Do Enhanced Compiler Error Messages Help Students? Results Inconclusive.

Raymond Pettit, John Homer, and Roger Gee
School of IT and Computing
Abilene Christian University
Abilene, TX
1-325-674-2070
{raymond.pettit,john.homer,rpg11a}@acu.edu

ABSTRACT

One common frustration students face when first learning to program in a compiled language is the difficulty in interpreting the compiler error messages they receive. Attempts to improve error messages have produced differing results. Two recently published papers showed conflicting results, with one showing measurable change in student behavior, and the other showing no measurable change. We conducted an experiment comparable to these two over the course of several semesters in a CS1 course. This paper presents our results in the context of previous work in this area. We improved the clarity of the compiler error messages the students receive, so that they may more readily understand their mistakes and be able to make effective corrections. Our goal was to help students better understand their syntax mistakes and, as a reasonable measure of our success, we expected to document a decrease in the number of times students made consecutive submissions with the same compilation error. By doing this, we could demonstrate that this enhancement is effective. After collecting and thoroughly analyzing our own experimental data, we found that—despite anecdotal stories, student survey responses, and instructor opinions testifying to the tool's helpfulness—enhancing compiler error messages shows no measurable benefit to students. Our results validate one of the existing studies and contradict another. We discuss some of the reasons for these results and conclude with projections for future research.

CCS Concepts

• ***Social and professional topics~CS1***
• ***Social and professional topics~Student assessment***
• ***Applied computing~Computer-assisted instruction***
• ***Applied computing~Interactive learning environments***

Keywords

computer science education; computer aided instruction; automated feedback; automated assessment tools; error messages

SIGCSE '17, March 8–11, 2017, Seattle, WA, USA.

ACM 978-1-4503-4698-6/17/03…$15.00 .
DOI: http://dx.doi.org/10.1145/3017680.3017768

1. INTRODUCTION

As automated tools for grading programming assignments become more widely used, learning opportunities may be leveraged by strategically modifying these tools to increase the quality of feedback to students, particularly feedback regarding their submission errors. Known enhancements include software metrics and analyzing the contribution level of each new submission for new features. We were particularly interested in making the language of compiler error messages more understandable for student users, who can be confused by technical messages, particularly in an introductory course. Several related papers have claimed success in this endeavor, based on feedback from students and faculty members, but without providing quantitative data concerning student submission behavior.

We enhanced our current automated assessment tool (AAT), named Athene, based on information from existing research concerning compiler error frequency and ways that researchers have tackled this problem in the past. We also analyzed our own past data to inform our decisions in improving our system, examining the frequency of compiler error types, and focusing on the most common errors to improve the messages students would receive when submitting similar code. We rolled out our enhanced error messages over the course of two semesters and have collected four semesters worth of data. The improvement received mostly positive verbal feedback from both students and instructors.

In this paper, we show the ways in which we improved Athene, consider several metrics of student behavior, and discuss our analysis of the data. We also compare our results to similar work and offer several possible explanations as to the apparent ineffectiveness of enhanced compiler messages.

2. RELATED WORKS

2.1. Student Frustrations

Most instructors will readily agree that syntax and compiler error messages are a great source of frustration to students. Traver addresses problems with compiler error messages, highlighting some of the challenges in improving messages and showing many actual examples of the misleading messages that compilers produce [18]. He offers suggestions on improving these messages based on HCI research and sound pedagogy. Murphy was part of a large multi-institution group analyzing debugging strategies of novice programmers. Observations from class sessions and one-on-one interviews make apparent the frustrations student have, related to misunderstanding errors in programming code [16]. Finally,

Marceau discusses how poor error messages lead to student frustrations, one issue researchers sought to address in creating and improving DrRacket [15]. Furthermore, Marceau observes that some languages used to teach introductory programming, such as Alice [12] and Scratch [14] were created with a goal of protecting students from any possibility of creating syntax errors in their early programs.

2.2. Compile Error Frequency

An examination of compiler errors that students receive in early programming courses shows that some errors occur with much more frequency than others. This pattern becomes especially important as we set priorities in improving standard error messages.

