THE HOLISTIC COURSE DELIVERY: A NOVEL PEDAGOGY FOR COLLEGIATE INTRODUCTORY COMPUTER PROGRAMMING

A DISSERTATION IN
Curriculum and Instruction
and
Computer Science

Presented to the Faculty of the University
of Missouri-Kansas City in partial fulfillment of
the requirements for the degree

DOCTOR OF PHILOSOPHY

by
BELINDA JOAN COPUS
M.S., University of Central Missouri, 2013
B.S., University of Texas at Austin, 1993

Kansas City, Missouri
2020

© 2020

BELINDA JOAN COPUS

ALL RIGHTS RESERVED

THE HOLISTIC COURSE DELIVERY: A NOVEL PEDAGOGY FOR COLLEGIATE INTRODUCTORY COMPUTER PROGRAMMING

Belinda Joan Copus, Candidate for the Doctor of Philosophy Degree

University of Missouri-Kansas City, 2020

ABSTRACT

For many years there have not been enough computer science graduates to fill open positions. One of the chief barriers to the formation of computer science graduates is that many students are unsuccessful in the introductory programming course. Unsuccessful students often change their major field of study or terminate their collegiate studies. A chief concern is therefore to minimize the DFW rate (grade of D or F, or withdrawal from a course).

Student characteristics have been extensively studied to explain, and sometimes justify, the high DFW rate in introductory programming courses. Pairs programming, flipped classrooms, choice of programming language, and a variety of other modifications and novel methods have been devised in efforts to reduce the DFW rate. The collective conclusion has been that there is no silver bullet that has been demonstrated to be universally effective.

This quasi-experimental study incorporates four learning theories that inform the design and delivery of an introductory programming course: Neo-Piagetian Theory, Cognitive Apprenticeship Theory, Cognitive Load Theory, and Self-Efficacy Theory. The objective was

to (1) design a course from the top-down that integrates several pedagogical elements in a holistic way, and (2) deliver it to a group of nascent programming students.

The Holistic Course Delivery was implemented in three class sections of an introductory programming course at a midwestern university in which a total of 96 students were enrolled. The Holistic Course Delivery had a significantly lower DFW rate compared to both historic DFW rates at the institution and established international norms and students indicated they felt prepared for subsequent computer science coursework.

APPROVAL PAGE

The faculty listed below, appointed by the Dean of the School of Graduate Studies, have examined a dissertation titled "The Holistic Course Delivery: A Novel Pedagogy for Collegiate Introductory Computer Programming," presented by Belinda Joan Copus, candidate for the Doctor of Philosophy degree, and certify that in their opinion it is worthy of acceptance.

Supervisory Committee

Rita Barger, Ph.D., Committee Chair
School of Education

Carolyn Barber, Ph.D.
School of Education

Raol Taft, Ph.D.
School of Education

Yugyung Lee, Ph.D.
Department of Computer Science and Electrical Engineering

Praveen Rao, Ph.D.
Electrical Engineering and Computer Science,
University of Missouri School of Medicine

CONTENTS

ABSTRACT iii

ILLUSTRATIONS ix

TABLES x

ACKNOWLEDGMENTS xi

Chapter

1. INTRODUCTION 1

Statement of Problem 6

Theoretical Framework 8

Purpose Statement 11

Research Questions 13

Definition of Terms 14

Study Limitations 18

2. LITERATURE REVIEW 20

Brief Summary of History & Progression of Computer Science Education & Curriculum 20

Important Observations in the Past 20 Years in Introductory Programming 24

Predictors of Student Success 26

Prior Research on Challenges Programming Novices Face 29

Reasons Students Dropped an Introductory Programming Course 31

Education Theory that Informs Computer Science Education 31

3. METHODOLOGY 38

Statement of Significance of the Problem 38

Research Questions 40

Participants 40

Detailed Description of the Holistic Course Delivery 43

Data Sources and Sample Instruments 48

Data Analysis Procedures 54

Institutional Review Board and Permissions 54

Risks and Limitations 54

4. ANALYSIS 56

RQ1 56

RQ2 57

RQ3 62

RQ4 68

RQ5 77

5. DISCUSSION 81

RQ1 81

RQ2 84

RQ3 88

RQ4 93

RQ5 102

Conclusion 104

Future Research 105

Appendix

A. PRE-COURSE SURVEY 108

B. PRE-COURSE TEST & POST-COURSE TEST 115

C. POST-COURSE SURVEY 124

D. HOMEWORK REFLECTION 132
E. HOMEWORK ASSIGNMENTS 136
F. CODING SPRINTS 160
G. FINAL EXAM PROGRAMMING QUESTIONS 172
H. QUALITATIVE INTERVIEW QUESTIONS 175
I. CURRICULUM OUTLINE 177
J. SYLLABUS 180
REFERENCES 184
VITA 198

ILLUSTRATIONS

Figure Page

1.1 Theoretical Framework 8

3.1 Quasi-experimental design static-group pretest-posttest design 39

4.1 Confidence in Java Programming 60

4.2 Student Confidence in Problem Solving 61

4.3 Histograms: Student Self-Reported Java Programming Confidence 58

4.4 Histogram, Final Course Grade (Points out of 1000) 63

4.5 Homework First Problem Perceived as Difficult or Requiring Substantial Effort 72

4.6 Histogram, Time Spent on HW6 73

4.7 Time Spent on Homework Assignments 74

4.8 Frequency of Guessing on Quizzes 75

4.9 Sprint 5, Question 2 Score Histogram 76

4.10 Sprint Code Tracing and Programming Performance 77

4.11 Sample Question Sensorimotor/Preoperational Staging 79

4.12 Sample Question Concrete Operational Staging 79

TABLES

Table | Page

3.1 Participant Demographics 42

3.2 Summary of course elements and theory that motivated inclusion 48

4.1 DFW Rate of Holistic Course Delivery vs. Institutional and International Averages 57

4.2 Student Self-Reported Preparedness for the Next Programming Course 62

4.3 Self-Reported Programming Ability at Start and End of Course 60

4.4 Student Continuous Characteristics and Final Course Grade 64

4.5 Student Discrete Characteristics and Final Course Grade 66

4.6 Course Pass/Fail and Class Rank for Course Completers 67

4.7 Class Rank and Major Field of Study 68

4.8 End-of-Course Survey, Part 1 68

4.9 End-of-Course Survey, Part 2 69

4.10 End-of-Course Survey, Part 3 69

4.11 End-of-Course Survey, Part 4 69

4.12 End-of-Course Survey, Part 5 70

4.13 End-of-Course Survey, Part 6 70

4.14 End-of-Course Survey, Part 7 70

4.15 Cross-Tabulation of Neo-Piagetian Stage, Pre- and Post-Course 78

4.16 Improvement in Students' Neo-Piagetian Stage from Pre- to Post-Course 80

ACKNOWLEDGMENTS

I have been most fortunate to be surrounded by many persons who, knowingly or not, aided me in achieving this milestone, and I would like to express my gratitude for making this possible.

- First, I would like to thank my Committee Chair and Advisor, Dr. Rita Barger, for her guidance; and the other members of my committee: Drs. Carolyn Barber, Yugyung Lee, Praveen Rao, and Raol Taft, for their constructive suggestions for my research, time reviewing the artifacts leading to this dissertation, and guidance in my education.
- A major theme in this research involved mentors and role models. I had several mentors and role models in this journey in three capacities. Thank you Dr. Rita Barger, Dr. Carolyn Barber, and Dr. Yugyung Lee for sharing your experiences in higher education and helping me to refine my thinking.
- Thank you to my UCM mentors -Dr. Xiaodong Yue, Dr. Mahmoud Yousef and Dean Alice Greife – without you I would never have considered this path; and without your support, I could never have finished.
- Thank you to my coworkers - Dr. Ann McCoy, Dr. Rhonda McKee, and Ms. Becky White for your words of encouragement that were always spoken at just the right moment.
- Thank you to the teachers who started me down the path of Computer Science and sparked my love for programming and software development: Dr. Tess Stewart – Plano East Senior High School, Dr. Nell Dale and Dr. Laurie Werth – The University of Texas at Austin.

