Beyond Automated Assessment:
Building Metacognitive Awareness in Novice Programmers in CS1

by

James Prather

A dissertation submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy
in
Computer Science

College of Engineering and Computing
Nova Southeastern University

2018

We hereby certify that this dissertation, submitted by James Prather, conforms to acceptable standards and is fully adequate in scope and quality to fulfill the dissertation requirements for the degree of Doctor of Philosophy.

Maxine S. Cohen, Ph.D.
Chairperson of Dissertation Committee

3/2/2018
Date

Raymond Pettit, Ph.D.
Dissertation Committee Member

3/2/2018
Date

Michael J. Laszlo, Ph.D.
Dissertation Committee Member

3/2/2018
Date

Approved:

Yong X. Tao, Ph.D., P.E., FASME
Dean, College of Engineering and Computing

3/2/2018
Date

College of Engineering and Computing
Nova Southeastern University

2018

An Abstract of a Dissertation Submitted to Nova Southeastern University
in Partial Fulfillment of the Requirements for the Degree of Doctor of Philosophy

Beyond Automated Assessment:
Building Metacognitive Awareness in Novice Programmers in CS1
by
James Prather
March 2018

ABSTRACT

The primary task of learning to program in introductory computer science courses (CS1) cognitively overloads novices and must be better supported. Several recent studies have attempted to address this problem by understanding the role of metacognitive awareness in novices learning programming. These studies have focused on teaching metacognitive awareness to students by helping them understand the six stages of learning so students can know where they are in the problem-solving process, but these approaches are not scalable. One way to address scalability is to implement features in an automated assessment tool (AAT) that build metacognitive awareness in novice programmers. Currently, AATs that provide feedback messages to students can be said to implement the fifth and sixth learning stages integral to metacognitive awareness: implement solution (compilation) and evaluate implemented solution (test cases). The computer science education (CSed) community is actively engaged in research on the efficacy of compile error messages (CEMs) and how best to enhance them to maximize student learning and it is currently heavily disputed whether or not enhanced compile error messages (ECEMs) in AATs actually improve student learning. The discussion on the effectiveness of ECEMs in AATs remains focused on only one learning stage critical to metacognitive awareness in novices: implement solution. This research carries out an ethnomethodologically-informed study of CS1 students via think-aloud studies and interviews in order to propose a framework for designing an AAT that builds metacognitive awareness by supporting novices through all six stages of learning.

The results of this study provide two important contributions. The first is the confirmation that ECEMs that are designed from a human-factors approach are more helpful for students than standard compiler error messages. The second important contribution is that the results from the observations and post-assessment interviews revealed the difficulties novice programmers often face to developing metacognitive awareness when using an AAT. Understanding these barriers revealed concrete ways to help novice programmers through all six stages of the problem-solving process. This was presented above as a framework of features, which when implemented properly, provides a scalable way to implicitly produce metacognitive awareness in novice programmers.

Acknowledgements

So many people have contributed to my success in completing this dissertation that it will be difficult to name them all. First and foremost, my wonderful wife, Erin, has been my rock and defender during these arduous years. Her unflagging and constant support as I pursued my dreams, even when it was very hard on her, was what made this possible from beginning to end. To my children, Zeke, Micah, and Eve, who will mostly not remember all the nights that daddy had to go back work after dinner, I would like to say thanks for your patience, even when you didn't entirely understand why I was absent so often. I would also like to thank my sister, Shelley Doremus and her husband, Brian, who together helped take care of the kids on those many days and nights that I was gone. My parents, Jim and Fran, were also a great source of encouragement and help. From my earliest days, I can remember my mother instilling in me the sense and pride in attaining a high level of education. My father continues to be an inspiration to me in his life's dedication to teaching. I would also like to thank my wife's parents, Ken and Tami Beach, who allowed me to work on this dissertation for weeks at a time in their basement while they entertained grandkids above. Truly, this dissertation was an exercise in village life.

There are many others who deserve my immense gratitude. My colleagues, Dr. John Homer, Dr. Brent Reeves, Dr. Ryan Jessup, and Dr. Brad Crisp, all helped me through my doctoral work in some way, including helping to sharpen ideas for, or proofing the various conference papers, that would form the body of this dissertation. Thank you to Dr. Rick Lytle who called me out and made possible my pursuit of the Ph.D. in Computer Science. Thanks also to my very talented and capable research assistants, Kayla McMurry and Alani Peters, without whom I would not have been able to complete this project so quickly and who both worked very hard and frequently went above and beyond. I am eternally grateful to my dear friend and colleague, Dr. Robert Nix, for his constant help and support at all levels of my education. I also had the help and support of friends in my NSU cohort, David Fleig, Garret Moore, and Ben Geisler. Thanks also to my friends Zach Fisher, Jarrod Jones, Aaron Gamble, Ryan Garner, Kipp Swinney, and Noemí Palomares for their constant friendship and encouragement. I am truly blessed to call you friends.

Several mentors have shaped me into a scholar and refined my capabilities. Thank you to Erika Orrick who first taught me to love the study of Human-Computer Interaction. My gratitude is also extended to Dr. Curt Niccum for mentoring and shaping me into the scholar I am today. And thank you to Dr. Jeff Childers who took me under his wing, trained me, and taught me what it means to pursue the life of a scholar while also being a family man and mentor to others.

Finally, my deepest thanks go to my committee. My gratitude goes out to Dr. Michael Laszlo for taking on this project and providing his excellent feedback. My sincere thanks also go to committee member and friend, Dr. Ray Pettit, who walked alongside of me during the entire process and provided detailed feedback along every step of the way. Finally, I am most grateful to my dissertation chair, Dr. Maxine Cohen, for taking me on as her student, for her careful shepherding of the project, for her flexibility in how it was carried out, and for her careful attention to detail with each iteration of the dissertation. It was a delight to work with her.

This dissertation is dedicated to my daughter, Micah. May the world see the way you glorify God and say in awe, "Who is like you, oh Lord?"

Table of Contents

List of Tables

Tables

List of Figures

Figures

Chapter 1

Introduction

Background

In a 2008 Australasian Computing Education conference (ACE) keynote paper, Lister grappled with the complete lack of a robust scholarship of teaching in computer science. Instead, he argued that most professors use "folk pedagogies" out of their own experiences as a student or their experiences as a professor of what has and has not worked in the classroom. In doing so, many professors who teach computer science and related subjects base their pedagogy on assumptions that have been shown to be false such as how novices read, interpret, and understand code. This is highly damaging to the students who fail, or worse, those who pass but do not form the correct cognitive models of programming as the base upon which to build further content. When programming is taught incorrectly, it leads to high failure rates and eventually a shifting of students away from the discipline, such as in the early 2000's. Eventually, Lister argued, the undergraduate students majoring in computer science shrinks, leading to fewer graduate students, leading to fewer faculty positions. Lister tracked this boom-to-bust cycle and targets folk pedagogy as one contributing factor. To combat this, Lister suggested computer science teachers must understand how novices in this field learn and then build classroom instruction around that process. He suggested that not only will it lower the failure rate of introductory courses, but it could also save the discipline entirely from collapsing upon itself (Lister, 2008). A little dramatic, perhaps, but his points are well made.

So how can one follow Lister's advice? He suggested turning to learning theory. Novices and experts alike only have a certain amount of "working memory," also called short-term memory. Miller (1956) famously quantified human short-term memory capacity: seven, plus or minus two chunks. A chunk represents any bit of information, such as a digit of a telephone number or one step in a series of directions. However, the *way* in which these chunks are stored and retrieved is different in experts and novices who draw relationships between items differently. Experts organize their knowledge in much more complex ways than novices (Chi et al., 1988; Ericsson & Smith, 1991) which Lister described as relating directly to computer programming. Therefore, a chunk for a novice reading previously unseen code may be one single statement or line while a chunk for an expert might be a set of lines or an entire function (Lister, 2008). This relates directly to Cognitive Load Theory developed by Sweller (1999) which is a theory of human learning where each person has a certain cognitive processing limit and if overwhelmed the person will cease to learn or understand. This means not overburdening a novice's cognitive load, which is probably somewhere around seven chunks, when learning new concepts. Mark Guzdial (2015a) wrote that the usual teaching method of introductory computer science courses – writing programs from scratch, also called the "constructivist" approach – was overwhelming the cognitive load of novices. In other words, asking students to learn by doing what experts do is ineffectual instruction. Furthermore, the kinds of feedback that novices receive when incorrect is often cryptic and built for experts and professionals in the field, which also or further overwhelms their cognitive load.

The constructivist approach to teaching novices programming is unhelpful at the very least, but there are even more barriers to teaching an excellent CS1 course. In the March 2015 issue of *Communications of the ACM*, Guzdial reflected on a blog post by Ko (2014) who argued that programming languages are the least usable interfaces ever created and therefore their learnability is incredibly low. Guzdial listed multiple barriers in the way of people learning to effectively use programming languages, such as esoteric features with a high cognitive load and a low expected payout. He ended with a call to educators: "We improve the usability and learnability of our programming languages by working with our users, figuring out what they want to do, and help them to do it" (Guzdial, 2015b, p.1). Computer science educators have attempted to lower cognitive load in several ways, such as using graduated exposure to programming concepts (Gray, 2007), changing the modality in which students receive programming instruction (Morrison, 2016), and using exercises such as Parsons Problems (Karavirta, 2012; Morrison, et al., 2016). Another approach to lowering cognitive load is to support users in creating a better cognitive model of programming languages as they learn. This is because learning how to code is more than just syntax and data structures, but also about assisting the novice in building a mental scaffold around which they can correctly place knowledge and develop metacognitive awareness (Eteläpelto, 1993; Roll et al., 2012; Mani & Mazumder, 2013).

Metacognitive awareness is the ability to not only understand the problem but also understand *where* one is in the problem-solving process and the ability to reflect on that state. In his 1945 seminal book, Polya identified multiple stages that learners move through while solving a math problem, hoping to make learners more explicitly aware of

their movement through these stages (Polya, 2014). Dijkstra attempted to affect this process in his students, saying, "I want you to gain, for the rest of your lives, the insight that beautiful proofs are not 'found' by trial and error but are the result of a consciously applied design discipline." (Dijkstra, 1995, p.1). Most recently and most relevant to this research, Loksa et al. applied a similar framework of metacognitive awareness to novices learning programming. They identified six specific stages in learning to code of which students should be aware in order to understand where they are in the problem-solving process: (1) reinterpret the prompt, (2) search for analogous problems, (3) search for solutions, (4) evaluate a potential solution, (5) implement a solution, and (6) evaluate implemented solution (Loksa et al., 2016). See Table 1 for how Loksa's learning stages roughly correspond to Polya's. The approach of Loksa et al. was to coach students on these stages and help them identify which stage they were in when they became stuck, although this approach is difficult to scale, such as in a massively open online course, where individual interaction with a teacher is not possible for everyone to receive.

Table 1. *Learning stages by Polya and Loksa, roughly correlated.*

Polya's Stages (Polya, 2014)	Loksa's Stages (Loksa et al., 2016)
1. Understand the problem	1. Reinterpret the prompt
	2. Search for analogous problems
2. Devise a plan	3. Search for solutions
	4. Evaluate a potential solution
3. Carry out the plan	5. Implement a solution
4. Look back on your work	6. Evaluate implemented solution

One way to address the scalability issue found in the approach of Loksa et al. is to use an automated assessment tool (AAT), which grades student programming assignments, to implement features that help novices through all six stages of learning and therefore build metacognitive awareness into the process of using the tool itself. AATs have a long history of development starting in 1960 and continuing up to the present (Douce et al., 2005; Ihantola et al., 2010; Pettit et al., 2012; Pettit & Prather, 2017). These tools currently only truly support the fifth and sixth stage discussed by Loksa et al. (2016). The fifth stage, implement solution, is supported via compile error message (CEM) feedback when a student submits their assignment and it contains compilation errors. The sixth stage, evaluate implemented solution, is sometimes supported as well via feedback messages when a student's program fails a particular test case. Some have attempted to create enhanced compiler error messages (ECEMs) to be more helpful and decrease student error rate, but most of these have done so based on what Lister terms folk pedagogy instead of empirical data. The few who have offered empirical data have shown mixed results and it is currently disputed in the literature whether ECEMs are effectual or not. Some have argued that ECEMs have no effect on student learning (Denny et al., 2014; Pettit et al., 2017). However, one very thorough study counters these claims (Becker, 2016a) and has generated substantial discussion in the computer science education community (Becker, 2016b; Guzdial, 2014). Some have also attempted to understand these messages from the perspective of the users, as Guzdial suggested, performing human-factors studies on error messages and novice interaction with them (Nienaltowski et al., 2008; Hartmann et al., 2010; Marceau et al., 2011b).

Discussion in the literature of how to use an AAT to guide students through the other five learning stages is non-existent.

Problem Statement

Learning to program is a hard task and novices are constantly cognitively overburdened (Lister, 2008; Guzdial, 2015a). This can be alleviated by supporting novices in building cognitive scaffolding and metacognitive awareness through six distinct learning stages (Loksa et al., 2016). A scalable implementation of the method of Loksa et al. would be to use AATs, which many universities are already using to help students learn programming and is therefore a somewhat ubiquitous place to start. Some AATs have been improved to support the fifth learning stage by providing usable feedback for student program submissions. A few studies have attempted to approach the design of these feedback messages from a usability or human-factors perspective (Nienaltowski et al., 2008; Hartmann et al., 2010; Marceau et al., 2011a). However, it is currently debated in the literature whether enhancing compiler error message feedback empirically improves student learning (Denny et al., 2014; Guzdial, 2014; Becker 2016a; Pettit et al., 2017). However, there is no discussion in the literature on implementing in AATs the means to help students through the other five learning stages.

Dissertation Goal

The goal of this research is to propose a framework for improving metacognitive awareness through AATs. First, since compiler error feedback in AATs is the only learning stage currently discussed by the literature, the feedback messages in an AAT

were enhanced according to current best practices and iterative testing. Then the AAT with enhanced feedback messages was tested by novices in a CS1 course via an ethnomethodologically-informed study utilizing a usability test with a think-aloud protocol and post-testing interviews. Qualitative analysis revealed the difficulties that novice programmers faced in developing metacognitive awareness. Finally, these difficulties informed a proposal, based on the analysis of the quantitative and qualitative data collected during the study, for implementing in an AAT features that can positively impact metacognitive awareness.

Research Questions

By the time a novice programmer in CS1 submits their code to the AAT, they have already mentally crossed five hurdles and are ready to evaluate their potential solution. However, many students who get this far still fundamentally misunderstand the problem they've been asked to solve or have implemented an incorrect solution, something generic error messages cannot usually correct. Moreover, students often have no idea where in the problem-solving process they actually are (i.e. they lack metacognitive awareness) and therefore feel as if they are close to a solution when they might have diverged at the very first learning stage.

Given these issues and the discussion in the literature surrounding feedback messages in AATs, this study asks the following research questions:

RQ1. When students diverge on a specific learning stage, what factors caused them to do that?

RQ2. Are ECEMs helping students evaluate their potential solution?

RQ2.a. Are students reading the enhanced messages?

RQ2.b. If students are reading the enhanced messages, how do the enhanced messages help them better understand the error?

RQ3. When students diverge on a specific learning stage, submit their program, and receive an ECEM, how do they interpret it?

RQ4. How can AATs be augmented to support metacognition in novice programmers in CS1?

Relevance and Significance

While there has been substantial discussion of the role of metacognition in learning to program (Eteläpelto, 1993; Roll et al., 2012; Mani & Mazumder, 2013; Loksa et al., 2016), and even suggestions as to how to apply them to intelligent tutoring systems for geometry (Roll et al., 2011), there has yet to be a framework for implementing them in AATs for introductory programming courses. The present research project proposes such a framework. As a starting place, this work picks up where the computer science education (CSed) community is currently discussing the only related piece of metacognition in AATs: enhanced compiler error messages. By beginning there, this study provides additional evidence as to the efficacy of ECEMs. Whether ECEMs provide no impact on student learning (Denny et al., 2014; Pettit et al., 2017) or do provide a positive impact (Becker, 2016), this additional evidence is an important step forward in understanding the role of ECEMs in metacognitive awareness in AATs. The current research project then expands to discuss the other five learning stages and how AATs can support students through each one.

Barriers and Issues

Since this study involves human participants, the main barrier is IRB compliance. This is overcome through a combination of IRB application for some portions of the study and classroom enhancement for other parts (see Appendix F). The primary issue for this study is in determining significance because only one AAT will be modified and tested. This is overcome through careful consideration of *what* was tested and how that may be compared to *what* was tested in other AATs. It is also overcome through the impact the present research study has already had on the literature through publication. A secondary issue is with sample size because this study will be carried out at one university where the introductory computer science course (CS1) is typically 30 to 40 students. This is overcome by comparing data from the past several semesters of CS1 courses at the same university where the exact same problem was used.

Assumptions

It is assumed that participants will provide honest feedback in ethnographic interviews. It is also assumed that participants will attempt to work hard at the provided task during the usability studies. Finally, it is assumed that observing participants in the lab can approximate their actual behavior wherever they usually work on their programming assignments. This last assumption is discussed in more detail in Chapter 2.

Limitations and Delimitations

The first limitation that potentially impacts validity involves the fundamental choices in CS1 curriculum underlying this study. Since the author is a computer science professor at Abilene Christian University (ACU), there are several factors beyond the author's control. The first is the programming language that is used, C++, was chosen by the department, not the professor. Much of the literature involving AATs uses Java and so there are some comparison issues. The second factor beyond control is the AAT that is used in CS1 at ACU, called Athene, is also selected by the department. Most of the literature discusses AATs that force students to compile inside of the AAT so that all behavior is captured. Athene is a service that only takes submissions which means that student behavior between submissions is not captured. This makes comparing results between Athene and many other tools a potential threat to validity. A related limitation is the way that data has been collected within the AAT. The AAT was created in-house eight years prior to commencing work on the present research project and the author had no control over what data was collected and how it was stored.

The first delimitation of this study is that it takes place in a single university computer science program. This naturally limits the sample size, but constrains the issues at hand for the sake of simplicity and for appropriate comparison. Performing the study at multiple universities would require controlling for differences in curriculum, academic preparation of students which can vary from university to university, and teaching style of professors in different departments where culture and values may differ.

List of Acronyms

AAT: Automated Assessment Tool

ACU: Abilene Christian University

CEM/ECEM: Compiler Error Message / Enhanced Compiler Error Message

CS1: Computer Science 1, the first course in a computer science curriculum

CSed: Computer Science Education

HCI: Human-Computer Interaction

UI: User Interface

UX: User Experience

Definition of Terms

Definitions of important terms used in this research are given below:

Athene – An automated assessment tool written by Dwayne Towell in 2009 that is currently in use at Abilene Christian University (Towell & Reeves, 2009).

Automated assessment tool – A tool that allows users to submit programs a receive instant feedback on syntax and correctness (Ihantola et al., 2010).

Cognitive overload/cognitive load theory – A theory of human learning where each person has a certain cognitive processing limit and if overwhelmed the person will cease to learn or understand (Sweller, 1999).

Compiler Error Message – The standard error message that is provided by the compiler by default (Becker, 2015).

Constructivist approach – An approach to teaching computer science in which it is thought that the most effective way for students to learn is by constructing their own solutions, from scratch, to programming problems (Lister, 2008).

Enhanced Compiler Error Message – A compiler error message that has been edited/updated to provide more information to the user, clarify difficult terminology, and provide feedback about how to fix the error (Becker, 2015).

Ethnographic study – Taken from the social sciences, specifically anthropology, ethnography is a practice that attempts to observe and understand human behavior, beliefs, and institutions (Angrosino, 2007).

Folk pedagogies – An approach to teaching that is "a mix of an oral tradition handed down by more experienced colleagues and [one]'s own intuitions about what would help the students, which [is] often a reflection of what had worked…when [one] was a student" (Lister, 2008, p. 5).

Mental/conceptual model – A user's perception of how the system works and why it works the way that it does (Norman, 2013).

Mental scaffold/cognitive scaffold – Providing help, tools, and feedback to learners while they move from novice to expert user (Eteläpelto, 1993).

Metacognitive awareness – The state of awareness of the problem, the problem-solving process, and where one is currently in that process (Loksa et al., 2016).

Post-testing interviews – After observing behavior during a usability test, researchers listen to users' perceptions of the task(s) by asking a set of focused questions (Miller & Crabtree, 1999).

Think aloud protocol/think aloud study – A study where a participant is asked to perform a task and to vocalize their thoughts while doing it (Ericsson and Simon, 1993).

Usability study/tests – Typically performed in a laboratory setting, usability tests "are about watching one person at a time ty to use something (whether it's a Web site, a prototype, or some sketches of a new design) to do typical tasks so you can detect and fix the things that confuse them or frustrate them" (Krug, 2014, p. 113).

Summary

The primary task of learning to program in CS1 courses cognitively overloads novices and must be better supported. Rather than relying on folk pedagogies to improve student learning, this must involve verifiable data collected through quantitative or qualitative research. One way that researchers are already trying to alleviate cognitive overload in CS1 students is by improving cognitive scaffolding and metacognitive awareness via enhancing the compiler error message feedback they receive in AATs. However, this only supports one stage of the learning process. Therefore, a robust framework for implementing features in an AAT that will improve students' metacognitive awareness through all learning stages is needed. By carrying out an ethnographic study of CS1 students via usability studies and post-testing interviews, this study provides such a framework.

Chapter 2

Review of the Literature

Introduction

This chapter first discusses the relevant literature surrounding automated assessment tools (AATs), specifically looking at feedback in AATs. This discussion is further narrowed to compiler error messages and the various attempts to enhance those messages for the sake of readability, understandability, and learnability for novice programmers. While many have written about the problem of enhancing compiler error messages, only a few researchers have approached the problem from a human-factors perspective. Similarly, only a small group of researchers have provided empirical data about the effectiveness of their attempts to enhance compiler error messages. The present research project intends to approach the problem from a human-factors standpoint and provide empirical data from the study. Therefore, it is necessary to examine this literature.

This chapter also examines relevant literature about the research methods used in the present research project, namely usability evaluations, think-aloud studies, and ethnography. Understanding how previous studies have used – and, in some cases, misused – these tools will provide insight for building the methodology of the present research project.

Automated Assessment Tools

Introduction

From very early in the discipline, computer scientists were attempting to find a way to write programs that could evaluate and grade other programs. In a much earlier volume of *Communications of the ACM*, Hollingsworth (1960) discussed his card-based FORTRAN homework grader. His assessment was that students learned more and learned faster with the help of his AAT. The grader was very rudimentary and technological limitations meant the computer would stop during run-time errors such as buffer overflows. Regardless of the problem in their code, it would always return one of three possibilities: nothing (stopped execution before reaching termination), "WRONG ANSWER," or "PROBLEM COMPLETE." One of the first modern AATs is described by Schorsch (1995), written to help students writing in Pascal at the United States Air Force Academy. Schorsch created the Code Analyzer for Pascal (CAP) which would find syntax, logic, and infinite loop errors and report them to students through a graphical user interface. At the end of the course, Schorsch provided the students with a questionnaire to attempt to find out if they felt using CAP helped them better learn how to program. Many students responded favorably, but Schorsch offers no metric or data to quantify the gains from using CAP. Contrary to student perception of CAP, Schorsch reported that he could tell that many students did not even read the feedback messages CAP provided, no matter how user-friendly they were made to be. Instead of learning how to program better, many students began using CAP as a crutch.

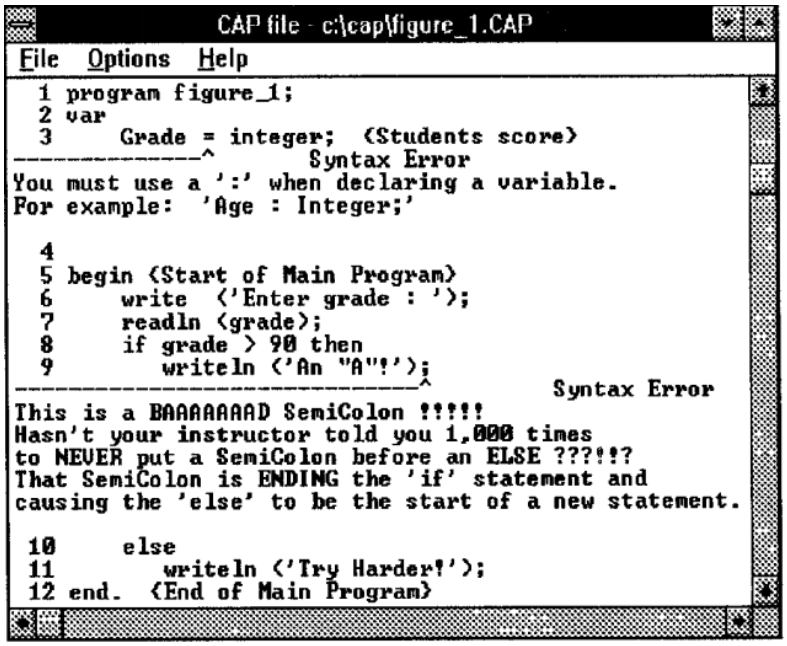


Figure 1. Feedback message from CAP (Schorsch 1995, p. 169).

Following in the footsteps of these early pioneers, Venables and Haywood (2003) identified the importance of feedback for the learnability of programming and created their own system for grading programs written in Java. Unfortunately, they provided no data to support the premise that students were learning from error messages. Rather, they focused on the results that faculty and staff had less work and therefore enjoyed using the AAT. Venables and Haywood found the same worrisome conclusion of Schorsch: students began to modify code to fit the expectations of the AAT, rather than do it right the first time. Many other studies, such as Joy and Luck (1998) and Foxley et al. (2001) have reported on similar attempts with mostly similar results.

Many have recognized the importance of proper feedback in any learning experience. Don Norman (2013) writes that feedback is an essential piece to bridging the "gulf of evaluation" where a user determines how to interpret the results of their actions. Feedback, then, is a fundamental piece in the learnability of any task, including

programming languages. If the research community is to answer Guzdial's call to bridge this gulf, then it would seem feedback is the natural choice. However, as is evident from the brief discussion above, one consistent result looms over all research into the use of AATs in computer science education: automated feedback has not definitively been proven to help students learn. This would lead to two possible conclusions. The first is that feedback in AATs is genuinely not helpful. If so, how could such a fundamental part of learning be unnecessary? The second is that perhaps it has not been fully understood from the perspective of the discipline of Human-Computer Interaction. These two possibilities are explored below.

Feedback in AATs

Several modern articles on the *status quaestionis* of AATs have presented readers with excellent reviews of past tools and their capabilities. These studies identify the importance of feedback in learning and assume that instant feedback is implicitly helpful, yet most do not provide quantitative data on the efficacy of these feedback messages (Douce et al., 2005; Pears et al., 2007; Ihantola et al., 2010). Some do not agree that instant feedback is helpful for students (Butler et al., 2007).

The first study to focus exclusively on utilizing feedback in an AAT as a primary means of enhancing student learnability is Flowers et al. (2004). At the time of the study, the United States Military Academy at West Point required all freshmen to take an introductory programming course using Java. Instructors found that most students repeated the same minor syntax errors throughout the entire semester and created Gauntlet, a useful, easy to use, and often humorous AAT. By replacing cryptic compiler

messages with more user-friendly feedback, they claim that student performance dramatically increased (Flowers et al., 2004). Unfortunately, this claim is only substantiated anecdotally. If the perception of improvement by the instructors was correct, then it could be attributed to two separate but related factors. The first is that better error messages truly helped students learn from their mistakes and freed them from their syntax woes to focus on problem-solving skills. The second is that an automated tool allowed otherwise less talented or less motivated students to succeed by consistently freeing them from the responsibility of catching – and learning to move past – novice errors.

"You have misspelled printLine on line 22.
The method is spelled with a capital L
like the one on your forehead."
"You have learned well grasshopper.
Your code has passed the Gauntlet. You
may walk across the rice paper and leave
no footprints."

