Measuring the Effectiveness of Error Messages Designed for Novice Programmers

Guillaume Marceau
WPI
100 Institute Road
Worcester, MA, USA
+1 (508) 831-5357
gmarceau@wpi.edu

Kathi Fisler
WPI
100 Institute Road
Worcester, MA, USA
+1 (508) 831-5357
kfisler@cs.wpi.edu

Shriram Krishnamurthi
Brown University
115 Waterman St
Providence, RI, USA
+1 (401) 863-7600
sk@cs.brown.edu

ABSTRACT

Good error messages are critical for novice programmers. Recognizing this, the DrRacket programming environment provides a series of pedagogically-inspired language subsets with error messages customized to each subset. We apply human-factors research methods to explore the effectiveness of these messages. Unlike existing work in this area, we study messages at a fine-grained level by analyzing the edits students make in response to various classes of errors. We present a rubric (which is not language specific) to evaluate student responses, apply it to a course-worth of student lab work, and describe what we have learned about using the rubric effectively. We also discuss some concrete observations on the effectiveness of these messages.

Categories and Subject Descriptors K.3.2 [Computer and Education]: Computer and Information Science Education—Computer science education; H.5.2 [User Interfaces]: Evaluation/Methodology

General Terms Experimentation, Human Factors

Keywords Error messages, Novice programmers, User-studies

1. INTRODUCTION

In a compiler or programming environment, error messages are one of the most important points of contact between the system and the programmer. This is all the more critical in tools for novice programmers, who lack the experience to decipher complicated or poorly-constructed feedback. Thus, many research efforts have sought to make professional compilers more suitable for teaching by rewriting their error messages [10] or supplementing them with hints and explanations [3]. Such efforts complement more general research on improving error messages by means such as error recovery during parsing.

DrRacket[1] [5] goes farther. It defines several sublanguages around what students have learned at different stages [9]. Each sublanguage provides only (versions of) those constructs that make sense at that point, and similarly customizes error messages. The levels and messages have evolved over a decade of observation in lab, class, and office settings.

Despite this care, we still see novices struggle to work effectively with the messages. To understand why, we logged students' edits in response to errors over an entire college-level introductory course and coded whether the edits reflected understanding of the error message. Our work is novel in using fine-grained data about edits to assess the effectiveness of individual classes of error messages. Our coding rubric for assessing the performance of error messages through edits is a key contribution of this work. Our observations about how to use the coding results to reflect on our course is another. Finally, we also present some concrete observations on how students respond to these messages.

2. RESPONSES TO ERROR MESSAGES

We begin by showing a few examples of student responses to error messages during Lab #1. When Lab #1 begins, most students have not had any contact with programming beyond four hours of course lectures given in the days before and two short homeworks due the day before and evening after the lab.

Figure 1 (a) shows one function (excerpted from a larger program) submitted for execution 40 minutes after the start of the lab. The student is defining a function `label`, with one argument `name`. Most likely the student is missing a closing parenthesis after `name`, and another one after `"conservative"`. The nesting suggests that the student is struggling to remember how to combine two Boolean tests into one using the `or` operator. DrRacket provides a textual message (lower pane) and highlights (pink in upper pane) a code fragment that triggered the error.

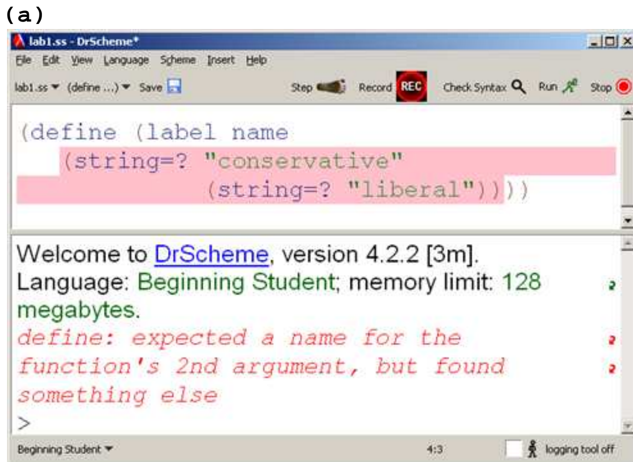
Figure 1 (b) shows the student's next edit. The student inserted `name` as an argument to the function call to `string=?`. An ambiguity in the error message might have prompted this mistake: the word "function" in the fragment "for the function's second argument" can refer to either the function being defined (`label`) or the function being called (`string=?`). DrRacket means the former, but the student seems to have inferred the latter (perhaps influenced by the highlighting). Our dataset illustrates several situations in which the association of the highlight to the error text is underspecified.