- I am grateful for my students and their willingness to participate in this study and their contagious curiosity in learning.
- And finally, I am grateful for, and to, my parents, sister, and immediate family. My partner in life, Perry – this was but a small challenge among others in our past few years. Your fortitude to stay alive in the face of great physical trial gave me courage to push through the tough days. As God may grant, may we continue this journey in life together. Thank you, Peter, Zachary, and Caleb, for the many sacrifices you made so that this could happen.

CHAPTER 1

INTRODUCTION

According to the Bureau of Labor Statistics (BLS) (2018), the demand for software developers in the United States will grow by 24% in the ten-year period from 2016 to 2026. This rate is significantly higher than the average expected growth rate of 7% for all other occupations. The software developer field is predicted to grow by 302,500 jobs during this ten-year period while other computing fields such as web development, database administration, information security analysis, and research science are also expecting substantial growth (U.S. Department of Labor, 2018). Most potential employers for software developers and related jobs require an applicant to have a bachelor's degree in computer science or a closely related field.

Employers are unable to recruit sufficient qualified candidates to fill open positions (Adams, 2014). Although enrollment in introductory computer programming classes and computer science major declarations have increased for each of the past eight years, with a recently-reported nationwide increase in computer science majors (Sax, Lehman, & Zavala, 2017), degree production is not increasing at the same pace and is insufficient to support current market demand. In fact, industry leaders have been pleading with political leaders to allocate funding for computer programming education in all U.S. schools so that more workers can be trained in this profession (CS Education Coalition, 2016).

A significant impediment to producing a sufficient number of Computer Science graduates is that a large number of students are unsuccessful in completing the introductory courses in Computer Science, and thus will never graduate in the discipline (Watson & Li, 2014). The first two computer science courses are collectively labeled as “Introductory Programming” or “CS1” and often represent a major barrier to entry into the field for freshman computer science students. The failure rate (“DFW,” or grades of D, F, or a withdrawal) has been widely reported to be approximately 33% worldwide for over a decade in the introductory computer science courses. At the minimum, DFWs often lead to changes of major to something other than computer science (Beaubouef & Mason, 2005; Watson & Li, 2014). If this rate could be improved by enabling more students to be successful in introductory computer programming courses, many more graduates could potentially fill the current and projected hiring gap.

There have been many attempts to understand why the DFW rate in introductory computer programming courses is so high. Some researchers have examined student demographics such as race, gender, and socioeconomic status. Women are underrepresented in obtaining computer science degrees, earning only 23% of all B.S. Computer Science degrees in 2004, and dropping to a modest 18% a decade later in 2014 (Espinosa, 2015). Non-white graduates, as reported by the National Science Foundation, represented 18% of B.S. Computer Science degree grants. That is an underrepresentation by half, as more than 36% of the U.S. population is non-white (National Science Foundation, 2017). A study by the Computing Research Association found that only 8% of those declaring the B.S.

Computer Science major were Black or Hispanic (National Academy of Sciences, 2018). Clearly, there is a diversity problem in computer science as in many STEM fields.

It is generally held that lack of exposure to computing is the primary issue driving the lack of diversity of computer science degree grants, and there are many working to expose underrepresented populations to Computer Science early in their primary education. For example, Code.org is a non-profit with a mission to expand computer science access in K-12 schools particularly concerned with increasing the participation of women and underrepresented minorities (Code.org). It is hoped that time will reveal a more diverse and larger computer science workforce.

It is not surprising that students from underrepresented populations have a higher rate of failure in computer science. Recent AP Computer Science data tracking underrepresented populations in an NCES study from the years 2003-2009 shows that initial majors in computer/information sciences had some of the highest percentages of departures from all STEM majors, with approximately 59% of majors choosing to leave the field. Additionally, this report shows that 52% of students who left a STEM bachelor's program without a certificate or degree were Black or Hispanic, while 62% of STEM majors that switched to another major were Black or Hispanic (U.S. Department of Education, 2013). While the data does not directly state the number of underrepresented persons who start a degree in computer science and fail to complete it, it is likely that underrepresented populations majoring in Computer Science experience much higher rates of failure than other populations and other STEM fields.

Researchers have also investigated the mechanisms of delivery for computer science courses as a contributing factor to the high failure rate. Studies on the class format, such as traditional vs. online formats, or traditional vs. flipped classrooms, have been conducted. Results are generally unhelpful. For example, Horton and Craig (2015) found no substantial difference in the failure rates obtained through traditional classrooms and flipped classrooms.

Researchers have also considered the choice of programming language that is taught in introductory programming courses and its effect on failure rates, with the evidence suggesting that the choice of programming language has no significant impact on the DFW rate (e.g., Watson & Li, 2014). Researchers have also considered pairs programming, a cooperative software development method that has emerged from recent developments in agile software engineering, as a potential solution, but found that it did not produce significantly higher rates of success (McDowell, Werner, Bullock, & Fernald, 2006).

Prior experience in programming, science, and/or mathematics has also been researched in an effort to determine if prior related experience is an indicator of subsequent success in introductory programming courses. Results have been mixed: prior programming experience was shown to be a positive predictor of success in CS1 in some cases, and a non-influencing, or neutral effect in others (Ventura & Ramamurthy, 2004; Horton & Craig, 2015). On the other hand, high-level mathematics experience, such as calculus, has been demonstrated to be a somewhat positive predictor of success in introductory programming courses (Owolabi, Olanipekun & Iwerima, 2014, p. 109). This may be because mathematics is suspected to contribute to the development of problem-solving skills (Hiebert et al.,

1996). This area of research has shown the most promise in finding a predictor of success in CS1. However, many universities are not able to be highly selective in terms of student population prerequisites, so it may be unrealistic to require students to have coursework in advanced mathematics prior to enrolling in an introductory computer programming course for many computer science programs.

As in other educational disciplines, a student's belief in their own capabilities is associated with their degree of success. Self-efficacy in computer programming is no different and should be considered "a significant aspect of learning" (Sharmain, Zingaro, Zhang, & Brett, 2019). Kallia and Sentance (2018 & 2019) and Ramalingam, LaBelle, and Wiedenbeck (2004) stress the importance of enhancing students' self-efficacy in computer science courses, particularly programming.

While there is much research on what to teach in an introductory programming course—such as procedural or object-first approach, the best programming language for a novice, the best format for delivery (face-to-face, online, flipped)—there is very little research that explores developing the novice student in a holistic way. A computer programmer needs to master three facets of computing: syntax knowledge, conceptual knowledge, and strategic knowledge (McGill & Volet, 1997). Most research in this field has focused on one of the three facets. Few studies have explored how to provide course content to address this triumvirate of skills holistically.

A few researchers have looked to educational theorists such as Piaget and his successors' neo-Piagetian theory for an explanation. Research has shown that neo-Piagetian theory is relevant to computer programming (Lister 2011; Teague, Corney, Ahadi, & Lister

2013). Lister (2011) was able to explore the relationship between novice programming students and neo-Piagetian theory. He found that the novice programmer progresses through learning stages, essentially from concrete to abstract. It is the objective for a learner to reach the abstract level in order to be a competent and independent software developer (Knight & Sutton, 2004). Lister and others often state in their concluding statements that neo-Piagetian theory should inform pedagogy. Additionally, educators of adult learners must "allow students to master programming skills using the type of reasoning indicative of the earlier stages of development before exposing them to significantly more abstract concepts" (Teague & Lister, 2014). Studies that take up this pedagogical problem in a comprehensive way have not yet materialized. Modifications to curriculum and pedagogy in light of neo-Piagetian theory is thus a viable area of exploration in pursuit of a way to address the DFW problem in introductory programming courses.

Statement of Problem

While many facets such as student characteristics, prior programming experience, programming language taught, etc. have been explored, nothing has been found to be a consistent predictor of success rates in introductory programming courses. Some research has been conducted regarding applying pedagogy to each facet (Knight & Sutton, 2004). These studies typically address one of the three major components (syntax, conceptual knowledge, and application to solve a problem) in learning computer programming. Little research exists that explores how educators can address success in introductory programming by understanding how its students learn and how an educator might best adapt pedagogy to enable and maximize attainment of the three components of ability that a

programmer should possess. We know that introductory computer programming students progress through stages as defined by neo-Piagetian theory (Lister, 2011; Teague, 2015). This knowledge has not yet produced a comprehensive application to educational practice (Knight & Sutton, 2004). Additionally, researchers often focus on improving one of the three components with little improvement in the success of novice programmers. Perhaps all three facets need to be addressed as individual parts of a whole. Perhaps a new approach to course design, informed by neo-Piagetian theory, and expressly holistic in its pedagogical approach, could provide more insight on how to improve DFW rates.