Figure 2. Two separate enhanced feedback messages from Gauntlet (Flowers 2004, p. 12)

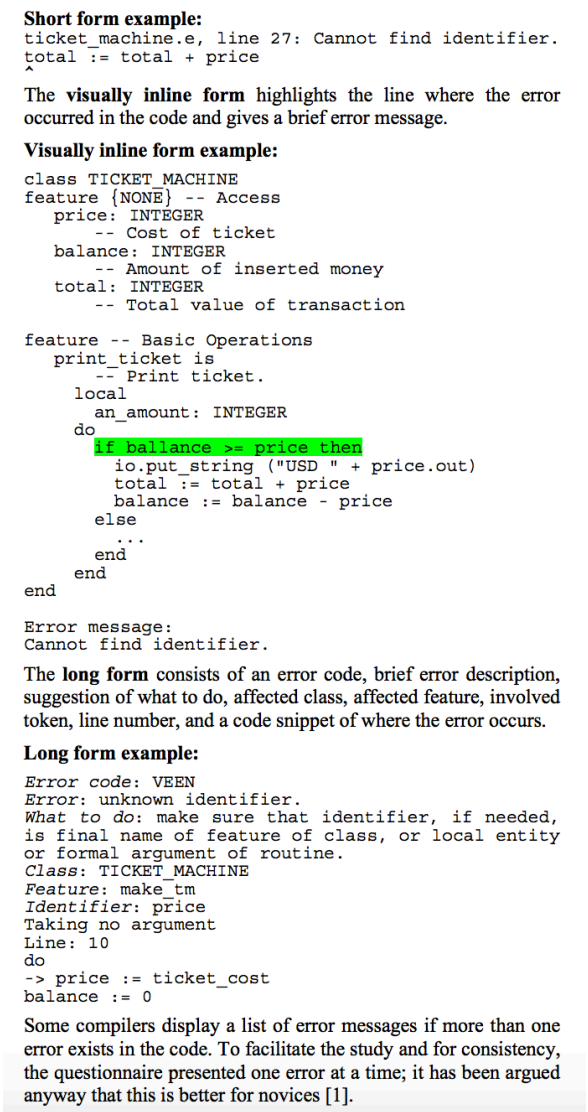
The first study to provide quantitative data on student error before and after the introduction of an AAT was Jadud (2005). Jadud introduced the tool BlueJ and recorded time between compilation and the number of errors at each compilation in order to explore what he calls "compilation behavior." The results were encouraging: student error rates dropped considerably after the introduction of the tool. Jadud felt that his data had gone a significant way toward describing the behavior of novice programmers and next wonders how educators might use this data to *shape* that behavior. However, Jadud rightly described the issue with using his study to infer anything about the causes of the

recorded student behavior. He writes: “Even if changes such as those proposed appeared to ‘improve’ novice programmer behaviour in some way, we don't want to condition students the way Tom Schorsch (1995) and his colleagues did in 1995 at the United States Air Force Academy,” (Jadud, 2005, p. 37) indicating the consistent fear that the tool is not enhancing learning but rather impeding it.

Many have reported similar results with an AAT. For instance, Nordquist (2007) did not address how automated feedback impacted student learnability, other than to mention that in an anonymous survey the students felt it helped. More likely, however, is the author’s instinct that students probably learned how to guess the test cases and play the system. In another case, Sherman et al. (2013) introduced an AAT into their introductory programming courses with one group using the vanilla version and another using a version with custom feedback messages. They found an increase in student submissions in the test group that had access to the AAT running the feedback messages, writing that students were “leveraging feedback to improve their programs” (Sherman et al., 2013, p. 1). While this sounds positive, it is once again possible that students were taught a reliance on a system and that their learning was not enhanced through feedback messages.

Nienaltowski et al. (2008) was the first to provide quantitative data on whether feedback messages increased learnability of programming languages for novices. Relevant to this study are three of their surprising findings. The first is that students with less prior programming experience did not benefit more from the enhanced feedback messages than students with more experience. Second, more information provided to students did not result in more correct answers. Third, novice students responded much

better to both long- and short-form error messages, but responded poorly to more visual or picture-related formats. The result that longer and more detailed feedback did not correlate to significantly improved student learning is troubling. One possible explanation for these findings which was not discussed in their study is that students did not benefit from the feedback messages because they did not read them. This possibility is striking because students often report very positive perceptions of AATs (Holton & Wallace, 2013; Rubio-Sánchez et al., 2014).



Short form example:

```
ticket_machine.e, line 27: Cannot find identifier.
total := total + price
^
```

The **visually inline form** highlights the line where the error occurred in the code and gives a brief error message.

Visually inline form example:

```
class TICKET_MACHINE
feature {NONE} -- Access
   price: INTEGER
       -- Cost of ticket
   balance: INTEGER
       -- Amount of inserted money
   total: INTEGER
       -- Total value of transaction

feature -- Basic Operations
   print_ticket is
       -- Print ticket.
     local
       an_amount: INTEGER
     do
       if ballance >= price then
         io.put_string ("USD " + price.out)
         total := total + price
         balance := balance - price
       else
         ...
       end
     end
end

Error message:
Cannot find identifier.
```

The **long form** consists of an error code, brief error description, suggestion of what to do, affected class, affected feature, involved token, line number, and a code snippet of where the error occurs.

Long form example:

```
Error code: VEEN
Error: unknown identifier.
What to do: make sure that identifier, if needed,
is final name of feature of class, or local entity
or formal argument of routine.
Class: TICKET_MACHINE
Feature: make_tm
Identifier: price
Taking no argument
Line: 10
do
-> price := ticket_cost
balance := 0
```

Some compilers display a list of error messages if more than one error exists in the code. To facilitate the study and for consistency, the questionnaire presented one error at a time; it has been argued anyway that this is better for novices [1].

Figure 3. Short form, visually inline form, and long form examples of enhanced compiler error message feedback from the AAT by Nienaltowski et al. (2008, p. 169).

Empirical Arguments: Conflicting Reports on the Helpfulness of Enhanced Feedback

The first to attempt an empirical study on whether enhanced feedback messages increases learnability of programming languages in an introductory course was Denny et al. (2014) using their own AAT, CodeWright. The enhanced feedback consisted of the offending line of code, a description of the error, a code block that contained a similar error, the same code block with that error fixed, and a discussion of what was fixed and why. They quantitatively analyzed student submissions and discovered that there was no statistically significant difference between the experimental and control groups. Furthermore, they found that enhanced feedback did not affect the average number of compiles needed to overcome any common error. What could explain their results? Why would students not benefit from such a fundamental piece of learnability? They present several possibilities. The first is that the types of errors which novices typically commit may be simple enough that the generic compiler error message may already provide adequate information. The second possibility they propose, already noted by Schorsch (1995), is that students did not read the enhanced error messages because of their verbosity. This second possibility seems more likely. They close by calling for a rigorous human factors study on how students are using these enhanced feedback messages in order to determine why they are not helping.

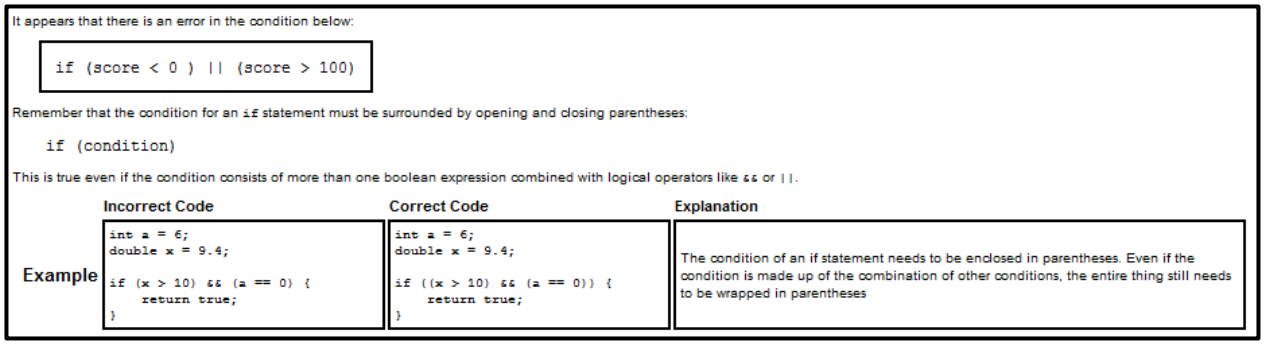


Figure 4. Enhanced compiler feedback message from Denny et al. (2014, p. 277).

Directly conflicting the findings of Denny et al. is the work of Becker (2016). Becker employed Decaf, a Java editor that presented students with enhanced compiler messages alongside the standard message. Two student groups, each with just over 100 students, were tested over a period of four weeks. The two groups were from separate consecutive academic years. There were three important results from this study. The first is that the overall number of student errors was lower for the group with the enhanced messages. Second, the number of errors per student was not lower with the group that received the enhanced messages. However, when the data are constrained to the top 15 errors, then the experimental group made less errors per student. Third, the number of consecutive error messages that students received was much lower in the group that received the enhanced messages. Becker's study was thorough and he carefully statistically compared his work to those that have gone before, leaving little doubt in its validity. The study also generated considerable discussion in the CSed community. Guzdial wrote in a blog post, "Is it really the case that enhancing error messages doesn't help students? Yes, if you do an ineffective job of enhancing the error messages" (Guzdial, 2014, p. 1). Guzdial wrote that Denny et al. did not first attempt to see if the enhanced messages in that study were more helpful and that it was disappointing they did

not consider the work of Marceau et al. (2011a) who provided a rubric for evaluating compiler feedback messages (see Appendix A). Becker commented on the discussion as well, digging deeper into the work of Denny et al., showing what was measured and how it could be compared. (Becker, 2016)

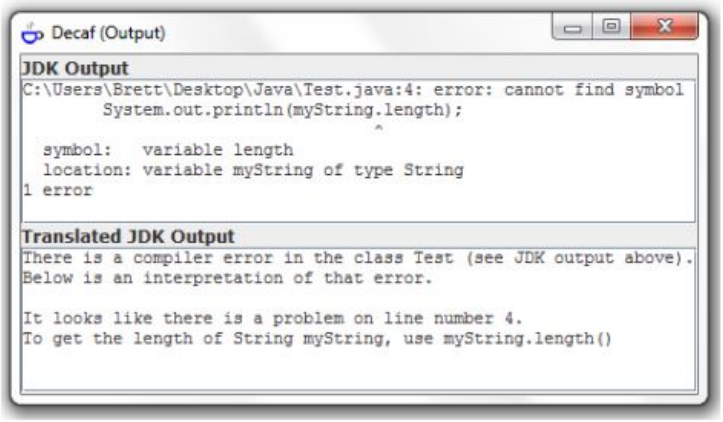


Figure 5. User Interface from Decaf showing both the standard and enhanced error feedback messages (Becker 2016, p. 127).

Agreeing with Denny et al. (2014), Pettit et al. (2017) performed an empirical study of enhanced feedback messages in Athene and found that it produced no significant change. They compared four consecutive semesters of CS1 classes without enhanced error messages to four consecutive semesters of CS1 classes with enhanced messages. They measured the likelihood of successive compilation errors, the occurrence of compiler errors within semesters, and student progress towards a successful completion of a programming assignment. Although there was no statistical evidence of increased learning in the experimental groups, students who saw the enhanced messages were overwhelmingly positive. Pettit et al. admitted multiple threats to validity, such as that students can compile offline and only use Athene for submissions, while CodeWright (Denny et al., 2014) and Decaf (Becker, 2016) are full code editors that capture all

student behavior. One issue that Pettit et al. did not address is whether or not the enhanced error messages themselves could be empirically shown to be more helpful; simply adding additional text does not necessarily make an error message more helpful (Nietawalski, 2008). Marceau et al. (2011a) provide a rubric for evaluating the effectiveness of error messages (see Appendix A), and like Denny et al., Pettit et al. did not consider it.

Even still, the works of Denny et al., Becker, and Pettit et al. are indicative of an open question in CSed: do enhanced error messages increase student learning? With two empirical studies against it and one for it, a rigorous human factors evaluation is needed to answer this question.

Human Factors Evaluations of AATs

If feedback messages are, as Denny et al. (2014) and Pettit et al. (2017) have said, ineffectual in increasing student learning, then what can be done to improve them so that they are helpful? Several studies have attempted to answer this question through a human factors approach. The literature discussed above often blames the confusing and terse compiler error messages as a source for student confusion and perhaps even why many students confess to not reading the enhanced feedback messages that AATs often provide. Hartmann et al. (2010) thought to solve this problem by creating their own AAT, HelpMeOut, which provides students with feedback similar to Denny et al. Instead of a traditional error message with a line number, an arrow to the offending character, or highlighting, 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, that was later found by Jackson et al. (2005) 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 AAT helped novice programmers create a better mental model of the errors they received or whether it increased learnability for novice programmers.

Marceau et al. (2011b) call out 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 ethnographic interviews with those same students. They discovered that students were grossly misinterpreting the feedback messages and were confused at the highly specialized vocabulary of their AAT, DrRacket. Guessing at why this is the case, 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 a follow-up paper, Marceau et al. (2011a) provide a rubric for evaluating the

effectiveness of error messages which will be used in the methodology of this study (see Appendix A).

Based on their results, Marceau et al. recommend the following 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 AAT. Lee and Ko (2011) discuss personifying feedback in a game that teaches programming. 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. Warren et al. (2014) discuss implementing their AAT within a Massive Open Online Course (MOOC) and the change in medium offers helpful insight into potential changes to feedback through enhancing user experience. Falkner et al. (2014) attempted to increase the granularity of feedback and observe its impact on students. Their results are promising, though they do not address student behavior in response to increased granularity of feedback – a goal also not addressed in many of the studies discussed above. Loksa et al. (2016) 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. They write, "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" (p. 1450). Broadly speaking, this ability is called "metacognitive awareness." They report that students trained in metacognitive awareness were significantly better able to understand feedback than students who were not. The work of Loksa et al. suggests that the cognitive scaffolding gains from improving feedback in AATs can be extended to the entire process of solving a programming problem using an AAT, though they do not describe what that would look like.

Finally, Singh et al. (2013) describe an AAT that automatically derives the solution of an error, creates a metric for measuring how *wrong* the student's code is, and provides that number to the student along with the appropriate error message. The AAT was tested with thousands of MIT students and the authors found that it could propose correct solutions to 64% of student errors. This sort of artificial intelligence technique on automatic grading is still new but is promising. The most interesting piece of this study is that when the tool can find a correct solution to propose, it allows students to quickly create an effective mental model of how far off they are from the solution. With other AATs, students will struggle against an error, find the solution, triumphantly expect their next submission to be correct, and then sadly run into the next error. This is because novices have not yet built a comprehensive mental model of programming and they therefore have no way to know how far off their submissions are from the correct solution. The feedback students receive from the AAT created by Singh, et al., help them with their immediate error, but as successive attempts occur it also provides clues about the direction the student is going. (e.g. is the number getting lower or higher?)

Unfortunately, they do not evaluate whether this novel approach actually enhanced student learnability.

Metacognitive Awareness in Novice Programmers

Introductory courses in programming often focus solely on syntax and data structures, but there is a growing consensus among computer science education researchers that it should also focus on assisting the novice in building a mental scaffold around which they can correctly place knowledge and develop metacognitive awareness (Eteläpelto, 1993; Shaft, 1995; Roll et al., 2012; Mani & Mazumder, 2013; Loksa et al., 2016). Metacognitive awareness is, simply put, knowing about knowing. Applied to programming, it is not just knowledge of the problem, but knowledge of where one is in the problem-solving process and self-reflection on that state (Metcalfe & Shimamura, 1994).

Incorporating metacognitive awareness into the instruction of novice programmers is rather scarce. In 2000, Vizcaíno et al. described HabiPro, an intelligent tutoring system (ITS) (Vizcaino et al., 2000). HabiPro included four exercises intended to help students develop good programming habits. In the first exercise, students were asked to find the mistake in a block of source code. The second exercise was, given a jumbled program with lines out of order, put the program in the correct order. The third exercise was to guess the result of executing some given source code without comments and with randomized variable names. The fourth exercise was, given source code with one line missing, write the one line to complete the program. The exercises in intelligent tutors can help build mental scaffolding in novices (Roll et al., 2012), although HabiPro was not

specifically designed to build metacognitive awareness. HabiPro is also not an automated assessment tool and that difference is an important distinction to make for the present research project. An ITS is designed to train novices in a particular skill, coding in this case, whereas an AAT is designed to assign and assess the correctness of student work.

A more recent study by Cao et al. reports on Idea Garden, an IDE that helps novices by providing mental scaffolding through just-in-time contextual hints (Cao et al., 2014). Their work focused on the development environment that coders use and how metacognition could be better engendered at that level. A follow-up by Jernigan et al. implemented these concepts into a larger prototype and reported that novices in the experimental group required substantially less help than the control group that did not use the prototype (Jernigan et al., 2015). Finally, Nelson et al. (2017) proposed a comprehension-first pedagogy paired with PLTutor, an ITS that would help novices better learn meta-programming skills such as code-tracing.

The most relevant study on promoting metacognitive awareness in novice programmers is by Loksa et al. (Loksa et al., 2016). They identified six distinct problem solving stages that learners usually progress through sequentially. Each stage is somewhat broad. Finer granularity inside each stage might be counterproductive because it is difficult to make fine-grained observations of people learning. See Table 1 for how these stages roughly correlate to stages proposed by Polya. See Table 2 for examples of the difficulties a student might face as they progress through each of the six learning stages. Loksa et al. reported on an intervention at a code camp where the control group was taught how to code and the experimental group was additionally trained in these six problem solving stages and the use of an IDE with an Idea Garden. They report that

students with this training were significantly more productive and required less help. As the literature indicates, pedagogical approaches and helpful coding environments should be pursued. These approaches, however, are difficult to scale or hard to implement for online learning technologies, such as massively open online courses, which Loksa et al. acknowledged as a limitation of their work. The intention of the present research project is to adapt the spirit of these interventions to automated assessment tools that can span this gap.

Table 2. *Learning stages by Loksa paired with examples of difficulties that novices might encounter at each stage.*

Loksa's Stages (Loksa et al., 2016)	Example Difficulty Faced by Novices at Each Learning Stage
1. Reinterpret the prompt	Fundamentally misunderstands the programming problem or mistakes it for a different problem.
2. Search for analogous problems	Decides to use a problem previously encountered that is too different from the current problem.
3. Search for solutions	Finds a solution that does not satisfactorily solve all possible test cases or solves the wrong problem.
4. Evaluate a potential solution	Fails to properly account for edge cases when mentally running through a selected algorithm.
5. Implement a solution	Incorrectly use of syntax.
6. Evaluate implemented solution	Incorrectly addresses failed test case by a making an ineffectual change in their code.

Human-Factors Tools: Ethnography, Usability Evaluation, and Think-Aloud Studies

Usability Evaluation

The design of a usability study has been discussed heavily in the literature since the late 1970's, but crystalized with the publication by Gould and Lewis (1985). They recommended the now standard early focus on users and their needs/desires, empirical measurements, and iterative design. Since Gould and Lewis' landmark paper, usability evaluations have become very important in the field of HCI because they test the systems through actual use to make sure that the user experience is what the researcher or designer imagined it to be when designing it (Dix, 2009; Shneiderman, Plaisant, Cohen, Jacobs, Elmqvist, 2017, p. 147).

Barkhuus and Rode (2007) examined 24 years of usability evaluations published at the ACM SIGCHI conference. They found that multiple kinds of usability evaluations, from qualitative think-aloud studies to quantitative lab studies to analytical studies utilizing measurements such as GOMS. However, the overwhelming majority of studies presented at SIGCHI used empirical quantitative methods. Their conclusions raise an important set of questions. First, is the research community biasing itself towards problems easily solved by empirical quantitative study? Second, is this biasing causing the research community to lose groundbreaking research that does not fit well into empirical quantitative testing? These two questions were answered the following year by Greenberg and Buxton (2008) who noted that usability evaluation had actually become "harmful" some of the time. They found that SIGCHI was indeed biased towards empirical quantitative analysis and this biasing had produced "weak science," i.e.

research questions were formulated for the method, rather than choosing a research method to answer the question. In light of the questions raised by Barkhaus and Rode and Greenberg and Buxton, in this study careful consideration is given to selection of methods, specifically the use of mixed methods. Greenberg and Buxton also highlight the need for replication of past studies, despite the less prestigious nature of the work. The discussion above highlights some of the pitfalls in usability testing that the present research project seeks to avoid. Of course, much more has been written on usability testing and how to perform it (Dumas & Loring, 2008; Tullis et al., 2008; Rubin & Chisnell, 2008; Krug, 2014). A related idea to usability evaluations is heuristic evaluation which is often done as part of a usability study.

Nielsen and Molich (1990) offer now-classic advice on using heuristic evaluation of user interfaces, namely that when it is done by a panel of experts, rather than one, it is highly reliable. This study will follow that advice when evaluating results of the usability study.

Think-Aloud Studies

One research tool often employed in evaluating changes made to CS1 classes is the think-aloud study. A few recent and relevant examples are considered. Yuen (2007) performed a think-aloud study on his CS1 class to understand the differences in how novices construct knowledge compared to experts. 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 first, "need to code," is the least desirable

response which is when the novice does not 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 this leads 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. (2013) perform a think-aloud study watching novices trace code and then determine in a single sentence what it does. They follow the classic think-aloud protocols by Ericsson and Simon (1993). Their 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. The present research project follows Ericsson and Simon (1993) for think-aloud protocol and follows Teague et al. (2013) in adding a short practice session at the beginning to hopefully offset cognitive load on novices.

Whalley and Kasto (2014) perform a think-aloud study watching novices solve three programming challenges. 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.

Ethnography

Taken from the social sciences, specifically anthropology, ethnography is a practice that attempts to observe and understand human behavior, beliefs, and institutions (Angrosino, 2007). Ethnography has been widely adopted outside of social sciences as a tool to understand business culture (Brannen & Salk, 2000; Cunliffe, 2009), theology (Wyche et al., 2007; Moschella, 2008), and is widely used in HCI (Button, 2000; Bell, 2001; Bell et al., 2003; Bell et al., 2005; Lazar et al., 2017). The goal of ethnographic study in HCI is to understand the user, from the user's perspective, situated in the user's context (Blomberg & Karasti, 2012). Perhaps the earliest and most cited example of ethnography in HCI is Suchman (1987) who showed that users do not form goals and then follow plans to reach those goals like machines, but rather use those plans as resources for "situated action" as it unfolds (Blomberg & Karasti, 2013). Suchman developed this theoretical model by observing office workers using a copy machine and was then able to redesign its interface to better support human needs. A recent example of ethnographic study in HCI observed those with cognitive disabilities in order to derive better means of supporting their daily tasks (Carmien & Fischer, 2008). The only ethnographic study that takes place in the computer science classroom is Garvin-Doxas

and Barker (2004) who observed students and professors in CS1 classrooms to determine what makes the atmosphere defensive or supportive.

However, when poorly applied, ethnography can be harmful for research. Crabtree et al. (2009) review ethnographic studies at SIGCHI and found that many poorly apply the tool to emerging areas of technology. They argue that ethnographic methods do not need to be changed for these new contexts and highlight the harm to the research that is often the result. Bell et al. (2003) called for new understandings of the new domains of technology as it rapidly progressed from the office to the home and beyond. Crabtree et al. (2009) write that, "it is important to recognize that new contexts of design do not necessarily demand the development of new approaches to develop new understandings" (p. 880). They caution against new methods that do not seek to understand the "lived work" of the user, transform ethnography into a literary critique through defamiliarization which does not constructively inform design, surface-level descriptions of contexts which they call "exotic tales from home and abroad" or "design tourism," and critical reflection on "new values." Crabtree argues that these new techniques are often used to support the researcher's concerns rather than the concerns of those being observed. It is for these reasons that the present research project will use traditional ethnomethodologically-informed practices, which are practices that seek to observe action and interaction wherever and whenever it occurs. In this case, it is the CS1 student working on his or her homework, which due to the mobility of modern computing, can take place *anywhere*: the dorm, the student center, the classroom, in a car, etc. This means that observing students working on their homework in a lab setting is not too different from any other particular setting on campus where they might work.

Summary

The environment in which many students learn to program is now almost fully automated. The feedback that students receive from these AATs now directly contributes to the learnability of programming and programming languages. And yet, even with AATs providing instant feedback, as Guzdial (2015) points out it remains a disproportionately difficult task to learn to code. Most who have implemented these AATs report increased student performance, but this hypothesis is currently disputed when quantitatively tested. It seems as though enhancing feedback can – but does not necessarily – enhance the learnability of programming for novices. The solution to finding out *why* is to understand the problem from the perspective of the users. What makes "enhanced feedback" more usable for students? Too often it is considered better simply because there is more information, but arbitrarily increasing the cognitive load on students does not usually lead to increased learnability. Meanwhile, very few have taken the advice of Marceau et al. (2011a) in using their rubric (see Appendix A) to verify that the enhanced messages are, in fact, more usable. Furthermore, enhanced feedback is just one piece of the metacognitive puzzle. The substantial research into enhanced feedback discussed above will allow this study to correctly implement enhanced feedback and then proceed to propose suggestions for implementing other features that more fully support metacognition in novice programmers.

This study will be carried out using standard research tools. Multiple research tools were discussed above and are employed in this study: usability studies, think-aloud protocol, and ethnography. This study refers to those mentioned above to help design the

methodology used. The pitfalls and incorrect applications of these methods are also considered and avoided.

Chapter 3

Methodology

Approach

The ethnomethodologically-informed study outlined in this chapter was carried out to address the research questions presented in Chapter 1. Learning to program in CS1 courses cognitively overloads novices, but helping novices become aware of their cognitive processes (metacognition) improves their performance (Loksa et al., 2016). One method for addressing this issue is to enhance the compiler error message feedback students receive in AATs, though this only effectively tackles one of six learning stages of which students should be made aware. Chapter 2 addressed the history of AATs (Douce et al., 2005; Pears et al., 2007), including student and professor opinions and quantifiable data regarding its efficacy at enhancing learning (Ihantola et al., 2010). However, it is currently heavily disputed whether or not ECEMs improve student learning (Denny et al., 2014; Pettit et al., 2017). The researcher carried out an ethnographic study of CS1 students via usability studies and interviews in order to arrive at verifiably helpful ECEMs and from there to understand students' metacognitive hang-ups in the other five learning stages. The reviewed literature on usability studies, think-aloud protocol, and ethnography informed the design of the present study.

This chapter first describes two pilot studies and their preliminary results in iterative stages. It next describes the design of the full study and how it was informed by the literature and the results of the pilot studies. Next, the procedure for this study is discussed along with the participant pool, recruitment, and data collection methods. This

chapter also revisits research questions from Chapter 1 to provide methods by which they may be answered through the evaluation of the full study's results. Finally, the methods by which the data will be analyzed are discussed. As listed above, the primary research questions for this study are:

RQ1. When students diverge on a specific learning stage, what factors caused them to do that?

RQ2. Are ECEMs helping students evaluate their potential solution?

RQ2.a. Are students reading the enhanced messages?

RQ2.b. If students are reading the enhanced messages, how do the enhanced messages help them better understand the error?

RQ3. When students diverge on a specific learning stage, submit their program, and receive an ECEM, how do they interpret it?

RQ4. How can AATs be augmented to support metacognition in novice programmers in CS1?

Pilot Studies

Two pilot studies were performed in the Fall of 2016 with 6 students in the first and 6 students in the second. These pilot studies helped determine necessary modifications to the ECEMs to improve its usability for the full study.

Participants and Recruitment

All participants in both pilot studies consisted of students from Abilene Christian University (ACU), a small private liberal arts university located in Abilene, Texas.

Students participated in the study in a lab on the third floor of the Mabee Business Building at ACU. All students were enrolled in computing, math, or related majors. The participant pool consisted of 12 students, 10 males and two females. The pilot studies were conducted under NSU IRB# 2016-399 and with coordinate and in conjunction with ACU's IRB (see Appendix F). All appropriate steps regarding recruitment and consent were followed as outlined in the IRB application.

The first pilot study was conducted early in the Fall semester and so using participants from CS1 would have been impossible or the task would have been too simple. Therefore, participants from CS2 were used instead. These participants had completed CS1, but were not very far along in CS2 to be significantly different from CS1 students and therefore still considered to be novice programmers.

The second pilot study was conducted near the end of the Fall semester. Therefore, it was possible to use the ideal target of CS1 students.

