is not accepted.
QUALIFIER (Section 5.3.6)	This is the degree of belief for a claim, often used to weaken a claim.
REBUTTAL (Section 5.3.7)	Exceptions to the claim or other components of the argument.

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 these argument components are presented in Table 4.

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" (A16686321) 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" (A16686321).

5.3.2 Resolution. A 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. An example of how answers provide resolutions is found in A8783031. 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) {
```

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" (A4114098), "`foo<T>` is a base class of `bar<T>`" (A27417378), and "local variables cannot have external linkage" (A5185870) 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 through the warrant *because*: "gets is deprecated *because* it's dangerous, it may cause a buffer overflow" (A26193030).

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 [46]. During our analysis, we would insert implicit "since" or "because" phrases during our reading of the error messages.

In some compilers, messages can bridge grounds with warrants through explicit concatenation, 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 demonstrates:

```
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.

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 [46]. For presenting error messages, we found that while grounds were typically natural language statements, backing was provided through the use of code examples (A51962187) and external resources. These resources include references to programming language specifications (A5005384), and occasionally, bug reports (A37835991) and tutorials (A2640747).

5.3.6 Qualifiers. Despite the usefulness of static analysis techniques for reporting compiler error messages to developers, many types of analysis are undecidable or computationally hard and necessitate the use of unsound simplifications [23]. Qualifiers include statements like "should" (A29189792), "likely" (A17980660), "try" (A7316556), and "probably" (A2841668, A7329198, A7942848). 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).

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 (A2858800). Another means of rebuttal occurs when the accepted answer provides reasons for *ignoring* a claim, as in A11180128. Here, the accepted answer suggests downgrading a ReSharper warning from a warning to a hint in order to not get "desensitized to their warnings, which are usually useful."

6 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 and limitations.

Comparative evaluation with Jikes and OpenJDK. We did not evaluate all combinations of the argument design space, for example, if developers prefer simple arguments against extended arguments. Thus, the error messages we asked participants to evaluate are not necessarily representative in terms of either difficulty or type of error message. Because we (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. Furthermore, the participants in this study were recruited from a single company, and processes and tools specific to this company may influence developers' expectations about error messages. The subsequent Stack Overflow study to some extent mitigates these threats, but does not eliminate them 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 error message comprehension [1]. Similarly, Flanagan et al. [13] use a diagrammatic representation on the source code to help developers understand code flow for an error. Other tools like Path Projection [21] and Theseus [25] use visual overlays on the source code, which are also 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, as typically found in terminal environments.

Generalizability. As a qualitative approach, our findings do not offer external validity in the traditional sense of nomothetic, sample-to-population, or *statistical generalization*. Rather, our findings are embedded within Stack Overflow and contextualized to how developers comprehend and resolve compiler error messages within these question-answer pairs. As one example of this limitation, 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* [34]. In analytic generalization, we generalize from individual statements within question-answer pairs to higher-order abstractions such as argumentation theory.

7 RELATED WORK

The work by Nasehi et al. [30] is the closest related work in terms of research approach and methodology. Nasehi and colleagues inspected Stack Overflow questions and accepted answers to identify components of good code examples: we use a similar methodology to understand how components of good compiler error messages correspond to structure and content drawn from argumentation theory.

Design criteria and guidelines. Several researchers have identified guidelines for compile errors. However, the history of design criteria for improving compiler error messages is both long and sometimes sordid; many of these guidelines are today considered to be pedestrian [5, 28].

Early work by Horning [17] suggested guidelines for the display of error messages, such as a "coordinate system" for relating the error message back to the source code listing, and revealing "memory addresses" relating to the error message. Shneiderman [39] focused less on the structural design of the error message and more on the holistic presentation, recommending that errors should have a positive tone, be specific using the developer's language, provide actionable information, and have a consistent, comprehensible format. In 1982, Dean [11] argued for design guidelines that emphasize humans goals and give people control over the messages they receive.

More recently, Traver [44] adapted criteria from Nielsen's heuristic evaluation [31] to compiler error messages, suggesting principles such as clarity, specificity, context-insensitivity, as well as previously-seen guidelines such as positive tone and matching the developers' language. Similarly, Sadowski et al. [36] present four design guidelines that inform when and how to incorporate new error messages into a static analysis platform, Tricoder. Like Kantorowitz and Laor [20], they suggest that program analysis tools minimize errors messages that are false positives.