Piagetian learning theory is a theory that addresses the intellectual development of children. Piaget proposed four stages of cognitive development that are aligned with the age of a child. The stages progress from reflexive to concrete to abstract:

> In Piaget's view, children's thought processes move from innate reflex actions (sensory-motor stage, birth to two years), to being able to represent concrete objects in symbols and words (preoperational stage, two to seven years), to an understanding of concepts and relationships of ideas (concrete operational state, seven to eleven years), to an ability to reason hypothetically, logically, and systematically (formal operational stage, twelve-plus years). (Merriam, Caffarella, & Baumgartner, 2007, p. 326).

Piaget's theory does not address an individual beyond 12 years old, including adult learners. Neo-Piagetian theory has expanded Piaget's original theory to better understand cognitive development in adulthood.

Neo-Piagetian theory accepts the four stages posed by Piaget: sensorimotor, preoperational, concrete operational, and formal operational. When relating this to the adult learner, the theme of age-related progression through stages is not applicable, but theorists believe that the novice programmer will still progress through the neo-Piagetian stages. Further they assert that, "[the] novice programmer actually positions these behaviors as

normal behaviors to be expected in the long and torturous cognitive development of the novice programmer" (Corney, Teague, Ahadi, & Lister, 2012, p. 86). Instructors of novice programming students could benefit by understanding that students develop and proceed through the stages established by neo-Piagetian theory.

Theoretical Framework

Several theoretical frameworks form the foundation of this study as shown in Figure 1.1.

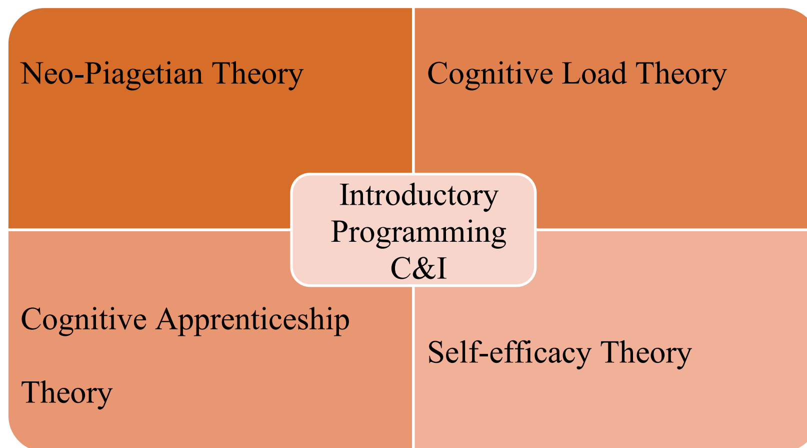

Figure 1.1
Theoretical Framework

Neo-Piagetian theory builds upon Piagetian theory. Neo-Piagetian theory includes the principles from Piaget: the idea of stages, that cognitive structures are actively created by learners, that these cognitive structures become increasingly complex through the intricate interaction of maturation and experience in a cyclical knowledge building process, and that cognitive levels build on and transform the lower, less complex levels (Knight & Sutton,

2004, p. 49). The schemes or stages from least abstract to abstract are sensorimotor, preoperational, concrete operational, and formal operational. The learner will progress through these four stages, regardless of age (Knight & Sutton, 2004, p. 49). This is a substantial difference from Piaget, who believed that progression from one stage to the next is connected with the age or maturation of the learner. Neo-Piagetian theorists further extend Piagetian theory through the observation that the adult learner can display different levels of cognition in different topics. The learner can display an "unevenness in development across different domains and contexts. . . . [this] is the norm and [is] to be expected." (Knight & Sutton, 2004, p. 49).

Vygotsky (1978) developed the concept of "Zone of Proximal Development" (ZPD), which posited that a novice's thinking is influenced by relationships with others who are more capable (Vygotsky, 1978, p. 87). Essentially, a novice learner needs assistance from an expert to initially demonstrate or share knowledge and skill, arrange experience for the novice to practice the skill and be guided through the exposure of knowledge, and to gradually provide less and less of this support. This modeling, a bit like that of a coach, enables the learner to obtain basic skills and internalize the skills/knowledge. The student is an active participant in their learning and the instructor is a collaborator with the learner. Bruner (1960) used the term "instructional scaffolding" and posited similar concepts regarding the need for guidance which is provided at just the right time and with just the right content (Bruner, 1960).

Cognitive Apprenticeship theory (Collins, Brown, & Newman, 1987) is similar to Zone of Proximal Development, but proposes six teaching methods: modeling, coaching,

scaffolding, articulation, reflection, and exploration. This theory also relies on an expert as the initiator or enabler in learning:

> In order to make a real difference in students' skill, we need both to understand the nature of expert practice and to devise methods that are appropriate to learning that practice. Thus, we must first recognize that cognitive and metacognitive strategies and processes, more centrally than low-level subskills or abstract conceptual factual knowledge, are the organizing principles of expertise, particularly in domains such as reading, writing, and basic mathematics. Further, because expert practice in these domains rests crucially on the integration of cognitive and metacognitive processes, we believe that it can best be taught through methods that emphasize what Lave (in preparation) calls successive approximation of mature practice, methods that have traditionally been employed in apprenticeship to transmit complex physical processes and skills (Collins et al., 1987, p. 2).

Cognitive apprenticeship theory addresses pedagogical methods, sequencing of learning activities, and the sociology of learning to replicate the traditional model of apprenticeship in the modern educational setting.

Sweller (1988) developed a Cognitive Load Theory which addresses how students process new information and how the instructor can best manage presentation of information to optimize student learning. Sweller studied whether working math problems of basic principles after demonstration from the instructor was an effective method for student learning. He suggests that this might not be the best method for the student and further suggests that more worked examples could be beneficial. He states that problem solving as a learning device is not necessarily effective. This theory has application to computer programming in that the learner needs direction in both learning basic principles and in applying those principles to solve a problem.

Bandura defines self-efficacy as "people's judgments of their capabilities to organize and execute courses of action required to attain designated types of performances" (Bandura, 1986, p. 391). Bandura suggests four ways a person can develop self-efficacy; two are quite

relevant to introductory computer programming. First, “mastery experience” (success) leads to greater self-efficacy. Second, “social modeling” (seeing others, especially those similar to ourselves, succeed) increases the belief in our own ability (Bandura in Lopez, 2008). In light of Bandura, Kinnunen and Simon (2011) determined four aspects for consideration in promoting self-efficacy in computer science. These include avoiding repeated failures at the beginning of the learning process, selecting tasks that are at the appropriate difficulty level and provide an attainable path of benchmarks that the student can trace, and intentionally managing the classroom environment to create a sense of belonging. Cognitive apprenticeship theory is closely related to self-efficacy in that the instructor can act as a social model and provide mastery experiences to the learner. In summary, one would expect that guided and careful exposure to programming exercises in the classroom and outside the classroom will enable small victories and lead to an increase in self-efficacy.

Each of the theories—neo-Piagetian, cognitive load, cognitive apprenticeship, and self-efficacy—can contribute to elements in a course design. The theories can be used to inform the design and delivery of an introductory computer programming course to holistically develop the student as a nascent computer programmer.

Purpose Statement

The purpose of this quasi-experimental study was to describe DFW rates and self-efficacy in students enrolled in an introductory computer programming course designed around the neo-Piagetian, cognitive apprenticeship, cognitive load, and self-efficacy theories. Students’ perceived belief of how the format of the class contributed to their own performance in the course was also studied. The study considered student characteristics

such as prior programming experience for 96 participants at a Midwestern university. The course design and delivery for this study, the Holistic Course Delivery, is generally defined as a course curriculum designed with planned topic presentation that utilizes minimal cognitive load and the components of cognitive apprenticeship. Student "success" is defined as obtaining a grade of A, B, or C in a first programming course. The aggregate number of students earning grades of D or F, and students who withdrew from the course after the last official free drop date, was divided by the total number of students who registered for the course and did not drop it before the last official drop date—this is referred to as the DFW rate. In addition to the DFW rate, student self-efficacy was measured throughout the course with a cadence corresponding to the homework assignments and at the beginning and the end of the course. Prior programming experience and a pre-collegiate standardized mathematics test score were also considered.