Procedure

Participants were presented with a programming assignment within the Athene, provided a file of buggy code, and asked to submit it until accepted as correct. The programming assignment chosen for this evaluation was a Fibonacci problem to be completed using a simple loop. The Fibonacci sequence is a mathematical series of numbers where every number after the first two, which are 0 and 1, respectively, is the sum of the previous two numbers. Each participant had completed the assignment in their CS1 class several weeks before the usability study. This problem was picked because its complexity was low enough to allow quick problem solving, but high enough to warrant

interesting feedback messages. Participants were provided with a code file that had been created specifically for this evaluation. The code had five specific errors that, as participants fixed each one, would lead them through six feedback messages (five enhanced messages and one message indication problem completion). These errors were chosen to represent a broad spectrum of programming errors and feedback messages.

Both pilot studies used Krug's (2014) format, including following his pre-testing checklists, format for testing, testing logic and prompting, and data collection methods. This report follows Rubin and Chisnell's (2008) suggested arrangement and all participants were anonymized as suggested by Dumas and Loring (2008). Participants were allowed into the testing room one at a time. Once the participant was called into the room, the evaluator read a script to the student and then guided them through the tasks. Participants were asked to submit the provided code file to Athene, fix any errors, and iteratively correct and resubmit until the program passed all test cases. During this time the evaluator used the think-aloud protocol (Teague et al., 2013; Whalley and Kasto, 2014) to help individual participants vocalize their thought process while completing the tasks. Each evaluation lasted for a maximum of ten minutes. If the participant did not pass all test cases by the end of the ten minute time window, the evaluation was stopped. In either case, the participant was thanked for their time and then exited the room.

A feedback message was determined to be useful to the participant if they corrected the error after reading it. The usefulness of each message did not change subsequent messages or change how these messages appeared. Rather, which feedback messages the participants found useful and which parts of the messages they utilized

were noted by the observer and that data was subsequently used to refine the messages for the next pilot study or the full usability study.

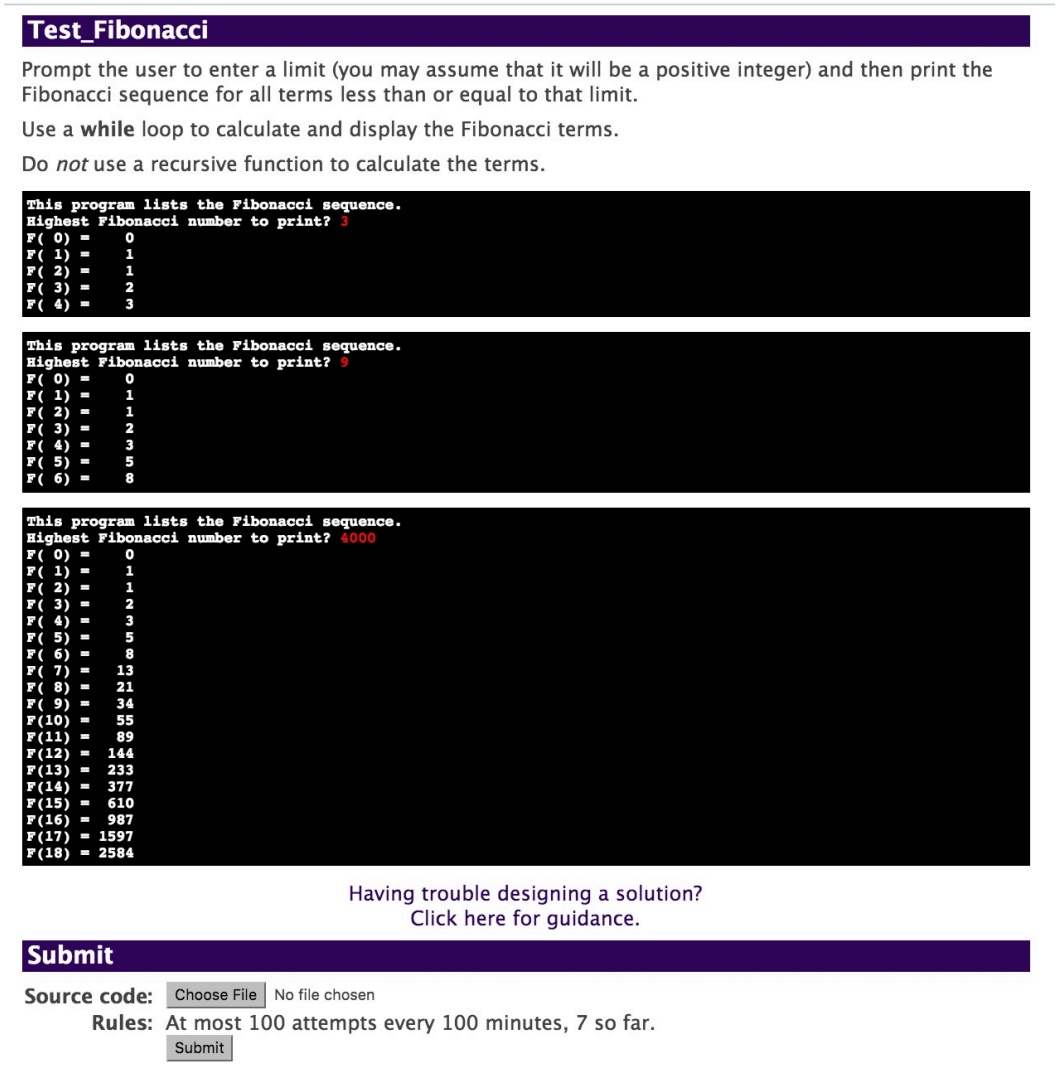


Figure 6. The Fibonacci problem in Athene that participants used.

Pre-Study Enhanced Messages

Error feedback messages in Athene were originally enhanced in the Fall of 2015 with results of that work published by Pettit et al. (2017). Prior to their work, compiler error feedback simply consisted of the standard compiler error message. Their study found that the enhanced messages had no effect on student learning. The motivation of

the original enhancement was simply to provide more information to students using the AAT and provide possible reasons for avenues of approach in their subsequent attempts. The design of these messages did not take into account any of the existing literature discussed in Chapter 2. For these reasons, these pre-study enhanced compiler error feedback messages will be referred to as "naïve enhanced messages."

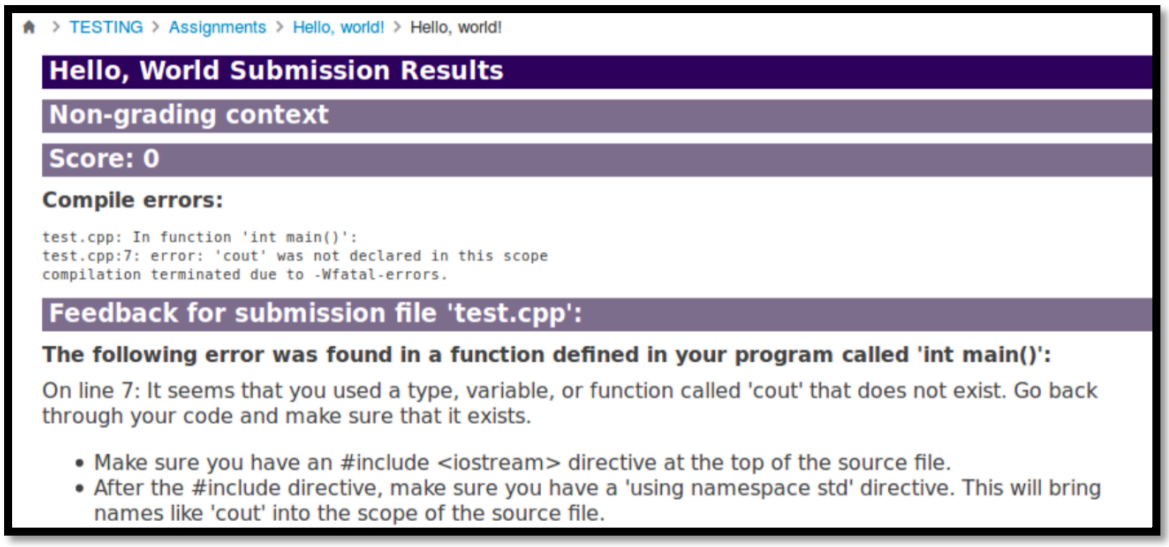


Figure 7. Example of standard compiler error feedback message with naïve enhanced message below it in Athene.

Pilot Study #1

The first pilot study tested the naïve enhanced messages through a usability test as discussed above. Six students participated. The five error messages are included below. The sixth feedback message simply stated that the problem was finished.

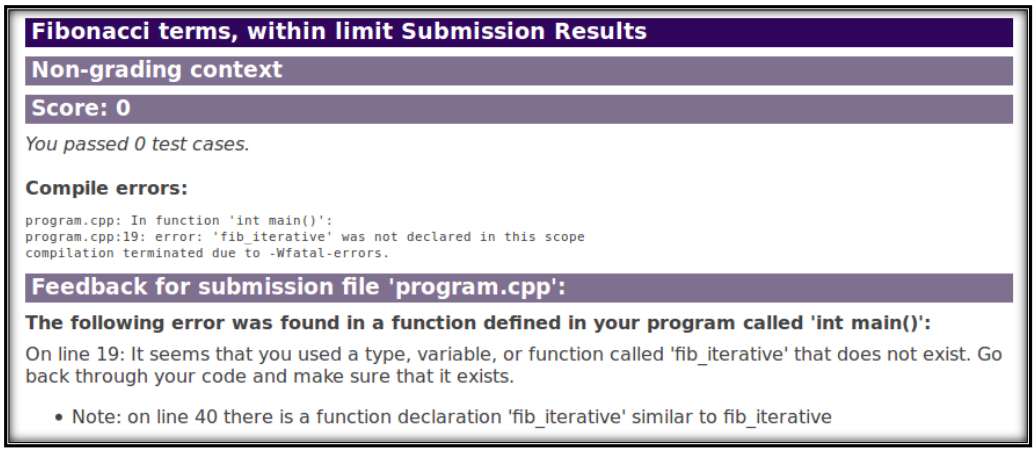


Figure 8. First feedback message that the participant encountered. The standard compiler message is in the top section. The enhanced compiler message is in the bottom section.

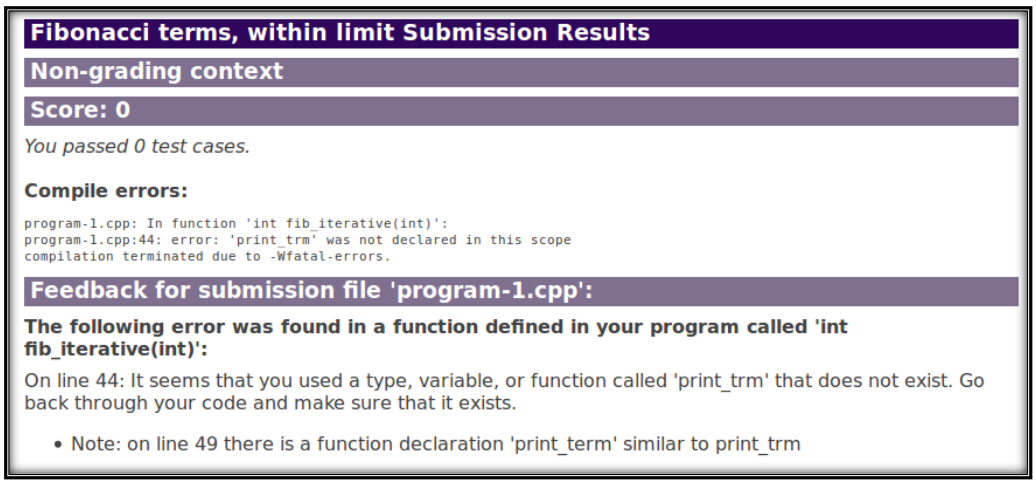
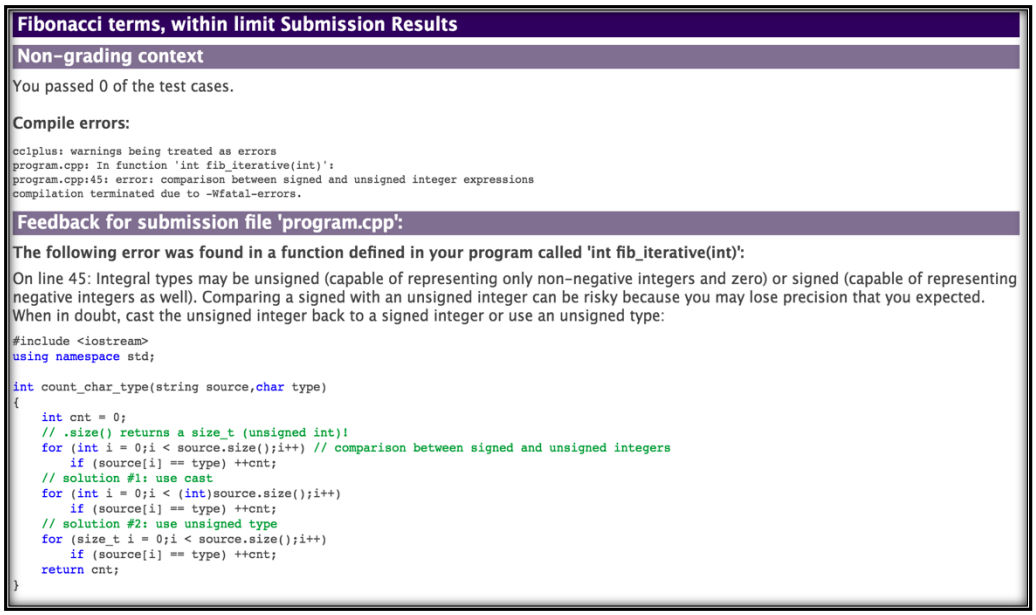


Figure 9. Second feedback message that the participant encountered after correcting the first error.



Fibonacci terms, within limit Submission Results

Non-grading context

You passed 0 of the test cases.

Compile errors:

```
cc1plus: warnings being treated as errors
program.cpp: In function 'int fib_iterative(int)':
program.cpp:45: error: comparison between signed and unsigned integer expressions
compilation terminated due to -Wfatal-errors.
```

Feedback for submission file 'program.cpp':

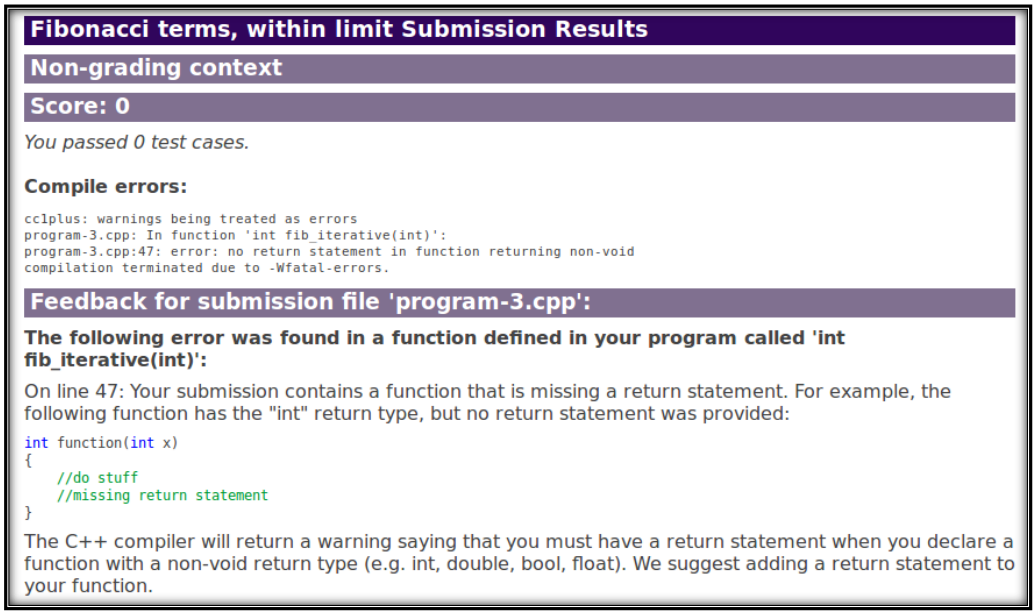
The following error was found in a function defined in your program called 'int fib_iterative(int)':

On line 45: Integral types may be unsigned (capable of representing only non-negative integers and zero) or signed (capable of representing negative integers as well). Comparing a signed with an unsigned integer can be risky because you may lose precision that you expected. When in doubt, cast the unsigned integer back to a signed integer or use an unsigned type:

```
#include <iostream>
using namespace std;

int count_char_type(string source,char type)
{
    int cnt = 0;
    // .size() returns a size_t (unsigned int)!
    for (int i = 0;i < source.size();i++) // comparison between signed and unsigned integers
        if (source[i] == type) ++cnt;
    // solution #1: use cast
    for (int i = 0;i < (int)source.size();i++)
        if (source[i] == type) ++cnt;
    // solution #2: use unsigned type
    for (size_t i = 0;i < source.size();i++)
        if (source[i] == type) ++cnt;
    return cnt;
}
```

Figure 10. Third feedback message that the participant encountered after correcting the second error.



Fibonacci terms, within limit Submission Results

Non-grading context

Score: 0

You passed 0 test cases.

Compile errors:

```
cc1plus: warnings being treated as errors
program-3.cpp: In function 'int fib_iterative(int)':
program-3.cpp:47: error: no return statement in function returning non-void
compilation terminated due to -Wfatal-errors.
```

Feedback for submission file 'program-3.cpp':

The following error was found in a function defined in your program called 'int fib_iterative(int)':

On line 47: Your submission contains a function that is missing a return statement. For example, the following function has the "int" return type, but no return statement was provided:

```
int function(int x)
{
    //do stuff
    //missing return statement
}
```

The C++ compiler will return a warning saying that you must have a return statement when you declare a function with a non-void return type (e.g. int, double, bool, float). We suggest adding a return statement to your function.

Figure 11. Fourth feedback message that the participant encountered after correcting the third error.

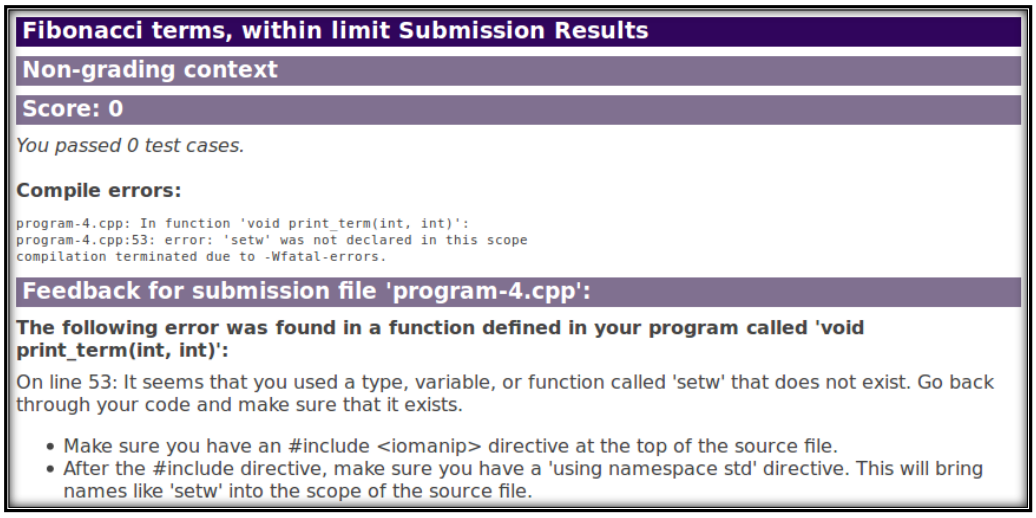


Figure 12. Fifth feedback message that the participant encountered after correcting the fourth error.

All six students successfully submitted the program and began receiving feedback for each error, one at a time. Each one moved through the errors at a different pace with only one correcting all five errors. Although each participant had already written and completed this program during CS1, only one participant completed the task of fixing all five errors because of a ten minute time constraint. A time constraint of ten minutes was intended to simulate actual student activity of submitting at the last possible moment. Below is a discussion of some of the more salient results from each participant.

Participant #1: Read the standard compiler messages first, then went back to his code to look, and made several attempted fixes before finally correcting the error. He repeated this for the second error message. He did not read the enhanced error messages until seeing an error he had never seen before (Figure 10), and only then remarked regarding the enhanced messages, “I guess I should read this now.” He glanced through it quickly and looked back to his code, saying, “That’s really helpful.” Even though he

thought the example code was his code, he quickly determined the source of the error and fixed it.

Participant #2: After reading the standard compiler message for the first error, he went back to the code, made an incorrect change, resubmitted, and got the same error again. Frustrated, he read the enhanced message and then was able to fix the bug that produced the error. For each successive error, participant skipped the standard compiler message and went right to the enhanced message. He expressed some frustration with one of the enhanced messages (Figure 10) that contained some vocabulary or syntax he had not yet learned in class and at first thought it was displaying his code.

Participant #3: Participant ignored the enhanced messages and only read the standard compiler messages. This was sufficient for fixing the first three errors. Upon seeing the third error (Figure 10), he remarked, "Whoa! That's a lot of text," and then ignored it. After becoming stuck on the fourth error, he finally read the enhanced message. He then went straight to the enhanced message for the fifth error (Figure 12), remarking, "Actually, that is really helpful." When asked what about it was helpful, he replied, "Because it tells me exactly what to do." This participant fixed the final bug with about ten seconds remaining.

Participant #4: Participant glanced through the first error message (Figure 8), both standard and enhanced, and was still confused. After multiple incorrect attempts, he went back and carefully read the enhanced message, but was still confused. After muddling through for a while longer, he found and corrected the first bug. He read the full second error message at length and again seemed puzzled, but eventually corrected it after

multiple attempts. Upon seeing the third error message (Figure 10) the evaluator asked what he was thinking and he remarked that he was, "a little overwhelmed."

Participant #5: Participant read the full error message at length and made multiple ineffectual changes. After half the test time had elapsed and he began changing more and more of the code, the evaluator had to undo these changes and move the participant on to the second error message. Normally the evaluator would not intervene in the evaluation, but the participant's changes were so substantial that he would have never fixed the bug or seen the additional error messages. He read the second message at length, but it did not seem to help him. He was very confused and did not correct any errors.

Participant #6: For the first error, participant read the standard compiler message, looked at the code, and after being unable to find the problem went back to the enhanced message. For the next two error messages, she consulted the enhanced message first. Upon seeing the third error message (Figure 10), the longest message, she said, "This is comforting to see an example with code because it's familiar. This is really helpful!" She was then able to follow the advice and correct the error. For the fourth error message she skipped the standard compiler message and immediately read the enhanced message.

There are three main observations from the results of the first pilot study. The first is that most participants did not read the entire error message before attempting to fix their code. Four of the participants did not bother with the enhanced messages until hitting a wall and only then did they read it, preferring instead a trial-and-error approach with minimal help. After reading the enhanced messages, each expressed delight – or even surprise – at how helpful they were. Two participants read the entire enhanced message each time, but it did not seem to be very helpful to them. These two participants

progressed the least in amount of errors corrected. From the above data, an interesting trend emerges. It appears that participants more comfortable with the material preferred to skip the enhanced messages until absolutely necessary while students with less grasp of the material preferred to start with the enhanced message but found it unhelpful.

The second observation concerns the design of the feedback messages themselves. As noted above, several participants thought that the example code was their code, even though it did not look anything like the code in the file that they were provided. This could be due to a lack of familiarity with the code because it was provided for them, rather than their usual experience with Athene (i.e. writing it themselves). However, an even more simple explanation is that the naïve feedback messages did not clearly label the sample code as a sample. This was especially confusing for the third feedback message (Figure 10).

The third observation from this study is that several participants mentioned how enhanced feedback messages had too much text. Participants seemed intimidated by longer blocks of text and code and were less likely to wade through it until absolutely necessary. In the only screenshot given in the paper by Becker (Figure 5), the enhanced error message is indeed brief, and part of that message can simply be copied and pasted directly into the students' code to correct it. In the example (Figure 4) shown in the paper by Denny et al. (2014), the enhanced message requires more reading and understanding. Students want to get to an answer quickly, so perhaps the enhanced feedback messages are sometimes simply too much on their cognitive load and so they prefer not to engage the feedback.

A related finding to the two above was a level of frustration that the study brought out in nearly every participant. Some mentioned that they did not enjoy being led through failure after failure. However, the difference in the number of errors used in this study and the number found in actual student submissions were not statistically significant. It is therefore possible that students already had a negative emotional association with Athene, having experienced this kind of frustration on dozens of assignments already at that point in the semester. Nothing could control for that variable because this study required novice students who knew enough about programming to correct some errors.

Pilot Study #2

The feedback messages were significantly updated with the findings of the first pilot study in mind, along with the suggestions from Marceau et al. (2011b) and Hartmann et al. (2010). The following changes were made:

- Drop down. To address the cognitive overload issues, the enhanced error messages were put into a collapsed drop-down with the text "Need more help?"
- Sample code. To clear up confusion about the sample code, all code in the enhanced message was clearly labeled as a sample.
- Similar errors. As suggested by Hartmann et al. (2010), each enhanced error message showed an example with a similar error alongside an example of the same code with the error corrected. These code snippets were pulled from Athene.

- Code highlighting. As suggested by Marceau et al. (2011b), the exact line where the error occurs and is fixed in the sample code snippets was highlighted to draw attention to it.
- Was this helpful? As suggested by Hartmann et al. (2010), the bottom of each enhanced error message included a way for students to rate the helpfulness of the error messages, potentially allowing a voting system where the most helpful example code snippets rise to the top. This rating was implemented by including the question "Was this helpful?" with simple "Yes" and "No" form boxes.
- Vocabulary. Marceau et al. (2011b) suggests paying careful attention to vocabulary and to make sure that students learn important words that they might encounter in the AAT early on in the class. To implement this, a blue circle with a question mark in it was placed next to any word in the enhanced error message feedback that might potentially confuse students. On mouseover, a bubble appears to explain the particular vocabulary word.

The following are examples of the same error messages as the naïve ones above (Figures 13 – 19), but redesigned given the above criteria. For the sake of succinctness, only the first message is shown with both the dropdown collapsed (default state) and expanded. Also included only once is an example of the vocabulary help that shows up on mouseover of the blue circles with question marks.

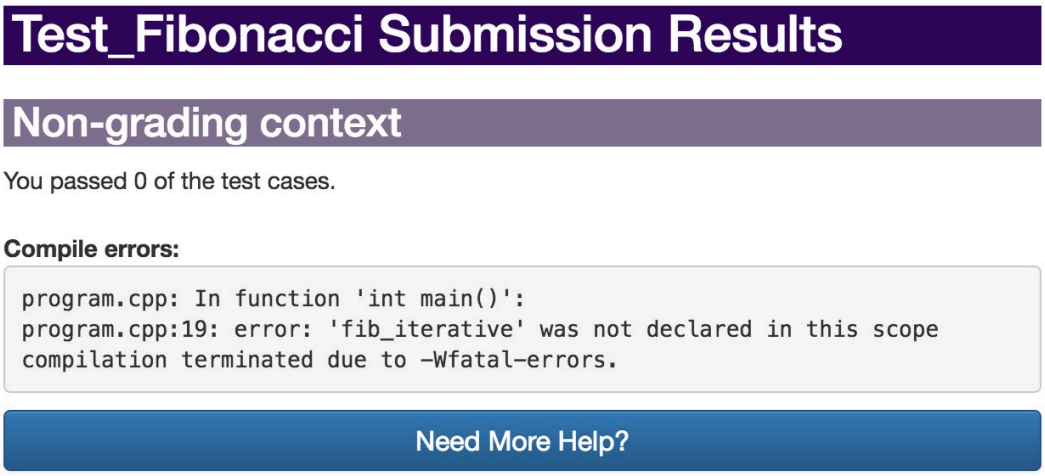


Figure 13. The first enhanced compiler error message from the second pilot study with the enhanced portion of the message collapsed.