Barriers to error message comprehension. Johnson et al. [19] conducted an interview study with developers to identify barriers to using static analysis tools. Their interviewers reported barriers such as "poorly presented" tool output and "false positives" as contributing to comprehension difficulties. One interviewee suggested that error messages be presented in some "distinct structure" to facilitate comprehension [19]. A follow-up study by Johnson et al. [18] identified that mismatches between developers' programming knowledge against information provided by the error message contribute to this confusion. A large-scale multi-method study at Microsoft identified several presentation "pain points," such as "bad warnings messages" and "bad visualization of warnings" [9].

Ko and Myers [22] found that many comprehension difficulties are due to programmers' false assumptions formed while trying to resolve errors [22]. Similarly, Lieber et al. [25] postulated that difficulties in resolving errors were due to faulty mental models, or misconceptions, that remained uncorrected until the developer manually requested information explicitly from their programming environment; they developed an always-on visualization in the IDE to proactively address misconceptions.

8 DESIGN PRINCIPLES

We synthesize and discuss our results through three design principles, which compiler authors can use to inform the design and evaluation of error messages:

Principle I—Allow developers the autonomy to elaborate arguments. From our comparative evaluation in Phase I, we found that with E3 developers preferred the OpenJDK version of the error message, despite the fact that the Jikes version is more explanatory (RQ1, Section 5.1). However, novice developers may still find the explanation from Jikes useful, and expert developers may still find the Jikes presentation useful if they want to understand the rationale for the fix. Thus, developers may selectively need more or less help in comprehending the problem, and we should support mechanisms to progressively elaborate error messages. The static analysis tool Error Prone implements such an approach; the tool initially provides a simple argument for the error messages, but also enables additional backing through a supporting link:

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

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

Principle II—Distinguish fixes from explanations. Accepted answers from Stack Overflow identified a dichotomy in argument structure: 1) claim-resolutions, which we can think of essentially as quick fixes that immediately *resolve the problem* for the developer, and 2) simple to extended arguments, which provide an *explanation* of the problem (Figure 3).

Both styles of argument structure are useful (RQ2, Section 5.2). A claim-resolution structure is appropriate when the resolution is obvious. For example, consider a C file with a missing semi-colon, as presented through the LLVM:

```
hello.c:4:26: error: expected ';' after expression
  printf("Hello, world!")
                        ^
                        ;
```

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

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

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

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

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

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

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

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

Now, it is apparent the actual claim is that `all elements of a list constructor expression must have the same type`, and the remainder of the error message is evidence to support that claim. There are also content differences (RQ3, Section 5.3): F# helpfully provides a code snippet that points to the position of the error in the result, whereas `ghci` indicates the location narratively through a series of `In the expression` statements.

9 CONCLUSION

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

ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation under Grant Nos. 1559593 and 1714538.

REFERENCES

[1] T. Barik, K. Lubick, S. Christie, and E. Murphy-Hill. 2014. How Developers Visualize Compiler Messages: A Foundational Approach to Notification Construction. In *IEEE Working Conference on Software Visualization (VISSOFT)*. 87–96. https://doi.org/10.1109/VISSOFT.2014.24

[2] Titus Barik, Justin Smith, Kevin Lubick, Elisabeth Holmes, Jing Feng, Emerson Murphy-Hill, and Chris Parnin. 2017. Do developers read compiler error messages?. In *International Conference on Software Engineering (ICSE)*. 575–585. https://doi.org/10.1109/ICSE.2017.59

[3] Melanie Birks, Ysanne Chapman, and Karen Francis. 2008. Memoing in qualitative research: Probing data and processes. *Journal of Research in Nursing* 13, 1 (Jan. 2008), 68–75. https://doi.org/10.1177/1744987107081254

[4] J Martin Bland and Douglas G Altman. 1986. Statistical methods for assessing agreement between two methods of clinical measurement. *The Lancet* 327, 8476 (1986), 307–310. https://doi.org/10.1016/S0140-6736(86)90837-8

[5] P. J. Brown. 1983. Error messages: The neglected area of the man/machine interface. *Commun. ACM* 26, 4 (April 1983), 246–249. https://doi.org/10.1145/2163.358083

[6] Joshua Charles Campbell, Abram Hindle, and José Nelson Amaral. 2014. Syntax errors just aren't natural: Improving error reporting with language models. In *Mining Software Repositories (MSR)*. 252–261. https://doi.org/10.1145/2597073.2597102

[7] John L. Campbell, Charles Quincy, Jordan Osserman, and Ove K. Pedersen. 2013. Coding in-depth semistructured interviews: Problems of unitization and intercoder reliability and agreement. *Sociological Methods & Research* 42, 3 (Aug. 2013), 294–320. https://doi.org/10.1177/0049124113500475

[8] Philippe Charles and David Shields. 1998. Frequently asked questions about Jikes. https://www.cc.gatech.edu/data_files/public/doc/jikesfaq.html

[9] Maria Christakis and Christian Bird. 2016. What developers want and need from program analysis: An empirical study. In *Automated Software Engineering (ASE)*. 332–343. https://doi.org/10.1145/2970276.2970347

[10] Ian F Darwin. 2004. *Java Cookbook*. O'Reilly.