As another example, Figure 2 shows a sequence of programs that each triggered the same error message. The topmost program was submitted first; what follows are the student's first four attempts to correct the problem. The student never identifies the actual problem, which is a missing open parenthesis before the `cond`. The entire sequence lasts 10 minutes, until the end of the lab ses-

[1] Formerly known as DrScheme. **`www.drracket.org`**

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. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SIGCSE'11, March 9–12, 2011, Dallas, Texas, USA.
Copyright 2011 ACM 978-1-4503-0500-6/11/03...$10.00.

(a)



(b)

```
(define (label name
   (string=? name "conservative"
             (string=? name "liberal"))))
```

Figure 1. (a) A student's program and its error message, (b) The student's response to the error message

```
(define (string-one-of? check-for-match stringOne stringTwo stringThree)
   cond [(and (string=? check-for-match stringOne))]
        [(and (string=? check-for-match stringTwo))])
```

define: expected only one expression for the function body, but found at least one extra part

```
(define (string-one-of? check-for-match stringOne stringTwo stringThree)
   cond [(string=? check-for-match stringOne)]
        [(and (string=? check-for-match stringTwo))]
        [(and (string=? check-for-match stringThree))])

(define (string-one-of? check-for-match stringOne stringTwo stringThree)
   cond [and ((string=? check-for-match stringOne))]
        [(and (string=? check-for-match stringTwo))]
        [(and (string=? check-for-match stringThree))])

(define (string-one-of? check-for-match stringOne stringTwo stringThree)
   cond [(string=? check-for-match stringOne)]
        [(string=? check-for-match stringTwo)]
        [(string=? check-for-match stringThree)])

(define (string-one-of? check-for-match stringOne stringTwo stringThree)
   cond [(string=? check-for-match)]
        [(string=? check-for-match stringTwo)]
        [(string=? check-for-match stringThree)])
```

Figure 2. A sequence of responses to an error message

sion. A few weeks later, when the student participated in an interview, he mentioned how frustrating the experience had been.

Even with our years of experience teaching with DrRacket, the state of the programs we collected was often surprising, if not humbling. Students manage to create quite mangled functions, which the error messages must attempt to help them sort out.

3. METHODOLOGY

In the spring of 2010, we ran a study in WPI's introductory programming course. We configured DrRacket to save a copy of each program each student tried to run, as well as the error message received (if any) plus any keystrokes that the student pressed in response to the error message, up to their next attempt at running the program. We collected data during the course's normal lab sessions, which ran for 50 minutes per week for six weeks (undergrad courses at WPI are 28 hour-long lectures over seven weeks, so the data cover the entire course). During labs, students worked on exercises covering recent lecture material.

Of the 140 students in the course, 64 agreed to participate in this data collection and 53 actually submitted data. We did not ask participants for demographics such as intended majors, but more than a third of the overall course population listed majors other than computer science. Student participation dropped to 24 students by the final week as a result of students changing lab seats and not reinitializing the software at their new stations; no students withdrew from the study.

4. THE DESIGN OF A CODING RUBRIC

There are many ways one might study the effectiveness of error messages. A common approach in the literature (as reviewed in Section 7) is to change the messages or their presentation and compare the impact on student grades at the end of the course. We are interested in a more fine-grained analysis that determines which error messages are effective and in what ways. There is no single metric for "effectiveness" of an error message. Possible metrics include whether students demonstrate learning after working with messages or whether the messages help novice programmers emulate experts. We have chosen a narrower metric: does the student make a reasonable edit, as judged by an experienced instructor, in response to the error message?

