Compiler Error Notifications Revisited

An Interaction-First Approach for Helping Developers More Effectively Comprehend and Resolve Error Notifications

Titus Barik, Jim Witschey, Brittany Johnson, Emerson Murphy-Hill
Department of Computer Science
North Carolina State University, USA
{tbarik, jim_witschey, bijohnso}@ncsu.edu, emerson@csc.ncsu.edu

ABSTRACT

Error notifications and their resolutions, as presented by modern IDEs, are still cryptic and confusing to developers. We propose an interaction-first approach to help developers more effectively comprehend and resolve compiler error notifications through a conceptual interaction framework. We propose novel taxonomies that can serve as controlled vocabularies for compiler notifications and their resolutions. We use preliminary taxonomies to demonstrate, through a prototype IDE, how the taxonomies make notifications and their resolutions more consistent and unified.

Categories and Subject Descriptors

D.3.0 [**Programming Languages**]: General; D.2.6 [**Software Engineering**]: Programming Environments; D.2.10 [**Software Engineering**]: Design

General Terms

Design, Human Factors, Languages

Keywords

Compiler error messages, IDE, visualization, taxonomy

1. INTRODUCTION

Programming is a cognitively demanding task [17]. One demanding subtask is the cycle of comprehending feedback as presented through compiler error notifications and articulating appropriate resolutions for each. In our experience, it seems nearly every developer has stories about impenetrable error notifications that caused them consternation. Unfortunately, today's integrated development environments (IDEs) emit cryptic and confusing messages, leaving their resolutions equally elusive, even for experts [16, 13].

When the IDE was itself a new idea, researchers recognized its potential to present and resolve compiler messages in innovative ways, by taking advantage of graphics and multiple windows [3]. Consider *quick fixes*, which augment text-based notifications by offering candidate resolutions. While more useful than text notifications alone, even quick fixes do not fully realize the potential of the IDE. Quick fixes only offer one mechanism for all errors, one that may not be appropriate for every error.

This leads us to ask: How should error notifications be presented, and for what errors? Does a single notification style work for all errors? Should every error have a unique notification, or is the best solution somewhere in between, where some errors should be presented in the same way? Do the resolutions offered by these tools actually align with the way developers articulate them? We suggest a principled approach to understanding this space will help us develop more effective tools.

We propose two taxonomies that formalize an interaction framework from human-computer interaction (HCI) research. These taxonomies model notifications and resolutions and can be used as a controlled vocabulary, which can be reasoned about both cognitively and computationally. Though preliminary, we believe this research is useful and will provide tool developers with new tools to design consistent, unified presentations of notifications and their resolutions. We show how abstract representations of notifications can be *computationalized*, or expressed in a form that machines can interpret, and how this representation supports tools that help developers comprehend and resolve error notifications. To demonstrate how tools can implement our formalism, we introduce a language-agnostic prototype system in which new visualizations, as well as quick fixes and other resolution strategies, can be implemented.

2. RELATED WORK

Though researchers have previously identified the problems developers encounter when interpreting and resolving error notifications [2, 8, 18], few provide concrete solutions. As early as the 1960s, researchers designed systems to support developer comprehension of compiler error messages [9, 12, 7], but much of this work has focused on text-only solutions that do not take advantage of IDEs. Other researchers have written guidelines for the development of error notifications [11, 15, 14], but these guidelines cannot be directly understood by machines. There has also been recent research on systems to help developers comprehend and resolve errors [10, 6, 13]. In particular, Hartmann and colleagues use a social recommender system which provides suggestions to developers that

ICSE '14, May 31 - June 7, 2014, Hyderabad, India

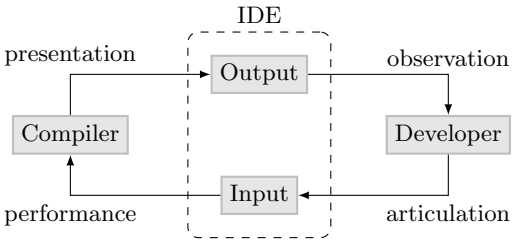


Figure 1: The interaction framework, instantiated for IDEs.

cognitively align with their comprehension and resolution process [6]. However, a limitation of this system is that the suggestions require a corpus of human-generated fixes.

3. OUR APPROACH

We describe Abowd and Beale's interaction framework [1], how it models the interaction between developers and IDEs (Section 3.1), and through which we identify areas for improvement. We propose two cognitively and computationally expressive taxonomies for building tools that allow developers to more effectively comprehend and resolve error notifications (Section 3.2).