Jadud reports on the most common error messages generated in an introductory programming course using BlueJ to teach Java programming [11]. Of the 1,926 errors generated during the semester he examined, there were a total of 42 different errors encountered, but 5 of these together accounted for 58% of the total errors. The most common errors were (1) missing semicolons, (2) unknown symbol: variable, (3) bracket expected, (4) illegal start of expression, and (5) unknown symbol: class.

Denny used CodeWrite in teaching a Java based course [4]. Most students worked on about 12 programming exercises, and the median number of lines of code for submissions that compiled was 8. For the semester Denny reported on for this paper, students submitted to CodeWrite code containing compiler errors in more than 60% of the attempts. Over 60% of the students experienced at least 4 successive compilation errors at least once during the course of the semester while working within the CodeWrite tool. This repetition gives some indication of the difficulty students have in understanding a given error message and being able to fix the related mistake within relatively small code fragments.

2.3. Similar Experiments

Previous experiments that are related to enhancing compile messages for novice students include (in chronological order): CAP [17], Thetis [8], HiC [9], Expresso [10], Gauntlet [7], a tool by Dy [6], BlueFix [19], LearnCS! [13], an IDE by Barik [1], CodeWrite [5], ITS-Debug [3], and Decaf [2].

2.3.1. Review of CodeWrite/Denny Experiment

Denny reported the results of an experiment using CodeWrite [5]. This experiment took place over one semester in a Java-based course and included 83 students. Students were randomly assigned to an experimental or control group. An independent recognizer was created to identify compiler errors which also included regular expression checking to disambiguate certain messages. By doing this, he was able to recognize the compiler errors in about 92% of all submissions that included compiler errors. During the course of the semester, each student experienced about 70 submissions that failed to compile. Although it was expected that these enhancements would increase student performance, a thorough analysis of the data between the two groups showed that there was no measurable effect in decreasing student compilation errors.

2.3.2. Review of Decaf/Becker Experiment

In contrast to the study done by Denny, a recent study by Becker [2] seems to indicate that enhancing compiler error messages can be done in a way which produces positive empirical results. By enhancing compiler error messages with Decaf, Becker was able to show a significantly lower number of student errors per compiler error message for the compiler error messages that had been enhanced. Becker's study also showed a significantly lower number of student errors per compiler error message for the group of students in the experimental group. Another finding was that students were less likely to generate the same compiler error, from the same line of code, on consecutive attempts. These results run counter to those of Denny, and provide context for the results that we are presenting.

3. METHODOLOGY

3.1. Implementation

In our experiment, we sought to improve compiler error feedback messages in a C++-based CS1 course by implementing changes to our automated-assessment tool. We also implemented an error message parser to analyze corresponding messages from the compiler. The automated assessment tool targets the C++ front-end to the GNU Compiler Collection (GCC).