To gain confidence that our metric and its application to our data were valid, we developed a formal rubric for assessing edits and subjected it to a test of *inter-coder reliability* [2] (where "coder" is the standard term for one who applies a rubric to data).[1] Intercoder reliability tests whether a rubric can be applied objectively: multiple coders independently apply it to data, measure their consistency, then revise the rubric until sufficient consistency is achieved. After describing how our rubric evolved, we present the standard measurement of inter-coder reliability that we used.

Our rubric attempts to distinguish ways in which error messages succeed or fail. Our design starts from a conceptual model of how error messages intend to help students: if an error message is effective, it is because a student reads it, can understand its meaning, and can then use the information to formulate a useful course of action. This is a three step sequence:

Read ▸ Understand ▸ Formulate

Students can get stuck at any of these steps. One interesting question is whether students get stuck earlier in the sequence with particular kinds of errors. To explore this, we would ideally like a rubric that identifies how far a student successfully went in the sequence when responding to a given error message. This would suggest a rubric with at least four categories: failure-on-read, failure-on-understand, failure-on-formulate, and fixed-the-error. Our initial attempts to distinguish failure-on-read from failure-on-understand were not successful (in that we could not achieve inter-coder reliability). Our recordings of student editing sessions lack attention-based data (such as eye-tracking) that indicate where a student looked or reacted when an error occurred; such data might have helped distinguish between read- and understand-failures. We concluded that a more realistic rubric would com-

[1] This paper uses "coder" exclusively as a social-science term; in particular, "coder" never refers to programmers.

[DEL]	Deletes the problematic code wholesale.
[UNR]	Unrelated to the error message, and does not help.
[DIFF]	Unrelated to the error message, but it correctly addresses a different error or makes progress in some other way.
[PART]	Evidence that the student has understood the error message (though perhaps not wholly) and is trying to take an appropriate action (though perhaps not well).
[FIX]	Fixes the proximate error (though other cringing errors might remain).

Figure 3. Rubric for responses to error messages

bine failure-on-read and failure-on-understand into a single category separate from failure-on-formulate.

Figure 3 presents our final rubric for assessing students' edits. The [UNR] and [PART] codes capture failure-on-read/understand and failure-on-formulate, respectively. All the responses in the sequence shown in Figure 2 were coded [UNR], for example, since none of the edits tried to change the number of parts in the function body position of the `define`, and nothing else suggested that the student had read or understood the message.

Earlier versions of our rubric attempted to discern two nuances of failure-on-understand: failure to understand the text as separate from failure to understand what the message *really* means in terms of the code. An error message can use simple words and simple grammar but still be hard to understand because the underlying problem is difficult or because the message inadequately describes the problem. Responding to these error messages requires students to read beyond the words and understand that "when DrRacket says X, it really means Y". Figure 4 shows an example. On its face, the message contradicts the text of the code: there definitely is a parenthesis before the `and`. To understand the message, one has to realize that the parenthesis before the `and` has been attributed to the `cond;` in the parser's view, the `and` stands on its own without a parenthesis. Predictably, the student failed to formulate a useful response to that message (they deleted the parenthesis before the `and`). Early versions of the rubric tried to capture how often students failed to formulate a response according to the deep meaning of the message (what an expert would understand from the message) because they were being misled by its literal meaning. However, coders were not sufficiently reliable when making these distinctions, and so the final rubric has only one code corresponding to a failure to formulate, namely [PART].