3.1 First Principles: Interaction Framework

Traver considered the difficulty of error message comprehension and resolution from an HCI perspective [16]. This perspective offers a useful insight: Traver describes modern compiler use, as enabled by IDEs, as a specialized instantiation of the Abowd and Beale's interaction framework [1]. However, conventional tools are modeled by the framework only incidentally, and not by design. We believe designing tools for interaction in a principled way will help us make better tools. We briefly describe here the framework (Figure 1) and its applicability to IDEs.

This framework comprises four components: a compiler, a developer, an input, and an output. The input and output constitute the interface, which, for developers, is the IDE. For example, imagine a compiler has identified an error in some C# code. The compiler *presents* this error to the developer through the IDE, and the IDE can augment this presentation by, e.g., visually underlining the offending code. This notification must then be *observed* and comprehended by the developer. The developer must then *articulate* a resolution through the IDE, either as a manual edit to the code or through an automated tool in the IDE. The IDE then invokes the compiler, which *performs* the compilation process, and the cycle repeats. Abowd and Beale call these four steps *translations*. This framework exposes interdependence between the translations: if the developer observes a cryptic message and misinterprets it, they are more likely to articulate an incorrect resolution.

3.2 Formalizing Translations: Taxonomies

While all four translations in the interaction cycle are important, we think research on some of these translations have made greater computational progress than others. In our opinion, the translations that have made less progress with regards to improving error notifications are presentation and articulation. As with other translations, there are likely many useful formalizations. We chose taxonomies from knowledge management theory as a starting point for designing our formalization, because they help people retrieve, manage, and improve complex problem spaces [5]. We intend to develop a *notification taxonomy* describing information content to include in the presentation of errors, and a *resolution taxonomy* describing how programmers articulate error resolutions to the compiler.

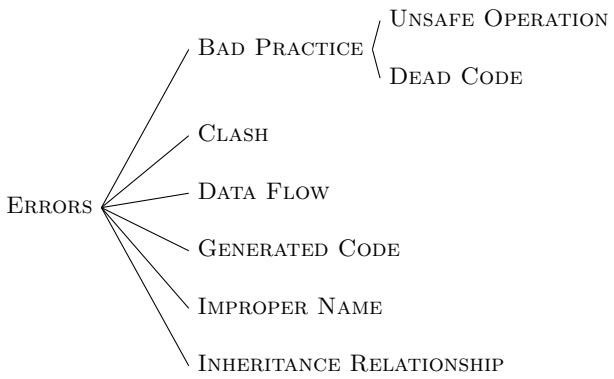


Figure 2: A partial taxonomy for categorizing notifications by presentation concerns.

Presentation: A Notification Taxonomy.

We have conducted preliminary research on the first taxonomy by randomly sampling and categorizing roughly 40% of the 500 possible OpenJDK compiler error notifications. The notifications are categorized based on information needed for developer comprehension. We categorized error notifications in a way that identifies the important information that would help a developer understand the underlying error irrespective of any potential resolution strategies. A subset of this taxonomy, which is still in preliminary stages of development, is shown in Figure 2.

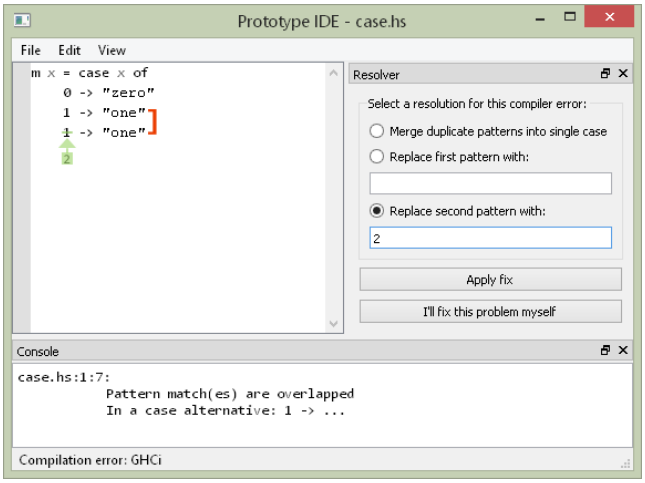
One category in this taxonomy is a Clash, which informs a developer that two elements cannot coexist in the program. As a result, a notification for a Clash should inform the developer which two program elements are in conflict. For instance, two local variables declared in the same scope in Java with the same name would cause a Clash.

Articulation: A Resolution Taxonomy.