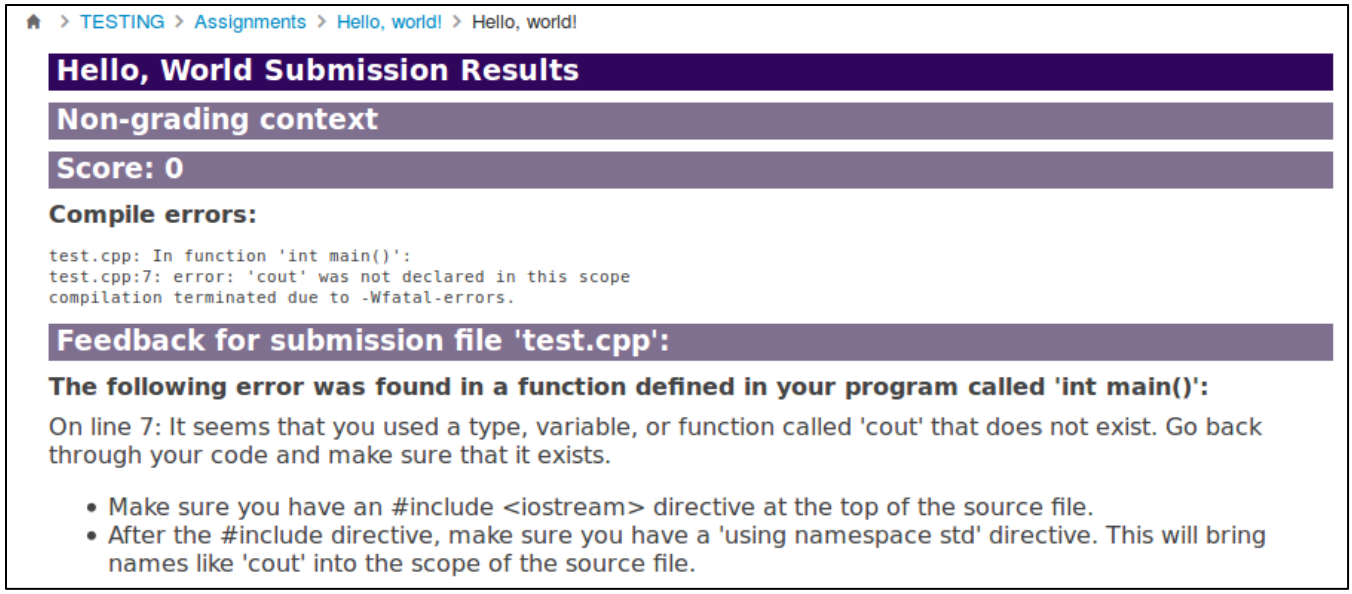
We considered historical submission data from previous iterations of the course to create a probability distribution for different error message types. In this way, we were able to determine which error cases occurred most frequently in the semesters we analyzed. For the set of most frequently occurring error messages, we then analyzed the source code to determine the most common cause for particular error messages. Not every error can be handled based upon the error message alone; some messages are either too indirect or non-pertinent to the actual cause of the issue and require independent analysis of the source code. At this stage, some cases scan a parse tree representation of the student's program, which is obtained from a context-free grammar parser that interprets a subset of the C++ programming language. However, most cases rely solely upon the original compiler error message for error case recognition.

Figure 1 shows an example of a response that includes an enhanced message. A student is still shown the original compiler error message under the section "Compile Errors:". As is the case with the example shown, most compiler messages contain a function context in which the error was found and a message line that contains a number of user-defined elements from the parse tree. We call these user-defined elements *variable tokens*. These tokens are always enclosed in single-quote marks within the message line. We created generic message strings by replacing variable tokens with generic placeholder names. For example, we reduce this error message generated by the compiler to the following generic format:

```
%1 was not declared in this scope
```

Using these message strings allows the system to recognize a general error case and then interpret the actual values of the variable tokens to identify a specific sub-case. In other words, the mapping from the compiler message to enhanced feedback message is not entirely static. Variable tokens from the error

Figure 1. Example feedback to student



message are used to provide specific error feedback. You can see the enhanced feedback under the section titled "Feedback for submission file 'test.cpp' ".

3.2. Data Gathering

We deployed the improvements to our automated-assessment tool in the CS1 programming course. This course contains numerous small assignments (around 75) that gradually progress in difficulty. Students are presented an assignment description and a simple input form for a file upload. The student is allowed and encouraged to build and test his or her programs offline before submitting to the tool. When the student submits a program, the tool attempts to compile the program and, if successful, executes the resulting program against a variety of problem-specific test cases. The system shows the student the status of their submission: compile errors, failed test cases, or successful completion. Each time a student submits, a database records the submit time, program code, score, and feedback given. Before our improvements, the student simply received compile errors "as-is" from the compiler. With the improvements, in response to commonly occurring error messages, the student now sees the compiler messages and the enhanced feedback messages. When these new messages were added to the system, students were shown examples of the message and were encouraged to read them.

4. METRICS AND RESULTS

4.1. Our results

Our study focused on three kinds of measurements:

- likelihood of successive compilation errors
- occurrence of compiler errors within semesters
- student progress towards a successful completion of a programming assignment

We expected that these measurements would demonstrate significant change in relation to the (historical) control group. Such a distinction would indicate that students learn more effectively from enhanced feedback messages and thus perform better with the tool overall.

