8-2017

On Novices' Interaction with Compiler Error Messages: A Human Factors Approach

James R. Prather
Abilene Christian University

Raymond Pettit

Kayla Holcomb McMurry

Alani Peters

John Homer

See next page for additional authors

Authors
James R. Prather, Raymond Pettit, Kayla Holcomb McMurry, Alani Peters, John Homer, Nevan Simone, and Maxine Cohen

On Novices' Interaction with Compiler Error Messages: A Human Factors Approach

James Prather, Raymond Pettit, Kayla Holcomb McMurry, Alani Peters, John Homer, Nevan Simone
Abilene Christian University
ACU Box 28036
Abilene, TX 79601
jrp09a,rsp05a,kmh12c,alp13d,jdh08a,nfs13a@acu.edu

Maxine Cohen
Nova Southeastern University
3301 College Avenue
Fort Lauderdale, FL 33314
cohenm@nova.edu

ABSTRACT

The difficulty in understanding compiler error messages can be a major impediment to novice student learning. To alleviate this issue, multiple researchers have run experiments enhancing compiler error messages in automated assessment tools for programming assignments. The conclusions reached by these published experiments appear to be conflicting. We examine these experiments and propose five potential reasons for the inconsistent conclusions concerning enhanced compiler error messages: (1) students do not read them, (2) researchers are measuring the wrong thing, (3) the effects are hard to measure, (4) the messages are not properly designed, (5) the messages are properly designed, but students do not understand them in context due to increased cognitive load. We constructed mixed-methods experiments designed to address reasons 1 and 5 with a specific automated assessment tool, Athene, that previously reported inconclusive results. Testing student comprehension of the enhanced compiler error messages outside the context of an automated assessment tool demonstrated their effectiveness over standard compiler error messages. Quantitative results from a 60 minute one-on-one think-aloud study with 31 students did not show substantial increase in student learning outcomes over the control. However, qualitative results from the one-on-one think-aloud study indicated that most students are reading the enhanced compiler error messages and generally make effective changes after encountering them.

CCS CONCEPTS

• **Human-centered computing → Empirical studies in HCI**; *User studies*; *Usability testing*; • **Social and professional topics → CS1**; *Student assessment*; • **Applied computing → Computer-assisted instruction**; *Interactive learning environments*;

ICER '17, August 18-20, 2017, Tacoma, WA, USA

ACM ISBN 978-1-4503-4968-0/17/08…$15.00
https://doi.org/10.1145/3105726.3106169

Table 1: Definition of Frequently Used Terms

Term	Meaning
CEM	Compiler Error Message
ECEM	Enhanced Compiler Error Message
Athene	Automated assessment tool used in this experiment

KEYWORDS

HCI, human factors, usability, automated assessment tools, education, CS1, ethnography

ACM Reference format:
James Prather, Raymond Pettit, Kayla Holcomb McMurry, Alani Peters, John Homer, Nevan Simone and Maxine Cohen. 2017. On Novices' Interaction with Compiler Error Messages: A Human Factors Approach. In *Proceedings of ICER '17, Tacoma, WA, USA, August 18-20, 2017*, 9 pages.
https://doi.org/10.1145/3105726.3106169

1 INTRODUCTION

It is well-documented that novice programmers often struggle in understanding compiler error messages (CEMs) caused by incorrect syntax [3, 7, 32, 42]. This is a motivating factor in the development of automated assessment tools [36]. In one very popular tool, BlueJ, students placed their difficulty in interpreting CEMs high amongst their other concerns [16]. Bennedsen has suggested that this is a contributing factor in the high failure rates in CS1 courses [5]. Specifically called out in 1976 [44] as an impediment to learning syntax, CEMs are still just as cryptic and hard to understand as they were forty years ago [4]. In an attempt to alleviate novice frustration, several papers have recently attempted to improve upon the design of standard CEMs in automated assessment tools from an HCI perspective [2, 31, 33, 42], which we will call enhanced compiler error messages (ECEMs).

Several recent studies have enhanced default CEMs and performed empirical experiments to determine if ECEMs have a positive impact on student learning in CS1 [4, 9, 37]. These studies all provide quantitative data and analysis of student performance, but stated conflicting conclusions. Denny et al. [9] and Pettit et al. [37] could not find conclusive evidence that enhancing error messages was helpful to students. Becker [4], however, was able to show that ECEMs were more helpful for those in his study.

We postulate five possible explanations for these conflicting results:

(1) students do not read ECEMs,
(2) researchers are measuring the wrong thing,
(3) the effects are hard to measure,
(4) the ECEMs are not properly designed, therefore students do not understand them,
(5) the ECEMs are properly designed but students do not understand ECEMs in context due to increased cognitive load.

For this study, we constructed experiments to test the possibilities that (1) students do not read ECEMs or (5) the ECEMs already implemented are properly designed but cognitive load in students reduces the positive effects. Our research questions are therefore:

- **RQ1:** Do novice students read ECEMs?
- **RQ2:** Are ECEMs helpful for novice students in a setting with low cognitive load?
- **RQ3:** Are the ECEMs helpful for novice students in a setting with high cognitive load?

We begin by reviewing related work on automated assessment tools, ECEMs in automated assessment tools, and the design of these ECEMs. We next describe the methodology by which we attempt to answer the research questions above. In section 4, we describe both quantitative and qualitative results from our mixed-methods study. In section 5 we summarize our conclusions.

2 RELATED WORK

Automated assessment tools for programming assignments have been around since at least 1960, when Hollingworth built a way to automatically assess the programs that students submitted in his course via punch cards [19]. Today, many different automated assessment tools exist, some focused more on assessment and some focused more on helping students learn to program. Still others are focused on test-driven development, such as the popular automated assessment tool Web-CAT [12]. For a general review of automated assessment tools see Ala-Mutka [1], Douce [10], and Ihantola [21].