For the remaining codes in Figure 3, [DEL] captures cases when students simply deleted error-inducing code rather than attempting to fix it, [DIFF] captures edits that were useful but unrelated to the reported error (such as fixing a different error or adding more code or test cases), and [FIX] captures successful completion of the read/understand/formulate sequence. These codes and their precise wordings reflect several design decisions that arose while developing the rubric:

The rubric should assess the performance of the error messages, not the students. Consider a situation in which a student's edit corrects a problem that had nothing to do with the original error message. While this is a positive outcome, it does not address our primary concern of how effective error messages are at guiding students through the read/understand/formulate sequence. Similarly, students may experience difficulties with problem solving or program design that should not be attributed to shortcomings of the error messages. To keep our coding focused on the error messages, we include the [DIFF] code for reasonable edits unrelated to the proximate error. Unreasonable edits unrelated to the proximate error are coded [UNR]. Our first rubric design had unified [DIFF] and [UNR]; we split them after considering when the error message could be held accountable. Sometimes, students simply avoid the proximate error by deleting their code (for example, deleting a test case that yields an error). To avoid judging the error message (as [UNR] might), we introduced the separate [DEL] code for such cases. When deletion is the appropriate action (e.g., when removing an extra function argument) and it is performed on a reasonable code fragment, we code it as [PART] or [FIX] as appropriate. Together, [DIFF] and [DEL] characterize situations in which the student's action provides no information about the quality of the error message.

Coding decisions have to be made narrowly, strictly in relation to the proximate error described in the message. DrRacket reports only one error at a time. Fixing the problem mentioned in the message sometimes makes the overall code worse (for example, a student might delete an extra expression rather than add an operator to combine it with the rest of the code). Frequently, an edit fixes the error mentioned but leaves other glaring errors in surrounding code untouched. We nevertheless code such edits as [FIX]. [FIX] implies neither mastery on the part of the student nor oracular accuracy on the part of the message. It simply means that the student formulated a reasonable response to the problem mentioned in the message. The text "though other cringing errors might remain" reminds coders to take this narrow interpretation. In practice, our coders needed the explicit reminder to be self-consistent in applying [FIX].

Coding needs a holistic view of multi-faceted error messages. DrRacket's error messages have two components: text and a highlight. In assessing whether a student had "read" or "understood" an error message, we had to decide whether editing within the highlighted expression sufficed (even if the edit showed no evidence of understanding the text). Some students glance first at the highlight for a quick overview of the error; this should be a credit to the error message, even though we have a bias towards the text when assessing "understanding". At the same time, students often made random edits in the highlighted code that were arguably unrelated to the proximate error. We ultimately decided that location was not sufficient justification for ascribing [PART] or [FIX].

We accepted liberally any evidence that the student read and understood something from the message. In some cases, making this determination required human judgment or teaching experience, as was the case with the "expect a name" example in Figure 1. Because we decided that the student probably got the idea of inserting "name" from having read the words "expected a name" in the message, we coded that response [PART] rather than [UNR]. We found such subjective decisions surprisingly consistent across the coders.

```
(define (label-near? name bias word1 word2 word3)
  (cond
    (and (cond [(string=? name word1) "Name Located"]
               [(string=? bias word1) "Bias Located"])
         (cond [(string=? name word2) "Name Located"]
               [(string=? bias word2) "Bias Located"])
  "Mark")
))
```

and: found a use of `and' that does not follow an open parenthesis

Figure 4. A counterfactual error message

During the design process, we also ruled out ideas that failed to survive inter-coder reliability tests or our own evaluation:

- *Distinguishing [FIX] codes based on elapsed time*: we considered factoring in students' response time by having separate codes for "fixed with hesitation" and "fixed without hesitation" (we have timestamp data on all edits, and can replay editing sessions at their original pace). In theory, errors to which students respond more slowly might be harder for students to process. We ultimately ruled this out for two main reasons. First, response time could be affected by corrupting interferences (such as a student taking a bathroom break or differences in working styles across students). Second, we lacked a good metric for the expected difficulty of each error message; without that, we would not be able to identify messages that were performing worse than expected.
- *Considering whether the edit yielded a new error message as a criterion for [FIX]:* this is a corollary to our observation about coding narrowly. In practice, we found cases in which the student really did fix the error, but had code of such a form that the same error applied after the edit. We chose to ignore this criterion in final coding.