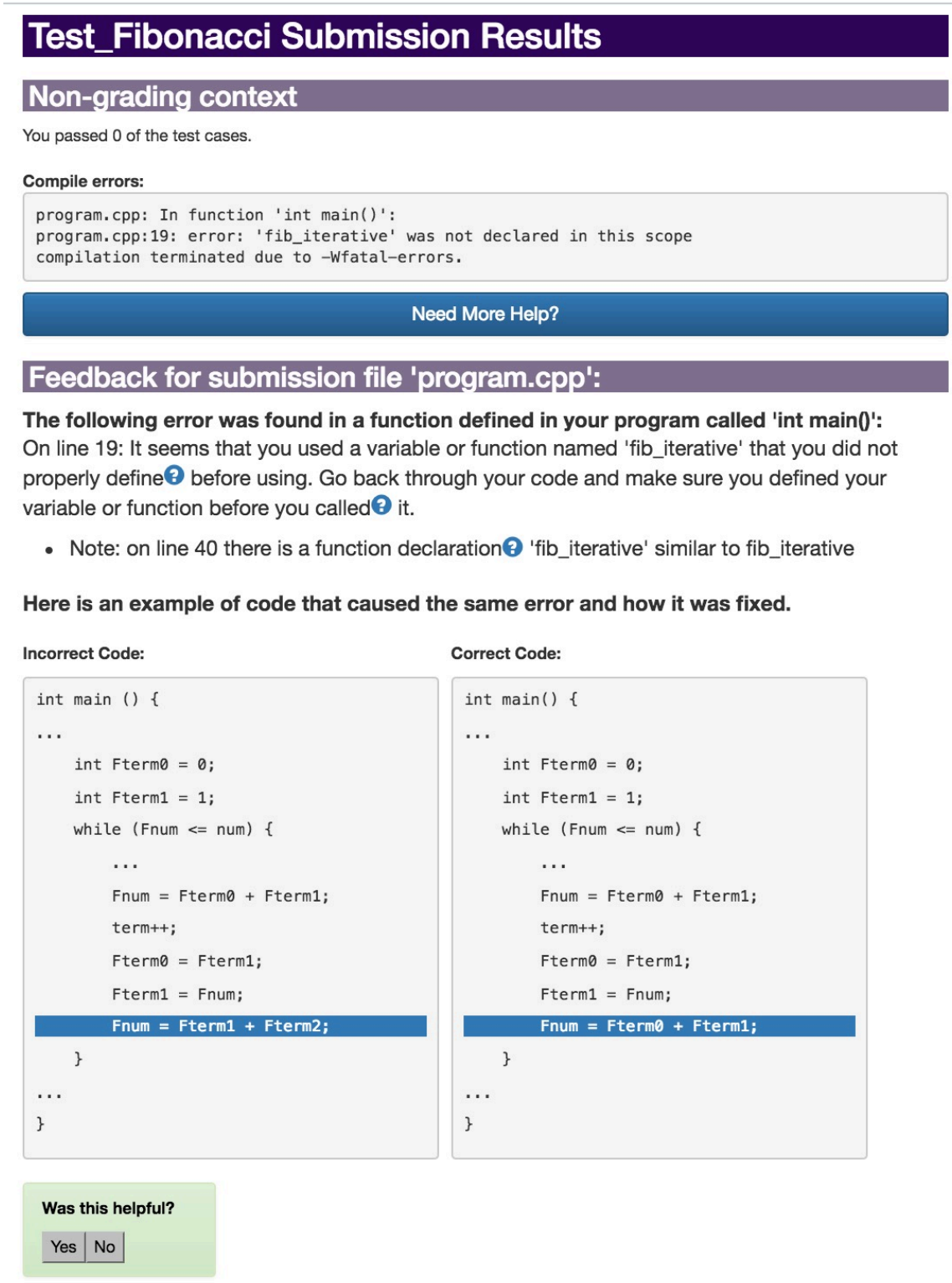


Figure 14. The first enhanced compiler error message from the second pilot study with the enhanced portion of the message expanded.

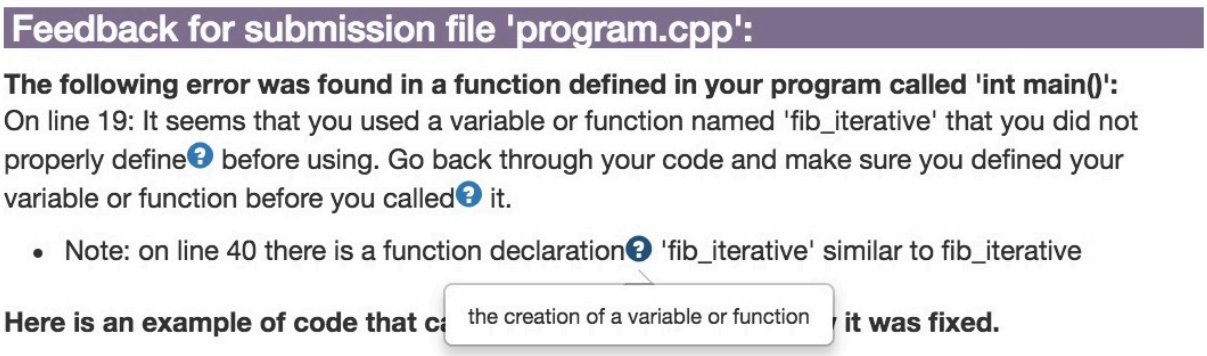


Figure 15. A portion of the first enhanced compiler error message from the second pilot study showing the vocabulary help bubbles appear on mouseover of the blue circle question mark.

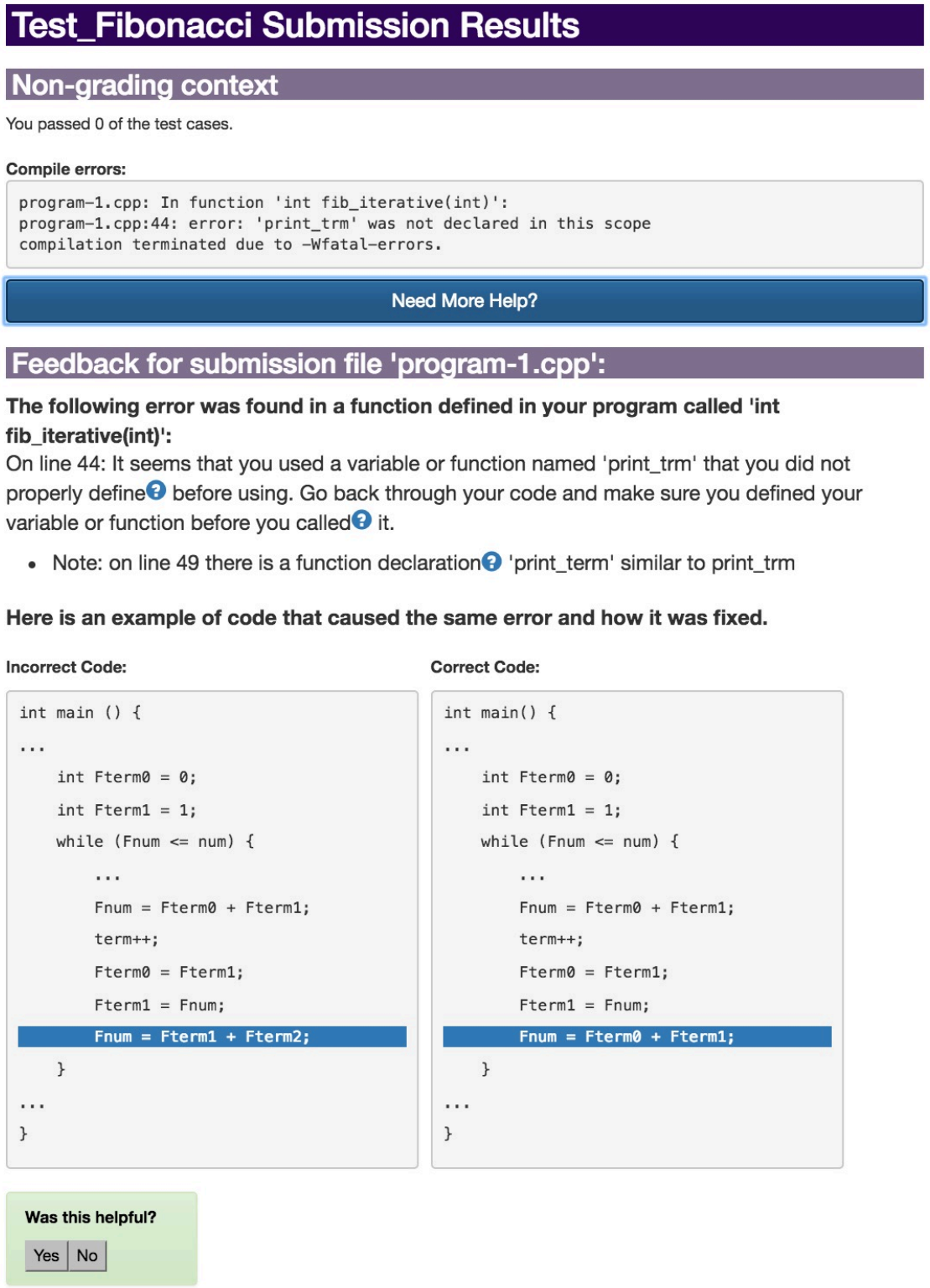


Figure 16. The second enhanced compiler error message from the second pilot study with the enhanced portion of the message expanded.

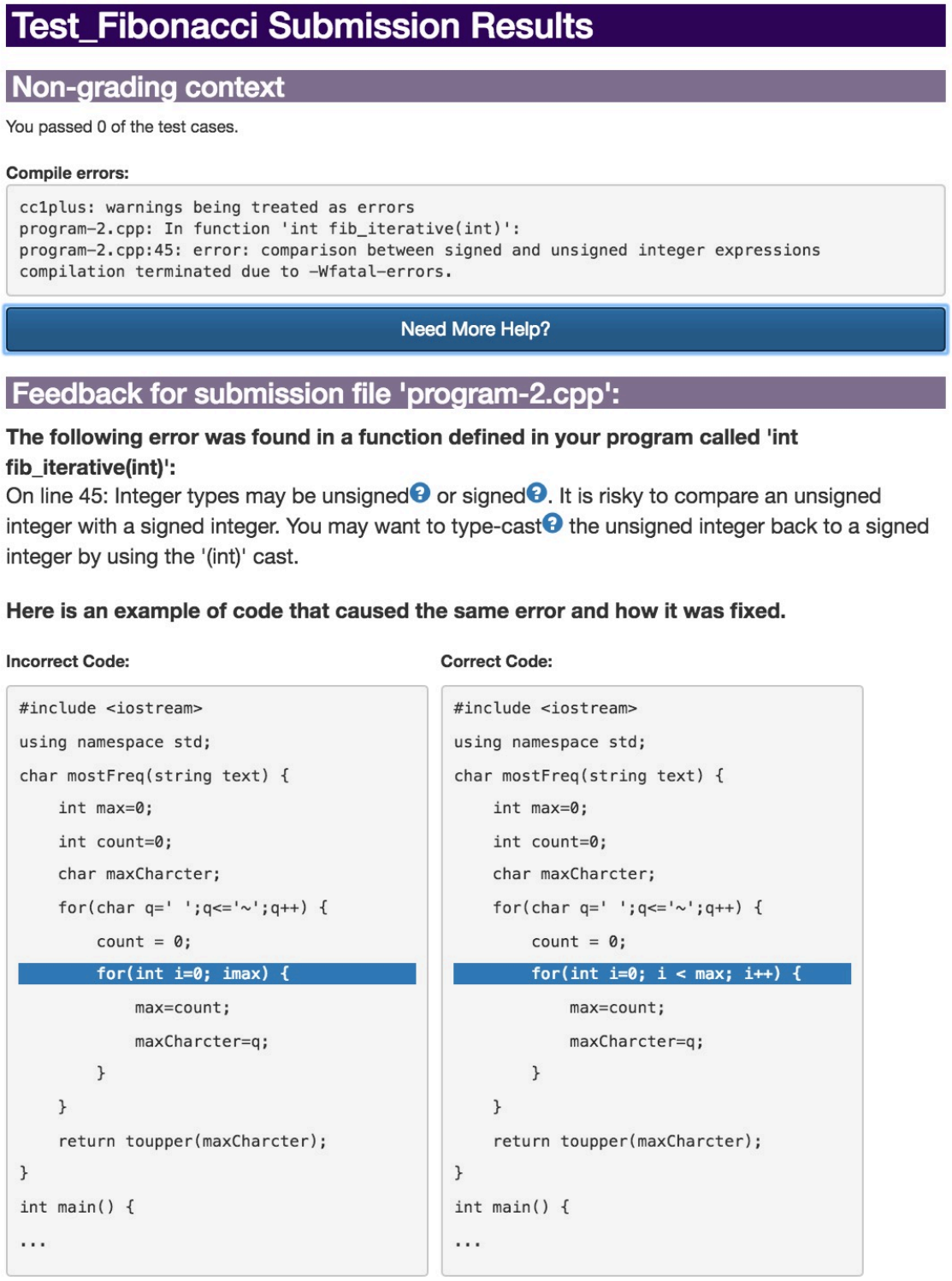


Figure 17. The third enhanced compiler error message from the second pilot study with the enhanced portion of the message expanded. Rating box could not be fit into this screenshot.

Test_Fibonacci Submission Results

Non-grading context

You passed 0 of the test cases.

Compile errors:

```
cc1plus: warnings being treated as errors
program-3.cpp: In function 'int fib_iterative(int)':
program-3.cpp:47: error: no return statement in function returning non-void
compilation terminated due to -Wfatal-errors.
```

Need More Help?

Feedback for submission file 'program-3.cpp':

The following error was found in a function defined in your program called 'int fib_iterative(int)':

On line 47: Your submission contains a function that is missing a return statement. Either change the return type to void or return a value of your function's type.

Here is an example of code that caused the same error and how it was fixed.

Incorrect Code:

```
#include <iostream>
using namespace std;
int explode(int number, int array[]) {
// populates array with digits from number
// and return the number of digits in arra
    int count = 0;
    while ( number != 0 ) {
        count++;
    }
    for ( int i = count ; i >= 0 ; i--) {
        array[i] = number % 10;
        number = number / 10;
    }
}
int main() {
...
```

Correct Code:

```
#include <iostream>
using namespace std;
int explode(int number, int array[]) {
    int count = 0, copy = number;
    while ( copy != 0 ) {
        count++;
        copy /= 10;
    }
    for ( int i = count-1 ; i >= 0 ; i--)
        array[i] = number % 10;
        number = number / 10;
    }
    return count;
}
int main() {
...
```

Was this helpful?

Yes | No

Figure 18. The fourth enhanced compiler error message from the second pilot study with the enhanced portion of the message expanded.

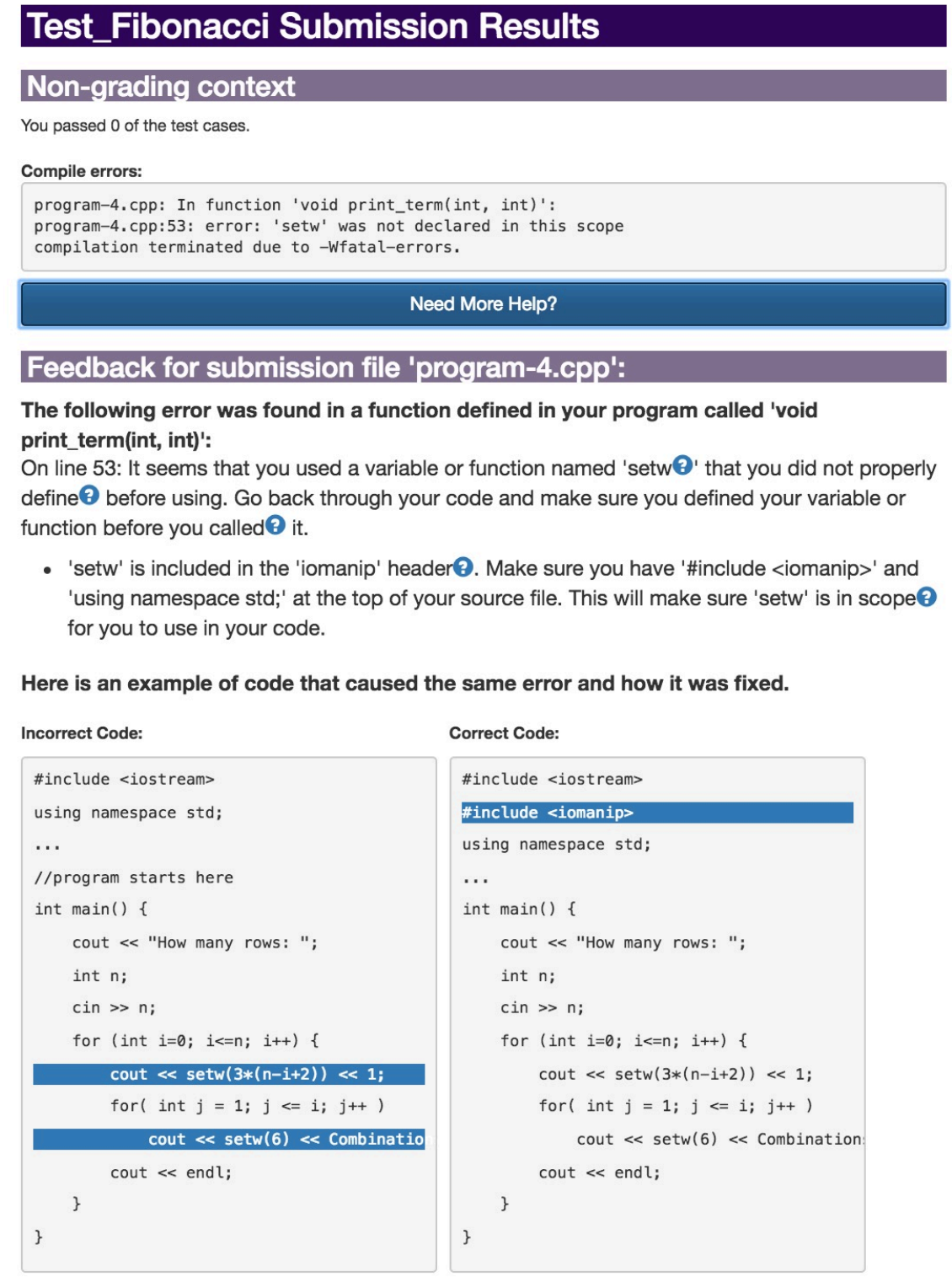


Figure 19. The fifth enhanced compiler error message from the second pilot study with the enhanced portion of the message expanded. Rating box could not be fit into this screenshot.

All six students during the second pilot study successfully submitted the program and began receiving feedback for each error, one at a time. Each one moved through the errors at a different pace with two correcting all five errors. Although each participant had already written and completed this program just a few weeks prior during CS1, only two participants completed the task of fixing all five errors because of a ten-minute time constraint. Below is a discussion of some of the more salient results from each participant during the second pilot study. Since the enhanced message is collapsed by default, it was easier than in the first pilot test to determine when the participant viewed it and this will be noted.

Participant #1: After reading the standard message, participant followed it too literally and solved the first error, but created more bugs in the process. After fixing those, he moved on and solved the second error immediately. For the third error, he found the line with the bug by reading the standard message, looked at the code, and said, "I don't understand." After stalling for several minutes, the evaluator asked if he had seen the "Need More Help?" button and the participant said he hadn't even seen it. After expanding the enhanced message, participant read it and immediately understood and fixed the error. For the fourth error, participant read the standard message first, looked at the code, and said, "I don't know what that means, so…" and clicked the help button. Unfortunately, he once again followed its advice too literally and got a totally different error message that was not enhanced for this study. After this, his time was up.

Participant #2: Participant was very comfortable with the standard messages and solved each error quickly. After the final submission and completion of the problem, evaluator asked participant if she had seen the "Need More Help?" button. She replied

that she saw it, but wanted to try it on her own first. She also noted that during the session she wasn't sure what it did and that it looked like it took her to another web page away from Athene. After expanding the enhanced feedback message, she said, "It's very nice. It explains everything clearly. I like how the color draws your eye to the code examples."

Participant #3: For the first error, participant rearranged function order, rather than uncomment the prototype at the top. This caused him to encounter the errors out of order. The next error encountered, the fifth in order (Figure 19), stumped him. He said, "The error message tells me where it is, but not what to do." After a few minutes of tinkering, participant was directed to the "Need More Help?" button. After expanding the enhanced message and reading it, he exclaimed, "Oh! That's really helpful." When asked why he had not yet used it, he said, "Because it looked like it took me to a different page," so he was suspicious of it and wanted to stay on task. Participant next encountered the fourth error (Figure 18) and immediately clicked to expand the enhanced message. At this point, his time was up.

Participant #4: Participant struggled with the first error for several minutes and was eventually directed to the "Need More Help?" button. After reading the message, he fixed the error. Participant immediately used the enhanced feedback for the next two errors and then his time ran out. After the session was over, participant was asked what he thought about the enhanced messages. He said, "They're helpful and straightforward."

Participant #5: Participant immediately solved the first error. On the second error he moused-over the help button initially but did not click it; instead, he found the error and fixed it. He also only used the standard message for errors three, four, and five, completing the task before the time limit. The evaluator asked if he had noticed the

"Need More Help?" button and participant replied that he had seen it, but didn't need it. Upon expanding the enhanced message, participant noted that the sample code comparison and blue question marks for vocabulary were helpful.

Participant #6: Solved the first error quickly. On the second error, participant said about the standard message, "I'm not sure what exactly this means." After tinkering for a few minutes, participant was directed to the "Need More Help?" button. In response he said, "Oh. I haven't seen that." After clicking on the button and reading the enhanced message, it became clear he thought the example code was his actual code. He said, "I'm going to search for this and replace it with that." He looked for the sample code from the enhanced message in the provided code file, but when he could not find it, he asked, "Am I supposed to delete my code and replace it with the sample?" After a few more minutes of tinkering he fixed the error. For the third error, participant paused to read the standard message before expanding the enhanced message. After reading the enhanced message, mousing-over the blue question marks and reading vocabulary help, he was still stumped. Then the time limit expired.

There are five important observations from the second pilot study. First, several students failed to even notice the large blue "Need More Help?" button that expanded the enhanced error messages. This was striking because it was assumed that the color and size of the button would be enough to draw their attention toward it. Clearly, more must be done or students should be told about it during class time as Marceau et al. (2011b) suggests. Related to this is the observation that the students who did notice its presence on the page thought that it took them to another page, away from the context of Athene. Here it seems that the button's design lacked clear signifiers to communicate its

affordance. Students understood it as a link and therefore thought that it afforded HTTP transport, potentially undoing their progress in the task, rather than expanding a drop-down.

Second, the observational data once again confirmed an observation from the first pilot study: students who were more comfortable with the material did not need the enhanced messages, while students who were very uncomfortable with the material did not benefit from the enhanced messages when read. One possible explanation for this result is that the students who did not benefit diverged at an earlier learning stage and were not metacognitively aware of their divergence. Therefore the feedback they receive would be confusing because they might be trying to solve a different problem or mapped the problem to the wrong domain. Therefore, as of the second pilot study, it seems that the enhanced messages only truly benefit the group of students that fall in the middle. This observation underlies the next one.

Third, despite clearly labeling the "example code," at least two students were confused enough by it to confirm it verbally. It's possible that other students might have been as well. Only the students most comfortable with the material (participants #2 and #5) thought the example and highlighting were useful. The rest either did not mention it or found it distracting or, even worse, misleading. Once again, it seems that a plausible reason for this frustration was that students may not have been cognitively ready to see example code and were still one or more stages back in the problem-solving process.

Fourth, no students used the ranking feature. It is unclear whether they noticed it at all or saw it and decided not to use it. This is despite many exclaiming how incredibly helpful they found the enhanced messages.

Finally, and most surprisingly, most students 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 students 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. This is related to anecdotal evidence regarding conversations that the researcher has had with students and other CS professors which indicated that computer science majors have fragile egos when it comes to solving their programming homework and do not want to ask for help until absolutely necessary. This point alone makes it clear that help must be built into the AAT such that novices are guided through the problem-solving stages one at a time until their solution is correct.

The lessons learned from the second pilot study resulted in several tweaks to the user interface:

- Drop-down signifiers. Placed a triangle pointing toward the text and left-aligned the text. When the enhanced error message is expanded, the triangle points down. This follows general web conventions about collapsible drop-downs by providing a signifier that clearly communicates the drop-down's affordance (i.e. it drops down).
- Color. The button to click to expand the enhanced error message was blue in the second pilot study. Since blues lie on the frequency that humans have the most difficulty perceiving (Ware, 2012), it may have led to poor

discoverability. Therefore its color was modified to be the same gray was that behind the standard message. The idea is that it would look as one would expect more of the same type of information to look.

- Neutral text. The label "Need More Help?" was too threatening to the ego of novice programmers. Even though the researcher thought that the text was neutral, it was not. It was therefore changed to "More information."
- Removed example code with comparison fix. The students who really needed it did not benefit from it and were generally worse off for it. Furthermore, it substantially increased the size of the enhanced messages. Results from the first pilot study indicated that students had difficulty with the enhanced messages because it substantially increased their cognitive load. Without a clear benefit to students and the possibility of increased cognitive load on the students struggling the most, the example code with a similar error and how that error was fixed was removed. However, some error messages may still have example code snippets. Great care was taken to make sure that the example was stripped-down enough to not be mistaken for the student's own code, yet substantial enough to be helpful.
- Removed ranking. Students did not use it and with the removal of the sample code it became obsolete.
- Background. The entire enhanced feedback message was put into an HTML element with the same gray background color as the one behind the standard message. This was done to further solidify in students' minds what was part of the enhanced message and what was not. In the second

pilot study, only the blue button separated the standard from the enhanced feedback and there was no clear functional grouping.

Standard and Enhanced Error Message Quizzes

After using the results of the pilot studies to update the user interface of the ECEMs, participants enrolled in CS1 for Spring 2017 at ACU were given ten quizzes in class to determine if the enhanced error messages were more helpful than the standard compiler messages. This helps to answer RQ2b: if students are reading the enhanced messages, how do the enhanced messages help them better understand the error? In order to provide a control group and an experimental group, the class of 31 was divided into two roughly equal groups: A and B. Each quiz contained a code snippet with a bug that would lead to a specific compile error, a feedback message from the AAT 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 odd-numbered quizzes, the students in group A saw only the standard compiler error message as feedback from the AAT, while the students in group B saw the standard compiler error message as well as the enhanced error message from the AAT. 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 ten quizzes. Thus, each student saw a standard message for five quizzes and the enhanced message for five quizzes. Each quiz contained a different code snippet with a different compile error and thus a different feedback message from the AAT.

The compile errors were chosen to be the ten most-encountered errors from this problem from the previous five years. These quizzes were given in order to determine if

the enhanced messages were actually more helpful than the standard messages alone, helping to answer RQ2b. This data was gathered as a classroom enhancement quiz and all results will be published in aggregate and anonymously. At the time of writing, only five quizzes have been given with the other five to be taken throughout the rest of the semester. The results of these quizzes are discussed in Chapter 4. Below is a sample of the one of the quizzes containing both the standard and enhanced messages:

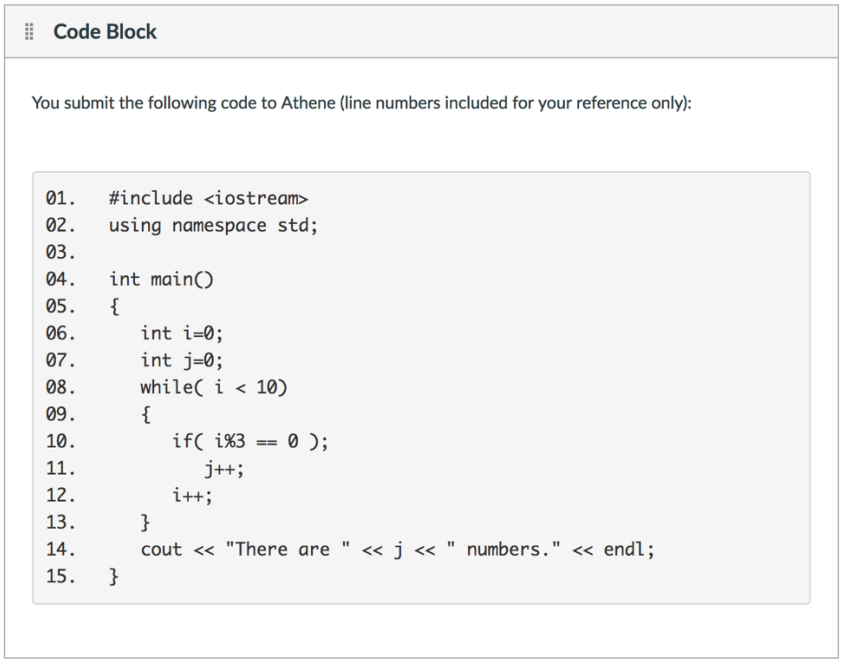


Figure 20. The Code Block segment from quiz "Athene Error Messages 4A."