We compared student use of the improved tool with enhanced error messages against historical data from the tool that generated only stock compiler error messages. We analyzed 4 semesters worth of historical data and 4 semesters with partial or full implementation of the enhancements. In all 8 semesters (36,050 submissions), students were presented generally the same set of programming assignments. Although we show data collected during all 8 semesters, the fall semesters (1210, 1310, 1410, and 1510) represent a larger number of students as well as a more uniform student group from year-to-year. The fall semesters typically are comprised mainly of computer science majors who are taking the course for the first time.

Table 1 shows all data collected from student submissions. A student submission is classified as either: correct, executing but with a wrong answer, generating a runtime error, or generating a compile error.

Over the 8 semester study, students using Athene in our CS1 course submitted programs that failed to compile 16.64% of the time. This number is lower than reported with other tools [4,11] due to students' opportunity to write and debug offline. If the enhanced messages help students avoid compiler errors over time, we would expect to see some decrease in the overall percentage of submissions that cause compiler errors as students learn how to avoid causing them. Over the 4 fall semester, this metric varied from 17%-14%, with no significant trend after enhanced messages were introduced.

Another analysis looked specifically at cases where a student received the *same* compile error in consecutive submissions. This measurement could indicate an improvement in student learning from the enhanced messages by immediately applying that knowledge to fix the error. After receiving a compile error, and given a standard error message, the student's next submission produced the same compile error in 13.71% of cases. When given an enhanced message, there was an insignificant increase to 13.99%.

Table 1. Data from 8 semesters of student submissions

SUBMISSIONS	1210	1220	1310	1320	1410	1420	1510	1520	Totals
Correct	1716/7725	972/4159	1381/3870	699/1704	1729/4676	1114/3814	2923/7678	896/2424	14826/36050
	22.21%	23.37%	35.68%	41.02%	36.98%	29.21%	38.07%	36.96%	31.71%
Program executed, wrong answer	4264/7725	1967/4159	1783/3870	667/1704	2149/4676	1880/3814	3406/7678	1157/2424	17273/36050
	55.20%	47.30%	46.07%	39.14%	45.96%	49.29%	44.36%	47.73%	47.91%
Generated runtime error	421/7725	152/4159	151/3870	85/1704	123/4676	118/3814	231/7678	69/2424	1350/36050
	5.45%	3.65%	3.90%	4.99%	2.63%	3.09%	3.01%	2.85%	3.74%
Generated compile error	1324/7725	1068/4159	555/3870	253/1704	675/4676	702/3814	1118/7678	302/2424	5997/36050
	17.14%	25.68%	14.34%	14.85%	14.44%	18.41%	14.56%	12.46%	16.64%
Given previous compile error, failed compile again with same error	125/1324	118/1068	91/555	32/253	105/675	103/702	183/1118	69/302	826/5997
	9.44%	11.05%	16.40%	12.65%	15.56%	14.67%	16.37%	22.85%	13.77%
… and had advanced feedback					10/125	53/395	100/696	30/164	193/1380
					8.00%	13.42%	14.37%	18.29%	13.99%
… and did not have advanced feedback	125/1324	118/1068	91/555	32/253	95/550	50/307	83/422	39/138	633/4617
	9.44%	11.05%	16.40%	12.65%	17.27%	16.29%	19.69%	28.26%	13.71%

Submitting the same error repeatedly is often a sign that a student does not understand his or her error. However, there are other explanations. We have witnessed students resubmitting a known non-compiling program without making changes in the hope that explanations. We have witnessed students resubmitting a known non-compiling program without making changes in the hope that resubmitting it will cause the tool to reconsider its previous assessment of the program. This type of persistence often works in real-life situations in dealings with other people. It may also just be a sign of frustration.

Over the course of a semester, we would expect students to encounter fewer compile errors as they learn from previous mistakes. When many of their compile error messages were enhanced, we expected to see fewer errors over the course of the semester. However, the percentage of submissions that generated errors did not significantly change after enhanced messages were introduced.

We also analyzed students' progress toward completing the programming assignment. The average number of submissions was used to determine the level of effort a student put forth to correctly debug compile errors and eventually solve the assignment. We counted attempts within each student-assignment -- the sequence of submissions that a particular student makes towards successful completion. Looking at the average number of submission attempts per student-assignment within the 8 semesters, we found no statistically significant trend. Looking

deeper at just the failed compilation attempts, still showed no significant trend. One explanation for not finding a decrease in submission attempts could have been that students increasingly used the tool as their primary compiler given the helpfulness of the enhanced messages. But in this case, we would have expected to see an increase in the values of failed compilation attempts.