The rubric as shown in Figure 3 meets standards of inter-coder reliability on the data from Lab #1. We used the standard metric of inter-coder reliability [2], κ, which is defined as

$$\kappa = \frac{\text{Agreement} - \text{Expected Agreement}}{1 - \text{Expected Agreement}}$$

κ compares the agreement of the human coders to the agreement that would be expected by chance according to the marginal probabilities. Because of this, it is a more demanding metric than the simple proportions of the number of times the coders agreed. Values of κ usually lie within 1.0 (meaning perfect agreement) and 0.0 (meaning agreement exactly as good as would be expected by chance), but values of κ can be negative if the human coders perform worse than chance. We executed this test of inter-coder reliability on each version of the rubric. The final version of the rubric (the one shown in Figure 3) was the first version which met the $\kappa > 0.8$ standard, with $\kappa = 0.84$ on 18 different responses.

5. APPLYING THE RUBRIC

Our rubric is designed to identify error messages that students respond to poorly. Given that multiple error messages can reflect the same underlying problem, we group messages into nine thematically-related categories, such as "parenthesis matching", "syntax of define", and "runtime type". For each lab, we sampled 15 edits per category (a sufficient size for statistical validity) and coded each against the rubric in Figure 3. We used Köksal, et al.'s edit-replay software [14] to replay programs during coding.

Table 1 shows the results of coding for the 9 most common error categories (the categories not shown occurred very rarely.) The top row shows the number of errors an average student received during each lab (${}^{\#}E_l$). The sub-table for each lab shows the percentage of the error messages presented during that lab that were of the given category (${}^{\%}P_{l,c}$); the percentage of error messages in the category that were badly responded to according to our coding (${}^{\%}B_{l,c}$); and an estimate of the number of errors in that category that each student responded to poorly during the lab. This estimate is the product of the preceding three values:

$${}^{\#}B_{l,c} = {}^{\#}E_l \cdot {}^{\%}P_{l,c} \cdot {}^{\%}B_{l,c}$$

When computing each variable for the table, we average across students to avoid due influence by the few students who compile much more frequently than the average. For instance, we compute ${}^{\%}B_{l,c}$ by first taking the ratio of $[\text{UNR}] + [\text{PART}]$ against the denominator $[\text{UNR}] + [\text{PART}] + [\text{FIX}]$ per individual student. Specifically, for student *s*, lab, *l*, and category *c*, we compute:

$${}^{\%}B_{s,l,c} = ([\text{UNR}] + [\text{PART}])/([\text{UNR}] + [\text{PART}] + [\text{FIX}])$$

Then we take the unweighted average across the n students who are represented in the selected samples:

$${}^{\%}B_{l,c} = \left(\sum {}^{\%}B_{s,l,c}\right)/\text{n}$$

The bar to the right of #bad in each cell indicates the relative magnitude of #bad values within that lab. The ten lab/error-class combinations with the highest number of poor responses appear in boxes. The sum of the #bad estimates within each lab (under the table) indicates the total estimated number of errors that a student would make in that lab. By this measure, we expect a student to respond poorly to 3.16 errors in Lab #1 and 5.51 errors in Lab #2. While these may seem small out of context, 5.51 errors in 50 minutes translates to a poorly-handled error message every 10 minutes. Worse, this is an average, so the frequency is much higher for some students.

In interpreting the data, we are interested in the cells with boxes (suggesting the highest likelihood of recurrence in a general population) and the cells with the highest %bad values (suggesting errors which were highly problematic for perhaps a small number of students).

We are particularly interested in interventions that might help students at the very beginning of the course (when students form their first impressions of programming and may elect to drop the course). Within Lab #1, syntax errors (including "paren matching") dominate the problematic cases. Based on other data (from interviews with students and a vocabulary quiz), we have good reason to suspect problems with the highlights and the vocabulary. Due to space constraints, we defer discussion of these findings to follow-up papers that report on interventions.