This research is significant and contributes knowledge to many different constituencies. The primary constituency to receive benefit from this study is the future students, who will obtain a deeper understanding of course material, have greater success in the course, and be better equipped to tackle higher level computer science coursework. Additionally, universities will benefit from an increased rate of retention for the computer science major, which is advantageous relationally, reputationally, and fiscally. Third, educators will benefit from the detailed study of this topic as it may help them to improve their own practices in teaching introductory programming courses, resulting in better outcomes and decreased vocational frustration. Fourth, this research will contribute to the sparse body of literature on this critical subject that has not been adequately examined in the

immediate past years. Finally, it is also an interest of this study to determine if the demographically defined population of students most likely to fail CS1 can be better served.

Research Questions

This study sought to describe DFW rates, final exam scores, and self-efficacy for students taking an introductory computer programming course designed and delivered according to principles informed by the Neo-Piagetian, Cognitive Apprenticeship, Cognitive Load, and Self-Efficacy theories. Such a course is referred to as a Holistic Course Delivery, as opposed to a course that is not intentionally developed according to insights to these four theories, referred to as a Traditional Course Delivery. Additional open-ended interviews were conducted to complement the quantitative results of the study and were used to help interpret findings in the discussion section. The research questions for this study included the following:

Research Question 1: How do DFW rates for the Holistic Course Delivery compare to historic DFW rates for introductory programming courses taught using Traditional Course Delivery at the institution at which the present study was conducted, and internationally?

Research Question 2: Does student self-efficacy with respect to Java programming and with respect to problem solving change over the course of the semester for students participating in the Holistic Course Delivery?

Research Question 3: Are there particular student characteristics associated with total points achieved in the Holistic Course Delivery?

Research Question 4: What course elements do students believe were helpful in the Holistic Course Delivery?

Research Question 5. Does the neo-Piagetian stage of students change from the beginning to the end of the Holistic Course Delivery?

Definition of Terms

Because this study addresses two primary audiences, both educators in general, and computer science educators in particular, definitions are given for terms that might be unfamiliar to one or another of these audiences.

Neo-Piagetian Theory

Neo-Piagetian Theory is a learning theory that builds from the seminal work of Jean Piaget on intellectual development. Piaget proposed that the learner advances through four cognitive stages: sensorimotor, preoperational, concrete operational, and formal operational. The stages are essentially organized from concrete to abstract and the learner will advance through these stages as they age or mature. Piaget described the basic unit of cognitive analysis, or a scheme. A scheme functions through a dual process or assimilation and accommodation (Mascolo, 2015). Essentially, that which is to be known is incorporated into an existing scheme and the existing scheme is modified to include the knowledge. Neo-Piagetians revised this theory to include skills rather than schemes that are products of context. Piagetian theory addresses cognitive growth in normal children by age and therefore does not allow the possibility that learners develop at substantially different rates. Neo-Piagetian theory provides an alternative and more universal model and allows that a learner, regardless of age, will advance through the four stages of cognitive development as new domains of knowledge are encountered. This effectively extends Piaget's theory into the realm of adult learning.

Cognitive Apprenticeship Theory

Cognitive Apprenticeship Theory (Collins et al., 1987) proposes six teaching methods: modeling, coaching, scaffolding, articulation, reflection, and exploration. This theory relies on an expert as the initiator or enabler in learning. Essentially, the instructor models a skill and the learner is guided by the instructor with decreasing dependence on the instructor while a particular skill is attained.

Cognitive Load Theory

Cognitive Load Theory deals with how students process new information and how the instructor can best manage presentation of information to optimize student learning. It is believed that by minimizing extraneous information for the student that students can focus on learning important knowledge.

Traditional Course Delivery

Traditional course delivery is a course design which relies heavily on lectures by the instructor. Students are primarily passive during class meeting times. A delivery method often used in traditional courses is content knowledge shared via PowerPoint presentation. There could be opportunity for students to observe the instructor live coding, but students performing live coding during the lecture time is rare. The course will typically follow the concept ordering as presented in the textbook. Students will have weekly homework assignments that include problems requiring content knowledge not necessarily demonstrated during class meeting times. Midterm(s) and a final exam are the primary assessment instruments in a traditional course.

Holistic Course Delivery

The course design and delivery utilized in this study that incorporates Neo-Piagetian, Cognitive Apprenticeship, Cognitive Load, and Self-Efficacy theories in an introductory computer programming course.

DFW rate

DFW rate is the ratio of the total number of grades of D or F, and the number of withdrawals from a course divided by the number of students registering for the course who do not drop it before the last free drop date. A withdrawal occurs when a student drops or withdraws from a course after the formal add/drop period at the beginning of a semester.

Student Success

Student success is defined as a student completing an introductory computer programming course with a grade of C or higher.

Self-efficacy

Self-efficacy is defined as a student's individual belief in his or her ability to succeed.

Special Demographic Participants

Special demographic participants include female students and underrepresented minorities in an introductory programming course. Underrepresented minorities typically are females and persons who are non-white.

Historically Challenging Programming Questions

These include programming questions that deal with programming concepts of variable assignments, if-statements, loops, and lists.

Introductory Programming Course

An introductory programming course is a course in which programming fundamentals are introduced. Traditionally there are two courses (CS1 and CS2) that comprise the foundation in computer science, particularly the programming aspect. These two courses were originally developed in the ACM's 1978 *Computing Curricula*, and the names continue today with modernization of course content. While there is disagreement among educators as to what topics are typically included in CS1, the primary topics in CS1 include the programming constructs of variables, types, conditionals, loops, methods, and arrays (Hertz, 2010). CS2 is often a continuation of CS1 that adds object-oriented programming and/or an introduction to data structures. Both courses will have some level of problem decomposition, testing, and debugging. For the purpose of this writing, the description of topics typically included in CS1 (as mentioned above), will comprise an introductory programming course.

Triumvirate Skill Set

Many have stressed the importance that a computer programmer needs to master a triumvirate skill set. This set of three skills includes syntactical knowledge, conceptual knowledge, and strategic knowledge (McGill & Volet, 1997). Each skill is defined by Qian and Lehman (2017) as follows:

> Syntactical knowledge is the knowledge of the language features, basic facts, and rules. Conceptual knowledge refers to the knowledge of how programming constructs and principles work and what happens inside the computer. Strategic knowledge refers to how to apply syntactic and conceptual knowledge of programming to solve novel problems. (p. 3-4).

A novice programming student will need to learn the syntax of a programming language, be able to apply multiple programming constructs, and ultimately be able to solve a variety of problems.

Study Limitations

This study has potential limitations. Random selection of participants is not possible, since students are able to self-enroll in course offerings. A non-randomized control group was considered but presented several concerns: (1) The researcher's belief that the Holistic Course Delivery may be significantly superior to the Traditional Course Delivery might have introduced an element of researcher bias in the delivery of the course, had the researcher employed both methods in different sections of the course; (2) The one other section of the course for which data might have been available to the researcher was taught by another instructor and met twice per week (instead of three times per week, as in the sections taught by the researcher). This additional section represented a small data set in comparison, met less frequently, and was taught by a different instructor—hardly a reasonable basis for comparison.

It has been argued in the literature that course grades are an inadequate measure of student knowledge acquisition. While this is acknowledged, this study used course grades because the DFW rate, which intrinsically incorporates letter grade earned in a course, is a metric that is tracked by, and important to, universities.

A pre-test and pre-course survey was used to establish a baseline and to understand the characteristics of the class make-up and each individual student. The pre-test and pre-course survey also served as post-test instruments, potentially allowing for threat of testing.

Due to the methodology used in this study, causation was not confirmable. The research findings may be generalizable to other similarly situated institutions.

CHAPTER 2

LITERATURE REVIEW

Brief Summary of History & Progression of Computer Science Education & Curriculum

Currently there is no required standard to teach in computer science. Most would agree that computer programming is at the heart of computer science. While there is a hardware aspect to computing, individuals with a degree in computer science will most likely take a professional role in software development. Computer science curriculum has developed over many decades, but only after computers became a reality and were determined to be a very useful tool for humanity.