Our final measurement attempted to gauge how the tool's enhancements affect the amount of time that students spend working on the program offline. A decrease in time between submissions could indicate that students are benefitting from the tool's improved feedback. Once again, the data showed that there was no evidence to suggest any significant learning from the enhanced messages is taking place; in fact, the mean length of time between submissions showed an increase from about 150 seconds to almost 250 seconds.

5. STUDENT PERCEPTIONS

We requested student feedback about the enhanced compile error messages (from the semesters they were shown) and received 28 responses. The low number and the subjective nature of responses make this data anecdotal, but it can provide indications of student perspective on the enhanced messages.

Students were asked about the level of detail in the enhanced messages, with possible responses ranging from 1 ("too simple") to 5 ("too detailed") and the average response was 3.14, close to

the desired 3.0 balance between simplicity and detail. No responses of 1 or 5 were given.

Most students (67%) indicated that they saw the enhanced messages "occasionally" while others indicated that they saw the messages at least once in a typical assignment; only one student claimed to see the messages six or more times in a typical assignment. When asked how often they read the enhanced messages when they appeared, with possible responses ranging from 1 ("never") to 4 ("always"), the average responses was 3.42. Only one student selected 1, and that same student later seemed to contradict themselves by admitting to submitting homework occasionally just to see if a message helped.

When asked if the enhanced messages helped identify how to fix the problem, 78% (22) of the students responded affirmatively. When asked to identify what (if anything) made the enhanced messages helpful or easier to understand than regular messages, one student responded, "*The messages accurately identified my errors and reported them in concise, easily readable statements. The suggestions on how to fix the errors were also helpful, even when I knew from the error what to do.*" Most responses similarly identified the clarity and comprehensibility of the enhanced

messages, describing them as "*more readable*," "*human language*," "*more familiar wording*," "*clearly worded and in complete sentences*," "*in simpler language*," "*without using too much computer language*," *etc*. A few responses were ambivalent, stating "*I think it is the same to me*" or "*The original ones are easy to pretty easy once you ignore all the stuff that doesn't make sense*." Only one survey response was negative, stating a desire for "*a simple sentence [rather] than some complex rant from the computer about it not wanting to do my program because I have some type of error*."

When asked how often they submitted a program specifically to see an enhanced message, 60% (17) of students acknowledged that they had done this at least once. Of these, many claimed that they had done this only occasionally (1-10 times in the semester) while only a few admitted to following this path often (more than 20 times in semester). Referring to this behavior, students said, "*Sometimes when I kept getting an error after compiling, I would send it in to see if it could point out what my error was*" and "*I couldn't understand what my computer was trying to tell me was wrong*."

Corresponding to this belief in the helpfulness of the enhanced messages, 75% (21) of students agreed that the enhanced messages helped them to "prevent making those mistakes in other programs."

6. CONCLUSION AND FUTURE WORK

6.1. Conclusions and Questions Raised

Given the data that we collected and analyzed, it appears that enhancing compiler error messages does not make students less likely to repeat the same compiler errors. Despite a difference in the language, number of assignments, and the automated assessment tool that we used, we were able to reproduce the same counterintuitive lack of significant effect demonstrated by Denny [5].

These results, however, do not support the work done by Becker [2]. At this time, we are looking more closely at the details of Becker's work to see how his experiment differed from our experiment and Denny's experiment. For example, not all of our compiler messages were enhanced, and it may be that expanding coverage of messages that are enhanced would produce a measurable effect. There are also differences in the way that enhanced messages were displayed to students.

It is interesting to note that even in our experiment, students generally believe the enhanced messages to be helpful, although the quantitative data shows no significant improvement against similar course sections where these messages were not delivered. There are some possible explanations for this apparent contradiction.