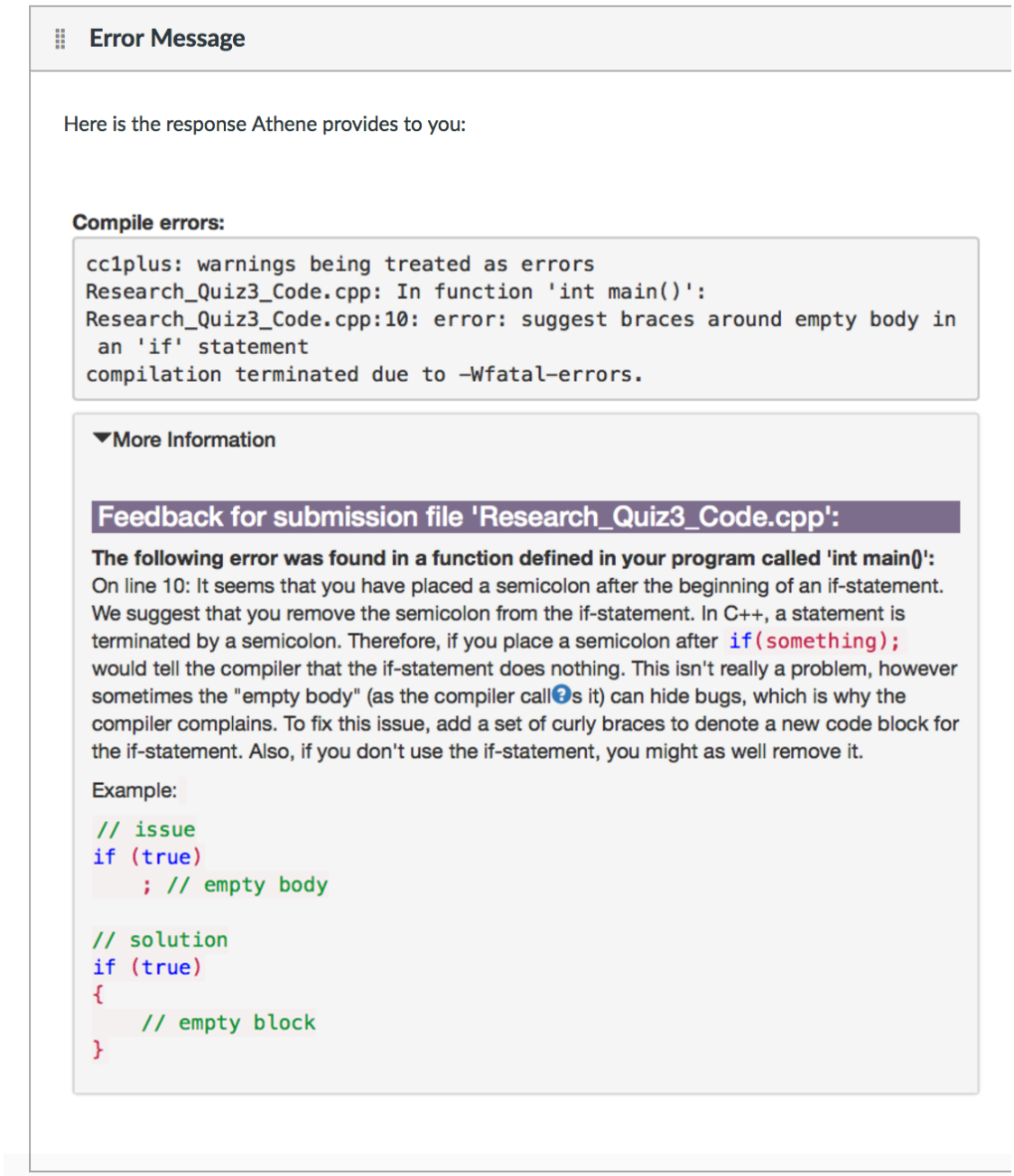


Figure 21. The Error Message segment from quiz "Athene Error Messages 4A."

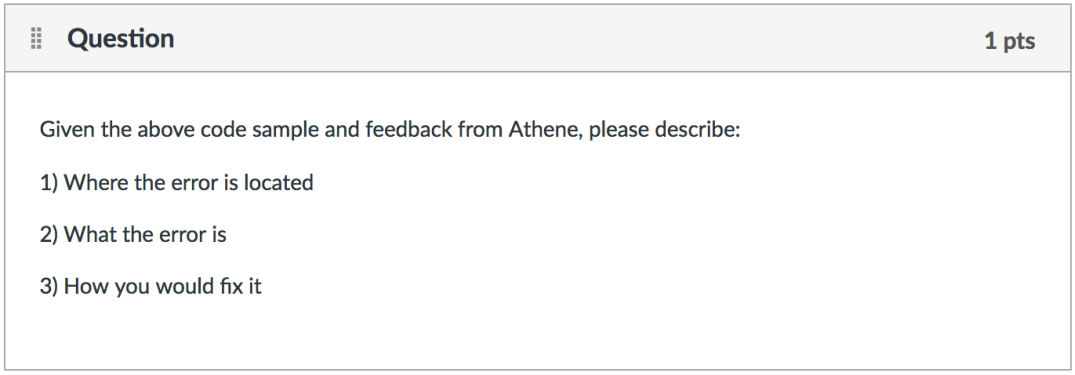


Figure 22. The open-ended short-answer segment from quiz "Athene Error Messages 4A."

Full Usability Study

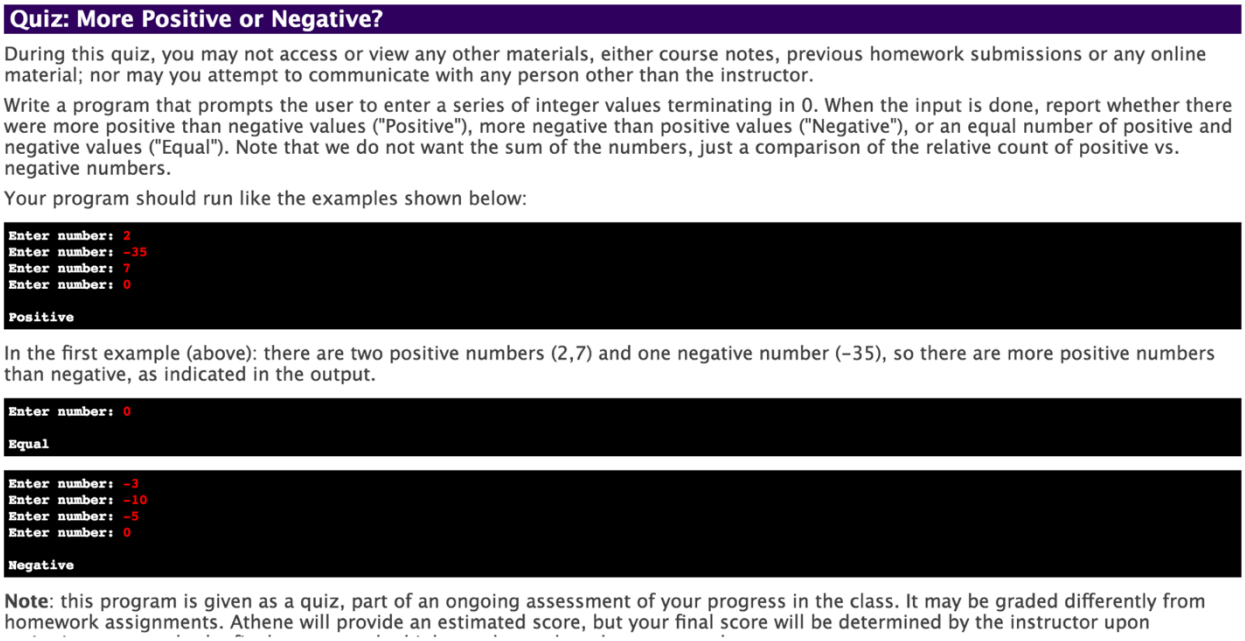
Introduction

The English department at ACU usually has a week during the semester where the freshmen writing courses do not hold class and instead each student meets one-on-one with the professor to discuss their writing and receive personal feedback. As a classroom enhancement, the same was done in CS1 during week 6 of classes for Spring 2017. The participants were therefore all 31 students from that particular CS1 class. Each student met one-on-one with the professor or the professor's TA's 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 an Athene problem and must solve it in 35 minutes. Each one-on-one meeting lasted 60 minutes. Students were asked to think aloud while they solved the problem, especially when they see the enhanced feedback messages.

Procedure

The general format of the usability test follows Rubin and Chisnell (2008) and Krug (2014), 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. (2013) and Whalley and Kasto (2014). 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, in 35 minutes. This particular problem 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. See screenshot in Figure 23 showing this problem in Athene:



Quiz: More Positive or Negative?

During this quiz, you may not access or view any other materials, either course notes, previous homework submissions or any online material; nor may you attempt to communicate with any person other than the instructor.

Write a program that prompts the user to enter a series of integer values terminating in 0. When the input is done, report whether there were more positive than negative values ("Positive"), more negative than positive values ("Negative"), or an equal number of positive and negative values ("Equal"). Note that we do not want the sum of the numbers, just a comparison of the relative count of positive vs. negative numbers.

Your program should run like the examples shown below:

```
Enter number: 2
Enter number: -35
Enter number: 7
Enter number: 0

Positive
```

In the first example (above): there are two positive numbers (2,7) and one negative number (-35), so there are more positive numbers than negative, as indicated in the output.

```
Enter number: 0

Equal
```

```
Enter number: -3
Enter number: -10
Enter number: -5
Enter number: 0

Negative
```

Note: this program is given as a quiz, part of an ongoing assessment of your progress in the class. It may be graded differently from homework assignments. Athene will provide an estimated score, but your final score will be determined by the instructor upon

Figure 23. Screenshot of the quiz in Athene. The remaining text that is cut off could not be fit into the screenshot, but is not relevant to the problem itself, but rather pertains to grading.

While solving the problem, the observer took extensive notes on what the student did and what they said they were thinking. Ericsson and Simon (1993) recommended the following important methodological guidelines, which were all followed closely:

1. This is not a social encounter. Make that clear by sitting behind the participant. The focus is on the participant completing the task, not the interaction between observer and participant.
2. Give a short practice session so the participant can become familiar with the "think-aloud" protocol.
3. Social interaction is minimized. For instance, if the participant stops talking, saying, "keep talking" instead of "tell me what you are thinking," the latter of which might be understood by the participant as an invitation to be social with the observer.

4. Participants always told to focus on completing the task. The only way to obtain the same result using a think-aloud protocol as compared to when the participant is silently thinking is by keeping them singularly focused on completing the task.

After the students completed the problem or the time limit expired, students were asked up to five questions and their responses were recorded. Some questions may not have pertained to that particular student, depending on their experience solving the quiz. These questions were (as exactly reproduced from the observation sheet):

1. When you encountered the enhanced feedback messages (with the "More information" drop-down), were they helpful?
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? (probe: 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 soon before the deadline do you usually make your first attempt (uploading your program to Athene) on your homework?

Finally, the student received feedback and encouragement on their programming and problem-solving process for approximately 20 minutes. This has important benefits for the students. First, because an expert watched them do a homework problem, they can help pinpoint places in the process where the student is weakest (e.g. problem-solving, syntax, tracing, test cases, etc.). Second, if the student tinkered with the code until a

solution came about, or tried to throw syntax at the wall until something stuck, the expert can discuss this approach and help clarify or demystify certain elements. Finally, if the student did not read the enhanced messages (or perhaps read them sparingly) and could have gained obvious benefit from doing so, this resource can be discussed with the student so they might use it more effectively in the future. In this way, the exercise was designed with the hope of catching bad habits, poor mental models, and bad practice early in the semester and to correct it.

Because this was run as a classroom enhancement, it was not run under IRB. However, because data was still collected from the process, IRB representatives at both NSU and ACU were consulted and strict precautions were taken to protect student rights:

1. Students were given consent forms where they could opt-in for their data to be used for publication. It was assumed that students were opted-out by default and therefore must opt-in in order for their data to be used. Since the researcher is also the professor for CS1 in Spring 2017, a different professor within the department handed out a form to each student and explained what they meant while the researcher was not present. This different professor then collected and kept the forms in his office until after grades were submitted at the end of the semester. In this way, if a student chooses not to opt-in, the researcher has no way of knowing and therefore no way of potentially influencing the student's grade or biasing the researcher's actions towards said student.
2. Students were also given FERPA release forms at the same time as the general research opt-in. The FERPA release form was provided to allow students to opt-in

to let the researcher use their grades anonymously, in aggregate, in published research.

3. All data from the classroom enhancement that is published (including in the present research project) will be done so in aggregate and/or anonymously.

Instrumentation and Data Collection

Athene

Athene is the AAT developed and used extensively at ACU (Towell & Reeves, 2009). It is also used at Lipscomb University in Nashville, TN. Students view the problem specifications and then are allowed to submit their code. After submission, students receive detailed feedback regarding their code's failure to compile, failure to pass a particular cases, or successful completion of all test cases. Originally, Athene allowed as many submissions as students wanted. However, it was shown empirically that adding throttling to assignments, e.g. no more than 3 submissions in 15 minutes, improved student learning outcomes (Pettit, et al., 2015). Every submission is stored in a database with the oldest entries being from 2009.

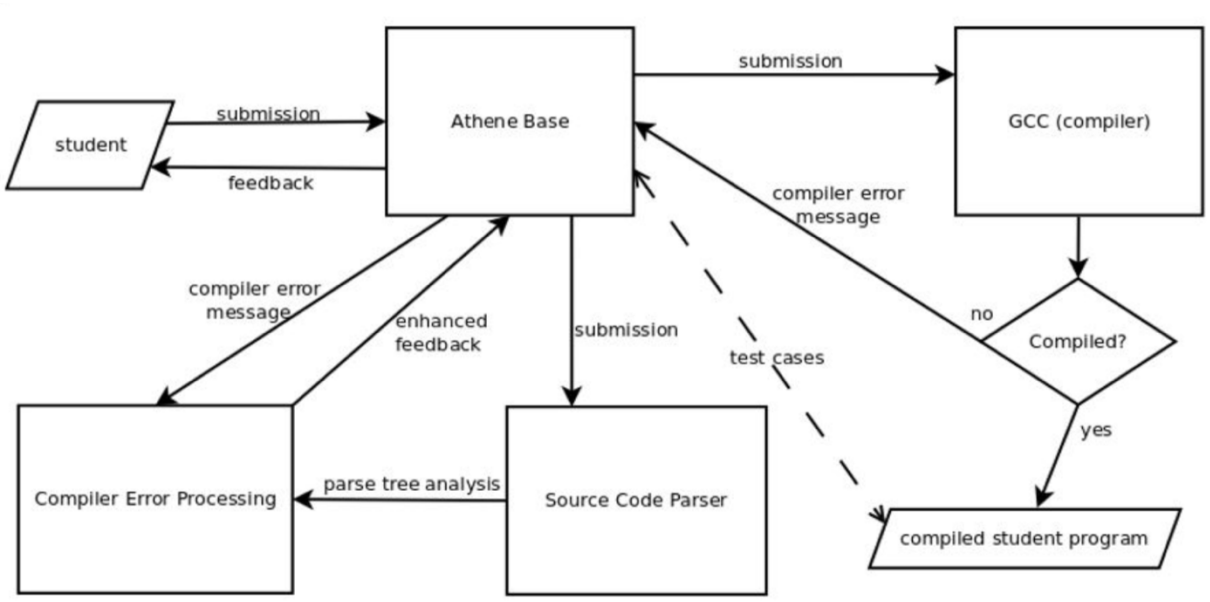


Figure 24. The design of the Athene system. This is the latest version of the system, updated by a senior Computer Science major, Roger Gee, in 2016.

Therefore, all submissions from the two pilot studies and the full study are stored in the Athene database which can then be pulled down and analyzed. Important data from the database for this research are: user, problem id, problem name, submission time, submitted source code, feedback response to submitted source code, and score.

Canvas

ACU uses Canvas, a learning management system (LMS) created by Instructure (https://www.instructure.com/). This tool facilitates the error message quizzes. For each quiz, the important data from Canvas for this research are: student, response to short-answer question.

Google Drive

The researcher has a Google Drive folder where all data is securely stored. This includes the observation sheets that were used during the full usability study, tabulated results from quizzes, and both raw and filtered Athene database data. Google Drive keeps track of when each edit is made, making it possible to create a timeline of participant movement through the six learning stages (see Appendix D).

ATLAS.ti

The qualitative observations of student behavior and verbalizations during the full usability study and ethnographic interview questions afterward were put into ATLAS.ti for coding and tagging.

Analysis

A common analysis rubric used in ethnographic and ethnomethodologically-informed studies is triangulation, which attempts to explain "the richness and complexity of human behavior by studying it from more than one standpoint…by making use of both quantitative and qualitative data" (Cohen et al., 2011, p. 195.). Therefore, the approach the discussed above uses mixed-methods triangulation: pilot studies to refine the ECEM UI, error message quizzes to determine if those changes were successful, and a think-aloud study to observe novices work through the six problem-solving stages while using an AAT.

Quantitative

The data collected from the error message quizzes and full usability study was quantitatively analyzed in the following ways.

Quizzes were graded on a pass/fail or, rather, understand/does not understand basis. This grading was done liberally such that any student who seems to understand the error was given credit for it as if they did, which follows the approach to grading taken by Marceau et al. (2011a). In order to answer RQ2b, the number of students who do not understand the error in the control group was be compared to the students who do not understand the error in the experimental group. The researcher predicted a statistically significant difference between the two groups with the experimental group having fewer incorrect responses overall.

The data from the full usability study was also quantitatively analyzed. Here, the full usability study is considered to be the experimental group whereas the control group is the previous three semesters (Fall 2015, Spring 2016, Fall 2016), which used the naïve enhanced feedback messages. Enhancements made for the pilot studies in the Fall of 2016 were kept to a private sandbox and so students in CS1 during that semester did not see anything other than the naïve enhanced messages. Even though Athene has data from 2009, the feedback messages from Spring 2015 and before were not enhanced at all. Therefore, the data from Fall 2009 – Spring 2015 is not the same as from Fall 2015 – Fall 2016. The practical quiz "More Positive or Negative?" (Figure 23) for these two groups was compared in the following ways:

1. Average completion time. The average time it took students in CS1 for each semester to complete the practical quiz.

2. Average number of completions. The average number of students in CS1 for each semester that completed the practical quiz.
3. Average score. The average number of students in CS1 for each semester that completed the practical quiz.
4. Repeated compile errors after enhanced message. The number of students in CS1 per semester who repeated the same compile error after seeing an enhanced message while taking the practical quiz.

Qualitative

The observational and ethnographic data taken from the full usability study was analyzed using ATLAS.ti. The researcher used the software to discover the "big picture" that organically arises from the data, which helped identify concrete features that can be implemented in the AAT to improve novice metacognitive awareness. This is process of letting the big picture arise organically from the data is known as "grounded theory" (Glaser & Strauss, 2009). Before performing this analysis, it was impossible to know what that big picture would be. According to Lazar et al. (2017), grounded theory requires coding, grouping of concepts, grouping of concepts into categories, and finally formation of a theory. The tagging stage consisted of identifying any interesting phenomena that appears in the raw data. This included: student confusion, encountering a specific enhanced error message, clicking to expand the enhanced error message, read enhanced error message and was helpful, read enhanced error message but was not helpful, student response to error message per the rubric of Marceau et al. (2011a), and student movement through the six problem-solving stages. Using the rubric of Marceau et

al. (see Appendix A) is included in the tagging for the sake of external validity. Some of these tags regard whether or not students read the messages and this data answers RQ2a. The rest of the tags help to build a picture of student metacognitive awareness that contributes toward answering RQ1, RQ3, and RQ4. The full list of tags is provided below, listed in related groups:

- **Learning stages**: stage one, stage two, stage three, stage four, stage five, stage six
- **Metacognition**: successfully created conceptual model, had difficulty with conceptual model, struggles to select correct algorithm, mentally tests selected algorithm, does not mentally test selected algorithm, struggles with syntax errors, struggles with test cases
- **Marceau**: DEL, DIFF, FIX, PART, UNR
- **Completion time**: <10 min, <=15 min, <=20 min, <=25 min, <=30 min, <=35 min, did not finish
- **Enhanced messages**: did not need enhanced, did not notice enhanced, opened enhanced, does not click enhanced, enhanced helpful, enhanced not helpful
- **Emotion**: indifferent toward error messages, like toward error messages, dislike toward error messages, blames self, disparages self
- **Error messages**: # of passed test cases, error message 1, error message 2, error message 3, error message 4, error message 5, error message 6, error message 7, error message 8, error message 9, error message 10, error message ?, error message ? without enhanced

- **Overall**: overall click on enhanced first, overall enhanced messages helpful, overall enhanced messages not helpful, overall wait to click enhanced, overall displays metacognition, overall does not display metacognition

Once the data was tagged and coded, the general concepts that emerged were grouped into categories. These categories help to determine the overall effectiveness of the enhanced error messages, identify difficulties students had in the problem-solving process, and help identify where students diverge in that process while solving a programming problem using an AAT.

Contributions of this Study

Effectiveness of ECEMs in AATs

The first contribution of this study is confirmatory evidence that ECEMs in AATs either enhance learning or do not enhance learning for novice students in CS1. As discussed above, this is currently hotly debated in the CSed community. This evidence is presented in graphs, charts, and statistical analysis of the results of the full usability study discussed above, supported by the error message quizzes. This contribution answers RQ2. See Chapter 4 for this data.

Proposing a Metacognitive Framework

The second artifact produced by this study is a framework for improving metacognitive awareness through AATs. This framework is a set of design guidelines for implementing features in an AAT that guide students through the six learning stages and

help make students more aware of where they are in that process. Each of the six learning stages has specific design guidelines that come from the quantitative and qualitative data gathered by this study. This contributes toward answering RQ1, RQ3, and RQ4. See Chapter 5 for the framework.

Summary

The standard compiler error message feedback in Athene was enhanced in Fall 2015. This naïve enhanced message was tested and not found to significantly positively increase student learning (Pettit et al., 2017). However, human factors analysis was not considered in the previous study. To begin addressing this gap, two pilot studies were carried out in Fall 2016, each with six participants. The first pilot study tested the existing naïve enhanced message through a usability test. After the test, the user interface was significantly updated to include lessons learned from the usability test and suggestions from Marceau et al. (2011b) and Hartmann et al. (2010). The second pilot study tested the updated user interface of the enhanced error messages. Lessons learned from the second pilot study resulted in a slimmed-down and streamlined version in order to significantly lower cognitive load.

In order to determine if the new UI was more helpful than standard compiler error messages, ten quizzes were given to the CS1 class at ACU in Spring 2017. These ten quizzes represented the ten most encountered errors for the practical quiz that would be used in the full usability study. Each student saw five quizzes with only the standard compiler error message (control) and five quizzes with the standard message plus the newly updated enhanced error message (experimental). The quizzes were graded and

analyzed to determine if more students understood the error in the experimental group than in the control group. This helps to determine if students find the enhanced error messages more helpful (RQ2b).

A full usability study was conducted with a total of 31 participants. Each session lasted for an hour. The session consisted of observing a student complete a preliminary task to become familiar with the think-aloud protocol, working on a practical quiz for at most 35 minutes, post-quiz interview questions, and then around 20 minutes of personal feedback to encourage the student and correct any bad habits or misconceptions. The results of the full study (experimental) were compared quantitatively to the previous three semesters (control). The observational and interview data was qualitatively analyzed through application of grounded theory: coding, developing concepts, grouping, and formation of theory. The tags discussed helped to identify phenomena that helped to answer RQ1, RQ3, and RQ4. The data was also tagged using the rubric of Marceau et al. (2011a) to provide external validity (see Appendix A).

Chapter 4

Results

Introduction

The results of the study described in Chapter 3 are presented here. The present research project investigates the experiences of novices using an AAT to navigate the six stages of learning to write code and the metacognitive awareness that they display while doing so.[1] Since current research on AATs has only focused on what amounts to stages five and six of the learning stages proposed by Loksa et al. (2016), this chapter will begin by discussing the results of participant engagement and reaction to the newly refined ECEMs (see Chapter 3 for information on the pilot studies and iterative refining of the ECEMs), answering RQ2. This will be internally verified by the results of the error message quizzes (quantitative), analysis of the submission data pulled from Athene from the full usability study (quantitative), the results of observed participant usage of the ECEMs during the full usability study (qualitative), and interviews from after the practical quiz portion of the full usability study (qualitative). It will also be externally verified through the application of the rubric of Marceau et al. (2011a) on observed participant interaction with the ECEMs (qualitative).

[1] Some material in this chapter was published at ICER 2017: Prather, J., Pettit, R., McMurry, K. H., Peters, A., Homer, J., Simone, N., & Cohen, M. (2017, August). On Novices' Interaction with Compiler Error Messages: A Human Factors Approach. In *Proceedings of the 2017 ACM Conference on International Computing Education Research* (pp. 74-82). ACM.

Building from the current starting point in the research literature surrounding stages five and six, this chapter will next discuss the qualitative results of the full usability study regarding metacognitive awareness displayed by participants. This data was tagged, coded, and put into ATLAS.ti. The observational data was tagged in 433 places and from that ATLAS.ti identified 39 unique first order concepts that emerged directly from the observations. ATLAS.ti helped group these first order concepts which allowed the researcher to move toward five distinct second order concepts, which informs the discussion below.

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 25) 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).

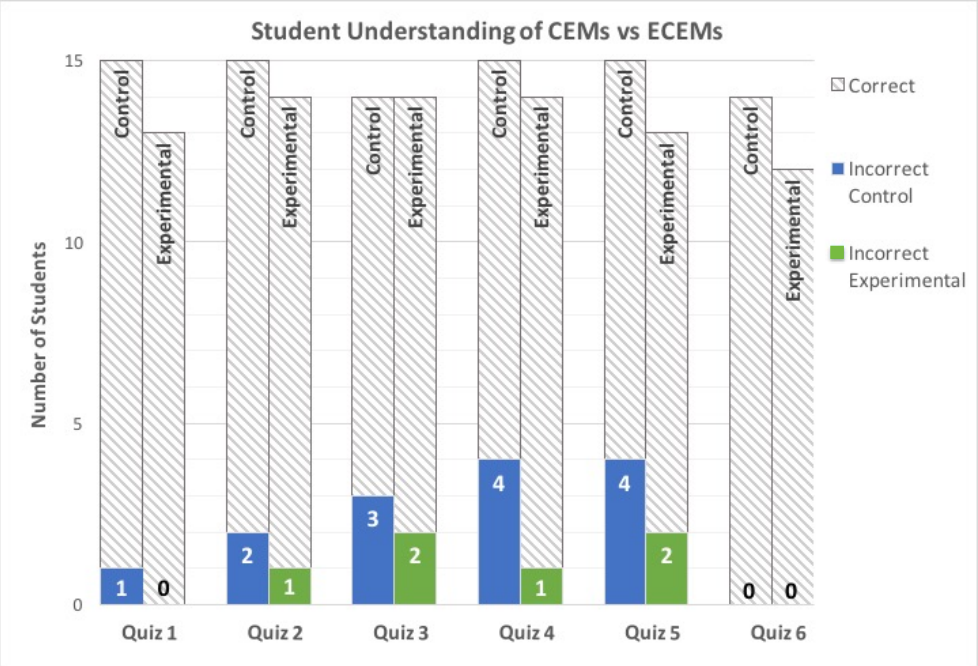


Figure 25. The number of incorrect responses for each quiz in both control (blue) and experimental (green) conditions.

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 26 (rows three, five and six), nine 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 (Figure 26, row six). Another outlier in the opposite direction was the student who incorrectly answered two experimental quizzes, but correctly answered all in the control condition (Figure 26, row two).