We can begin the story of the origin of computer programming with Ada Lovelace in 1843. Lovelace was working as an assistant to Charles Babbage while he was developing the Analytical Engine Difference Machine. The concept of a computer program (a set of instructions a machine could execute to solve a problem) existed prior to the first computing machine. A computer was originally a machine that would perform calculations. One hundred years would pass and a world war (WWII) would occur before significant developments in computing would take root. The Eastern Association for Computing Machinery was formed on the east coast of the U.S. to promote and advance science, development, construction, and application of new machinery for computing, reasoning, and

handling of information. This was on the heels of WWII when computers had names such as ENIAC and MARK I. The mathematicians and scientists managing these machines--Hopper, Aiken (IBM)--used them to calculate ballistic firing tables. The computer used pre-punched paper for arithmetic instructions. Data was stored mechanically, and results were then transferred to paper through an electronic typewriter, an early version of printer. While this was a specialized, almost single-purpose machine, scientists were quite aware of the potential computers represented and conceived of ways to make the computer versatile, general, and more powerful. Thus, the concept of a compilable language was born and computing expanded quickly in the realm of software and programs.

The first computer programmers were predominantly mathematicians, so early computers were applied to solving mathematical problems. During the 1950s, George Forsythe with Stanford wanted to apply computers to a general set of problems that went beyond mathematical calculation. The Division of Computer Science was born, but under the Mathematics Department. Purdue University established the first Computer Science department as such in 1962.

During this period (1950s-1960s), there was ongoing conflict on what computer science was and where it should be housed at universities. Is it a tool for mathematics? Is it a natural science? An individual science? Is it engineering? And so on. Strong opinions fueled substantial controversy about the nature of computer science and the way new practitioners should be taught the discipline.

It was by the work of people organized under the Association for Computing Machinery (ACM), formerly the Eastern Association for Computing Machinery, that formal

computer science curriculum was defined. In 1968, the first curriculum for higher education was released. This curriculum aligned heavily with mathematics and remained math-focused until the late 70s and early 80s, during which there was a new push for CS to be "professionally-focused" and to become a field in its own right. ACM began to define more curriculum that was application oriented and less fundamentally theoretical or mathematical. During this time the Institute for Electrical and Electronics Engineers (IEEE) published their own curriculum for computer science that was more hardware based. This was useful for those studying computer engineering, but it failed to connect hardware and software and missed the mark for computer science.

While ACM continued to refine what constituted a computer science curriculum in higher education, a movement to accredit computer science programs began. In 1990, ABET, under the CSAB (Computer Science Accrediting Body), developed criteria to accredit undergraduate CS programs. These criteria established minimum requirements for mathematics (1.5 years – 45 credits). The criteria for curriculum was informed by the ACM curriculum. There are requirements for minimum credits in science, math, and computing subjects, but the latter allows for a wide variety of topics, focus, and definition for individual programs.

During this time period, computer science was increasingly taught in public high schools—primarily something arranged along the lines of the first programming sequence course from higher education, and an Advanced Placement (AP) exam was offered. High schools varied widely in how they credited computer science course work. Some districts counted the credit as elective while others counted the course as a math, science, or foreign

language credit. Similar to the controversy of where to house computer science in higher education, how to allow credit for Computer Science in public high schools has been just as controversial. Computer science didn't fit any of the traditional areas and those traditional areas (mathematics and science) perceived a threat in allowing CS to exist at all.

After the dot-com bubble of the 2000s there was an increasing realization that the need for persons trained in computer science had not perished with the venture capital-fueled startups of the era. Computing was integral to modern life and was here to stay. Private non-profit organizations such as Code.org were formed to promote computer science in K-12 schools. These organizations placed computing skills at the same level as reading, writing, and mathematics. They believed that computational thinking and programming are skills to which all people should be exposed. States also began to awaken and address this need by examining how to bring computer science into schools and how to give credit for high school computer science courses. Some states now require computer science to be taught in high school. Many educational bodies pushed for exposure in K-12, a pathway to teacher certification, and credit as an advanced science, math, or foreign language. Some fear that a student could forego mathematics coursework in high school, replace it with CS, and enter college mathematically underprepared. Nonetheless, Code.org estimates that there are over 500,000 computing job openings in the U.S. right now, and that we produce 63,000 graduates into the workforce each year (Code.org). (Brief history summarized from Misa, 2017, *Communities of Computing*).

Important Observations in the Past 20 Years in Introductory Programming

Soloway, Ehrlich, Bonar, and Greenspan (1982) examined the disconnect that novice programmers exhibit between natural language and programming language. They suggest that instruction could be a contributing factor (Soloway, Ehrlich, Bonar & Greenspan, 1982, p. 12). More recently, McCracken et al. (2001) published a seminal paper on the programming competency of students in higher education who recently completed the first two programming courses in computer science. In the study, the McCracken group created a set of five student learning objectives that first year students should attain. These include the following: abstraction of a problem from a description, decomposition of a problem into sub-problems, creation of solutions for sub-problems, combining sub-problem solutions, and iterative evaluation and correction until a final solution is produced. After this framework was determined, the group developed strategies for assessing the attainment of the outcomes. Charettes, or short programming assignments completed in a lab setting were used to assess the participants. A set of three charettes were created and the instructors from four universities administered at least one of the three to their class. A common rubric was created to maintain consistency of grading across participating institutions. There was also an optional student questionnaire that gathered demographic information, programming background, and reaction to the exercise(s). The McCracken group addressed the challenge of comparing results from participating institutions. After considering differences among student prior experience, time allowed, hints given, etc., they were able to conclude that students did much more poorly than predicted and that students in introductory computer programming courses do not know how to program at the expected skill level. The authors

did acknowledge that their expectations might have been too high and that this could have contributed to the results. The study did show that students often were unable to successfully deliver a program that would even compile (a step in which program syntax is checked for errors). A major point in the conclusion was that “issues of how the course is taught and who the students are influence the outcome” and that this is more relevant than the programming language used in the course (McCracken et al., p. 132). While there were other interesting findings in this study, it suggests that we overestimate the types and complexities of problems that we can expect first year students to complete and that how students are taught—including scope, sequence, and delivery method—is intimately tied to student success. The McCracken paper continues to be extensively cited and was a major contribution in highlighting where first year programming students are weak, and a possible disconnect in what students should be expected to know at the end of introductory programming coursework. A curriculum designer cannot consider course content for an introductory programming course without reflection on the McCracken Report.

Lister (2011) boldly concluded in his study that teachers of introductory programming students should reset their expectations for skill attainment and, specifically, that the first semester student should not be expected to develop programming skills at a highly abstract level. Lister took a step back to analyze how students’ cognitive thinking develops in a first semester programming course. He found that a programming student develops skills from concrete to more abstract concepts which aligned with neo-Piagetian learning theory. His study refined the major conclusions from the McCracken report by showing that both students’ cognition and programming-specific skill mature in a particular

order. He suggests that the computer science educator should be aware of the progression and consequently adapt introductory pedagogy to incorporate this theory (Lister, 2011, p. 17).

Brown and Altadmri (2017) support Lister. They examined programming mistakes that students frequently made and found that certain syntactical errors decrease, but that as programming problems became more complex some mistakes continue to be made. The fact is that some mistakes become easily identified, corrected, and no longer made. When coding problems become more complex, the novice student has difficulty managing syntax and has a need to apply more abstract thinking.

Predictors of Student Success

Previous Programming Experience

Prior programming experience would seem to be an obvious factor promoting student success. If a student has had some experience programming, whether formal or informal, one would expect the student to be more likely to succeed in an introductory programming course in higher education than if they had no such prior experience. Indeed, Petersen, Craig, Campbell, and Tafliovich (2016), indicate that lack of prior programming experience is a key contributor to students dropping or withdrawing from an introductory programming course. This agrees with a study conducted by American Association of University Women it was determined that students in introductory programming were more likely to succeed if they were enrolled in a section based on prior experience (Corbett & Hill, 2015), and another that found that students with no prior programming knowledge score lower than students with some prior programming knowledge (Veerasamy, D'Souza,

Linden, and Laakso, 2018). But not all studies are agreed on this point. A 2004 study points in a different direction: Prior programming experience did not benefit students taking an objects-first introductory programming course (Ventura & Ramamurthy, 2004).