Perhaps students don't attentively read the standard compiler messages or our new enhanced error messages. Although students overwhelmingly reported reading these enhanced messages, this may be just bad reporting or wishful thinking on the students' part. Since there were no reports of attempting to measure a quantitative learning effect for students using CAP [17] or Gauntlet [7], we don't know if these tools produced positive measurable effects or not. But it could be that their use of humor contributed to greater student attention to these messages.
Another explanation of the apparent contradiction may be that the higher achieving students who would be the best at understanding enhanced messages and then applying the appropriate fixes don't often submit non-compilable programs to our tool. Perhaps the majority of the submitted non-compilable code is from the lower achieving students who are not conscientious and thus are less likely to spend time reading any error message.

We recognized that we had certain students who are outliers, accounting for a disproportionate number of the compiler errors. For example, we have discovered that in every semester for which we have data, the single student who generates the most enhanced messages sees more than 15% of the total enhanced messages for that class. Given an average class size of 35, a few outlying data points could significantly skew the data concerning the benefit of these messages.

Referring back to the student survey, we want to highlight one student anecdote to describe in a bit more detail to show another possible question raised by our research. This student describes a working session in which she was first attempting to write a given assigned program and achieve some level of functionality before submitting it to the tool. Although some students use the tool as their compiler, most students write the program with their own local compiler and try to create a running program before submitting their program to the tool. She stated that she was having difficulty in understanding a compiler error message that she was receiving from the compiler on her personal machine, but she knew that the enhanced compiler messages given by our tool were usually more helpful, so she purposely submitted non-compilable code simply to receive a better quality error message. And she indeed reported receiving a better message that helped her get past the present error and continue the assignment. As our tool is not normally used as a student's default compiler, we are attempting to find ways to test to see if this student's behavior may be skewing some of the data from the experimental group that is now expecting better compiler error messages from the tool. Perhaps students in this group are now more likely to submit known non-compilable code than the (historical) control group, who would receive no extra benefit from doing this.

6.2. Future Work

Each time a student receives feedback from the tool, we should measure how long he or she views the page with or without an enhanced error message. This may give us some indication of whether or not a typical student is really reading the error messages. With this information we could check for a correlation between reading the enhanced messages and successful resolution of error. Eye movement tracking may also be a possibility in determining if students are reading the enhanced messages.

Alternatively, after being given an enhanced compiler error message, we could ask the student a simple question to see if he or she did indeed read and understand the message. A single multiple choice question related to the given error could be used. Answering this question could tell us two things: did the student really read the message, and did he or she understand what the message said. The student's success at answering the question could be used in conjunction with the above mentioned timing data to further correlate with his or her success at fixing the error.
Perhaps interjecting humor into the error messages does have an effect on how much students will read them. For the given database of error messages that we have produced already, we could make alternative forms of the existing enhanced messages

which added humor. Analysis could then be performed to look for measurable difference in student behaviors and performance.

7. REFERENCES

[1] Titus Barik, Jim Witschey, Brittany Johnson, and Emerson Murphy-Hill. 2014. Compiler error notifications revisited: an interaction-first approach for helping developers more effectively comprehend and resolve error notifications. In *Companion Proceedings of the 36th International Conference on Software Engineering* (ICSE Companion 2014). ACM, New York, NY, USA, 536-539. DOI= http://doi.acm.org/10.1145/2591062.2591124

[2] Brett A. Becker. 2016. An Effective Approach to Enhancing Compiler Error Messages. In *Proceedings of the 47th ACM Technical Symposium on Computing Science Education* (SIGCSE '16). ACM, New York, NY, USA, 126-131. DOI=http://dx.doi.org/10.1145/2839509.2844584

[3] Elizabeth Carter. 2015. ITS debug: practical results. *J. Comput. Sci. Coll.* 30, 3 (January 2015), 9-15.

[4] Paul Denny, Andrew Luxton-Reilly, Ewan Tempero, and Jacob Hendrickx. 2011. Understanding the syntax barrier for novices. In *Proceedings of the 16th annual joint conference on Innovation and technology in computer science education* (ITiCSE '11). ACM, New York, NY, USA, 208-212. DOI= http://doi.acm.org/10.1145/1999747.1999807

[5] Paul Denny, Andrew Luxton-Reilly, and Dave Carpenter. 2014. Enhancing syntax error messages appears ineffectual. In *Proceedings of the 2014 conference on Innovation & technology in computer science education* (ITiCSE '14). ACM, New York, NY, USA, 273-278. DOI= http://doi.acm.org/10.1145/2591708.2591748