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 26. Incorrect understanding of CEM vs ECEM.

Full Usability Study: Program Logs Data from Athene

Data that can be pulled from Athene's database on assessment submissions has been previously reported by Pettit et al. (2015, 2017). 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 full usability study allowed this data to be gathered using Athene for the first time. It is expected that student behavior will 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 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 27). This data indicate that the experiment did not adversely affect student outcomes.

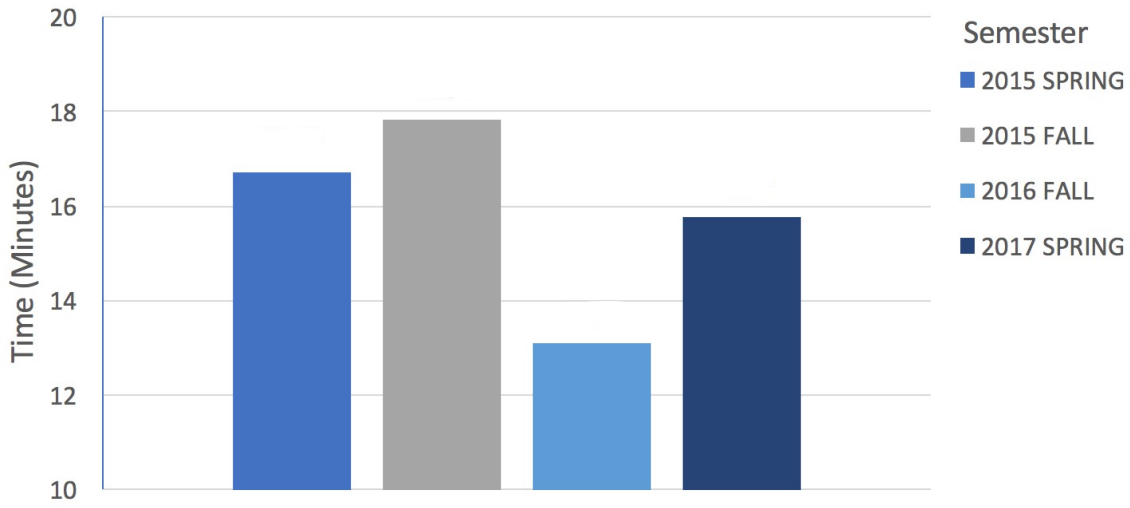


Figure 27. Average time to complete the problem by semester.

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, ten students did not complete the quiz at all, six of which suffered from problems with the basic structure of their code. All of these six students could not remember basic "#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 the choices about which messages to enhance were based 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 six students from the group brings the average score up to 84.8%, which is in range of the control semesters.

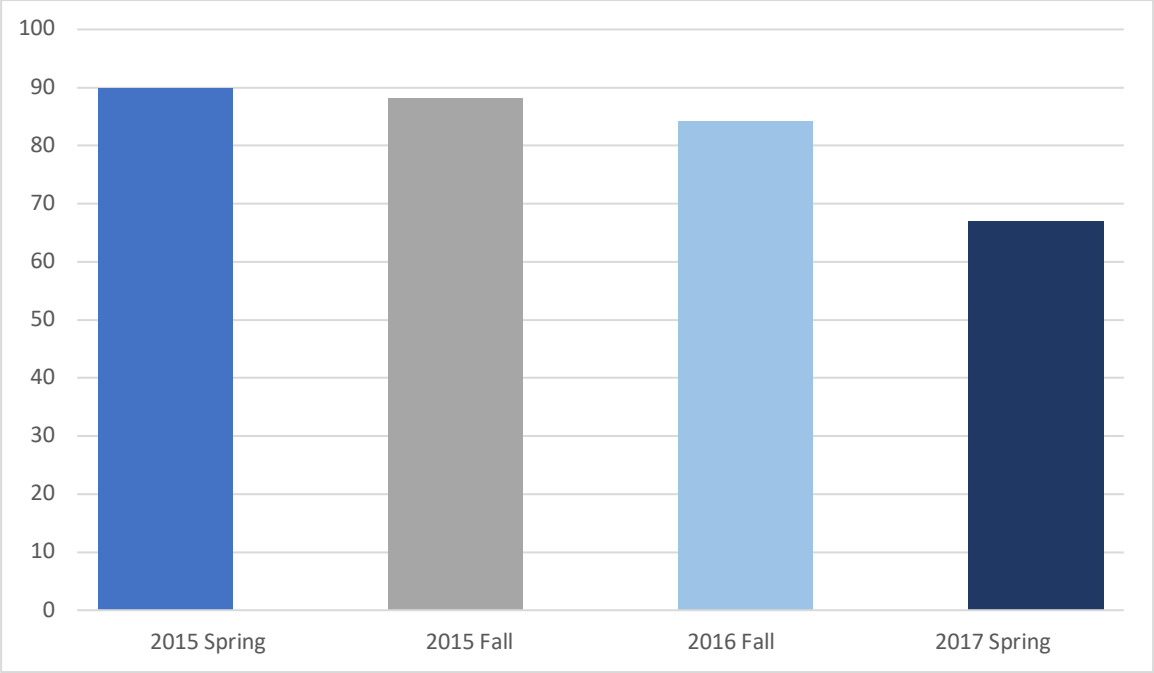


Figure 28. Average score by semester, raw.

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 from the full usability study illuminates a possible explanation.

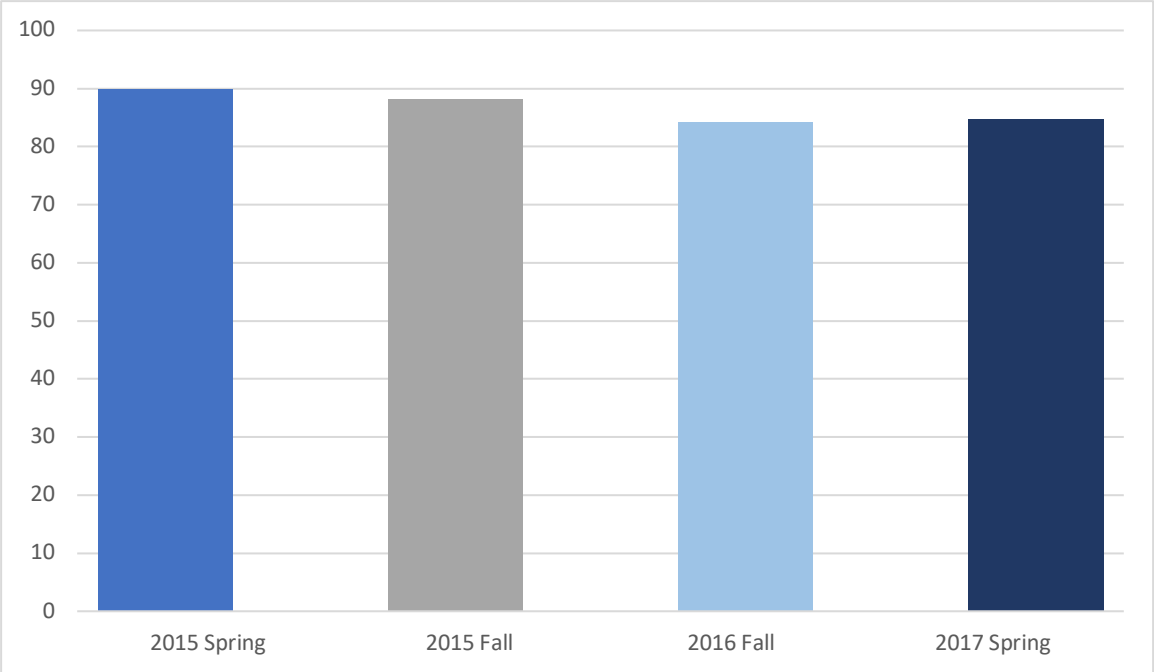


Figure 29. Average score by semester, adjusted by removing the six students who could not remember the most basic parts of their program, which contributed heavily to their failure to complete the assignment within the time limit.

Full Usability Study: Observation and Interview Data

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.

Observational

Although there were 21 students who completed the quiz and ten 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 30 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.

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 30. Student Perception of ECEMs in Complete vs. Incomplete Quizzes

Interviews: 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 31.

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 31. Student Perception of ECEMs in Complete vs. Incomplete Quizzes

Interviews: 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.

Discussion

The results of the error message quizzes compared with the quantitative program log results from the full usability study seem contradictory. The observational and interview data presented above tell a different story. The students who struggled, but ultimately succeeded 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 interview data shows that it was ultimately the ECEMs that helped them across the finish line. This is precisely what is wanted. 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. It is possible that the increased cognitive load of the assessment may have tipped the scales from helpful ECEMs to unhelpful.

Results of Tagging on the Rubric of Marceau et al. (2011)

The observational data from the full usability study were tagged according to the rubric of Marceau et al. (2011a) as a means to test for external validity (see Appendix A).

A successful set of ECEMs will allow students to understand the error and point them to the correction that must be made in the code. Therefore, the more times a student action in response to an ECEM can be tagged with either the PART or FIX tags, the better. As summarized in Table 3, the enhanced messages in Athene were overwhelmingly helpful to students who encountered them. Note that the numbers in Table 3 reflect only the instances where students *read* the enhanced message, a smaller number than the total number of enhanced messages encountered by students. This affirms the results above that the redesigned ECEMs were indeed more helpful for novices involved in the study.

Table 3. *Results of ECEMs tagged on the Rubric of Marceau et al. (2011a)*

Tag	Meaning	Result
DEL	Deletes problematic code	0
UNR	Change unrelated to current error	1
DIFF	Fixes a different error	0
PART	Attempts to take the correct action	8
FIX	Fixes the error	22

Full Usability Study: Observations of Metacognitive Awareness

This section describes from observation during the think-aloud study how students working in Athene moved through the six learning stages outlined by Loksa et al. (2016). This qualitative data from the full usability study will highlight the relevant ways in which automated assessment tools, like Athene, fail to help students implicitly build the cognitive scaffolding necessary for metacognitive awareness. Using these ethnographic stories, the framework for implementing features in an automated

assessment tool to increase metacognitive awareness among novice programmers will be proposed. First, the metacognitive awareness of students who successfully completed the quiz will be examined in order to look for the ways in which they cognitively augmented the shortcomings of Athene. After this group, the metacognitive awareness of students that did not complete the quiz will be examined in order to better understand how the lack of cognitive scaffolding in Athene negatively impacted their performance. Finally, the two groups will be contrasted. Student names in this section were changed for the sake of anonymity.

Students that Completed the Quiz

Several students in the group that completed the quiz finished it in under 10 minutes and surprisingly displayed a similar set of traits. The first was a consistent approach to starting the problem. Observation notes report that at the outset these students, "interpreted the instructions for the problem," and "immediately verbalized a clear conceptual model for the problem." This is followed by a similar pattern of thinking through the problem, thinking about how to solve it, choosing a solution (in this case, using a *while loop*), implementing a solution, and tracing their code with specific test cases in mind. Several students in this group were observed taking a few pauses to think about their chosen solution and their process to solve the problem. One student in this group received one ECEM; the others received none. The student who received this message, Bill, did not read the enhanced portion at first, but read the standard portion, successfully edited his code (tagged as a [FIX] using Marceau's rubric), and then double-checked his edit by opening the enhanced portion, reading it, and agreeing that his fix

was correct. Bill's experience is ideal. By the time these students were receiving feedback regarding test cases, they had successfully moved through the first five stages of problem solving and therefore any failed test cases were quickly interpreted, and the offending code was fixed.

Jane, who took 14 minutes to finish the quiz, thought through the problem, immediately decided on a solution, and proceeded to go through her code and place comments about what she planned to do and then went back and filled it out. This student received one ECEM, did not read the enhanced portion, and immediately successfully edited the code (a [FIX]). The next submission compiled, passed two of the test cases, and failed on the third. She made an edit to her code and resubmitted, receiving the same failed test case message, and then repeated this once more. Finally, after receiving the same test case failure three times, she stopped and carefully walked through her code with specific test cases in mind, found the issue, fixed the code, and finished.

Another student, Patricia, who finished in 18 minutes, read the problem prompt too quickly and immediately began solving it as if it was a problem students in CS1 had previously encountered that semester, "Even or Odd?": given *n* numbers, compute whether there were more even or odd integer numbers provided as input. It seems as though she initially failed to correctly move through problem solving stage one, *reinterpret problem prompt*, which led to moving on to stage two, *search for analogous problems*, correctly choosing of the "Even or Odd?" problem, and finally moving onto stage three, *search for solutions*, but choosing to use the solution for the "Even or Odd?" itself instead of using it as the basis from which to form a new solution to a different problem. However, as Patricia began to write her solution, this approach made less and

less sense, and she quickly realized something was off. She checked the instructions again, but still didn't understand what was wrong - a fascinating case of how forming the wrong conceptual model early on can make it difficult to fundamentally change how one views the programming problem at hand. Finally, after being stuck for a few more minutes, she re-read the instructions a third time and understood. After this, Patricia solved the problem very quickly.

Another interesting group of students that completed the quiz were those that took 30-35 minutes, coming right up against the time limit. Adam, completing the quiz in 30 minutes, read the prompt and immediately showed a clear conceptual model of what the problem required and how to solve it. However, Adam ran into extensive issues with syntax and therefore became stuck on stage five, *implement a solution*. He recognized his deficiency in the particulars of syntax correctness and utilized the enhanced portion of the ECEMs to his advantage, finally solving it on the seventh submission.

Finally, Wayne, who completed the problem in 33 minutes, ran into the same issue as Patricia, confusing the problem for "Even or Odd?" However, Wayne did not realize his mistake early on. At multiple points in the session he carefully talked through his algorithm, revealing his incorrect conceptual model. After writing his solution, built for the "Even or Odd?" problem, he encountered one compile error, fixed it, and moved on to the final stage, *evaluate implemented solution*, where he failed the first test case. Wayne looked at the expected output compared to the actual output of his program, made an edit, and then passed the next test case. He continued failing test cases, adding to his code to create the right output, and failing the next test case. His code grew longer until he had passed 10 test cases, a process that took just over 30 minutes in which he became

increasingly frustrated. Finally, at 31 minutes he re-read the problem prompt and exclaimed, "Oh! Wait! This just hit me that it's doing positive and negative rather than evens and odds. I don't know why that happened," and very quickly solved the problem. In this case, failing to correctly navigate the first few stages of problem solving led to an incorrect feeling of accomplishment and an incorrect conception of location in the problem-solving process. By solving compilation problems and working through multiple test cases, Wayne felt as if he was very close to solving the problem when he was actually very far away. In this case, his lack of metacognitive awareness almost cost him the quiz.

Students that Did Not Complete the Quiz

The 11 students who did not complete the quiz all failed to successfully move through at least one of the problem-solving stages. If the way to a correct solution can be thought of like a path from stage to stage, these students often diverged very early, backtracked frequently, and never returned to the crucial juncture to take the correct path. The most frequent issue these students encountered was a failure to build a correct conceptual model of the problem. Unable or unwilling to spend the time to successfully navigate stage one, *reinterpret problem prompt*, many of these students searched for analogous problems and solutions to the wrong problem. And, unlike Wayne above, these students did not stumble into the realization they had the wrong conceptual model. The automated assessment tool, Athene, did not alert these students to this failure of metacognitive awareness, allowing them to meander down the wrong path, totally lost until the quiz time had expired.

The most obvious example of a failure to create a correct conceptual model can be seen in the experience of Theo. Theo spent nearly a third of his quiz time reading and re-reading the quiz prompt. At one point, halfway into the quiz time, the researcher noted that he, "just keeps repeating the same phrase from the instructions, 'if the number of positive is greater than the number of negative,' over and over again." Eventually, Theo wrote some code and submitted it, and received a standard CEM. He spent the rest of his time trying to understand this message. Since it was his only syntax error, if he had corrected and submitted it again, Athene would have begun running his code against the set of test cases. This was not the feedback that Theo needed in order to succeed. His time expired while he was re-reading the prompt for the eighth time.

One student, Neil, is a good example of what happens when one fails to navigate each of the stages. After skimming the problem prompt, Neil immediately began coding without stopping to think through stage two, *search for analogous problems*, stage three, *search for solutions*, or stage four, *evaluate a potential solution*, jumping right to stage five, *implement a solution*. This was evidenced by his statement out loud after a few minutes, "What I'm wondering is if I need the prompt for input to be in the loop or not," followed quickly by removing the prompt entirely. A minute later he created two variables and said, "Somehow I'm going to let those represent positive and negative values. I think I'll have to do that in my while loop." At that point in the quiz, his code was structured to accept two integer values and report if they were positive or negative, which is not the correct problem. Minutes later he said, "I'm going to mentally run through it now," but did so without any specific test cases. All of this shows a confusion about what problem he was trying to solve, how to solve the problem he thought it was,

and an inability to evaluate his own solution. Finally, he submitted his code to Athene and spent the rest of his quiz time working through compiler errors. Slowly working through seven CEMs/ECEMs seems to have provided a false sense of progress to Neil because his program, even without syntax errors, was very far away from a correct solution.

Thomas successfully navigated stages one and two, failed to solve stage three, and was subsequently totally unprepared to move into stages four through six. Early on Thomas said things like, "I'm trying to figure out how to...that's not going to work," and, "I'm trying to figure out how to make it count the positive ones. I don't know how to...that's going to be my issue." He continued tinkering with his code and said, "I just don't know how to see if there's more positive or negative." Thomas' comments reveal that he understood what he needed to do, but had great difficulty successfully getting through stage three, *search for solutions*. Frustrated and eager for some feedback, Thomas submitted his code, saying, "I guess I'll run it just to see what it will say." His code was syntactically valid and so Athene began running test cases. Once in stage six, *evaluate implemented solution*, Thomas struggled with the first test case for the remainder of the time, unsure as to how to convert the specified input into the correct output. Near the end of his quiz time, Thomas said, "I feel like I'm close, but I just don't know how to count up positive and negatives. Why is this not working?" The feedback from Athene had given Thomas a false sense of progression through the problem. He felt very close, but without finding a solution in stage three from which to build his own solution, he was actually quite far from completion. Thomas' experience was almost

exactly repeated in the experience of two other students, with one saying, "really close to finishing this, I think," when he was quite far away.

Several other observations are worth mentioning as well. A few students said that they usually solve the problem through trial and error. This behavior shows that automated assessment tools, such as Athene, allow for submission of code immediately without any assurances that the student understands the problem - they are focused solely on correctness via syntax and test cases. Another issue researchers noticed is that several students became very frustrated with Athene and the quiz, with one student even calling herself and her code "stupid." This highlights that without appropriate feedback from Athene, students can feel hopelessly lost, become frustrated, and form a very negative opinion of the discipline. Finally, one positive behavior in this group was displayed by Jenny who successfully navigated stages one through three, stalled in stage four, *evaluate a potential solution*, and finally got out a piece of scratch paper and sketched the flow of the program. This helped her immensely and she was able to immediately move on to stage five, *implement a solution*. Unfortunately, by the time Jenny successfully moved into stage five, her quiz time was nearly over.

Comparing Complete vs. Incomplete Students

The most glaring inconsistency between those who completed the quiz and those who did not is in the initial formation of a correct conceptual model for the problem, which corresponds to stage one, *reinterpret problem prompt*. This is perhaps the single greatest weakness in modern automated assessment tools: the tool merely presents the problem and trusts that the successful student will eventually conceptualize the problem

correctly. Furthermore, there are no measures between viewing the problem and submitting source code to ensure that the student understands what they're being asked to do. As it is, tools like Athene treat every student submission the same: as if they are just a few syntax errors and edge case fixes away from a correct submission. The experiences of Wayne and Patricia, who both realized their incorrect conceptual model, were also seen in multiple students who did not complete the quiz, only these others were not fortunate enough to realize their error. It's very possible that these students would have completed the quiz if Athene had offered to help them form the correct conceptual model at the outset. This does not only benefit the poorer performing students; Wayne completed the quiz, but just barely. It's likely that Wayne would not have taken 33 minutes to solve the problem had he been operating under the correct conceptual model the entire time. Both Patricia and Wayne also illustrate that re-reading the problem prompt may not help a student that has formed an incorrect conceptual model due to the difficulty in dislodging it once formed.

After forming a correct conceptual model, Jane and several other students who completed the quiz took the time at the outset to build out some scaffolding inside their code by placing comments about how they intended to solve the problem. These students used this technique to navigate stage two, *search for analogous problems*, by thinking back to similar problems they have encountered, and stage three, *search for solutions*, by thinking through how those previous problems were solved, and finally stage four, *evaluate a potential solution*, by sketching the solution in comments before actually implementing it. This strategy proved to be a helpful way of thinking through an approach before committing to any code. Jenny, who did not complete the quiz, also

employed this strategy, but did so far too late into the quiz time to be of any benefit. Meanwhile, many of the students who did not complete the quiz read the prompt (often briefly) and jumped directly to coding, skipping stages two through four entirely. This proved disastrous for them as they wandered aimlessly, hoping to eventually stumble on a solution.

Another important distinction can be drawn in stage five, *implement a solution*, and the number of enhanced messages read by students. The incomplete quiz participants read 9 enhanced messages, while the participants that completed the quiz read 23 enhanced messages. Even though the complete and incomplete group received roughly the same number of ECEMs, the complete group read them far more often. From the quiz data, reading the enhanced messages seems to have been a defining factor to complete the quiz for several students who might not otherwise have done so. Students such as Adam, who correctly navigated stages one through four, but became stuck on stage five with syntax errors, heavily relied upon and successfully utilized the enhanced messages to reach a correct solution. As discussed above when tagging the enhanced messages against the rubric by Marceau et al. (2011a), the success of Athene's enhanced messages proved to be very helpful for many of the students who completed the quiz. Most perplexing is the general behavior of the students who didn't utilize the enhanced messages. One student in particular saw the same ECEM 15 times, but never once clicked on the enhanced message to expand it and read it.

Also in stage five, *implement a solution*, some in the incomplete group attempted to work through the received CEMs/ECEMs, but had no idea they had incorrectly navigated all previous stages. Having used Athene for the assigned homework problems

in weeks one through five of CS1 thus far, these students associated receiving CEMs/ECEMs with being mostly complete, which many admitted during the think-aloud session or in the post-session interviews. This poor sense of location in the problem-solving process ultimately distracted them from the real issue at hand: even if they could get their code to be syntactically correct, it was not going to solve the problem.

Finally, the experiences of Neil and Thomas can be juxtaposed with the experience of Jane to offer a window into stage six, *evaluate implemented solution*. When Jane began receiving test case feedback from Athene, she had already successfully navigated stages one through five and was therefore ready to incorporate the feedback accordingly. Because she was solving the right problem, had chosen an approach that could solve the problem, and had correctly implemented the code for her solution, the feedback about failed test cases that she received enabled her to tweak her code and quickly arrive at a correct solution. Neil and Thomas, on the other hand, both also reached stage six, but because they had incorrectly navigated stages one through five, the feedback they received was misleading, at best. Because Athene told them which test cases they had failed, Neil and Thomas assumed that they should evaluate these feedback messages and that doing so would lead them to a correct solution. Unfortunately, because some test cases in Athene are randomly generated, no amount of failed test case feedback would have helped them correct their fundamental misunderstanding of the problem.

The above discussion reveals the most common difficulties faced by the students in the study and are summarized in Table 4. These are the second order concepts mentioned above.

Table 4. *Observed difficulties to metacognitive awareness by novices using AATs*

Metacognitive Difficulty	**Explanation**
Forming	Forming the wrong conceptual model about the right problem
Dislodging	Dislodging an incorrect conceptual model of the problem may not be solved by re-reading the prompt
Assumption	Forming the correct conceptual model for the wrong problem
Location	Moving too quickly through one or more stages incorrectly leads to a false sense of accomplishment and poor conception of location in the problem-solving process
Achievement	Unwillingness to abandon a wrong solution due to a false sense of being nearly done

Summary

Beginning where the current literature ends, the present research study first sought to take a human-factors approach to enhancing CEMs in an AAT, Athene. Through user testing and the small number of research papers on the subject, the ECEMs in Athene were iteratively refined. Following this, a larger ethnomethodologically-informed usability study using a think-aloud protocol was conducted among novice programmers in the Spring of 2017 at Abilene Christian University with 31 participants. This was augmented by a series of error message quizzes given to the students. The quantitative data, such as student performance on error message quizzes and submission data in Athene's database, were combined with qualitative data from observations and interviews in order to answer RQ2: *Are ECEMs helping students evaluate their potential solution?*

Specifically, this question was broken down into two sub-questions, RQ2a: *Are students reading the enhanced messages?* and RQ2b: *If students are reading the enhanced messages, how do the enhanced messages help them better understand the error?* The results above show that, yes, students do read the ECEMs more often than not and that the ECEMs are helping them to better understand the error.

Qualitative data from the full usability study were analyzed to answer RQ1: *When students diverge on a specific learning stage, what factors caused them to do that?* and RQ3: *When students diverge on a specific learning stage, submit their program, and receive an ECEM, how do they interpret it?* The ethnographic stories presented above provide an in-depth look at student problem-solving ability and their metacognitive awareness while doing so throughout all six stages. These stories highlight how some students who already have some amount of metacognitive awareness were able to mentally augment Athene as they solved the problem. The stories also highlight how many students who have never developed the skill of metacognitive awareness diverged on specific learning stages, which may have been prevented given some kind of cognitive scaffolding in Athene. Finally, the stories also highlighted how some students struggled to complete the assignment, but due to the successful use of ECEMs, they were able to do so.

With all of the data presented above in mind, the present research project now turns to RQ4: *How can AATs be augmented to support metacognition in novice programmers in CS1?* As mentioned above, the answer to this question will take the shape of a framework of features that can be implemented in any AAT that will implicitly

reinforce metacognitive awareness in novice programmers. This is presented in Chapter 5, Conclusions.

Chapter 5

Conclusions, Implications, Recommendations, and Summary

Conclusions

Learning to program is a hard task and novices are constantly cognitively overburdened (Lister, 2008; Guzdial, 2015a). This can be alleviated by supporting novices in building cognitive scaffolding and metacognitive awareness through six distinct learning stages (Loksa et al., 2016). A scalable implementation of the method of Loksa et al. would be to use AATs, which many universities are already using to help students learn programming and is therefore a somewhat ubiquitous place to start. Some AATs have been improved to support the fifth learning stage by providing usable feedback for student program submissions. A few studies have attempted to approach the design of these feedback messages from a usability or human-factors perspective (Nienaltowski et al., 2008; Hartmann et al., 2010; Marceau et al., 2011a). However, it is currently debated in the literature whether enhancing compiler error message feedback empirically improves student learning (Denny et al., 2014; Guzdial, 2014; Becker 2016a; Pettit et al., 2017). However, there is no discussion in the literature on implementing in AATs the means to help students through the other five learning stages.

To address this problem and answer the research questions posed above, the present research project presented an ethnomethodologically-informed usability study with a think-aloud protocol where CS1 students were observed solving a programming problem with an AAT, Athene. The results detailed above show how some students effectively used the enhancements made to support stage five (ECEMs), how some

students mentally augmented the tool when it did not implicitly support their metacognitive awareness, and how some stumbled due in part to the tool's lack of such support. This chapter will revisit each research question discuss them more fully in light of the results presented above.

RQ1. *When students diverge on a specific learning stage, what factors caused them to do that?*

From the ethnographic stories presented above, no definitive answer can be ascertained. However, there are also clearly likely contributing factors that were observed in the participants. The largest contributing factor is a lack of cognitive scaffolding in Athene to guide students through the six learning stages and, after using it to write a semester's worth of programs, implicitly build their metacognitive awareness. This can be seen clearly in the prevalent tendency among students who did not complete the quiz to skim the assignment instructions (stage one) and jump straight into coding (stage five), fitting into Yuen's (2007) *need to code* category. This *need to code* tendency could be mitigated by placing a few distinct hurdles in front of the student before they can begin coding.