2.1 Error messages in automated assessment tools

Many creators of automated assessment tools have attempted to enhance the standard syntax/compiler error messages that students receive. One of the earliest examples is CAP developed by Schorsh in 1995 [39]. The intent of CAP was to provide students in an introductory programming course with user-friendly feedback pertaining to syntax, logic, and style errors. In 2012, Watson discussed the tool BlueFix, which applied his principle of adapting the compiler messages to the level of the students [43]. Upon the first encounter of a compiler error, students saw the standard error. If the student generated the same error a second time in a row, they received an enhanced version of the error message. After a third consecutive generation of the same compiler error, the student received a suggested fix to their code. This adaptive process involved an extensive analysis of the student's existing code and prediction of the student's intent. Students were then able to vote on whether or not the suggested fix worked. In this way, Watson was able to introduce crowdsourcing techniques into automated assessment tools. Other examples of enhancing compiler error messages for novice students include Thetis [15], HiC [18], Expresso [20], Gauntlet [14], a tool by Dy [11], LearnCS! [27], an IDE by Barik [3], and ITS-Debug [6].

2.2 Students have trouble with CEMs

Syntax and compiler error messages have long been documented to be a great source of confusion and 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 [42]. He offers suggestions on improving these messages based on HCI research and sound pedagogy. Murphy et al. were part of a large multi-institution group analyzing debugging strategies of novice programmers [32]. Observations from class sessions and one-on-one interviews make apparent the frustrations student have related to misunderstanding errors in programming code. Finally, Marceau et al. discuss how poor error messages lead to student frustrations, one issue researchers sought to address in creating and improving DrRacket [30]. Furthermore, Marceau observes that some languages used to teach introductory programming, such as Alice [23] and Scratch [29] were created with a goal of protecting students from any possibility of creating syntax errors in their early programs.

2.3 Empirical evidence of helpfulness of ECEM (Denny, Becker, Pettit)

In 2014, Denny et al. reported on the tool CodeWrite and the enhanced error messages the compiler generates [9]. Researchers used the CodeWrite tool for Java programmers, intercepting the compiler error messages that the tool returned. The researchers replaced existing compiler error messages with much more descriptive error messages geared to the novice programmer. The conclusion of the experiment was that there was no statistically significant difference in the students' behavior: students submitted as often as others had before to get past the same compile errors. These results were unexpected and seemed non-intuitive. In contrast, Becker [4] similarly enhanced error messages in the automated assessment tool, Decaf, also used for Java programming. His findings showed that these enhanced messages actually did change student behavior. After viewing an enhanced error message, students were less likely to generate the same error in the future. Finally, Pettit et al. enhanced CEMs in an automated assessment tool, Athene, used for C++ programming [37]. They could not find conclusive results that the ECEMs were more helpful than standard CEMs.

2.4 Design of ECEMs in automated assessment tools

Hartmann et al. [17] created their own automated assessment tool, HelpMeOut, which provides students with feedback similar to Denny et al. [9]. HelpMeOut queries a database of similar errors and presents users with examples and how to fix them. Previous approaches, such as those discussed above, have implemented enhanced feedback through a selection of top errors provided by instructors. These lists of potential errors are driven by experts and not user observation. A weakness to this approach is evidenced by

one such implementation discussed above, Gauntlet [14], that was later found by Jackson et al. [22] to not contain the most commonly encountered errors by novices. HelpMeOut overcomes this weakness through a dynamic list of real student bugs that can better reflect actual user experience. Furthermore, the suggestion that appears at the top of the list is accomplished through crowdsourced voting by students. In other words, the dominating metric of which examples of similar bugs that students will see is based heavily on user experience. While the solution is quite novel, Hartmann, et al., do not attempt to measure whether their automated assessment tool helped novice programmers create a better mental model of the errors they received or whether it increased learnability for novice programmers. Furthermore, as Traver et al. note, this requires a large database of student suggestions and crowd-sourced data [42].

Marceau et al. [31] questioned the computer science education research community for investigating whether or not feedback messages helped users learn without approaching it from the perspective of users. They provide both a quantitative and qualitative human factors approach via a statistical analysis of user errors after introducing enhanced feedback and follow-up interviews with four of those same students. They discovered that students were grossly misinterpreting the feedback messages and were confused at the highly specialized vocabulary of their automated assessment tool, DrRacket. They postulate that perhaps students do not take the time to read the messages, but rather use it only as an "oracle" that somehow knows how to fix their code or that students prefer to read only the code highlights that indicate the necessary change. In following work, Marceau et al. [30] provide a rubric for evaluating the effectiveness of error messages based on student behavior after encountering them. They recommend changes to error messages: simplify vocabulary, be more explicit in pointing to the problem, help students match terms in the error message to parts of their code (e.g. using color coded highlighting), design the programming course with error messages in mind (rather than an afterthought), and teach students how to read and understand error messages during class time.

Several other recent studies utilize aspects of a human factors approach to an automated assessment tool. Traver et al. discuss the theory behind error message design and propose eight specific principles for the design of ECEMs [42]. Lee and Ko discuss personifying feedback in a game that teaches programming [25]. Their tool, Gidget, personifies feedback by accepting blame when a program works incorrectly. Participants in the experimental group where personification was increased completed more levels of the game in a similar amount of time compared to the control group. Barik et al. performed an experiment with eye tracking software to determine if students read error messages [2]. These students were a mix of undergraduate and graduate students who had an average of 1.4 years of professional software engineering experience within a company. While the study by Barik et al. is closely related to the present study, we examine the problem from the perspective of novice programmers in their very first programming course. Barik et al. found that intermediate students do indeed read CEMs. They also found that, as error messages become more difficult and cryptic, programmers cycled between the error message and the offending code more times, negatively correlating with success. Their findings provide empirical justification for the necessity of enhancing CEMs. Loksa et al. performed a study on a code camp where the control group was taught to program and the experimental group was additionally trained in the cognitive aspects of coding [28]. They suggest training students in metacognitive awareness, upon the assumption that "programming is not merely about language syntax and semantics, but more fundamentally about the iterative process of refining mental representations of computational problems and solutions and expressing those representations as code," and report that students with this training were significantly better able to understand error feedback. Loksa's work suggests an entirely different direction for this research, one that is beyond the scope of this study.