Lab #2 contains four of the ten boxes; students struggled most with the messages in this lab. This might be expected, as students confront many new syntactic constructs and concepts in the preceding week. However, the boxes occur in somewhat surprising

Lab number	#1			#2			#3			#4			#5			#6		
# of errors a student received during this lab (on average)	8.5			16.3			14.4			9.0			9.8			9.8		
	% error	% bad	# bad	% error	% bad	# bad	% error	% bad	# bad	% error	% bad	# bad	% error	% bad	# bad	% error	% bad	# bad
arg. count	5%	48%	0.22	17%	27%	**0.74**	14%	17%	0.33	13%	20%	0.24	35%	21%	**0.74**	12%	31%	0.36
parens matching	28%	24%	0.58	12%	14%	0.27	17%	0%	0.00	14%	0%	0.00	13%	0%	0.00	10%	15%	0.15
runtime cond	3%	0%	0.00	3%	100%	0.49	4%	20%	0.12	6%	72%	0.40	8%	78%	**0.62**	1%	100%	0.06
runtime type	2%	100%	0.15	8%	73%	**0.91**	16%	40%	**0.93**	8%	22%	0.17	6%	44%	0.26	3%	38%	0.13
syntax cond	14%	51%	0.59	4%	50%	0.31	6%	26%	0.24	10%	28%	0.25	9%	20%	0.17	11%	11%	0.12
syntax define	16%	50%	**0.68**	14%	50%	**1.14**	6%	15%	0.14	7%	24%	0.14	2%	17%	0.03	3%	38%	0.10
syntax func. call	14%	64%	**0.74**	14%	17%	0.37	12%	14%	0.26	23%	27%	0.55	4%	29%	0.12	13%	38%	0.48
syntax struct	0%	0%	0.00	8%	32%	0.43	5%	92%	**0.73**	0%	0%	0.00	1%	0%	0.00	0%	0%	0.00
unbound id.	16%	16%	0.21	13%	40%	**0.85**	16%	14%	0.32	16%	0%	0.00	20%	7%	0.14	34%	13%	0.44
Total:			3.16			5.51			3.07			1.75			2.07			1.84

Table 1. Coding results.

categories (e.g., argument count and unbound identifier); manual inspection of the data for those cells might point to the problems.

Most of the boxes occur in the first three weeks. This would be consistent with students improving at error handling. However, curricular aspects of the labs also affect error patterns across cells. For example, the "arg(ument) count" errors in Lab #5 trace to mistakes students made while defining *n*-ary tree data. Constructing *n*-ary trees requires nesting lists inside of structures; students often close expressions for multiple branches incorrectly while nesting calls to the constructors. Understanding the context in which a class of errors becomes problematic is critical to designing good interventions based on this data.

It is tempting to interpret these graphs as indicating students' *conceptual* difficulties in the course. This interpretation is invalid because the error *message* that a student sees is often not a direct indicator of the underlying *error*. For instance, Lab #6 had numerous "unbound-id" errors, but a careful manual analysis revealed that many of them were from students improperly using structures (e.g., using field reference operators incorrectly in various ways), not merely being bad typists. To further confound matters, the precise error that is shown is a function of parsing strategies, because many invalid expressions can be flagged in multiple ways. Therefore, an additional manual analysis is necessary to understand what actual errors students were making.

6. PERSPECTIVE

Fine-grained analysis of student edits has given us detailed insight into student performance both with DrRacket and in our course overall. The data obtained from our rubric identifies errors that students find challenging at different points in the course; refining this with manual inspection of the edit files suggests concrete tasks (such as working with nested data structures) that pose hurdles for students. This information is much harder to identify from looking at the results of homeworks and exams, since these represent a final—rather than ongoing—snapshot of student work.

Effectively, this paper proposes assessing IDE and curricular changes not just on *student achievement* (e.g, course or assignment grades) but to also include assessment based on *error effectiveness*. While our work measures error effectiveness using student actions (edits), we grade the errors, not the students. This has several advantages. Interpreting the success of an action requires a benchmark against which to measure it. When measuring an edit, the benchmark of "made progress" is less subjective and easier to recognize than, say, benchmarking student behaviors against models of behaviors of domain experts.