[11] M. Dean. 1982. How a computer should talk to people. *IBM Systems Journal* 21, 4 (1982), 424–453. https://doi.org/10.1147/sj.214.0424

[12] Sara Dolnicar, Bettina Grün, and Friedrich Leisch. 2011. Quick, simple and reliable: Forced binary survey questions. *International Journal of Market Research* 53, 2 (2011), 231–252. https://doi.org/10.2501/IJMR-53-2-231-252

[13] Cormac Flanagan, Matthew Flatt, Shriram Krishnamurthi, Stephanie Weirich, and Matthias Felleisen. 1996. Catching bugs in the web of program invariants. In *Programming Language Design and Implementation (PLDI)*, Vol. 31. 23–32. https://doi.org/10.1145/231379.231387

[14] Joseph L Fleiss, Bruce Levin, and Myunghee Cho Paik. 2013. *Statistical Methods for Rates and Proportions*. John Wiley & Sons.

[15] Claire Hardaker. 2010. Trolling in asynchronous computer-mediated communication: From user discussions to academic definitions. *Journal of Politeness Research. Language, Behaviour, Culture* 6, 2 (Jan. 2010), 215–242. https://doi.org/10.1515/jplr.2010.011

[16] Michael Hilton, Nicholas Nelson, Danny Dig, Timothy Tunnell, and Darko Marinov. 2016. *Continuous integration (CI) needs and wishes for developers of proprietary code*. Technical Report. Oregon State University.

[17] James J. Horning. 1974. What the compiler should tell the user. In *Compiler Construction*. Vol. 21. Springer, 525–548. https://doi.org/10.1007/3-540-06958-5

[18] Brittany Johnson, Rahul Pandita, Justin Smith, Denae Ford, Sarah Elder, Emerson Murphy-Hill, Sarah Heckman, and Caitlin Sadowski. 2016. A cross-tool communication study on program analysis tool notifications. In *Foundations of Software Engineering (FSE)*. 73–84. https://doi.org/10.1145/2950290.2950304

[19] Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. 2013. Why don't software developers use static analysis tools to find bugs?. In *International Conference on Software Engineering (ICSE)*. 672–681. https://doi.org/10.1109/ICSE.2013.6606613

[20] E. Kantorowitz and H. Laor. 1986. Automatic generation of useful syntax error messages. *Software: Practice and Experience* 16, 7 (July 1986), 627–640. https://doi.org/10.1002/spe.4380160703

[21] Yit Phang Khoo, Jeffrey S. Foster, Michael Hicks, and Vibha Sazawal. 2008. Path projection for user-centered static analysis tools. In *Program Analysis for Software Tools and Engineering (PASTE)*. 57–63. https://doi.org/10.1145/1512475.1512488

[22] Andrew J. Ko and Brad A. Myers. 2003. Development and evaluation of a model of programming errors. In *Human Centric Computing Languages and Environments (HCC)*. 7–14. https://doi.org/10.1109/HCC.2003.1260196

[23] William Landi. 1992. Undecidability of static analysis. *ACM Letters on Programming Languages and Systems* 1, 4 (Dec. 1992), 323–337. https://doi.org/10.1145/161494.161501

[24] J. Richard Landis and Gary G. Koch. 1977. The measurement of observer agreement for categorical data. *Biometrics* 33, 1 (1977), 159–174. https://doi.org/10.2307/2529310

[25] Tom Lieber, Joel R. Brandt, and Rob C. Miller. 2014. Addressing misconceptions about code with always-on programming visualizations. In *Human Factors in Computing Systems (CHI)*. 2481–2490. https://doi.org/10.1145/2556288.2557409

[26] Lena Mamykina, Bella Manoim, Manas Mittal, George Hripcsak, and Björn Hartmann. 2011. Design lessons from the fastest Q&A site in the west. In *Human Factors in Computing Systems (CHI)*. 2857–2866. https://doi.org/10.1145/1978942.1979366

[27] Joseph A. Maxwell. 2010. Using numbers in qualitative research. *Qualitative Inquiry* 16, 6 (July 2010), 475–482. https://doi.org/10.1177/1077800410364740

[28] P. G. Moulton and M. E. Muller. 1967. DITRAN—a compiler emphasizing diagnostics. *Commun. ACM* 10, 1 (Jan. 1967), 45–52. https://doi.org/10.1145/363018.363060