[6] Thomas Dy and Ma. Mercedes Rodrigo. 2010. A detector for non-literal Java errors. In *Proceedings of the 10th Koli Calling International Conference on Computing Education Research* (Koli Calling '10). ACM, New York, NY, USA, 118-122. DOI= http://doi.acm.org/10.1145/1930464.1930485

[7] T. Flowers, C. A. Carver, and J. Jackson. 2004. Empowering students and building confidence in novice programmers through Gauntlet. In *Frontiers in Education, 2004* (FIE 2004). Vol. 1, 20-23. DOI= http://dx.doi.org/10.1109/FIE.2004.1408551

[8] Stephen N. Freund and Eric S. Roberts. 1996. Thetis: an ANSI C programming environment designed for introductory use. *SIGCSE Bull.* 28, 1 (March 1996), 300-304. DOI= http://doi.acm.org/10.1145/236462.236560

[9] Robert W. Hasker. 2002. HiC: a C++ compiler for CS1. *J. Comput. Sci. Coll.* 18, 1 (October 2002), 56-64.

[10] Maria Hristova, Ananya Misra, Megan Rutter, and Rebecca Mercuri. 2003. Identifying and correcting Java programming errors for introductory computer science students. *SIGCSE Bull.*35, 1 (January 2003), 153-156. DOI= http://doi.acm.org/10.1145/792548.611956

[11] M. C. Jadud. A first look at novice compilation behaviour using BlueJ. *Computer Science Education* 15,1 (2005), 25-40.

[12] Caitlin Kelleher, Randy Pausch, and Sara Kiesler. 2007. Storytelling alice motivates middle school girls to learn computer programming. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (CHI '07). ACM, New York, NY, USA, 1455-1464. DOI= http://doi.acm.org/10.1145/1240624.1240844

[13] Derrell Lipman. 2014. LearnCS!: a new, browser-based C programming environment for CS1. *J. Comput. Sci. Coll.* 29, 6 (June 2014), 144-150.

[14] John Maloney, Mitchel Resnick, Natalie Rusk, Brian Silverman, and Evelyn Eastmond. 2010. The Scratch Programming Language and Environment. *Trans. Comput. Educ.* 10, 4, Article 16 (November 2010), 15 pages. DOI= http://doi.acm.org/10.1145/1868358.1868363

[15] Guillaume Marceau, Kathi Fisler, and Shriram Krishnamurthi. 2011. Measuring the effectiveness of error messages designed for novice programmers. In *Proceedings of the 42nd ACM technical symposium on Computer science education* (SIGCSE '11). ACM, New York, NY, USA, 499-504. DOI= http://doi.acm.org/10.1145/1953163.1953308

[16] Laurie Murphy, Gary Lewandowski, Renée McCauley, Beth Simon, Lynda Thomas, and Carol Zander. 2008. Debugging: the good, the bad, and the quirky -- a qualitative analysis of novices' strategies. *SIGCSE Bull.* 40, 1 (March 2008), 163-167. DOI= http://doi.acm.org/10.1145/1352322.1352191

[17] Tom Schorsch. 1995. CAP: an automated self-assessment tool to check Pascal programs for syntax, logic and style errors. *SIGCSE Bull.* 27, 1 (March 1995), 168-172. DOI= http://doi.acm.org/10.1145/199691.199769

[18] V. Javier Traver. 2010. On compiler error messages: what they *say* and what they *mean*. *Adv. in Hum.-Comp. Int.* 2010, Article 3 (January 2010), 26 pages. DOI= http://dx.doi.org/10.1155/2010/602570

[19] Christopher Watson, Frederick W. B. Li, and Jamie L. Godwin. 2012. BlueFix: using crowd-sourced feedback to support programming students in error diagnosis and repair. In *Proceedings of the 11th international conference on Advances in Web-Based Learning* (ICWL'12), Elvira Popescu, Qing Li, Ralf Klamma, Howard Leung, and Marcus Specht (Eds.). Springer-Verlag, Berlin, Heidelberg, 228-239. DOI= http://dx.doi.org/10.1007/978-3-642-33642-3_25