Our approach does require additional data analysis to identify the conceptual problems underlying the edit message performance. Additional studies are required to determine which problems are best solved by changes to the IDE or curriculum, respectively.

Although our work studies edits to Racket/Scheme programs, our rubric is not language-specific. Our read-understand-formulate model of processing errors is general, as are the categories of edits that appear in the rubric. We have not checked inter-coder reliability of the rubric against edits in other languages, but we hypothesize that the rubric could apply more broadly.

7. COMPARISON TO RELATED WORK

Due to space constraints, we omit citations to efforts to improve error feedback that lack user-oriented evaluation. Early work on the pedagogy of programming sought to classify the errors novice programmers make when using assembly [1] or Pascal [24]. More recent work along the same lines studies BlueJ [12, 22], Java [11], Eiffel [19], and Helium [8]. Others have studied novices' behavior during debugging sessions [18]. These observational studies depend on ad hoc methodologies which impair their reproducibility and makes drawing general conclusions difficult. Our work differs in not studying the students' mistakes broadly and qualitatively; rather, we constructed a quantitative evaluation of the influence of error messages on students' behavior which (we hope) will invite reproduction.

There are still relatively few efforts to evaluate the learning impact of pedagogic IDEs [21]. Gross and Powers survey recent efforts [6], including, notably, those on Lego Mindstorms [4], and on Jeliot 2000 [16], though neither works found a statistically significant improvement. Jadud [13] found a weak relationship between the errors students' received and their final course grade. These studies suffer from having too many confounding factors underneath their dependent variable (the course grades). In contrast, we attempted to tease out the effect of error messages alone.

Other groups have proposed subjective rubrics to study students' development process [14], or to evaluate students' success during individual programming sessions [3, 15]. Coull further added explanatory notes to the error messages of the standard Java compiler based on their observations. These notes made experimental subjects significantly more likely to complete successfully short exercises.

Researchers have interviewed students about the error messages in each of Alice [17] and BlueJ [7]; the difficulty of interpreting the error messages fared amongst the students' primary complaints [23]. One evaluation of BlueJ asked students whether they found the messages useful [25]. Most did, but it is unclear what this means since the evaluation did not gather any comparative data against which to calibrate the students' remarks.

Nienaltowski et al. [20] compared the impact of adding long-form explanation to an error message, and of adding a highlight on three different error messages, in a short web-based experiment. They found that the former has no impact, while the later impairs performance slightly. Unfortunately, the experiment's design has many threats to validity, some of which the paper acknowledged.

8. ACKNOWLEDGMENTS

Fatih Köksal generously provided his software. Ryan S. de Baker, Nate Krach, and Janice Gobert offered extensive advice on social science methodology. We thank the students who participated in our study and Glynis Hamel for enabling it. Tamara Munzner discussed study design and provided useful references. Matthias Felleisen offered valuable discussion and comments on an early draft. This work is partially supported by the US National Science Foundation and Google.

9. REFERENCES

[1] Chabert, J.M. and Higginbotham, T.F. An Investigation of Novice Programmer Errors in IBM 370 (OS) Assembly Language. In *Proceedings of the ACM Southeast Regional Conference*, pages 319-323. 1976.

[2] Cohen, J. A coefficient of agreement for nominal scales. *Educational and Psychological Measurement*, 20(1):37–46, 1960.

[3] Coull, N.J. *SNOOPIE: Development Of A Learning Support Tool For Novice Programmers Within A Conceptual Framework*. PhD Thesis, School of Computer Science, University Of St. Andrews, 2008.

[4] Fagin, B.S. and Merkle, L. Quantitative analysis of the effects of robots on introductory Computer Science education. *Journal on Educational Resources in Computing*, 2(4):1-18, 2002.

[5] Findler, R.B., Clements, J., Flanagan, C., Flatt, M., Krishnamurthi, S., Steckler, P., and Felleisen, M. DrScheme: A programming environment for Scheme. *Journal of Functional Programming*, 12(02):159–182, 2002.

[6] Gross, P. and Powers, K. Evaluating assessments of novice programming environments. In *Proceedings of the International Workshop on Computing Education Research*, pages 99-110. 2005.