The lack of cognitive scaffolding as a contributing factor to divergence at a specific learning stage can also be seen in the students who failed to correctly navigate stage four, *evaluate a potential solution*, and therefore the code they wrote had no chance of success. The observations above discuss how these students submitted their code, received compiler errors, fixed those errors, and then began receiving test case error feedback. Students in this situation often misinterpreted this feedback from Athene, that the submission now compiled and was being verified by test cases, as evidence that they

were on the right path and close to a correct solution. The students who shared this experience unanimously remarked that they felt they were very close when they were all very far from a correct solution. From this example, it is clear that students are already interpreting the two stages of responses that Athene currently offers, compilation feedback followed by testing feedback, but that they are doing so in a way that reflects a poor mental model of how Athene works. A stronger cognitive scaffold to guide students through the entire problem-solving process would mitigate at least some, if not most, of the student frustration with Athene's, and many other AATs for that matter, somewhat misleading feedback.

There are other contributing factors to divergence in specific learning stages that can be found in the results above, such as an overall lack of preparedness and a reliance on previous work without committing necessary minutiae to memory. These factors are outside of the scope of this research project and any programming course. If students are unwilling to do their work or take it seriously, not much can be done about that. However, improving the cognitive scaffolding in current AATs would be beneficial and easily incorporated into existing curriculum.

RQ2. *Are ECEMs helping students evaluate their potential solution?*

To better answer RQ2, this question was broken down into two sub-questions, RQ2a and RQ2b.

RQ2.a. *Are students reading the enhanced messages?*

Observational and ethnographic data above seem to indicate that novices in CS1 do, in fact, read ECEMs. Students also generally find the ECEMs more helpful than the standard CEMs. Barik et al. (2017) performed an eye tracking study with intermediate

students to determine if they read CEMs from standard compilers. They found that overwhelmingly, more experienced students do read the messages. While their study centered entirely around intermediate students and could quantitatively answer the question due to the use of eye-tracking equipment, their findings lend more weight to the findings of the present research project.

RQ2.b. *If students are reading enhanced messages, how do the enhanced messages help them better understand the error?*

Since the answer to RQ2a was "yes," this allows RQ2b to now be answered. Although the students who completed the quiz received about the same number of errors as the students who did not complete the quiz, 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). 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 number of instances for those who did not complete the quiz (3). From this data, it seems that reading the ECEMs proved helpful for the completion of the quiz. Another explanation is that the students who completed the quiz simply had a lower intrinsic cognitive load than those who did not complete the quiz. Since error rates have been used as an indirect measure of cognitive load (Ayres & Sweller, 1990; Ayers, 2001) and two-thirds of the students who completed the quiz received roughly as many errors as those who did not complete the quiz (see Figure 30), this seems possible. If so, it could mean that the enhanced messages had nothing to do with whether or not they were helpful. A

related explanation is known as the "conscientious student effect" (Salleh et al., 2010), which is that conscientious students naturally perform better on these types of assessments. This explanation would also mean that reading the ECEMs has nothing to do with the design of the ECEMs, but instead is dependent on the individual student. The present research study attempted to control for these other explanations through the error message quizzes, which were tested via a between-subjects test (Lazar et al., 2017), and independently verified that the ECEMs were more helpful.

Results from the interviews also helped to elucidate the answer to this question. The small group of students arrived at a correct solution with little to no compilation errors mentioned in the post-assessment interview that they did not use the ECEMs, even though they saw them, because they didn't need them. On the opposite end of the spectrum is the small group of students who struggled to complete the quiz due to compilation errors. These students mentioned how much the enhanced portion of the CEM helped them across the finish line. This is precisely the kind of student experience at which enhancing stage five is targeted; these are students who need more cognitive scaffolding.

RQ3. *When students diverge on a specific learning stage, submit their program, and receive an ECEM, how do they interpret it?*

The ethnographic stories above show that students were overwhelmingly helped by the feedback they received. This is ideal because regardless of what problem a student thinks they're solving (stage one) or the way they have chosen to solve the problem (stage three), the ECEMs should help students better understand and therefore more easily fix compiler errors. However, as discussed in this chapter under the conclusions for

RQ1, this success led to the discovery of another issue: students who might not have been able to work through a series of compiler errors without the availability of the ECEMs may have been led to a false sense of being on the right path. This issue could be mitigated by providing stronger cognitive scaffolding, which the ECEMs provide for stage five, for the other stages.

RQ4. *How can AATs be augmented to support metacognition in novice programmers in CS1?*

Using the usability study with think-aloud protocol, the present research project was able to see how successful students mentally augmented Athene and how unsuccessful students suffered from Athene's lack of cognitive scaffolding that could produce metacognitive awareness. Unfortunately, this lack of cognitive scaffolding in each of the problem-solving stages is common in most automated assessment tools. The only exception is in stage five, where some tools are moving to enhance the default CEMs, and even those that have done so are still often lacking in effective design. Therefore, the empirically-based framework for features to be implemented in an automated assessment tool that can help build metacognitive awareness in novices by assisting them through all six stages of the problem-solving process as described by Loksa et al. (2016) can now be described:

- **Stage 1: Reinterpret problem prompt.** The number one issue experienced by students in our study was a failure to form the correct conceptual model. Some caught their error in time while most never recovered. Some could not understand what the problem was asking them to do while others mistook the problem for a different, but very similar,

problem they had already encountered. Automated assessment tools, like Athene, allow students to submit code immediately, with no safeguards in place to be certain that the student is at least on the right track. In order to prevent a student with an incorrect conceptual model of the problem from submitting code, problem prompts in automated assessment tools should require students to correctly answer some simple, randomly generated, test cases. After successfully providing the output to the randomly generated input, the ability to submit is then unlocked. The idea that students should begin with test cases is not new. Since at least the 1950's, researchers have been discussing test-driven development (TDD), which is where a student is required to submit test cases along with their code (Edwards, 2003a). TDD is largely focused on helping students reflect on which test cases could break the code they have already written and encourages them to write their test cases as they write their code. The feature suggestion proposed here, which is to put a randomly generated test case at the front of the process and not allow students to proceed without first proving that they understand what the problem is asking them to do, is therefore different from TDD. However, if the research on TDD (Edwards, 2003b; Buffardi & Edwards, 2014; Buffardi & Edwards, 2015) can be incorporated into the very beginning of the process, it could massively improve metacognitive awareness. Students who misunderstood the problem would be immediately made aware of it and forced to re-evaluate their mistake.

- **Stage 2: Search for analogous problems.** Several students in the study skipped stage two (as well as stages three and four) and moved directly to coding in stage five. This proved disastrous for them because they focused on the compiler error message feedback or test case feedback that they received, rather than re-evaluating the problem they chose to build their solution upon. The students who solved the quiz probably thought through problems they had previously encountered without realizing it. This is called "learning by analogy" and occurs when one can successfully reflect on previous problems and understand which portions are relevant to be applied to the current situation (Hoc & Nguyen-Xuan, 1990). In order to facilitate this in an automated assessment tool, students could be required to look at a short list of previous problems and select the ones most relevant to the current problem. This list must be curated because each of the correct selections should help the student realize that some part of that program is relevant to finding a solution. This task must only be made available after the student has successfully answered the test case question from stage one.
- **Stage 3: Search for solutions.** After thinking through previous relevant problems, the student must decide on an approach. The experience of Jane in stage three was vastly different than those of Neil and Thomas. Jane went through the empty code file and placed comments throughout, which indicated her intentions. After she was done, she merely needed to fill it out. Neil was confused about how to construct a basic input loop and spent

too much time pondering a basic element of the solution. Thomas understood the problem, but could not conceptualize how to determine if there were more positive or negative numbers. With invalid assumptions about the problem, this led to insurmountable barriers. Thomas and Neil suffered from what Ko et al. (2004) described as a selection problem: students know what to do, but not how to do it. It is therefore recommended that the automated assessment tool help students lay out their solution, similarly to Jane's approach. This can be accomplished via Parsons Problems where the different programming elements necessary to solve the problem (e.g. input, while loop, condition, output) exist in a list on the left and students are required to drag and drop them into the list on the right in the correct order (Denny et al., 2008; Karavirta et al., 2012). Distractors should not be used in the list as this decreases learning in novice programmers (Harms et al., 2016).

- **Stage 4: Evaluate a potential solution.** The above observations show multiple students attempting to think through their chosen solution with varying degrees of success. Two of the students who did not complete the problem were observed doing so without any particular test cases in mind. In addition, several students were observed skipping this stage and jumping directly to stage five. But mentally running through a chosen algorithm to test its viability is a crucial step that determines if one can continue onward to stage five or, if the chosen solution fails a quick check, one must return to stage three to search for another. This can easily be

facilitated by following from the feature recommendation in stage three above. After the student has solved the Parsons Problem by arranging the different elementary programming elements in the correct order, this can be used to generate a series of comments in a code file similar to what Jane did on her own. This could be taken a step further by generating a basic code skeleton from the arrangement of the pieces of the Parsons Problem. One might worry that this could provide too much help to the student, or perhaps become a crutch. However, the student has solved the Parsons Problem and therefore generating a skeleton of comments or code from their solution is not providing to them anything they did not already have. Furthermore, doing so affords the student the chance to see their solution in a code file and reflect on it before they begin filling it out.

- **Stage 5: Implement a solution.** During this phase, students are writing their code and working to get it to compile. The greatest challenge observed in the present research project is in overcoming issues of esoteric syntax, though students can also run into other types of barriers (Ko et al., 2004). As discussed above, multiple students who completed the problem were helped over the finish line by the ECEMs provided by Athene. Even though this has been discussed by researchers for the past decade, many modern automated assessment tools still do not make any attempt to make the compiler error message feedback any more understandable to novices (Pettit et al., 2017). It is therefore recommended that compiler error messages be enhanced to increase novice comprehension and accessibility.

In order to do this, the following design recommendations, generated by the pilot studies discussed above, should be followed: reduce cognitive load by keeping messages as short as possible, explain esoteric terminology, use clear signifiers, place enhanced message below the standard message, and default the enhanced message to collapsed such that novices will see the standard message first (Hartmann et al., 2010; Marceau et al., 2011a). These are the basic suggestions, though other optional design recommendations depend on context, language choice, and IDE, such as showing example code with suggested fix, code highlighting, and crowd-sourcing suggestions. These should be used sparingly because, although seemingly helpful, often only serve to increase cognitive load, as discussed in the results of the pilot studies above.

- **Stage 6: Evaluate implemented solution.** Students at this stage tend to tinker until correct, often dreadfully unaware of why their code is failing edge cases. In addition to this, those who did not complete the quiz who also made it to this stage were often extremely confused because they had fundamentally misunderstood the problem. If the suggestions from the previous five stages are followed, this will be eliminated because these students will not be allowed to progress to this stage with an incorrect conceptual model. However, this does not solve the issue of tinkering through the test cases. The feature proposed for stage one was to have the AAT generate a random test case and the student provide the correct output. For the sixth stage, a return to test cases is necessary. However, in

> this final stage, it is the student who should be generating test cases, as in test-driven development (TDD) (Edwards, 2003). TDD can be implemented into automated assessment tools to prevent student reliance on instructor-provided test cases and instead encourage robust reflection on the student's solution code and its test case coverage (Buffardi & Edwards, 2015). This should take the form of adaptive feedback to reinforce incremental testing behavior, rather than allowing students the ability to write all their test cases at the end of their development process (Buffardi & Edwards, 2014).

The addition of all of these hoops for students to jump through could become confusing or frustrating. Therefore, there should be some unifying conceptual model to hold it all together. This could take the form of a progress bar, if one was seeking gamification, or a checklist. Regardless of the skeuomorphism chosen, it should be constantly visible and easily indicate at a glance the student's current location in the problem-solving process. Each of the above feature suggestions is designed to cause students to reflect on the problem and their current state in the problem-solving process. The unifying conceptual model, such as a progress bar, reinforces metacognition by tying the six stages together, allowing the student to explicitly know where they are in the overall problem-solving process, and provides a sense of progress and accomplishment as they move on to each stage.

Table 5. *Summary of AAT features to implicitly produce metacognitive awareness*

Learning Stage	Behavior Observed	Proposed Feature
1. Reinterpret problem prompt	Skimming the prompt or misunderstanding it.	Solve randomly generated test case before proceeding.
2. Search for analogous problems	Skipping this stage entirely.	Select similar problems from a list of previous problems that will inform the solution approach.
3. Search for solutions	Landing on a solution that is too close, or exactly like, a previous problem.	Solve a Parsons Problem to create a code-block outline of solution.
4. Evaluate a potential solution	Mentally running through an idea for a solution without a specific test case in mind or skipping this step entirely.	AAT generates a series of comments or basic code skeleton from student's Parsons Problem solution.
5. Implement a solution	Being intimidated by the esoteric nature of compiler error messages.	Enhance the standard compiler error messages according to a human-factors approach.
6. Evaluate implemented solution	Tinkering with the implemented solution as it is tested against various test cases.	Write and submit test cases alongside of the implemented solution code.

The framework above answers RQ4 and, if properly implemented, should increase metacognitive awareness in novice programmers. Even though this was a generative study that has proposed a framework from empirical observations, the present research project has not endeavored to build a prototype to test this framework, which is the most obvious threat to validity to this answer to RQ4. In order to do this well, it would take many years. First, each suggested feature would need to be independently tested. Then the features for all six stages would need to be combined and thoroughly

tested. Finally, the fully implemented framework would need to be tested at multiple institutions. This is obviously beyond the scope of this dissertation.

Implications

Some of the implications of this research have already been made apparent through engagement in the research literature, while other implications are still waiting to be carried forward. The first implication of the present research project is in its contribution to the ongoing discussion about ECEMs in AATs. Some of these results have already been published (Prather et al., 2017) and these results have already been favorably engaged in the ongoing discussions about ECEMs (Becker, Goslin, & Glanville, 2018; Becker et al., 2018). The experiment conducted by Becker, Goslin, and Glanville (2018) confirms the findings of the present research project. Their study wades into the current discussion by asking whether researchers are measuring the same things and whether they are measuring the *right* things. Becker, Goslin, and Glanville found that the effects of ECEMs are so difficult to quantitatively measure that there may be too much noise in the data. Their experiment, designed to address this issue, confirmed Denny et al. (2015), Becker et al. (2016), Pettit et al. (2017), and Prather et al. (2017) by discovering that ECEMs do matter, but one must measure the correct variables to notice the difference. Becker, Goslin, and Glanville (2018) write, "We also observe effects that may corroborate observations made by Prather et al…[but] we do not observe statistically significant effects of ECEMs on the number of compiling submissions, supporting the results of Denny et al. and Pettit et al." (p. 645). The present research project has also

been cited as supporting evidence of the different kinds of common logic errors made by novice programmers (Ettles et al., 2018).

The second implication of the present research project in its potential impact on computer science curricula. The method of Loksa et al. (2016) was to provide students with a chart of the six learning stages and explicitly coach the students to use the stages as a way to cope with bugs while learning to code. However, this relies on instruction from an expert and continued reinforcement through interaction with students when stuck. As discussed above, this is not easily scalable to an intro class of several hundred, and impossible to do in a MOOC. Therefore, AATs are an effective place to focus because hundreds of universities already utilize AATs, there is a wealth of published literature about these tools, and most AATs support stage five and some support stage six, making modification to support the first four stages seems reasonable and beneficial and easily incorporated into existing curriculum. This is perhaps the greatest implication of the present research project: that the findings presented here can be easily adapted into existing tools and curricula and yet stand to make a large impact on the discipline on the order of magnitude called for by Lister (2008). Lister suggested that computer science teachers must understand how novices learn and teach their courses accordingly. The framework proposed above, when implemented in an AAT, allows professors to incorporate Lister's ideas into their curriculum without becoming experts on novice learning. As opposed to the proposal by Loksa et al. (2016), which would require a computer science teacher to understand the stages of learning and how to coach students through them, the framework proposed above implicitly reinforces metacognitive awareness in novice learners. Over the course of a semester, completing program and

program, most of these novices would eventually think in terms of the six stages, even after their course has ended.

Limitations

There are several limitations to these findings. First, the control groups for the think-aloud study took place over multiple semesters and had two different professors. The researcher attempted to minimize this threat by keeping the curriculum (assignments, schedule, the use of Athene, etc.) roughly the same from semester to semester. This was also hopefully mitigated by including as many semesters of data as possible from Athene's database. However, it is possible that some of these differences affected the data and harm its generalizability.

A second limitation was that 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. The researcher attempted to offset this by adding in the warm-up exercise as suggested by Teague et al. (2013). In order to investigate this further, a second practical quiz was carried out on April 19, 2017. This was conducted in a classroom setting and students were not allowed to use previous code or an offline compiler. However, students were not asked to think aloud during the quiz and were not in a one-on-one setting. They were also told to use Athene as their compiler exclusively, as they had done in the first practical quiz. Out of 21 students present in class on the day of the quiz, 10 were able to complete the problem in the 35-minute time window. The

problem was also used in Fall 2015, which can serve as control data for the comparison. In the control semester, 23 out of 35 completed the quiz within the 35-minute time window. Average scores for the control and experimental groups were 65.7% and 47.6%, respectively. This represents a 38% drop in scores from the control semester to the experimental semester. These numbers approximately line up with those from the first practical quiz taken during the full usability study, which saw a 31% drop from the three control semesters to the experimental semester. Additionally, there were 257 submissions in the 2015 control group, compared to 306 submissions for the 2017 experimental group. While it makes sense that there would be an increase in submissions when not allowed to compile offline. However, it seems reasonable to expect a larger increase in submissions than 18%. The reasons for this and its implications cannot be determined here as it is outside of the scope of the present research project. However, the above data from the second practical quiz does confirm that the think-aloud protocol and laboratory observation of users was likely not the factor that caused the significant drop in student scores during the full usability study. While it's possible that it could be due to only allowing students to compile through Athene, this seems unlikely given the somewhat small increase in submissions from control to experimental groups. Most likely it can be explained by the lack of student access to previous work. This explanation, when used above in considering the scores of the full usability study, corrected the experimental average scores to be much closer to the expected scores of the control groups. Therefore, the above data from the second practical quiz strengthens that explanation and severely constrains the methodology as a limitation.

Finally, the low number of student participants in the full usability study (n=31) is another possible limitation. Though it is helpful in quantitative studies to increase the number of participants, this is not necessarily as helpful in qualitative work. Qualitative work focuses on the depth, not breadth, of the interactions and records. However, this small sample size at just one university does limit the generalizability of this study.

Recommendations

The participant pool was rather limited by the university's small number of students taking CS1. A study performed at a university with a larger – and more diverse – CS1 pool would be beneficial for verifying the results of the present research project. Furthermore, a multi-university study would also help with generalizability concerns. Obviously, the most important recommendation of this study is that the framework presented above be implemented. Each feature should be independently tested with a control and experimental group in the same semester. Only after each of the six features have been independently tested could they be combined and then tested again. The experiment on the entire implemented framework would probably need to take place longitudinally over the course of an entire semester, rather than one quiz during the sixth week. All of this will take at least six years to carry out and verify. Finally, future research should also continue to explore the relationship between the presence and design of ECEMs in AATs and student usage of them correlated with their success.

Summary

The extremely high failure rates in introductory programming courses (CS1) can be traced to several factors, one of which being a lack of pedagogical rigor by those who teach it (Lister, 2008). Lister writes that this crisis could eventually collapse the entire discipline if left unchecked. The way forward for many researchers in the computer science education (CSed) community is to better understand learning theory (Sweller, 1999) and apply that to novice programming (Guzdial 2015a; Ko, 2014). Understanding how novices learn is more than just language choice, syntax, data structures, and algorithms, but also *how* they learn these concepts, internalize them, and become aware of that process, called metacognition (Metcalfe & Shimamura, 1994). One recent study attempted to understand the six problem-solving stages of writing code and explicitly teach those to novices, including some tools in an IDE to help them reflect on those steps, so that instructors could more reliably refer to those stages when helping students debug (Loksa et al., 2016). The present research project attempted to adapt the spirit of their experiment to an online learning setting, specifically using automated assessment tools (AATs). AATs offer a somewhat ubiquitous place to start since many hundreds of universities already utilize them (Pettit & Prather, 2017) and, therefore, any new features proposed by the present research project could be easily implemented into existing curriculum.

A survey of existing literature on AATs revealed that only stages five and six are discussed at all, though they are not addressed in those terms. For stage five, the CSed research community has attempted to create enhanced compiler error messages (ECEMs) in an attempt to help students better understand the esoteric error messages that they

receive and to reflect on why they received it (Becker, 2015). However, the helpfulness of ECEMs in novice learning is disputed (Denny et al., 2015; Becker, 2016a; Pettit et al., 2017). The discussion in the literature that could pertain to stage six revolves around test-driven development (TDD), which attempts to have novices write test cases while writing their code (Edwards, 2003a). However, the discussion in the literature on ECEMs or TDD has never been done so in terms of the benefits these could have on novice development of metacognitive awareness of the process of programming. Furthermore, there is nothing in the literature about using AATs to help novices develop metacognitive awareness of the first four stages.

To address this problem, the present research study proposed and carried out a series of experiments designed to start where the current literature ends at ECEMs. The first was a set of pilot studies that iteratively developed the ECEMs in a specific AAT, Athene. These user testing sessions helped determine what students were using and how they were using the additional information. Additional refinements were made based on the scant research literature on human factors studies of error messages (Nienaltowski et al., 2008; Hartmann et al., 2010; Marceau et al., 2011b). The second was a series of error message quizzes given to students in CS1 in the Spring of 2017. These were designed to see if the error messages were genuinely helpful, outside of the context of a homework or quiz problem and the stress, and therefore cognitive load, associated with them. These first two parts were designed to get as much right about stage five of the problem-solving process as possible, so as to make the observations regarding the other five stages more salient. Finally, a larger ethnomethodologically-informed usability study was conducted in the same CS1 class using a think-aloud protocol. This study was designed to observe

students moving through all six of the problem-solving stages as presented by Loksa et al. (2016).

The results of the present research project provide two important contributions. The first is the confirmation that ECEMs that are designed from a human-factors approach are more helpful for students than standard compiler error messages. The iterative improvement of the ECEMs and the results of the full usability study also provide a window into how understanding user behavior can make ECEMs more helpful than ECEMs which are not built from a human-centered design approach. These results were published (Prather et al., 2017) and have already been engaged with and confirmed in the research literature (Becker, Goslin, & Glanville, 2018; Becker et al., 2018; Ettles et al., 2018). The second important contribution is that the results from the observations and post-assessment interviews of the full usability study revealed ways in which students could be helped through the entire problem-solving process. This was presented above as a framework of features, which when implemented properly, could implicitly produce metacognitive awareness in novice programmers (see Table 4 for a summary). This generative proposal should be taken up, implemented, and thoroughly tested over the next few years as it stands to make a substantial impact on the ability of novice programmers to create better conceptual models of how to code, which could improve on the abysmal failure rates seen throughout the discipline.

Appendix A: Rubric for Tagging ECEMs

Marceau et al. (2011a) asked the following question regarding the effectiveness of ECEMs: "does the student make a reasonable edit, as judged by an experienced instructor, in response to the error message?" (p. 500). To answer this question, they developed the following rubric to tag and code student responses to their ECEMs.

[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).

In the study, when a student encountered an error message, their response was recorded and later tagged with one of the five tags above. This tagging was done through the a three-step conceptual model of encountering and fixing errors. They write, "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" (p. 500).

Appendix B: Data Collection

Table B1. *Data collection summary*

Study	Sample Size	Date Performed	Contribution to Dissertation
Pilot Study 1	6	Sept 15, 2016	Testing current ECEMs in Athene.
Pilot Study 2	6	Nov 16, 2016	Testing revised ECEMs in Athene.
Error Message Quizzes	27	Jan 17, 2017 – May 5, 2017	Determine if newly enhanced ECEMs (after the two pilot studies) were more helpful to students than standard CEMs.
Full Usability Study	31	Feb 20-24, 2017	Observe student behavior in problem-solving process in order to make recommendations for improving metacognitive awareness.
Practical Quiz #2	21	Apr 19, 2017	This second "practical quiz" was done as a follow-up to the full usability study in order to control for having students think-aloud. In practical quiz #2, students were not asked to think aloud, but like the full usability study, were required to only compile online using Athene and could not use their own previous material.

Appendix C: Publication Timeline of Dissertation Data

Table C1. *How conference articles contribute to the primary dissertation artifact*

Conference Article	**Primary Dissertation Artifact:** A framework for implementing features in an AAT that implicitly improves metacognitive awareness in novice programmers.
The first four learning stages are not currently represented in any existing AAT. **ITiCSE Article: (Submitted: 1/15/18)** After showing successful implementation of features to improve stage five (as discussed in the ICER article, already published), this study presents the observational data from the full usability study to make suggestions for features to implement that would improve metacognitive stages 1-4 and 6. Stage 6 is covered in more detail in the SIGCSE article (see below)	1. A suggestion for learning stage 1: reinterpret prompt.
	2. A suggestion for learning stage 2: search for analogous problems.
	3. A suggestion for learning stage 3: search for solutions.
	4. A suggestion for learning stage 4: evaluate a potential solution.
In an AAT, "implementing a solution" occurs while students are writing code and fixing any syntax errors. If their code has syntax errors, they see an error message. Stage five is the only one of the six learning stages that is discussed in the literature for AATs. **ICER Article: (Published: 8/20/17)** Were the redesigned ECEMs effective at helping students arrive at a syntactically correct solution? This article examines previous semester control data, the iterative design work of the two pilot studies, the full usability study, and the error message quizzes. It concludes that the redesigned ECEMs were helpful.	5. A suggestion for learning stage 5: implement a solution. This will be design guidelines for ECEMs.
In an AAT, "evaluating an implemented solution" happens after the submission is syntactically correct and is then run against multiple test cases to demonstrate correctness. **SIGCSE Article: (Deadline: 8/25/18)** How does student behavior change when they can only compile online compared to when students can compile offline before submitting? How does this change interaction with failed test cases? This article examines previous semester control data, the full usability study, and practical quiz #2.	6. A suggestion for learning stage 6: evaluate implemented solution.