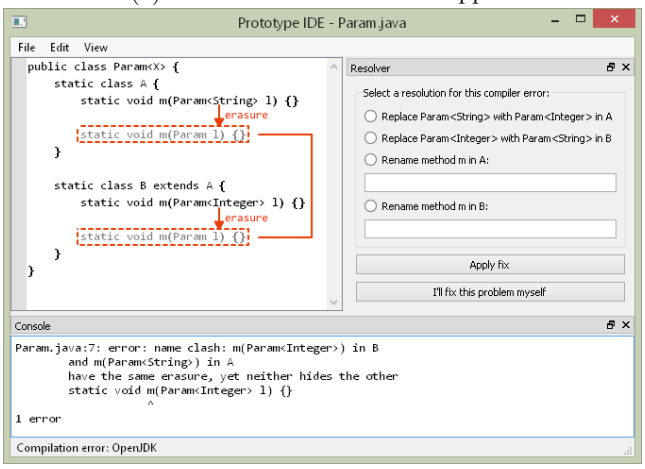
This taxonomy describes strategies developers use to articulate resolutions to errors. By observing and capturing developer resolution strategies, we can potentially generate interfaces which help developers articulate resolutions more naturally and at the appropriate level of abstraction. Table 1 shows a subset of these taxonomy elements as *tasks*, and how they might be used as GUI widgets or operations. The ChooseOneOf task occurs because a developer must delete all but one of a set of elements from a program, as when a method has been inadvertently assigned both `public` and `private` qualifiers. In this case, the compiler can populate the ChooseOneOf task with the two qualifiers as arguments. This task may be offered as a resolution through an error notification during presentation. Future empirical studies will evaluate the appropriateness of these tasks.

Table 1: A Partial Taxonomy of Developer Resolution Tasks

Semantic Task	Description	GUI Example
ChooseOneOf$(X, Y, \ldots)$	Chooses an argument from the provided arguments.	Dropdown
Merge$(X, Y, \ldots)$	Merges a set of identical arguments.	Radio button
Remove(X)	Removes a subtree from the source, e.g., dead code.	Radio button
Replace(X, Y)	Replaces the first argument with the second.	Radio button or text field
Move(X)	Moves the argument to a different location in the code.	Drag and drop



(a) Pattern matches are overlapped



(b) Name clash: Methods have same erasure

Figure 3: A prototype IDE for notifications and resolutions. The prototype leverages the notification and resolution taxonomies to reuse visualization components. The resolver is a single component, and generates appropriate resolutions using the resolution taxonomy. The text with a red, dashed border is generated code added by the system to help explain the error.

4. EMERGING RESULTS

We present a prototype IDE, shown in Figure 3. The prototype consists of a text editor pane that can be augmented with visualizations, a resolver pane that the IDE can use to present candidate resolutions, and a traditional console pane containing the raw text error message. Internally, the prototype leverages the presentation and articulation taxonomies by passing notifications as *error objects*. For this paper, it is sufficient to think of these objects as elements from the taxonomies augmented with additional semantic information such as line number, location, or other relevant information for use by the IDE. Here, we demonstrate the use of these taxonomies with two examples.

The IDE in Figure 3a presents an overlapping pattern error in Haskell, where the first '`1`' pattern overlaps the second '`1`' completely, such that the second '`1`' case can never execute. This error is presented to the IDE as a Clash error object (Figure 2). It displays the clash between the conflicting cases with a red bracket. The error object also contains resolution tasks (Table 1) for the developer to articulate: Merge, Replace (for the first conflicting pattern), and Replace (for the second conflicting pattern). With this information, the IDE presents a set of resolution tasks, with graphical widgets (radio buttons, text fields, and so on) as appropriate for articulating each task.

In this example, the developer has chosen the third resolution, so the IDE also displays, in green, the effect of articulating a Replace task: a strike-through for the case that will change, along with boxed text indicating what the case will be changed to.

A second, more complex Java example, is shown in Figure 3b. This error is tricky in that it results from *generics*-related type information that is available at compilation time but not at runtime – this is called *type erasure*. Roughly, during the compilation process, parameterized classes are turned into raw classes – for example, `Param<Integer>` and `Param<String>` will both resemble `Param`, at which point, their signatures would be the same (or `have the same erasure`, according to OpenJDK) and the runtime would not be able to tell them apart because both are still available (`neither hides the other`).

At first glance this notification may appear to be a completely different error than that in Figure 3a. However, if we consider it from a user-centric perspective, we can reduce it to a Clash error – a clash between method signatures after type erasure. Because the effect of the error is not apparent in the source code, but is manifested in the compiled code, we also consider it a Generated Code notification.

The error text provided by OpenJDK describes this error, but in a cryptic way. Our visualization is clearer because it directly shows the effect of the type erasure using red arrows labeled `erasure` and inserts a representation of the generated

code, which is displayed in a box with a red, dashed border to differentiate it from the developer's code. The IDE also displays red brackets, as before, to show that the methods are in conflict after erasure. Without this visualization, a developer would have thought through the erasure step of compilation and come to the same conclusion, but our visualization reduces the cognitive burden by performing this reasoning for them.