2.5 Think-aloud studies in CS1

One research tool often employed in evaluating changes made to CS1 classes is the think-aloud protocol where students are observed writing code and asked to verbalize their thoughts while doing so. Yuen performed a think-aloud study on his CS1 class to understand the differences in how novices construct knowledge compared to experts [46]. He collected data from four sources: an initial survey, participants' work on paper, transcripts of the interviews, and the researcher's field notes. Their results show three kinds of student behavior in response to various levels of knowledge construction. The least desirable response, "need to code," is when the student does not seek to first understand and determine a solution, but instead turns directly to the code. A better response is the second, "generalizing the problem," where the novice is able to take what they have previously learned and try to generalize it to the present scenario, sometimes leading to a valid solution. The third and most desirable behavior, "designing effective solutions," is when the student is able to properly take their knowledge construction and apply it to create a working solution. These three categories will be useful in this study's data analysis.

Teague et al. perform a think-aloud study watching novices trace code and then attempt to explain in a single sentence what it does [40]. They follow the classic think-aloud protocols by Ericsson and Simon [13]. Teague's results suggest that students who cannot trace code cannot build appropriate abstractions to understand complex programming tasks. One important contribution they make is in noting that think-aloud studies are difficult for novices. The task of programming is already cognitively overloading novices and therefore asking them to also think-aloud during a study could threaten the ability to replicate the same silent attempt. To offset this, they began their study with a short think-aloud practice session so the participant could become familiar with the think-aloud protocol and the interviewer. We follow Ericsson and Simon for think-aloud protocol and follow Teague in adding a short practice session at the beginning to hopefully offset cognitive load on novices.

Whalley and Kasto perform a think-aloud study watching novices solve three programming challenges [45]. Researchers narrated the problem-solving process and showed how some students who might otherwise get stuck were able to solve the challenges with some redirection and scaffolding. They also note that think-aloud studies are difficult with novice programmers because the

cognitive load is already very high and so they have a difficult time concentrating on solving the problem and can't continually verbalize their thoughts. In order to offset this, they also used a short practice session so participants could get used to the think-aloud protocol.

Qualitative research methods complement our quantitative methods in this study. These qualitative methods will give us access to information that submission data and quiz results do not, such as the students motivations for behaving in a certain way, their beliefs about what they are doing, and the feelings that they are having when performing certain actions. All of these serve to give us a better explanation of why students make certain choices. For a thorough review of qualitative research approaches, see Lincoln and Guba [26] or Patton [34]. For more information specifically on mixed method approaches, see Creswell [8].

3 METHODOLOGY

3.1 Pilot Studies

Two usability pilot studies were conducted in fall 2016, each with six participants in ten minute sessions that used a simple Fibonacci problem requiring a while-loop to solve. The automated assessment tool used in this study was Athene due to its adoption and use at the university where the study was conducted [35, 37, 41]. Participants were provided with a code file that contained six bugs and asked to submit the code, fixing errors as they found them, until Athene accepted the program as complete. The program used a method that takes *n* as a parameter and returns the *nth* Fibonacci number, computed using a while loop. After the first pilot study, the ECEMs in Athene were redesigned using the guidelines from the literature [17, 31, 42]. The second pilot study performed the same experiment with the redesigned ECEMs which led to a further refining of their design. This included changing wording to better match user expectations, streamlining the design, and removing pieces that confused users. Perhaps the most interesting change was to the text in the title of the enhanced portion of the message from "Need More Help?" to "More Information." Most participants in the second pilot study indicated that they did not want more help - even the ones that struggled. The design of the button used the words "Need More Help?" and most participants balked at that phrasing as a threat to their ego. Follow-up questions about why they felt this way revealed they thought that looking at something titled "Need More Help?" was almost like cheating or like giving up and they wanted to do it themselves without looking at the answer. The phrasing was picked to be a neutral and clear label about the button's function, but it clearly was not perceived this way by participants. Thus, it was changed to "More Information" as in Figure 1 and Figure 2. The attitude among the participants led to the creation of question #3 in the follow-up questions in the think-aloud study.

3.2 Error Message Quizzes

Participants that were enrolled in CS1 for Spring 2017 at Abilene Christian University were given six quizzes in class to determine if the newly refined ECEMs were more helpful than the standard compiler messages. This was done in an attempt to answer RQ2: are ECEMs helpful for novice students in a setting with low cognitive load. These quizzes all took place outside of the context of Athene as part of a replacement for a daily quiz where cognitive load would be much lower. In order to provide a control group and an experimental group, the class of 31 was divided in to two groups of similar demographics: A and B. Each quiz contained a code snippet with a bug that would lead to a specific compile error, a feedback message produced by Athene when that code is submitted, and a short-answer question asking students to determine where the error is, what the error is, and how they would fix it. In every case students had seven minutes to solve the quiz. In odd-numbered quizzes, the students in group A saw only the standard compiler