Appendix D: Participant Movement Through Problem-Solving Stages Over Time in Think-Aloud Study

Table D1. *Participant movement through problem-solving stages over time in think-aloud study*

Participant	Stage 1	Stage 2	Stage 3	Stage 4	Stage 5	Stage 6	Total
P1	1 min						
		<1 min					
			<1 min				
				<1 min			
					3 min		
						<1 min	
							5 min
P2 "Bill"	3 min						
		<1 min					
			1 min				
				1 min			
					1 min		
						<1 min	
							7 min
P3	2 min						
		<1 min					
			<1 min				
				<1 min			
					1 min		
				<1 min			
					1 min		
				<1 min			
					1 min		
						1 min	
							8 min
P4	1 min						
		<1 min					
			<1 min				
				1 min			
					<1 min		
	<1 min						
					<1 min		
	<1 min						
					1 min		

				<1 min			
					<1 min		
				<1 min			
					<1 min		
						1 min	
				<1 min			
					1 min		
						1 min	
							9 min
P5	2 min						
		<1 min					
			1 min				
				<1 min			
					1 min		
				1 min			
					2 min		
				1 min			
					1 min		
						<1 min	
							10 min
P6	2 min						
		<1 min					
			<1 min				
				<1 min			
					6 min		
						1 min	
					1 min		
						1 min	
							12 min
P7	<1 min						
		<1 min					
			<1 min				
				1 min			
					5 min		
				<1 min			
					1 min		
						5 min	
							13 min
P8	<1 min						
		<1 min					
			<1 min				

				<1 min			
					7 min		
				1 min			
					1 min		
						4 min	
							13 min
P9							
							14 min
P10	2 min						
		<1 min					
			<1 min				
				<1 min			
					3 min		
				1 min			
					3 min		
				2 min			
					2 min		
						<1 min	
							14 min
P10	2 min						
		<1 min					
			<1 min				
				<1 min			
					3 min		
				2 min			
					6 min		
						<1 min	
							14 min
P11 “Jane”	2 min						
		<1 min					
			1 min				
				1 min			
					4 min		
						3 min	
				1 min			
						1 min	

							14 min
P12	3 min						
		<1 min					
			1 min				
				<1 min			
					11 min		
						<1 min	
							15 min
P13	1 min						
		<1 min					
			<1 min				
				<1 min			
					4 min		
	1 min						
				1 min			
					9 min		
							16 min
P14	2 min						
		<1 min					
			3 min				
				3 min			
					1 min		
				<1 min			
					4 min		
				<1 min			
					2 min		
						<1 min	
							16 min
P15 “Patricia”	1 min						
		<1 min					
			<1 min				
				<1 min			
					1 min		
	<1 min						
				<1 min			
					1 min		
	<1 min						
					14 min		
						<1 min	
							18 min

P16	5 min						
		<1 min					
			<1 min				
				<1 min			
					8 min		
				<1 min			
					5min		
				<1 min			
					1 min		
							19 min
P17	<1 min						
		<1 min					
			1 min				
				1 min			
					8 min		
				1 min			
					5 min		
						2 min	
	1 min						
			<1 min				
				<1 min			
						1 min	
							20 min
P18	2 min						
		<1 min					
			1 min				
				1 min			
					4 min		
				<1 min			
					5 min		
				<1 min			
	1 min						
				3 min			
					3 min		
						<1 min	
							22 min
P19	3 min						
		<1 min					
			<1 min				
				<1 min			
					19 min		

						<1 min	
							23 min
P20 "Adam"	3 min						
		<1 min					
			2 min				
				1 min			
					5 min		
				2 min			
					4 min		
				5 min			
					1 min		
						6 min	
							30 min
P21 "Wayne"	<1 min						
		<1 min					
			<1 min				
				3 min			
			1 min				
				5 min			
			5 min				
				2 min			
					7 min		
				2 min			
					6 min		
	1 min						
			<1 min				
				1 min			
					<1 min		
							33 min
P22 "Neil"	1 min						
		<1 min					
			<1 min				
				<1 min			
					2 min		
	<1 min						
				1 min			
					2 min		
				2 min			**35 min**
					2 min		
				1 min			
					5 min		
				1 min			

					7 min		
						3 min	
			<1 min				
				<1 min			
					1 min		
						3 min	
				<1 min			
					1 min		
				1 min			
			1 min				
							35 min
P23	2 min						
		<1 min					
			<1 min				
				<1 min			
					2 min		
			1 min				
					2 min		
	1 min						
					2 min		
	1 min						
				<1 min			
					4 min		
				<1 min			
					1 min		
				2 min			
					8 min		
			1 min				
					7 min		
							35 min
P24	3 min						
		<1 min					
			<1 min				
				<1 min			
					6 min		
	1 min						
					24 min		
							35 min
P25 “Theo”	2 min						
		<1 min					
			<1 min				
				<1 min			

					3 min		
				1 min			
	2 min						
				1 min			
					2 min		
	4 min						
					3 min		
	2 min						
					9 min		
	<1 min						
			1 min				
	<1 min						
			1 min				
	<1 min						
			1 min				
	1 min						
							35 min
P26	2 min						
		<1 min					
			1 min				
				<1 min			
					12 min		
		<1 min					
			1 min				
					16 min		
				<1 min			
					<1 min		
						2 min	
							35 min
P27 “Thomas”	1 min						
		<1 min					
			1 min				
				<1 min			
					1 min		
			<1 min				
					2 min		
			2 min				
				4 min			
					6 min		
				1 min			
					2 min		
						1 min	
					3 min		

						2 min	
					7 min		
							35 min
P28	3 min						
			<1 min				
					3 min		
				<1 min			
					2 min		
				1 min			
					8 min		
				1 min			
					10 min		
				3 min			
					4 min		
							35 min
P29 “Jenny”	2 min						
		<1 min					
			<1 min				
					14 min		
				6 min			
					7 min		
				2 min			
					4 min		
							35 min
P30	2 min						
					19 min		
						<1 min	
					4 min		
						<1 min	
				2 min			
					1 min		
						1 min	
					5 min		
							35 min
P31	5 min						
					29 min		
						1 min	
							35 min

Appendix E: Error Message Quiz Data

Table E1. *Participant data for each of the six error message quizzes*

	Quiz 1		Quiz 2		Quiz 3		Quiz 4		Quiz 5		Quiz 6	
	1A	**1B**	**2A**	**2B**	**3A**	**3B**	**4A**	**4B**	**5A**	**5B**	**6A**	**6B**
P1	yes		yes		yes		yes		yes		yes	
P2		yes		yes		yes		yes		yes		yes
P3	yes		no		yes		yes		yes		yes	
P4		yes		yes		yes		yes		yes		yes
P5		yes		yes		yes		yes		np		yes
P6		yes		no		yes		yes		yes		yes
P7		yes		yes		yes		yes		yes		yes
P8		yes		yes		yes		no		no		yes
P9	yes		yes		no		yes		no		yes	
P10		yes		yes		yes		yes		yes		yes
P11	yes		yes		yes		yes		yes		yes	
P12	yes		yes		yes		yes		yes		yes	
P13		yes		yes		yes		no		yes		yes
P14	yes		yes		yes		yes		yes		yes	
P15	yes		yes		yes		yes		yes		yes	
P16	yes		yes		yes		yes		yes		yes	
P17	yes		yes		np		no		yes		yes	
P18		yes		no		yes		yes		yes		yes
P19		yes		yes		yes		yes		yes		yes
P20	yes		yes		yes		yes		yes		yes	
P21	yes		yes		no		yes		yes		yes	
P22		yes		yes		yes		no		yes		yes
P24		yes		yes		yes		yes		yes		yes
P26		yes		yes		no		no		yes		yes
P27	no		yes		no		yes		no		yes	
P28	yes		yes		yes		yes		no		yes	
P29	yes		yes		yes		yes		no		yes	
P30	yes		yes		yes		yes		yes		yes	
P31		yes		yes		no		yes		no		yes
Total Incorrect	**1**	**0**	**1**	**2**	**3**	**2**	**1**	**4**	**4**	**2**	**0**	**0**

Note that for the error message quizzes, two participants, P23 and P25, did not participate and are therefore not listed in the table. A "yes" means the error message was

correctly interpreted, “no” means that it was incorrectly interpreted, and “np” means the student was not present that day. There are two students who were not present for all six quizzes and their data was not considered in the analysis provided in Chapter 4.

Appendix F: IRB Authorization Agreement Between NSU and ACU

Version Date: 03/9/2016

Institutional Review Board (IRB) Authorization Agreement

Name of Institution Providing IRB Review (Institution A):
Nova Southeastern University
IRB 2016-399 FWA XXXX

Name of Institution Relying on the Designated IRB (Institution B):
Abilene Christian University
IRB 00009869

The Officials signing below agree that James Prather may rely on the designated IRB for review and continuing oversight of its human subjects research described below: (*check one*)

☐ This agreement applies to all human subjects research covered by Institution B's FWA.

☒ This agreement is limited to the following specific protocol(s):
Name of Research Project: A usability study on the feedback mechanism of an automated assessment tool for student programming
Name of Principal Investigator: James Prather
Sponsor or Funding Agency: N/A
☐ Other (*describe*):_____

The review performed by the designated IRB will meet the human subject protection requirements of Institution A's OHRP-approved FWA. The IRB at Institution A will follow written procedures for reporting its findings and actions to appropriate officials at Institution B. Relevant minutes of IRB meetings will be made available to Institution B upon request. Institution B remains responsible for ensuring compliance with the IRB's determinations and with the Terms of its OHRP-approved FWA. This document must be kept on file by both parties and provided to OHRP upon request.

Signature of Signatory Official (Institution/Organization A):

Date: 09/14/2016

Print Full Name: Ling Wang Institutional Title: Professor and IRB representative of Nova Southeastern University

Signature of Signatory Official (Institution B):

Date: 9/15/16

Print Full Name: Susan Lewis Institutional Title: Vice Provost

References

Angrosino, M. (2007). *Doing ethnographic and observational research*. Sage.

Ayres, P., & Sweller, J. (1990). Locus of difficulty in multistage mathematics problems. *The American Journal of Psychology*, 167-193.

Ayres, P. L. (2001). Systematic mathematical errors and cognitive load. *Contemporary Educational Psychology*, *26*(2), 227-248.

Barik, T., Smith, J., Lubick, K., Holmes, E., Feng, J., Murphy-Hill, E., & Parnin, C. (2017, May). Do developers read compiler error messages?. In *Proceedings of the 39th International Conference on Software Engineering* (pp. 575-585). IEEE Press.

Barkhuus, L., & Rode, J. A. (2007, April). From mice to men-24 years of evaluation in CHI. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (pp. 1-16).

Becker, B. A. (2015). *An exploration of the effects of enhanced compiler error messages for computer programming novices.* Unpublished master's thesis, Dublin Institute of Technology, Dublin, Ireland.

Becker, B. A. (2016a, February). An effective approach to enhancing compiler error messages. In Proceedings of the 47th ACM Technical Symposium on Computing Science Education (pp. 126-131). ACM.

Becker, B. A. (2016b, November). *You are what you measure: Enhancing compiler error messages effectively*. Retrieved from https://cszero.wordpress.com/2016/11/18/you-are-what-you-measure-enhancing-compiler-error-messages-effectively/

Becker, B. A., Goslin, K., & Glanville, G. (2018, February). The Effects of Enhanced Compiler Error Messages on a Syntax Error Debugging Test. In *Proceedings of the 49th ACM Technical Symposium on Computer Science Education* (pp. 640-645). ACM.

Becker, B. A., Murray, C., Tao, T., Song, C., McCartney, R., & Sanders, K. (2018, February). Fix the First, Ignore the Rest: Dealing with Multiple Compiler Error Messages. In *Proceedings of the 49th ACM Technical Symposium on Computer Science Education* (pp. 634-639). ACM.

Bell, G. (2001). Looking across the Atlantic: Using ethnographic methods to make sense of Europe. *Intel Technology Journal*, *5*(3), 1-10.

Bell, G., Blythe, M., Gaver, B., Sengers, P., & Wright, P. (2003, April). Designing culturally situated technologies for the home. In *CHI'03 extended abstracts on Human factors in computing systems* (pp. 1062-1063). ACM.

Bell, G., Blythe, M., & Sengers, P. (2005). Making by making strange: Defamiliarization and the design of domestic technologies. *ACM Transactions on Computer-Human Interaction (TOCHI)*, *12*(2), 149-173.

Blomberg, J., & Karasti, H. (2012). Positioning ethnography within participatory design. *Routledge international handbook of participatory design*, 86-116.

Blomberg, J., & Karasti, H. (2013). Reflections on 25 years of ethnography in CSCW. *Computer supported cooperative work (CSCW)*, *22*(4-6), 373-423.

Brannen, M. Y., & Salk, J. E. (2000). Partnering across borders: Negotiating organizational culture in a German-Japanese joint venture. *Human relations*, *53*(4), 451-487.

Buffardi, K., & Edwards, S. H. (2014, March). A formative study of influences on student testing behaviors. In *Proceedings of the 45th ACM technical symposium on Computer science education* (pp. 597-602). ACM.

Buffardi, K., & Edwards, S. H. (2015, February). Reconsidering Automated Feedback: A Test-Driven Approach. In *Proceedings of the 46th ACM Technical Symposium on Computer Science Education* (pp. 416-420). ACM.

Butler, A. C., Karpicke, J. D., & Roediger III, H. L. (2007). The effect of type and timing of feedback on learning from multiple-choice tests. *Journal of Experimental Psychology: Applied*, *13*(4), 273.

Button, G. (2000). The ethnographic tradition and design. *Design studies*, *21*(4), 319-332.

Cao, J., Fleming, S. D., Burnett, M., & Scaffidi, C. (2014). Idea Garden: Situated support for problem solving by end-user programmers. *Interacting with Computers*, *27*(6), 640-660.

Carmien, S. P., & Fischer, G. (2008, April). Design, adoption, and assessment of a socio-technical environment supporting independence for persons with cognitive disabilities. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (pp. 597-606). ACM.

Chi, M. T. H., Glaser, R. & Farr, M. J. (Eds.) (1998) *The nature of expertise*. Hillsdale, NJ, Lawrence Erlbaum Associates.

Cohen, L., Manion, L., & Morrison, K. (2011). Research methods in education. Routledge, 7th edition.

Crabtree, A., Rodden, T., Tolmie, P., & Button, G. (2009, April). Ethnography considered harmful. In *Proceedings of the SIGCHI conference on human factors in computing systems* (pp. 879-888). ACM.

Cunliffe, A. L. (2010). Retelling Tales of the Field In Search of Organizational Ethnography 20 Years On. *Organizational Research Methods*, *13*(2), 224-239.

Denny, P., Luxton-Reilly, A., & Simon, B. (2008, September). Evaluating a new exam question: Parsons problems. In *Proceedings of the fourth international workshop on computing education research* (pp. 113-124). ACM.

Denny, P., Luxton-Reilly, A., and Carpenter, D. (2014). Enhancing Syntax Error Messages Appears Ineffectual. In *Proceedings of the 2014 Conference on Innovation & Technology in Computer Science Education* (ITiCSE '14), 273-278.

Dijkstra, E. W. (1995, August). Introducing a course on calculi. Retrieved September 25, 2017, from https://www.cs.utexas.edu/users/EWD/transcriptions/EWD12xx/EWD1213.html Remarks by Edsger Dijkstra at Department of Computer Sciences, The University of Texas at Austin

Dix, A. (2009). *Human-computer interaction* (pp. 1327-1331). Springer US.

Douce, C., Livingstone, D., and Orwell, J. (2005). Automatic Test-Based Assessment of Programming: A Review. *J. Educ. Resour. Comput. 5, 3, Article 4* (September 2005).

Dumas, J. S., & Loring, B. A. (2008). *Moderating usability tests: Principles and practices for interacting*. Morgan Kaufmann.

Edwards, S. H. (2003a, October). Rethinking computer science education from a test-first perspective. In *Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications* (pp. 148-155). ACM.

Edwards, S. H. (2003b, October). Teaching software testing: automatic grading meets test-first coding. In *Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications* (pp. 318-319). ACM.

Eteläpelto, A. (1993). Metacognition and the expertise of computer program comprehension. *Scandinavian Journal of Educational Research*, *37*(3), 243-254.

Ettles, A., Luxton-Reilly, A., & Denny, P. (2018, January). Common logic errors made by novice programmers. In *Proceedings of the 20th Australasian Computing Education Conference* (pp. 83-89). ACM.

Ericsson, K. A., & Simon, H. A. (1993). *Protocol analysis*. Cambridge, MA: MIT press.

Ericsson K, and Smith, J. (Eds) (1991) *Toward a General Theory of Expertise: Prospects and Limits*. Cambridge University Press, England.

Falkner, N., Vivian, R., Piper D., and Falkner, K. (2014) Increasing the Effectiveness of Automated Assessment By Increasing Marking Granularity and Feedback Units. *Proceedings of the 45th ACM Technical Symposium on Computer Science Education*, SIGCSE (2014), 9-14.

Flowers. T. (2004). Empowering Students and Building Confidence in Novice Programmers through Gauntlet. In *Proceedings of the 34th Annual IEEE Frontiers in Education Conference* (FIE '04), Session T3H, 10 –13.

Foxley, E., Higgins, C., Symeonidis, P, Tsintsifas, A, The CourseMaster Automated Assessment System - a next generation Ceilidh, *Conference on Computer Assisted Assessment,* University of Warwick, 2001.

Garvin-Doxas, K., & Barker, L. J. (2004). Communication in computer science classrooms: understanding defensive climates as a means of creating supportive behaviors. *Journal on Educational Resources in Computing (JERIC)*, *4*(1), 2.

Glaser, B. G., & Strauss, A. L. (2009). *The discovery of grounded theory: Strategies for qualitative research*. Transaction publishers.

Gould, J. D., & Lewis, C. (1985). Designing for usability: key principles and what designers think. *Communications of the ACM*, *28*(3), 300-311.

Gray, S., St Clair, C., James, R., & Mead, J. (2007, September). Suggestions for graduated exposure to programming concepts using fading worked examples. In *Proceedings of the third international workshop on Computing education research* (pp. 99-110). ACM.

Guzdial, M. (2014). *Enhancing syntax error messages appears ineffectual — if you enhance the error messages poorly*. Retrieved from https://computinged.wordpress.com/2014/07/29/enhancing-syntax-error-messages-appears-ineffectual-if-you-enhance-the-error-messages-poorly/

Guzdial, M. (2015a). What's the best way to teach computer science to beginners?. In *Communications of the ACM* (February, 2015), 12-13.

Guzdial, M. (2015b). Programming Languages are the Most Powerful, and Least Usable and Learnable User Interfaces. In *Communications of the ACM* (March, 2015), 9.

Harms, K. J., Chen, J., & Kelleher, C. L. (2016, August). Distractors in Parsons Problems Decrease Learning Efficiency for Young Novice Programmers. In *Proceedings of the 2016 ACM Conference on International Computing Education Research* (pp. 241-250). ACM.

Hartmann, B., MacDougall, D., Brandt, J., and Klemmer, S. R. (2010). What Would Other Programmers Do: Suggesting Solutions to Error Messages. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (CHI '10), 1019-1028.

Hoc, J.M., & Nguyen-Xuan, A. (1990). Language semantics, mental models and analogy. In D.J. Gilmore, T Green, J.M. Hoc, & R. Samurcay (Eds.), *Psychology of Programming* (pp. 139–156). Cambridge, MA: Academic Press.

Hollingsworth, J. (1960). Automatic Graders for Programming Classes. *Communications of the ACM 3, 10* (October 1960), 528-529.

Holton, C., and Wallace, S. A. (2013). Investigating the Use of an Online Assignment Submission and Assessment System in the CS Classroom. *J. Comput. Sci. Coll. 29, 1* (October 2013), 123-129.

Ihantola, P., Ahoniemi, T., Karavirta, V., and Seppälä, O. (2010). Review of Recent Systems for Automatic Assessment of Programming Assignments. In *Proceedings of the 10th Koli Calling International Conference on Computing Education Research* (Koli Calling '10), 86-93.

Jackson, J., Cobb, M., and Carver, C. (2005). Identifying Top Java Errors for Novice Programmers. *Proceedings of the Frontiers in Education Conference*, 2005, pp. T4C–24.

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

Jernigan, W., Horvath, A., Lee, M., Burnett, M., Cuilty, T., Kuttal, S., ... & Ko, A. (2015, October). A principled evaluation for a principled Idea Garden. In *Visual Languages and Human-Centric Computing (VL/HCC), 2015 IEEE Symposium on* (pp. 235-243). IEEE.

Joy, M. S., Luck, M., The BOSS system for online submission and assessment monitor, *Journal of the CTI Centre for Computing*, 10, 27-29. 1998.

Karavirta, V., Helminen, J., & Ihantola, P. (2012, November). A mobile learning application for parsons problems with automatic feedback. In *Proceedings of the*

12th Koli Calling International Conference on Computing Education Research (pp. 11-18). ACM.

Ko, A. J., Myers, B. A., & Aung, H. H. (2004, September). Six learning barriers in end-user programming systems. In *Visual Languages and Human Centric Computing, 2004 IEEE Symposium on* (pp. 199-206). IEEE.

Ko, A. (March 25, 2014). *Programming languages are the least usable, but most powerful human-computer interfaces ever invented.* Retrieved from http://blogs.uw.edu/ajko/2014/03/25/programming-languages-are-the-least-usable-but-most-powerful-human-computer-interfaces-ever-invented/

Krug, S. (2014). *Don't Make Me Think, Revisited: A Common Sense Approach to Web and Mobile Usability*. New Riders.

Lazar, J., Feng, J. H., & Hochheiser, H. (2017). *Research methods in human-computer interaction* (2nd edition). Cambridge, MA: Morgan Kaufmann.

Lee, M. J., & Ko, A. J. (2011, August). Personifying programming tool feedback improves novice programmers' learning. In *Proceedings of the seventh international workshop on Computing education research* (pp. 109-116). ACM.

Lister, R. (2008, January). After the gold rush: toward sustainable scholarship in computing. In *Proceedings of the tenth conference on Australasian computing education-Volume 78* (pp. 3-17). Australian Computer Society, Inc..

Loksa, D., Ko, A. J., Jernigan, W., Oleson, A., Mendez, C. J., & Burnett, M. M. (2016, May). Programming, Problem Solving, and Self-Awareness: Effects of Explicit Guidance. In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems* (pp. 1449-1461). ACM.

Mani, M., & Mazumder, Q. (2013, March). Incorporating metacognition into learning. In *Proceeding of the 44th ACM technical symposium on Computer science education* (pp. 53-58). ACM.

Marceau, G., Fisler, K., & Krishnamurthi, S. (2011a). Measuring the effectiveness of error messages designed for novice programmers. In *Proceedings of the 42nd ACM technical symposium on Computer science education* (pp. 499-504). ACM.

Marceau, G., Fisler, K., and Krishnamurthi, S. (2011b) Mind your language: on novices' interactions with error messages, *Proceedings of the 10th SIGPLAN symposium on New ideas, new paradigms, and reflections on programming and software* (Onward!), ACM, New York, NY, 3 18, 2011.

Metcalfe, J., & Shimamura, A. P. (Eds.). (1994). *Metacognition: Knowing about knowing*. MIT press.

Miller, G. A. (1956). The magical number seven, plus or minus two: some limits on our capacity for processing information. *Psychological review*, *63*(2), 81.

Miller, W.L. and Crabtree, B.F. (1999) Clinical research: A multimethod typology and qualitative roadmap. In B.F. Crabtree and W.L. Miller (eds), *Doing Qualitative Research*. Thousand Oaks, CA: Sage Publications.

Morrison, B. B. (2016). *Replicating experiments from educational psychology to develop insights into computing education: cognitive load as a significant problem in learning programming* (Doctoral dissertation, Georgia Institute of Technology).

Morrison, B. B., Margulieux, L. E., Ericson, B., & Guzdial, M. (2016, February). Subgoals help students solve Parsons problems. In *Proceedings of the 47th ACM Technical Symposium on Computing Science Education* (pp. 42-47). ACM.

Moschella, M. C. (2008). *Ethnography as a pastoral practice: An introduction*. Cleveland: Pilgrim Press.

Nelson, G. L., Xie, B., & Ko, A. J. (2017, August). Comprehension First: Evaluating a Novel Pedagogy and Tutoring System for Program Tracing in CS1. In *Proceedings of the 2017 ACM Conference on International Computing Education Research* (pp. 2-11). ACM.

Nielsen, J., & Molich, R. (1990, March). Heuristic evaluation of user interfaces. In *Proceedings of the SIGCHI conference on Human factors in computing systems* (pp. 249-256). ACM.

Nienaltowski, M., Pedroni, M., and Meyer, B. (2008). Compiler error messages: what can help novices?. In *Proceedings of the 39th SIGCSE Technical Symposium on Computer Science Education* (SIGCSE '08), 168-172.

Nordquist, P. (2007). Providing accurate and timely feedback by automatically grading student programming labs. *J. Comput. Sci. Coll. 23, 2* (December 2007), 16-23.

Norman, D., *The Design of Everyday Things*, New York, NY: Basic Books, 2013.

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

Pettit, R., Homer, J., & Gee, R. (2017, March). Do Enhanced Compiler Error Messages Help Students?: Results Inconclusive. In *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Educatio*n (pp. 465-470). ACM.

Pettit, R., Homer, J., Gee, R., Mengel, S., & Starbuck, A. (2015, February). An empirical study of iterative improvement in programming assignments. In *Proceedings of the 46th ACM Technical Symposium on Computer Science Education* (pp. 410-415). ACM.

Pettit, R.S., Homer, J. D., Holcomb, K. M., Simone, N., & Mengel, D. S. A. (2015, June). Are automated assessment tools helpful in programming courses? In *Proceedings of the 122nd ASEE Annual Conference & Exposition*. ACM.

Pettit, R., & Prather, J. (2017, April). Automated Assessment Tools: Too Many Cooks, Not Enough Collaboration. In *Proceedings of the 28th Consortium for Computer Sciences in Colleges: South Central Region*. ACM.

Pieterse, V. (2013). Automated Assessment of Programming Assignments. In *Proceedings of the 3rd Computer Science Education Research Conference on Computer Science Education Research* (CSERC '13). Article 4, 12 pages. Marko van Eekelen, Erik Barendsen, Peter Sloep, and Gerrit van der Veer (Eds.). Open Universiteit, Heerlen, The Netherlands.

Polya, G. (2014). *How to solve it: A new aspect of mathematical method* (2nd edition expanded). Princeton, NJ: Princeton University Press.

Prather, J., Pettit, R., McMurry, K. H., Peters, A., Homer, J., Simone, N., & Cohen, M. (2017, August). On Novices' Interaction with Compiler Error Messages: A Human Factors Approach. In *Proceedings of the 2017 ACM Conference on International Computing Education Research* (pp. 74-82). ACM.

Roll, I., Aleven, V., McLaren, B. M., & Koedinger, K. R. (2011). Improving students' help-seeking skills using metacognitive feedback in an intelligent tutoring system. *Learning and Instruction*, *21*(2), 267-280.

Roll, I., Holmes, N. G., Day, J., & Bonn, D. (2012). Evaluating metacognitive scaffolding in guided invention activities. *Instructional science*, *40*(4), 691-710.

Rubin, J., & Chisnell, D. (2008). *Handbook of usability testing: how to plan, design and conduct effective tests* (2nd ed.). John Wiley & Sons.

Rubio-Sánchez, M., Kinnunen, P., Pareja-Flores, C., and Velázquez-Iturbide, Á. (2014). Student perception and usage of an automated programming assessment tool. *Computer Human Behavior 31* (February 2014), 453-460.

Salleh, N., Mendes, E., Grundy, J., & Burch, G. S. J. (2010, May). An empirical study of the effects of conscientiousness in pair programming using the five-factor personality model. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1* (pp. 577-586). ACM.

Schorsch, T. (1995). Cap: An Automated Self-Assessment Tool To Check Pascal Programs For Syntax, Logic And Style Errors. *Proceedings of the 26th ACM Technical Symposium on Computer Science Education*, SIGCSE (1995), 168-172.

Shaft, T. M. (1995). Helping programmers understand computer programs: the use of metacognition. *ACM SIGMIS Database*, *26*(4), 25-46.

Sherman, M., Bassil, S., Lipman, D., Tuck, N., and Martin, F. (2013). Impact of auto grading on an introductory computing course. *J. Comput. Sci. Coll.* 28, 6 (June 2013), 69-75.

Shneiderman, B., Plaisant, C., Cohen, M., Jacobs, S., & Elmqvist, N. (2017). Designing the User Interface: Strategies for Effective Human-Computer Interaction, Pearson Publ. Co. *Boston, MA, USA*.

Singh, R., Sumit, G., and Solar-Lezama, A. (2013). Automated feedback generation for introductory programming assignments. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation* (PLDI '13). ACM, New York, NY, USA, 15-26.

Suchman, Lucy. Plans and situated action (1987). Cambridge University Press, Cambridge. 149-70.

Sweller, J. (1999). *Instructional design in technical areas*. Camberwell, Australia: ACER Press.

Teague, D., Corney, M., Ahadi, A., & Lister, R. (2013, January). 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* (pp. 87-95). Australian Computer Society, Inc..

Towell, D., and Reeves, R. (2009). From Walls to Steps: Using online automatic homework checking tools to improve learning in introductory programming courses. In *Proceedings of the Association for Computer Educators in Texas* (ACET) 2009, Corpus Christi, Texas.

Tullis, T., Albert, W., Dumas, J. S., & Loring, B. A. (2008). *Measuring the User Experience: Collecting Analyzing, and Presenting Usability.* Newnes.

Venables A., and Haywood, L. (2003). Programming students NEED instant feedback!. In *Proceedings of the fifth Australasian Conference on Computing Education* (ACE '03), Ton Greening and Raymond Lister (Eds.), Vol. 20. Australian Computer Society, Inc., Darlinghurst Australia, Australia, 267-272.

Vizcaíno, A., Contreras, J., Favela, J., & Prieto, M. (2000, June). An adaptive, collaborative environment to develop good habits in programming. In *Intelligent Tutoring Systems* (pp. 262-271).

Ware, C. (2012). *Information visualization: perception for design.* Elsevier.

Warren, J., Rixner, S., Greiner, J., Wong, S. (2014). Facilitating Human Interaction in an Online Programming Course. *Proceedings of the 45th ACM Technical Symposium on Computer Science Education*, SIGCSE (2014), 665-670.

Wyche, S. P., Medynskiy, Y., & Grinter, R. E. (2007, April). Exploring the use of large displays in American megachurches. In *CHI'07 extended abstracts on Human factors in computing systems* (pp. 2771-2776). ACM.

Yuen, T. T. (2007). Novices' knowledge construction of difficult concepts in CS1. *ACM SIGCSE Bulletin*, *39*(4), 49-53.