Our taxonomy helped us recognize the semantic similarity between these two notifications. Then, we used it to represent them both computationally. Our visualization can reuse the same infrastructure for the CLASH semantics of the errors, despite being different errors in different languages. In addition, our example demonstrates the composability of the notification semantics, which allows Figure 3b to augment the CLASH visualization with an explanation that the CLASH occurs in GENERATED CODE.

This demonstrates the immediate benefit of using these taxonomies – they give consistent and unified semantics to notifications and resolutions. This, in turn, allows IDE developers to add presentation and articulation features to their tools for minimal incremental cost.

5. CHALLENGES

The discovery of tasks within the resolution taxonomy may reveal non-reusable tasks that are only applicable to a particular error notification. If special cases occur frequently, then it will negate the advantages of unification that taxonomies would otherwise provide. We must make design tradeoffs in the number of categories. A small number of highly abstract categories allows for greater consistency between notifications, but at the cost of detailed information about individual errors. It is also possible that some elements of these taxonomies cannot be computationalized. For example, consider possible BAD PRACTICE notifications relating to "code smells", which are not necessarily errors, but may indicate general flaws [4]. Resolving code smells is often wholly subjective, preventing any computational solution. So far, we have only used the OpenJDK in creating our taxonomies, though we intend to incorporate more languages in the future. Though we expect our taxonomies to capture a broad range of languages, new language features may require revisions to the taxonomies. A final challenge of our approach is that its effectiveness is constrained by the accuracy of error diagnostics provided by the compiler.

6. CONCLUSIONS

We think our taxonomies will provide developers and compilers with controlled and expressive vocabularies with which to communicate about errors and their resolutions. The taxonomies give consistent and unified semantics to error objects, which in turn allows IDE developers to easily add presentation and articulation features to their tools. More importantly, the taxonomies allow IDE developers to also design presentation and articulation of notifications in a consistent and unified way. In doing so, tools can offer error notifications and resolutions that align more closely with the way in which developers observe and resolve notifications in their programming activities.

7. ACKNOWLEDGMENTS

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

8. REFERENCES

[1] G. D. Abowd and R. Beale. Users, systems and interfaces: A unifying framework for interaction. In *People and Computers VI*, pages 73–87, 1991.

[2] P. J. Brown. 'My system gives excellent error messages'–or does it? *Software: Practice and Experience*, 12(1):91–94, Jan. 1982.

[3] P. J. Brown. Error messages: The neglected area of the man/machine interface. *Communications of the ACM*, 26(4):246–249, Apr. 1983.

[4] M. Fowler. *Refactoring: Improving the Design of Existing Code*. 1999.

[5] M. Frické. *Logic and the Organization of Information*. Springer New York, New York, NY, 2012.

[6] B. Hartmann, D. MacDougall, J. Brandt, and S. R. Klemmer. What would other programmers do. In *CHI '10*, page 1019, Apr. 2010.

[7] M. Hristova, A. Misra, M. Rutter, and R. Mercuri. Identifying and correcting Java programming errors for introductory computer science students. *ACM SIGCSE Bulletin*, 35(1):153, Jan. 2003.

[8] J. Jackson, M. Cobb, and C. Carver. Identifying top Java errors for novice programmers. In *FIE '05'*, pages 24–27, 2005.

[9] C. Litecky. An expert system for Cobol program debugging. *ACM SIGMIS Database*, 20(1):1–6, Apr. 1989.

[10] G. Marceau, K. Fisler, and S. Krishnamurthi. Mind your language: On novices' interactions with error messages. In *ONWARD '11*, page 3, Oct. 2011.

[11] R. Molich and J. Nielsen. Improving a human-computer dialogue. *Communications of the ACM*, 33(3):338–348, Mar. 1990.

[12] P. G. Moulton and M. E. Muller. DITRAN—a compiler emphasizing diagnostics. *Communications of the ACM*, 10(1):45–52, Jan. 1967.

[13] K. Muşlu, Y. Brun, R. Holmes, M. D. Ernst, and D. Notkin. Speculative analysis of integrated development environment recommendations. In *OOPSLA '12*, pages 669–682, Oct. 2012.

[14] S. Packowski. A lightweight and flexible process for designing intuitive error handling and effective error messages. In *CASCON '09*, pages 149–163, Nov. 2009.

[15] B. Shneiderman. System message design: Guidelines and experimental results. In *Directions in Human-Computer Interaction*, pages 55–78. 1982.

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

[17] G. M. Weinberg. *The Psychology of Computer Programming*. Dorset House Publications, 1998.

[18] E. A. Youngs. Human errors in programming. *International Journal of Man-Machine Studies*, 6(3):361–376, 1974.