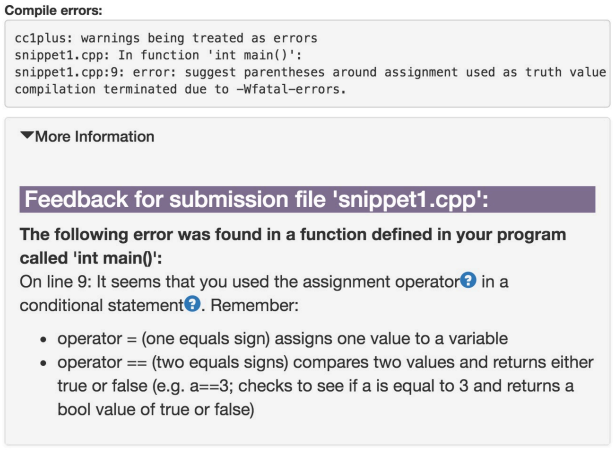

Figure 1: The error message from error message quiz 1B (enhanced) with the enhanced message expanded

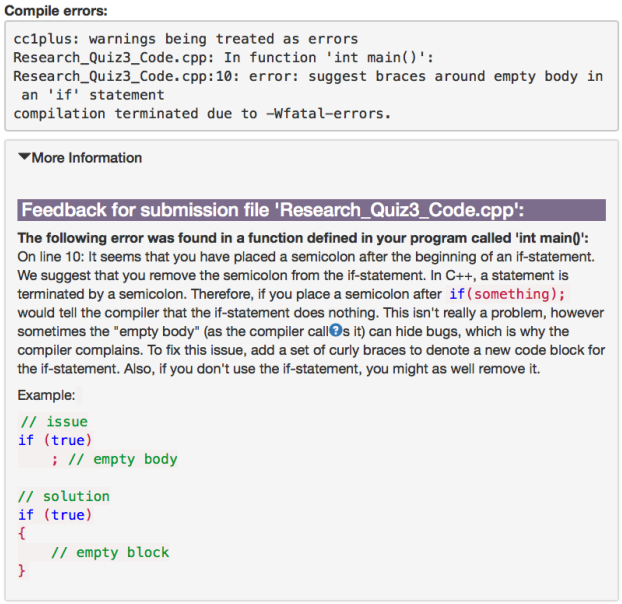

Figure 2: The error message from error message quiz 4A (enhanced) with the enhanced message expanded

error message as feedback from Athene, while the students in group B saw the standard compiler error message as well as the enhanced error message from Athene. For the even-numbered quizzes, group A saw the standard and enhanced messages while group B saw only the standard messages. This was repeated for all six quizzes. Thus, each student saw a standard message for three quizzes and the enhanced message for three quizzes. Each quiz contained a different code snippet with a different compile error and thus a different feedback message from Athene. The compile errors chosen for the six quizzes were the six errors students encountered most frequently while attempting the programming assignment used in the think-aloud study below. The difficulty of the code snippets that students were asked to analyze scaled along with the content of the semester in an effort to make them consistently challenging.

3.3 Think-Alouds

As a classroom enhancement, the researcher canceled class and instead held hour-long one-on-one sessions in CS1 during week 6 of classes for Spring 2017 at Abilene Christian University. All 31 students participated. Each student met one-on-one with a researcher where the student was observed completing a "practical quiz" and received feedback on their process. A practical quiz is similar to a homework assignment - students receive a programming problem in Athene, but must solve it in a proctored 35 minute time window. Students were asked to verbalize their thoughts while they solved the problem, especially when they encountered the enhanced feedback messages. The primary researcher supervised two researchers conducting these one-on-one studies. Occasionally a second supervisor also observed these sessions. This was done to ensure standardization of practices between observers. In an effort to control for differing development environments and the help students might or might not receive from certain ones, students were only allowed to type their code in the default Windows notepad application.

The general format of the think-aloud study follows the usability testing guidelines found in Rubin and Chisnell [38] and Krug [24], including pre- and post-testing checklists and scripts. At the beginning of each session, the evaluator read from a script outlining the reason for the session, the goal of the session, and what was expected of the student. Students were then given a very simple task and asked to think-aloud so they can get used to verbalizing their thoughts, the observer, and the process, as suggested by Teague et al. [40] and Whalley and Kasto [45]. This simple task was to write a program that would output "Hello, world." This particular task was chosen because it was cognitively the easiest code to write for any level of student at that point in the semester, so practicing the think-aloud protocol would be easier during this time.

Students were then asked to complete a practical quiz, similar to a simple homework assignment, with a time limit of 35 minutes. The task was this: given n numbers, compute whether there were more positive or negative integer numbers provided as input. Students would need to understand the following concepts: console input, console output, conditionals, and loops. This particular problem, rather than the Fibonacci problem used for the pilot studies, was chosen because it has been used as an in-class assessment in previous semesters and a majority of students from those previous semesters completed the problem within the same 35 minute time limit. While solving the problem, a researcher took extensive notes on what the student did and said. This study follows the recommendations made by Ericsson and Simon [13] for carrying out think-aloud studies, specifically minimizing social interaction with participants and trying to gently keep them focused on the task at hand. For instance, if the student stopped talking, then the researcher would say, "Keep speaking," and not "Tell me what you're thinking." This encouraged speaking, but the participant was not asked to formulate responses or socially interactive dialogue.

After the students successfully completed the problem or the time limit expired, they were asked up to five interview questions and their responses were recorded. They were asked up to five because some questions may not have pertained to that particular student, depending on their experience solving the quiz. For instance, students could not be asked about their perception of the ECEMs if they did not encounter any of them. Below are the questions that the students were (potentially) asked:

(1) When you encountered the enhanced feedback messages (with the "More information" drop-down), were they helpful? Why or why not?
(2) When you see a feedback message from Athene, how does it make you feel?
(3) Would you rather read the enhanced message under "More information" first, or would you rather wait until you can't figure it out yourself? Why?
(4) (If they saw an enhanced message and did not click it) When you saw the enhanced message, why did you choose not to click on it?
(5) In this class, how often before the deadline do you usually make your first attempt (uploading your program to Athene) on your homework?