Programming Language Taught

Programming language that is taught in a course is often considered a possible contributing factor to student failures in an introductory course. Watson & Li (2014) showed that pass rates were independent of programming language taught in the course.

Pair Programming

Some educators have turned to pair programming as an intervention in an introductory programming course to enable more student success. Petersen et al. (2016) conducted qualitative interviews of students who dropped an introductory programming course that utilized pair programming. They discovered that the student had come to rely on the partner during lab and that they had been unsuccessful in working independently. Pair programming has not consistently been shown to be a reliable intervention.

Gender

Women are underrepresented in introductory programming courses. According to a study by Sankar, Gilmartin, and Sobel (2015), women are less likely to answer course-related questions and choose to use tools that maintain anonymity. This points to a potential connection with self-efficacy and a difference in male and female students. One study found that women performed similarly to their male peers in an introductory programming course (Pillay & Jugoo, 2005). It seems that performance is gender independent, but self-efficacy might not be level between genders.

Others have developed courses in an attempt to better engage and retain female students. An introductory computer programming course was conducted as an experiment among Biology majors. Based on prior literature, female students are found to perceive an introductory programming course more difficult than the male students in the course. Additionally, male students are more likely to pursue additional programming coursework. The researchers theorized that the pedagogy of the introductory programming course contributed to these perceptions. They designed an experimental section of a course that used physical computing with Arduinos. The researchers were hoping that the use of contextual programming through Arduinos would be more engaging and less intimidating for female students. The researchers were able to conclude that the gap between male and female students' perception and learning outcomes was reduced with their pedagogy (Rubio, Zaliz, Manoso, & Madrid, 2015).

Class Size

Class size has been explored as a possible contributing factor toward the DFW rate. Watson and Li (2014) determined that classes with less than 30 students will typically have a lower DFW rate.

Delivery Format

Delivery formats may comprise of pure lecture, lecture and code, or lecture and lab. Hawi (2010) demonstrated that students who practiced coding concepts by creating programs were much more likely to be successful in an introductory programming course. This supports having the student participate with the instructor in writing working computer programs in the classroom. It is through this instructor/student interaction that the student

learns how to learn computer programming. This study explored other factors related to lack of success in an introductory programming course, such as “lack of study,” “subject difficulty,” “exam anxiety,” and others, but found that students who were high achievers believed their success was related to their “learning strategy” or intentional coding practice.

Prior Research on Challenges Programming Novices Face

Code Tracing

Code tracing is an important skill for the computer programmer. Code tracing is reading a computer program in order to predict what the program will do. Kaczmarczyk, Petrick, East, and Herman (2010) studied students and misconceptions they have in introductory programming. These researchers found that students form strong assumptions about a specific piece of code—in this case, variables and memory models—and are unable to recognize true programming issues because of their assumptions. The researchers indicate that the instructor is key in correcting and preventing this common weakness in novice programmers (Kaczmarczyk, Petrick, East, & Herman, 2010).

Veerasamy, D’Souza, and Laakso (2016) studied misconceptions of novice Python programmers and found that students who were unable to trace code were also unable to write code. Code tracing is a skill that precedes code writing.

Coding Constructs

Programming includes many constructs. Examples of coding constructs include selection statements to determine a logical path within a program and looping constructs which allow portions of a computer program to be repeated until a condition is met to terminate repetition. It was found that understanding loop programming constructs requires

cognitive skills beyond the ability to type a looping construct syntactically correctly (Veerasamy, D'Souza, & Laakso, 2016).

Veerasamy et al. (2016) did not find that students had significant issues with defining methods and passing parameters to methods. Other researchers referred to in Veerasamy did find otherwise.

Understanding logic is key to success in using programming constructs such as selection statements and loops. VanDeGrift et al. (2010) found that instructors should not assume students have prior knowledge or mastery in logic. Students often exhibit fragility in applying logic to programming. Students might understand logic concepts in English but have difficulty in translating the concept to a computing language. Based on this observation, my study will incorporate instruction and student practice problems on the basic truth tables and careful wording when explaining selection and loop constructs.

Strategic Knowledge

Veerasamy et al. (2016) found that two thirds of the students in their study struggled to solve mathematics-based problems, in which the students used coding syntax and constructs to solve said problems. This discovery supports the notion that prior knowledge outside of computing, i.e. mathematics, cannot be taken for granted. Perhaps novice programmers need additional support in math review or more considerate choice of problem selection on behalf of the instructor.

Reasons Students Dropped an Introductory Programming Course

Petersen et al. (2016), indicate that lack of prior programming experience, lack of time, lack of motivation, and pace of course were all reasons that students dropped introductory programming.

Education Theory that Informs Computer Science Education

Cognitive Load Theory

Caspersen and Bennedsen (2007) explored the benefits of designing an introductory programming course with consideration for cognitive load and cognitive apprenticeship. The course design was for an objects-first approach to teaching introductory programming. Their writing serves as a primer for CS educators in understanding learning theories of cognitive load and cognitive apprenticeship and how these theories can be applied in the introductory programming classroom. The study did not provide any evidence of success of this model but indicated that it has been used successfully with over 400 students in four years. Some would disagree with an objects-first approach to teaching programming, but this is incidental to the chief value of their study, which is that it provides a way to reconsider and improve pedagogy. Although the research in this study did not use an objects-first approach it has intentionally incorporated Cognitive Load Theory.

Bouvier et al. (2016), demonstrated that designing programming problems with a complicated theme or real-world context contributed to a greater cognitive load and had a negative impact on learning in a CS1 course. While CS educators seek to make problem sets interesting and relevant, the instructor will need to weigh the cognitive load to the potential return on learning for the student.

Debugging computer programs is closely related to problem solving. While problem solving involves designing a solution and producing an artifact, debugging involves searching for cause when the artifact does not perform correctly, or is syntactically incorrect. Becker et al. (2018) analyzed error messages from student produced programs. A student program might list several errors in a single compilation. The Becker study found that having students focus on the first reported error and to ignore any other errors listed minimized the cognitive load for the student. Often, corrections made in relation to the first error message will drastically alter the successive list of errors. They advise instructors to demonstrate the technique of only solving the first compilation error and then recompile and repeat the process.

Brown and Altadmri (2017) studied student programming mistakes. They believed that instructors often rely on intuition, knowledge, and experience to determine which programming mistakes are commonly made and how long it takes students to correct the error. Brown and Altadmri (2017) surveyed computing educators on these two points. They compared the results of the survey to actual data collected on student mistakes and time to correction. The study concluded that instructors, regardless of experience level, often mispredicted the frequency of errors that students make and the time needed for correction. The researchers suggest that computing educators should rely on more than intuition to better understand errors that students make and incorporate this knowledge into course content. This study will be reflected in my own practice that is connected with this dissertation.

Bergersen and Gustafsson (2011) studied computing professionals and whether working memory was mediated through programming knowledge. They administered twelve programming tests and a working memory test to each participant. They found that an increased level of programming knowledge allows for greater cognitive load. While this study focused on the professional, the concept is applicable to the novice learner in that the novice will also be able to manage a greater cognitive load as foundational concepts are stored in long-term memory.

Stachel et al. (2013) examined the use of scaffolding tools on cognitive load. Their study was conducted on students in a programming course. The experimental study involved the treatment group receiving worksheets, video presentations, and simulations/animations as part of their instruction. The analysis of collected data showed an improvement in course grades and a decrease in frustration over the control group.

Cognitive Apprenticeship

Vihavainen, Paksula, and Luukkainen (2011) built on the idea of cognitive apprenticeship and developed a course utilizing extreme apprenticeship. Scaffolding and continuous feedback was a key component of their course design. They chose to avoid “preaching” during class lecture times and to replace traditional lecture with a substantial number of worked examples. The study was conducted on two sections of two courses during a single semester and demonstrated a higher passing rate and a lower dropout rate by using their method.

Neo-Piagetian Theory

Fisher and Kenny (1986) examined the mathematics thinking of students ranging from elementary age through adulthood. They found that five levels of cognitive ability could be defined and that the learner advances from one level to the next, which progresses from concrete to abstract. Their study adds an additional stage that refines formal operational thinking. The study concluded that the development of "mental muscle" is related to the amount of support and practice of the learner and that practice is necessary for the learner to achieve the next cognitive level. It is also the case that students of computer science also progress through cognitive stages as they gain greater abilities of abstraction.