[29] Meiyappan Nagappan, Thomas Zimmermann, and Christian Bird. 2013. Diversity in software engineering research. In *Foundations of Software Engineering (FSE)*. 466–476. https://doi.org/10.1145/2491411.2491415

[30] Seyed Mehdi Nasehi, Jonathan Sillito, Frank Maurer, and Chris Burns. 2012. What makes a good code example? A study of programming Q&A in Stack Overflow. In *International Conference on Software Maintenance (ICSM)*. 25–34. https://doi.org/10.1109/ICSM.2012.6405249

[31] Jakob Nielsen. 1994. Heuristic evaluation. In *Usability Inspection Methods*. John Wiley & Sons, 25–64.

[32] Marie-Hélène Nienaltowski, Michela Pedroni, and Bertrand Meyer. 2008. Compiler error messages: What can help novices?. In *ACM Technical Symposium on Computer Science Education (SIGCSE)*. 168–172. https://doi.org/10.1145/1352322.1352192

[33] Niels Pinkwart, Kevin Ashley, Collin Lynch, and Vincent Aleven. 2009. Evaluating an intelligent tutoring system for making legal arguments with hypotheticals. *International Journal of Artificial Intelligence in Education* 19, 4 (Dec. 2009), 401–424.

[34] Denise F. Polit and Cheryl Tatano Beck. 2010. Generalization in quantitative and qualitative research: Myths and strategies. *International Journal of Nursing Studies* 47, 11 (2010), 1451–1458. https://doi.org/10.1016/j.ijnurstu.2010.06.004

[35] Joseph Ponterotto. 2006. Brief note on the origins, evolution, and meaning of the qualitative research concept thick description. *The Qualitative Report* 11, 3 (2006), 538–549.

[36] Caitlin Sadowski, Jeffrey van Gogh, Ciera Jaspan, Emma Soderberg, and Collin Winter. 2015. Tricorder: Building a program analysis ecosystem. In *International Conference on Software Engineering (ICSE)*. 598–608. https://doi.org/10.1109/ICSE.2015.76

[37] Johnny Saldaña. 2009. *The Coding Manual for Qualitative Researchers*. SAGE Publications.

[38] Hyunmin Seo, Caitlin Sadowski, Sebastian Elbaum, Edward Aftandilian, and Robert Bowdidge. 2014. Programmers' build errors: A case study (at Google). In *International Conference on Software Engineering (ICSE)*. 724–734. https://doi.org/10.1145/2568225.2568255

[39] Ben Shneiderman. 1982. Designing computer system messages. *Commun. ACM* 25, 9 (Sept. 1982), 610–611. https://doi.org/10.1145/358628.358639

[40] Jeremy Siek and Andrew Lumsdaine. 2000. Concept checking: Binding parametric polymorphism in C++. In *Workshop on C++ Template Programming*.

[41] Sean L. Simpson, Robert G. Lyday, Satoru Hayasaka, Anthony P. Marsh, and Paul J. Laurienti. 2013. A permutation testing framework to compare groups of brain networks. *Frontiers in Computational Neuroscience* 7 (2013), 171:7–171:13. https://doi.org/10.3389/fncom.2013.00171

[42] Stack Overflow. 2018. Stack Overflow Developer Survey. https://insights.stackoverflow.com/survey/2018/

[43] Stephen Toulmin. 2003. *The Uses of Argument*. Cambridge University Press.

[44] V. Javier Traver. 2010. On compiler error messages: What they say and what they mean. *Advances in Human-Computer Interaction* 2010 (2010), 602570:1–602570:26. https://doi.org/10.1155/2010/602570

[45] Christoph Treude, Ohad Barzilay, and Margaret-Anne Storey. 2011. How do programmers ask and answer questions on the web?. In *International Conference on Software Engineering (ICSE)*. 804–807. https://doi.org/10.1145/1985793.1985907

[46] Frans H. van Eemeren, Bart Garssen, Erik C. W. Krabbe, A. Francisca Snoeck Henkemans, Bart Verheij, and Jean H. M. Wagemans. 2014. *Handbook of Argumentation Theory*. Springer Netherlands, Dordrecht. https://doi.org/10.1007/978-90-481-9473-5

[47] Douglas Walton. 2009. Explanations and arguments based on practical reasoning. In *International Joint Conferences on Artificial Intelligence (IJCAI)*. 72–83.

[48] Mitchell Wand. 1986. Finding the source of type errors. In *Principles of Programming Languages (POPL)*. 38–43. https://doi.org/10.1145/512644.512648