These questions were designed to get the student talking about their experience with the ECEMs that they potentially encountered. Getting at perception can be difficult, so the first three questions were all designed to hit around the same issue. The fourth question was for those students that we knew would see them and not use them. The final question was added in an attempt to try and correlate perceived work ethic with use of the ECEMs.

The think-aloud study with post-assessment interview was designed to answer research questions (1) Do Students read ECEMs? and (3) Are the ECEMs helpful for novice students in a setting with high cognitive load?

3.4 Think-Aloud Analysis

The ethnographic portion of the study consisted of participant observation and post-assessment interview questions. This allowed us to record the participants' actions, thought process, problem solving process, reactions to error messages, and their answers to the end-of-session interview questions. Because the participants were asked to use the think-aloud protocol, we were better able to record their thought-process as they solved the assessment. Participant-specific data were separately recorded and then phenomena were coded and grouped into categories. In this way, larger trends began to emerge from the natural groupings of the qualitative data.

We put all of the data into ATLAS.ti and used it for the coding process. We began the coding process by combing through each document and creating quotations and adding codes for participants receiving error messages. For each error message, we coded if it was or was not an enhanced error message. The ECEMs were then broken down into a numbering system so that we could see how many times students received each ECEM. The error messages that were not enhanced were coded as "unenhanced," but were not given a specific error message number. If the error message was enhanced, we then coded if the participant expanded the "more information" section (see Figure 1 and Figure 2 and note the collapsible "more information" section). If the "more information" section was expanded, we then coded whether it proved to be helpful or unhelpful based on their next code edit and submission. However, if the "more information" section remained collapsed, we did not code for that ECEM's helpfulness. Next, we coded for completion time. Completion time was divided into five groups: less than 10 minutes, 10-20 minutes, 20-30 minutes, 30-35 minutes, and incomplete. Each participant document received one of the completion time codes, which allowed us to analyze the progress of each student. At the conclusion of each one-on-one session, the evaluator asked several questions and then gave feedback on the participant's performance in the quiz. We therefore coded each participant's response to these follow-up questions in order to receive an overall perception for the error messages. There were 28 unique codes and these codes were used a total of 370 times.

3.5 Program Logs Analysis

The particular programming assignment that was given in our one-on-one think-aloud sessions was given as an in-class assessment in three other semesters of the same Programming 1 course. For each submission, Athene logged the student's information, a snapshot of the code submitted, the test results or error message produced by Athene, time submitted and current grade on the assessment. The logs of these semesters were pulled and analyzed as a control to compare against our one-on-one session results. For each semester, we measured the number of students who completed the programming assignment with a correct solution, average score, and average time until completion in both the control and experimental groups.

4 RESULTS

4.1 Error Message Quizzes

The error message quizzes were given to students outside of the context of an assessment in Athene to determine if the redesigned ECEMs, on their own, were more helpful than the standard CEMs. Twenty-seven students from the Spring 2017 CS1 class were present for all six quizzes. The results of these quizzes (see Figure 3) show that the experimental case (ECEMs) was more helpful than the control (standard CEMs). The mean percent of incorrect answers among participants in the control group was 17.28% while the mean percent of incorrect answers in the experimental condition was 6.17%. Therefore, the experimental condition displayed a statistically significant improvement over the control ($p < 0.035$, $n = 27$, paired two sample for means).

Out of the 27 participants present for all six quizzes, 13 students gave an incorrect answer on at least one quiz. As shown in Figure

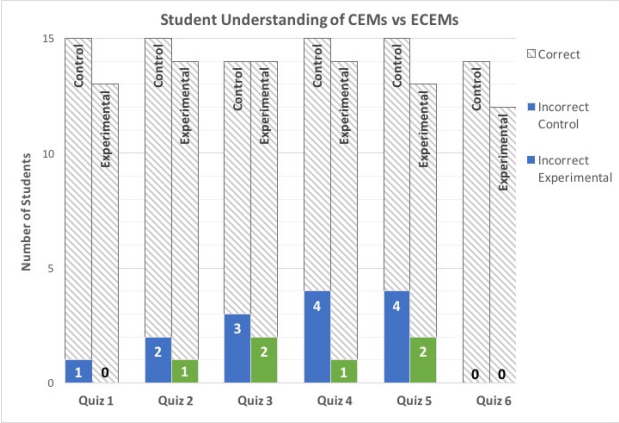

Figure 3: Number of incorrect responses for each condition by quiz

Control (CEM)	Experimental (ECEM)	# of Students
0	1	1
0	2	1
1	0	7
1	1	2
2	0	1
3	0	1

Figure 4: Incorrect Understanding of CEM vs. ECEM

4 (rows 3, 5 and 6), 9 of the 13 students were helped more by the ECEMs. One particularly interesting case is the student who incorrectly answered all three control quizzes, but correctly answered all three in the experimental condition (row 6). Another outlier in the opposite direction was the student who incorrectly answered two experimental quizzes, but correctly answered all in the control condition (row 2).

4.2 Program Logs

Data that can be pulled from Athene's database on assessment submissions has been previously reported by Pettit et al. [35, 37]. However, in previous studies, students were allowed to compile offline and only submitted their code to Athene when making an attempt at correctness. While other tools discussed above capture all student compilations, the automated assessment tool used in the present study, Athene, has previously not been able to report that data. The one-on-one think-aloud study allowed this data to be gathered using Athene for the first time. The difference in student behavior when they can only use Athene to compile versus when they can compile offline is an interesting subject for a future report, but cannot be discussed fully here due to space limitations. We do expect student behavior to change when the compiling constraints change, such as an increase in the number of submissions and therefore the number of errors encountered.