It is commonly agreed among computer science educators that "the process of abstraction will normally be prominent in all undergraduate curricula" (Turner, 1991). Kramer (2007) suggested that a test for ability in abstraction be a precursor to admittance into a computer science program. This reasoning firmly places abstract thinking as an innate trait rather than a concept that can be learned. Previous research sought to determine Piagetian level as a predictor of success and often would utilize instruments unrelated to programming to determine a participant's level of abstraction (Lister 2011). Lister (2011) looked for an alternative explanation by exploring learning theory as a way to develop abstract thinking. Specifically, he applied neo-Piagetian learning theory to learning introductory computer programming. He established that the student progresses through stages of learning that move from concrete to more abstraction and that the learner might exhibit different levels on different topics. Further, Lister says that we as computer science educators are flawed in thinking that the student will be able to reach the target level of

formal operational reasoning in an introductory course. He remarks that teachers of introductory programming students should seek for the student to reach the concrete operational level. The learner might not exhibit the formal operational stage of abstract ability of reasoning, but that a pedagogical approach informed by neo-Piagetian theory can enable the learner to transition from the concrete to abstract forms of thinking.

Lister (2011) also demonstrated that students at the preoperational stage can trace code with about 50% accuracy or greater. Students with less than a 50% accuracy are considered to be operating at the sensorimotor stage. Code tracing involves manually executing a piece of code to determine values in variables. The student must provide, or be provided, initial values for the variables and then track how they are modified through some number of lines of code. A novice at the preoperational level is somewhat successful at tracing code but is unable to provide a description of what the code is doing without performing a trace or making a guess (Lister, 2011). This particular characteristic of the preoperational stage learner will be tracked throughout this study to help place the student at the stage of sensorimotor or preoperational. Expert (formal operational stage) programmers have the ability to read a section of code and deduce what it is doing without manually tracing values of variables.

Teague and Lister (2014) conducted qualitative research through interviews with novice programmers. The participant led the interviewer through a common programming task by verbally walking through a solution. This format is called a “think aloud.” Two programming questions were used in the instrument that were based on an experiment by Piaget on cyclic series. The two problems would be considered a simple problem for an

experienced programmer but poses a challenge to a novice programmer. The two problems were closely related in that they performed the same task, but the second problem added one additional requirement. There were eleven students in the study, and four of the eleven participants demonstrated difficulties in solving the second problem. Analysis of the think aloud data confirmed that novice programmers struggle with preoperational reasoning. While the sample was small in order to allow for interview/think aloud, they were able to demonstrate that computer science educators must possess an awareness that novice programmers develop at different rates and will not necessarily reach the same neo-Piagetian stage at the end of a course. They also suggest that computer science educators assume the student begins the course at a preoperational stage when in actuality a sensorimotor level is likely. Computer science educators cannot assume that basic programming constructs are easily learned and that students immediately are able to reason at the concrete operational stage.

Self-efficacy in Computer Science

Self-efficacy is defined as a novice programmer's belief that he or she is able to succeed is a necessary element. Wilson and Shrock (2001) discovered twelve factors that predicted success or failure in an introductory computer science course. Of the twelve, three—which include comfort level, math experience, and attribution to luck—were associated with success or failure. Comfort level in the course and math experience had a positive association, while attribution to luck had a negative association. While math is not an area that can be altered significantly prior to taking an introductory programming course, comfort level and luck are addressable. As with Bandura (2008) and Kinnunen and Simon

(2011), classroom climate is an important factor in self-efficacy. Creating a classroom in which students are comfortable to engage and desire to engage should be a priority in any introductory programming course. Second, on the topic of luck, if an instructor utilizes cognitive apprenticeship, cognitive load management, and scaffolding, the perceived need for luck can be greatly minimized. Through intentional topic order, worked examples, etc. a student will be better prepared for working independently and not need to rely on luck, either through random changes to programming problems or guessing.

Watson, Li, and Godwin (2014) examined predictors of programming performance. The student was able to confirm prior research in that students who perform well in programming courses have “high levels of intrinsic motivation and self-efficacy” (p. 472).

Ramalingam and Wiedenbeck (1998) designed a survey to measure levels of self-efficacy in programming students. This survey is widely accepted by the community. This survey was originally created to address students learning C++ programming language and was administered at the beginning and end of an introductory course.

Ramalingam et al. (2004), hypothesized that self-efficacy increases as a student progresses through an introductory course. The pedagogical theme is to challenge but not overwhelm the student and to build confidence through accomplishment. Frequent assignments with quick and ample feedback and watching someone else complete a difficult task both increase self-efficacy. The researchers saw the greatest improvement in students with initially weak self-efficacy. They examined 75 students in sections of CS1, mixed majors, C++. They believe that inclusion of elements from Cognitive Load Theory and cognitive apprenticeship contributed to the results.

CHAPTER 3

METHODOLOGY

Statement of Significance of the Problem

Industry demand for graduates trained in software development is not currently met by universities in the United States. A contributing factor is that many beginning students are unsuccessful in the first programming course. While many potential reasons have been suggested as to why many students are unsuccessful, pedagogy has not been a primary focus. An analogy of trade apprenticeship can be applied to learning computer programming, specifically the idea that the learner interacts with an expert to gain and hone skills (Collins et al., 1987).

This study examined whether teaching an introductory computer programming course with a design informed by Neo-Piagetian and Cognitive Load theories, and with a cognitive apprenticeship methodology and intentional emphasis on building student self-efficacy, will improve DFW rates and self-efficacy. This study has a quasi-experimental design and uses a static group pretest-posttest, multiple measures design as illustrated in Figure 3.1.

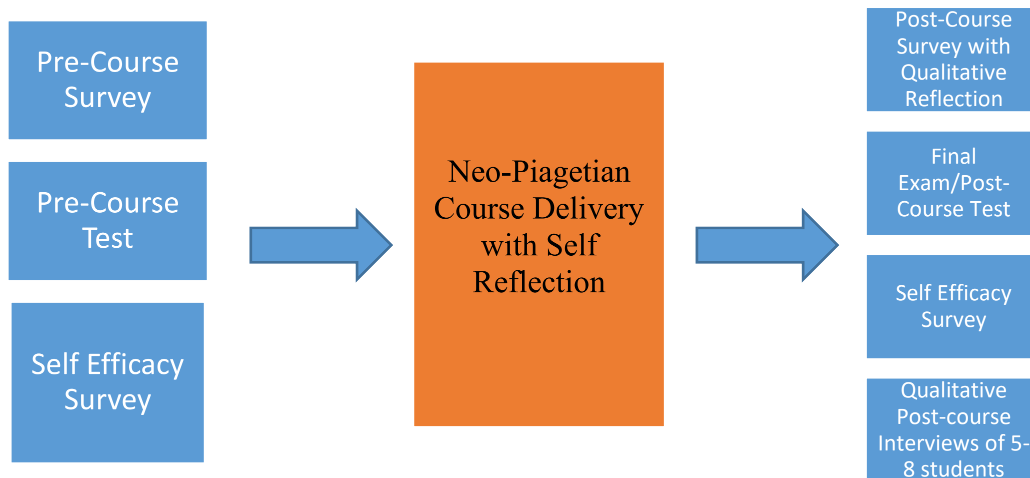

Figure 3.1
Quasi-experimental design static-group pretest-posttest design

A Randomized Controlled Trial is considered by some to be the "Gold Standard" for education-based research design. While researchers would prefer the advantages of a tightly-controlled experimental design that produces results indicating strong causal relationships, this is not always practical in the educational setting, and Randomized Controlled Trials are "quite limiting at best and inappropriate at worst" (Christ, 2014, p. 74). Given the impracticality of assigning students randomly to the study and ethical and pragmatic considerations concerning including a traditional course delivery method as a control (Cf. Study Limitations, p. 18, above), this research instead utilized a quasi-experimental design. Additionally, Christ states that Randomized Controlled Trials are often disadvantageous because the results are unable to "show how or why the intervention affected change in the participants, nor if the intervention was applicable, or desirable to the stakeholders" (Christ, 2014, p. 79). It was the intention of this study to analyze the relationship of course design

and delivery to DFW rate and self-efficacy, but also to capture the course elements students believed contributed to their success—the "how" and the "why."