[7] Hagan, D. and Markham, S. Teaching Java with the BlueJ environment. In *Proceedings of Australasian Society for Computers in Learning in Tertiary Education Conference*. 2000.

[8] Hage, J. and Keeken, P.V. Mining Helium programs with Neon. *Technical Report, Department of Information and Computing Sciences, Utrecht University*, 2007.

[9] Holt, R.C., Wortman, D.B., Barnard, D.T., and Cordy, J.R. SP/k: a system for teaching computer programming. *Communications of the ACM*, 20(5):301–309, 1977.

[10] Hristova, M., Misra, A., Rutter, M., and Mercuri, R. Identifying and correcting Java programming errors for introductory computer science students. In *Proceedings of the Symposium on Computer Science Education*, pages 153–156. 2003.

[11] Jackson, J., Cobb, M., and Carver, C. Identifying Top Java Errors for Novice Programmers. In *Proceedings of the Frontiers in Education Conference*, pages T4C–24. 2005.

[12] Jadud, M.C. A First Look at Novice Compilation Behaviour Using BlueJ. *Computer Science Education*, 15(1):25–40, 2005.

[13] Jadud, M.C. Methods and tools for exploring novice compilation behaviour. In *Proceedings of the International Workshop on Computing Education Research*, pages 73–84. 2006.

[14] Köksal, M.F., Başar, R.E., and Üsküdarlı, S. Screen-Replay: A Session Recording and Analysis Tool for DrScheme. *Proceedings of the Scheme and Functional Programming Workshop, Technical Report, California Polytechnic State University, CPSLO-CSC-09-03*, :103-110, 2009.

[15] Lane, H.C. and VanLehn, K. Intention-based scoring: An approach to measuring success at solving the composition problem. In *Proceedings of the Symposium on Computer Science Education*, pages 373-377. 2005.

[16] Levy, R.B., Ben-Ari, M., and Uronen, P.A. The Jeliot 2000 program animation system. *Computers & Education*, 40(1):1-15, 2003.

[17] Moskal, B., Lurie, D., and Cooper, S. Evaluating the effectiveness of a new instructional approach. In *Proceedings of the Symposium on Computer Science Education*, pages 75-79. 2004.

[18] Murphy, L., Lewandowski, G., McCauley, R., Simon, B., Thomas, L., and Zander, C. Debugging: the good, the bad, and the quirky — a qualitative analysis of novices' strategies. In *Proceedings of the Symposium on Computer Science Education*, pages 163-167. 2008.

[19] Ng Cheong Vee, M., Mannock, K., and Meyer, B. Empirical study of novice errors and error paths in object-oriented programming. In *Proceedings of the Conference of the Higher Education Academy, Subject Centre for Information and Computer Sciences*, pages 54-58. 2006.

[20] Nienaltowski, M., Pedroni, M., and Meyer, B. Compiler Error Messages: What Can Help Novices? In *Proceedings of the Technical Symposium on Computer Science Education*, pages 168-172. 2008.

[21] Pears, A., Seidman, S., Malmi, L., Mannila, L., Adams, E., Bennedsen, J., Devlin, M., and Paterson, J. A survey of literature on the teaching of introductory programming. *ACM SIGCSE Bulletin*, 39(4):204-223, 2007.

[22] Ragonis, N. and Ben-Ari, M. On understanding the statics and dynamics of object-oriented programs. *ACM SIGCSE Bulletin*, 37(1):226-230, 2005.

[23] Rey, J.S. *From Alice To BlueJ: A Transition To Java*. Master's thesis, School of Computing, Robert Gordon University, 2009.

[24] Spohrer, J.C. and Soloway, E. Novice mistakes: are the folk wisdoms correct? *Communications of the ACM*, 29(7)1986.

[25] Van Haaster, K. and Hagan, D. Teaching and Learning with BlueJ: an Evaluation of a Pedagogical Tool. *Issues in Informing Science and Information Technology*, 1:455-470, 2004.