For those students in the experimental section that completed the assessment during the 35 minute time limit, the average time

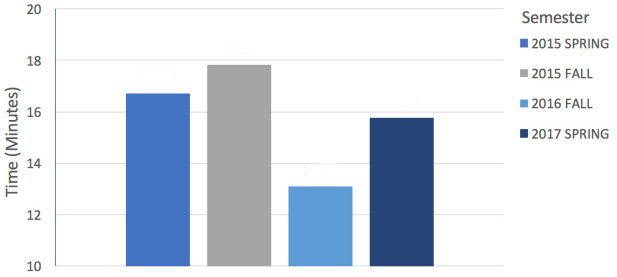

Figure 5: Average time to complete the problem by semester

to completion was 15:46 with a standard deviation of 7:03. In the control, the previous three semesters when this assessment was given the average completion times were: 16:44, 17:50, and 13:05 (Figure 5). This data indicates that the experiment did not adversely affect student outcomes.

The average score for all students in the experimental section was 67%. The average score for the previous three control semesters was 90%, 88.2%, and 84.2%. This seems to indicate that students in the experimental section may have been adversely affected. However, this may have been an artifact of the way the procedure was performed. As mentioned above, students have previously been able to compile offline and many students will use previous programs they have written as a bootstrap for any new program they attempt. In the case of the experimental group, 10 students did not complete the quiz at all, 6 of which suffered from problems with the basic structure of their code. All of these 6 students could not remember include statements and how to write their main function. If this assessment had been carried out in a previous semester, these students would have had access to previous programs and may have solved the problem. Instead, they could not move past the structural compiler errors. Furthermore, none of the structural compiler errors had been enhanced because we based our choices about which messages to enhance on the frequency with which a CEM was encountered in previous semesters. Since students in previous semesters had access to their prior programs before starting the quiz, none of these errors had been encountered in any of the control semesters. Therefore, it is interesting to note that removing these 6 students from the group brings the average score up to 84.8%, which is in range of the control semesters.

The error message quiz results above indicate that the ECEMs are more helpful than standard CEMs. However, the quantitative data from the program logs seems to contract this conclusion, or is inconclusive at best. This is where the qualitative data illuminates a possible explanation.

4.3 Think-Alouds

With regard to the errors that participants received, observational data - both spoken thought and behavior - allowed for the evaluator to be certain when ECEMs were expanded and read. An ECEM was marked as "helpful" in the observational data if the student solved that specific error or made steps towards solving it after reading the ECEM. Conversely, an ECEM was marked as "unhelpful" if the student made changes after viewing the ECEM that were not on the path to solving the error or the student read the message and didn't know how to proceed. Post-assessment ethnographic interviews and reflection revealed participants' feelings towards the ECEMs in greater depth, from gratefulness to frustration.

4.3.1 Observational. Although there were 21 students who completed the quiz and 10 students who did not complete the quiz, the total number of errors received was roughly equal at 56 for those who completed the quiz and 60 errors for those who did not, making 116 total errors tagged by evaluators. The group of participants that did not complete the quiz had a higher number of errors without enhanced messages (31) and a lower number of enhanced error messages (29), though this was dominated by a single participant who encountered the most (15). The incomplete quiz participants had under half of the number of read enhanced messages (9) when compared to the participants that completed the quiz (23). From this data it seems that encountering these messages really did prove helpful for the completion of the quiz.

The incomplete quiz participants also had over double the amount of unread enhanced messages (20) when compared to the completed quiz participants (8). For the participants that completed the quiz, there were 19 instances where the "more information" section of the ECEM proved helpful. This is over six times the amount of instances for those who did not complete the quiz (3). The incomplete quiz participants also contained more instances of unhelpful enhanced messages (6) when compared to the completed quiz participants (4).

The data presented in Figure 6 summarizes these observations and appears to indicate that the ECEMs helped students better understand the errors they were encountering, fix those errors, and ultimately complete the quiz.

4.3.2 Ethnographic: Perception of overall helpfulness comparing complete and incomplete. Of the ten students that did not solve the assessment in the 35 minute time limit, only two read the ECEMs and believed they were unhelpful. Another two students that did not complete the quiz read the ECEMs and believed them to helpful. The other six students did not receive an enhanced error message and were therefore unable to confirm whether or not the enhanced messages were helpful. See Figure 7.

4.3.3 Ethnographic: Perception of helpfulness of students with repeated error messages. There were four participants that received a repeated ECEM at least once and did not finish the quiz. One of them received three repeated ECEMs and thought that they were unhelpful. However, another one received the same ECEM ten times in succession, neglected to read the first nine, finally read it the tenth time, and subsequently corrected the error. Even though this participant did not finish the quiz, he still believed the ECEMs to be helpful. The other two participants that received repeated error messages and did not finish the quiz only received one repeated message and they both found the enhanced messages helpful.

4.4 Discussion

The results of the error message quizzes compared with the quantitative program log results from the assessment seem contradictory. The observational and ethnographic data presented above tells a different story. The students who struggled, but ultimately succeeded

Participants	# Students	# Errors	# Errors w/o Enhanced	# Errors w/ Enhanced	# Enhanced Read	# Unread	# Helpful	# Unhelpful	# Repeated
Total Complete	21	56	25	31	23	8	19	4	6
Total Incomplete	10	60	31	29	9	20	3	6	15

Figure 6: Student Perception of ECEMs in Complete vs. Incomplete Quizzes

Quiz Result	Interaction with Enhanced Messages	# of Students
Solved	Unread or Unreceived	10
Solved	Read and Helpful	9
Solved	Read and Unhelpful	2
Not Solved	Unread or Unreceived	6
Not Solved	Read and Helpful	2
Not Solved	Read and Unhelpful	2

Figure 7: Unique student groups based on quantitative and qualitative data

in completing the problem, brought down the average score and increased the average time to completion. However, these same students were helped the most by the ECEMs and expounded on this in great detail during the post-assessment interview. Although they struggled with the assessment, observational and ethnographic data shows that it was ultimately the ECEMs that helped them across the finish line. This is precisely what we want. Furthermore, a very small group of students who did not complete the quiz, and therefore brought down the average score, were not helped by the ECEMs and were frustrated by them in the post-assessment interviews. These two students were so unfamiliar with the material and so fundamentally lost that the additional information provided by the ECEMs only added insult to injury. We conjecture that the increased cognitive load of the assessment may have tipped the scales from helpful ECEMs to unhelpful.

From all of the data above, the students can be broken up into six distinct groups as seen in Figure 7.

5 CONCLUSIONS

This study has made several important contributions. First, do novice students read ECEMs? Observational and ethnographic data seem to indicate that novices in CS1 do, in fact, read ECEMs. Students also generally find the ECEMs more helpful than the standard CEMs. The corroborating evidence by Barik et al. [2] on eye tracking with intermediate students lends more weight to our finding. This helps to answer RQ1 and warrants further investigation.

Second, are ECEMs helpful for novice students in a setting with low cognitive load? The results of the error message quizzes shows a statistically significant decrease in incorrect understanding of ECEMs when compared to standard CEMs in an independent environment. This data answers RQ2.

Finally, are the ECEMs helpful for novice students in a setting with high cognitive load? Even though the quantitative program log data from the think-aloud study seems inconclusive, the qualitative data seems to indicate that ECEMs are also effective in an environment with a higher cognitive load. Students that struggled to complete the quiz successfully used the ECEMs to arrive at a correct solution. Here the qualitative think-aloud data provided a window into student behavior that the quantitative program log data could not provide. This helps to answer RQ3 and warrants further investigation.

There are several threats to validity of these findings. First, the control groups for the think-aloud study took place over multiple semesters and had two different professors. We attempted to minimize this threat by keeping the curriculum (assignments, schedule, the use of Athene, etc.) roughly the same from semester to semester. Second, control groups for the think-aloud study took the practical quiz in class, were not asked to think-aloud, and had access to previous code files to bootstrap their code. By contrast, students in the think-aloud study were in a one-on-one setting, were asked to think-aloud, and did not have access to previous code. It is possible that all of these factors increased student cognitive load in the think-aloud study and therefore skewed the results. We attempted to offset this by adding in the warm-up exercise as suggested by Teague et al. [40].

6 FUTURE WORK

This study should be replicated to further strengthen the findings presented here. Having more than one experimental group would add weight to these findings. It should also be replicated by those previous studies that found no evidence or inconclusive evidence for the helpfulness of ECEMs.

As mentioned above, Barik et al. [2] use eye tracking software to examine whether intermediate students read error messages. It would be helpful to replicate their work with novices in CS1.

Finally, even though this study followed the design guidelines by Marceau et al. [31] it would also be useful to use their rubric [30] to evaluate the helpfulness of the EECMs used in this study.

7 ACKNOWLEDGMENTS

The authors would like to thank Abilene Christian University (ACU) for providing necessary funding for this study. We would also like to thank the reviewers for their careful and thorough feedback.

REFERENCES

[1] Kirsti M Ala-Mutka. 2005. A survey of automated assessment approaches for programming assignments. *Computer science education* 15, 2 (2005), 83–102.
[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 *Proceedings of the International Conference of Software Engineering*. ACM.
[3] 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, 536–539.
[4] Brett A Becker. 2016. An effective approach to enhancing compiler error messages. In *Proceedings of the 47th ACM Technical Symposium on Computing Science Education*. ACM, 126–131.
[5] Jens Bennedsen and Michael E. Caspersen. 2007. Failure rates in introductory programming. *ACM SIGCSE Bulletin* 39, 2 (2007), 32–36.
[6] Elizabeth Carter. 2015. Its debug: practical results. *Journal of Computing Sciences in Colleges* 30, 3 (2015), 9–15.

[7] Maria Christakis and Christian Bird. 2016. What developers want and need from program analysis: An empirical study. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. ACM, 332–343.
[8] John W Creswell. 2013. *Research design: Qualitative, quantitative, and mixed methods approaches*. Sage publications.
[9] 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*. ACM, 273–278.
[10] Christopher Douce, David Livingstone, and James Orwell. 2005. Automatic test-based assessment of programming: A review. *Journal on Educational Resources in Computing (JERIC)* 5, 3 (2005), 4.
[11] 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*. ACM, 118–122.
[12] Stephen H Edwards and Manuel A Perez-Quinones. 2008. Web-CAT: automatically grading programming assignments. In *ACM SIGCSE Bulletin*, Vol. 40. ACM, 328–328.
[13] Karl Anders Ericsson and Herbert Alexander Simon. 1993. *Protocol analysis*. MIT press Cambridge, MA.
[14] Thomas Flowers, Curtis A Carver, and James Jackson. 2004. Empowering students and building confidence in novice programmers through Gauntlet. In *Frontiers in Education, 2004. FIE 2004. 34th Annual*. IEEE, T3H–10.
[15] Stephen N Freund and Eric S Roberts. 1996. Thetis: an ANSI C programming environment designed for introductory use. In *SIGCSE*, Vol. 96. 300–304.
[16] Dianne Hagan and Selby Markham. 2000. Teaching Java with the BlueJ environment. In *Proceedings of Australasian Society for Computers in Learning in Tertiary Education Conference ASCILITE 2000*. Citeseer.
[17] Björn Hartmann, Daniel MacDougall, Joel Brandt, and Scott R Klemmer. 2010. What would other programmers do: suggesting solutions to error messages. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, 1019–1028.
[18] Robert W Hasker. 2002. HiC: a C++ compiler for CS1. *Journal of Computing Sciences in Colleges* 18, 1 (2002), 56–64.
[19] Jack Hollingsworth. 1960. Automatic graders for programming classes. *Commun. ACM* 3, 10 (1960), 528–529.
[20] Maria Hristova, Ananya Misra, Megan Rutter, and Rebecca Mercuri. 2003. Identifying and correcting Java programming errors for introductory computer science students. In *ACM SIGCSE Bulletin*, Vol. 35. ACM, 153–156.
[21] Petri Ihantola, Tuukka Ahoniemi, Ville Karavirta, and Otto Seppälä. 2010. Review of recent systems for automatic assessment of programming assignments. In *Proceedings of the 10th Koli Calling International Conference on Computing Education Research*. ACM, 86–93.
[22] James Jackson, Michael Cobb, and Curtis Carver. 2005. Identifying top Java errors for novice programmers. In *Frontiers in Education, 2005. FIE'05. Proceedings 35th Annual Conference*. IEEE, T4C–T4C.
[23] 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*. ACM, 1455–1464.
[24] Steve Krug. 2014. Don't make me think revisited: A common sense approach to web and mobile usability. (2014).
[25] Michael J Lee and Andrew J Ko. 2011. Personifying programming tool feedback improves novice programmers' learning. In *Proceedings of the seventh international workshop on Computing education research*. ACM, 109–116.
[26] Yvonna S Lincoln and Egon G Guba. 1985. *Naturalistic inquiry*. Vol. 75. Sage.
[27] Derrell Lipman. 2014. LearnCS!: a new, browser-based C programming environment for CS1. *Journal of Computing Sciences in Colleges* 29, 6 (2014), 144–150.
[28] Dastyni Loksa, Andrew J Ko, Will Jernigan, Alannah Oleson, Christopher J Mendez, and Margaret M Burnett. 2016. Programming, Problem Solving, and Self-Awareness: Effects of Explicit Guidance. In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*. ACM, 1449–1461.
[29] John Maloney, Mitchel Resnick, Natalie Rusk, Brian Silverman, and Evelyn Eastmond. 2010. The scratch programming language and environment. *ACM Transactions on Computing Education (TOCE)* 10, 4 (2010), 16.
[30] 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*. ACM, 499–504.
[31] Guillaume Marceau, Kathi Fisler, and Shriram Krishnamurthi. 2011. Mind your language: on novices' interactions with error messages. In *Proceedings of the 10th SIGPLAN symposium on New ideas, new paradigms, and reflections on programming and software*. ACM, 3–18.
[32] 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. In *ACM SIGCSE Bulletin*, Vol. 40. ACM, 163–167.
[33] Marie-Hélène Nienaltowski, Michela Pedroni, and Bertrand Meyer. 2008. Compiler error messages: What can help novices?. In *ACM SIGCSE Bulletin*, Vol. 40. ACM, 168–172.
[34] Michael Quinn Patton. 2014. *Qualitative research & evaluation methods: Integrating theory and practice*. Sage.
[35] Raymond Pettit, John Homer, Roger Gee, Susan Mengel, and Adam Starbuck. 2015. An empirical study of iterative improvement in programming assignments. In *Proceedings of the 46th ACM Technical Symposium on Computer Science Education*. ACM, 410–415.
[36] Raymond Pettit and James Prather. 2017. Automated Assessment Tools: Too Many Cooks, Not Enough Collaboration. *J. Comput. Sci. Coll.* 32, 4 (April 2017), 113–121. http://dl.acm.org/citation.cfm?id=3055338.3079060
[37] Raymond S Pettit, John Homer, and Roger Gee. 2017. Do Enhanced Compiler Error Messages Help Students?: Results Inconclusive.. In *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education*. ACM, 465–470.
[38] Jeffrey Rubin and Dana Chisnell. 2008. *Handbook of usability testing: how to plan, design and conduct effective tests* (2 ed.). John Wiley & Sons.
[39] Tom Schorsch. 1995. CAP: an automated self-assessment tool to check Pascal programs for syntax, logic and style errors. In *ACM SIGCSE Bulletin*, Vol. 27. ACM, 168–172.
[40] Donna Teague, Malcolm Corney, Alireza Ahadi, and Raymond Lister. 2013. A qualitative think aloud study of the early neo-piagetian stages of reasoning in novice programmers. In *Proceedings of the Fifteenth Australasian Computing Education Conference-Volume 136*. Australian Computer Society, Inc., 87–95.
[41] Dwayne Towell and Brent Reeves. 2009. From Walls to Steps: Using online automatic homework checking tools to improve learning in introductory programming courses. (2009).
[42] V Javier Traver. 2010. On compiler error messages: what they say and what they mean. *Advances in Human-Computer Interaction* 2010 (2010).
[43] Christopher Watson, Frederick WB Li, and Jamie L Godwin. 2012. Bluefix: Using crowd-sourced feedback to support programming students in error diagnosis and repair. In *International Conference on Web-Based Learning*. Springer, 228–239.
[44] Richard L Wexelblat. 1976. Maxims for malfeasant designers, or how to design languages to make programming as difficult as possible. In *Proceedings of the 2nd international conference on Software engineering*. IEEE Computer Society Press, 331–336.
[45] Jacqueline Whalley and Nadia Kasto. 2014. A qualitative think-aloud study of novice programmers' code writing strategies. In *Proceedings of the 2014 conference on Innovation & technology in computer science education*. ACM, 279–284.
[46] Timothy T Yuen. 2007. Novices' knowledge construction of difficult concepts in CS1. *ACM SIGCSE Bulletin* 39, 4 (2007), 49–53.