Research Questions

Research Question 1: How do DFW rates for the Holistic Course Delivery compare to historic DFW rates for introductory programming courses taught using Traditional Course Delivery at the institution at which the present study was conducted, and internationally?

Research Question 2: Does student self-efficacy with respect to Java programming and with respect to problem solving change over the course of the semester for students participating in the Holistic Course Delivery?

Research Question 3: Are there particular student characteristics associated with total points achieved in the Holistic Course Delivery?

Research Question 4: What course elements do students believe were helpful in the Holistic Course Delivery?

Research Question 5. Does the neo-Piagetian stage of students change from the beginning to the end of the Holistic Course Delivery?

Participants

The target population for this study was introductory programming students. The accessible population for this study included three sections of the same introductory programming course with 96 students during the fall 2019 semester at a Midwestern university. The introductory programming course is the first course taken in computer programming. The students were not restricted to computer science majors. The students were primarily white, male, and traditional college-age; but other genders, races, and non-

traditional students took the course. Approximately 12% of the students were non-white and 21% female. There were no prerequisites for the course and students were from a variety of majors; but Computer Science, Cybersecurity, and Software Engineering majors predominated. The researcher served as the instructor for each of the three sections. There was one additional section of the course offered which was taught by a different instructor and was not part of this study.

There were 85 students who took the final exam and were considered to have completed the course, regardless of final grade. There were 11 participants in the study who either formally withdrew or effectively withdrew from the course. Participants who did not take the final exam and had stopped attending were considered to have effectively withdrawn from the course. Demographic information for the participants is shown in Table 3.1 and includes data for participants who started the course and for students who completed the course.

Table 3.1
Participant Demographics

		Initially Enrolled	Course Completers
Total Students		96	85
Gender			
	Male	78.1%	77.6%
	Female	20.8%	21.2%
	Unknown	1.0%	1.2%
Total		100%	100%
Race			
	White or Caucasian	80.2%	77.6%
	Asian	3.1%	3.5%
	Black or African American	6.3%	7.1%
	American Indian or Alaskan Native	2.1%	2.4%
	Unknown	8.3%	9.4%
Total		100%	100%
PELL Grant Eligible			
	Yes	75.0%	76.5%
	No	25.0%	23.5%
Total		100%	100%

Table 3.1—Continued

	Initially Enrolled	Course Completers
Year of Study		
Freshman	63.5%	63.5%
Sophomore	21.9%	21.2%
Junior	10.4%	10.6%
Senior	3.1%	3.5%
Other	1.0%	1.2%
Total	100%	100%
Major		
Computing Focused		
Computer Science	54.2%	51.8%
Cybersecurity	6.3%	5.9%
Software Engineering	9.4%	9.4%
Total Computing Focused	69.8%	67.1%
Other Majors		
Math	4.2%	4.7%
Actuarial Science	11.5%	11.8%
Math Education	5.2%	5.9%
Statistics	1.0%	1.2%
Bioinformatics	1.0%	1.2%
Music Tech/Music	3.1%	3.5%
Technology Management	1.0%	1.2%
Undecided	2.1%	2.4%
Unknown	1.0%	1.2%
Total Other Majors	30.2%	32.9%
Total	100%	100%
ACT Math Score		
N	72	64
Mean	23.0	23.3
Range	14-33	14-33

Detailed Description of the Holistic Course Delivery

The study included students in three sections of an introductory programming course that implements the following:

- Intentionally organized course content where topics were presented at just the right time. The breadth of each topic that was introduced consisted of that which was immediately needed, avoiding cognitive overload through too much detail.
- Cognitive load was intentionally managed so that the student was able to focus on one major concept at a time. New content knowledge was limited to only that which was needed at the moment.
- Half of the class time was dedicated to lectures. The remaining half of class meetings incorporated live coding, problem solving activities, and a bi-weekly code tracing and coding sprint.
- Live coding served as a scaffolding tool and was used when a new concept was introduced. The student worked alongside the instructor to create a program utilizing the content provided by the instructor (Cognitive Apprenticeship: Scaffolding). Live coding also allowed for the instructor to introduce errors or confusion into the program so that the students had to refine or restate their knowledge (Cognitive Apprenticeship: Articulating).
- Formative quizzes were given after lectures through Blackboard, an online learning management system. The quizzes provided feedback on missed questions so that the student could review lacking knowledge. Each quiz could be taken as many times as

desired over a 48-hour window (Cognitive Apprenticeship: Scaffolding and Reflection).

- There were no midterms. There was a final exam.
- Every two weeks, the student completed an in-class code tracing and program creation which was similar to an in-class quiz. These were called “Code Sprints.” The student had approximately 25 minutes to complete the two-part assessment. The first part required the student to trace code and select a response for the output of a computer program similar to those presented in the prior two weeks. The second part required the student to create a program that solves a given problem, relying on skills gained in the previous two weeks of lecture. The sprint was conducted through Blackboard during a regular class meeting.
- On the same day as the bi-weekly sprint, the students worked on a hands-on problem-solving activity with a partner, or as a class. These presented problems that were more complex relative to previous work. The students rearranged mis-ordered steps into a logical sequence or developed their own steps to solve the problem. (Cognitive Apprenticeship: Exploration). The students worked alongside the instructor (Cognitive Apprenticeship: Coaching), and every student left class with a working example that they had developed.
- The student was asked to provide pseudo-code prior to coding for a single problem on select homework problems for instructor review and feedback (Cognitive Apprenticeship: Coaching).

- The student completed a brief reflection survey after submitting a homework assignment. The self-reflection measured the student's confidence and perceived quality of their homework submission and the student stated challenges, or areas for improvement. The student provided an estimate of time spent on the homework assignment (Cognitive Apprenticeship: Reflection).
- The classroom climate was intentionally designed to encourage student questions, attendance, and a feeling of equality regardless of race, gender, or ability. This included questions of address (addressing students by their given names, flexibility in address of the instructor, encouragement for students to get to know their classmates); Strong and upbuilding engagement of student questions (encouragement of questions, embodying the notions of "no dumb questions" and "no dumb answers"; default assumption that the students do not understand because of a gap in communication originating with the instructor; Intentional effort to be approachable, available, and personable, including intentional identification with the cadence of an academic semester from the student's point of view and waxing/waning energy and stressors for the students; Instructor awareness of cultural differences and challenges, especially for the international students.

The process and instruments outlined in this chapter were piloted in a single course section in the semester prior to the semester engaged in this study. A summary of the course elements, and their relationships to the four fundamental learning theories, is shown below.

Table 3.2
Summary of course elements and theory that motivated inclusion

Course Element	Neo-Piagetian Theory	Cognitive Load Theory	Cognitive Apprent. Theory	Self Efficacy
Intentional content selection	•	•		•
Worked example and assigned problem progression	•		•	•
Formative quizzes	•		•	•
Minimal simultaneous new concepts	•	•		
Live coding with think-aloud		•	•	
Whole class live coding alongside instructor		•	•	•
Biweekly coding sprints (no midterms)	•	•	•	•
Problem-solving activities	•	•	•	•
Instructor accessibility			•	
Pseudocoding prior to coding				•
Self-reflections after homework submissions			•	•
Quick feedback on homework			•	•

Data Sources and Sample Instruments

Pre-Course Survey

The survey prompted the participant for demographic information, such as age, class level, gender, prior programming experience, and prior math experience. This survey also

captured perceptions and attitudes about computer programming. This survey was created by the researcher (Copus, 2015) under the guidance of a research advisor and used in an unpublished study on freshman and sophomore college students' perceptions of computer science. The survey included categorical and Likert-type data. This data was used to control for differences between participants. The Pre-course survey can be found in Appendix A.

Pre- and Post-Course Test

The test covered technical knowledge in programming and was administered at the beginning and the end of the course. The pre-course test served as a baseline for the skill level and neo-Piagetian staging for each participant. Problems were similar to those utilized by Teague and Lister (2014) and Kutscha (2017). The post-course test was a repeat of the pre-test and demonstrated level of achievement or course content and neo-Piagetian staging. The post-test was included as a portion of the final exam. Neo-Piagetian staging was determined from success on groupings of problems. The questions included on the pre-course and post-course test can be found in Appendix B.

To determine neo-Piagetian staging, questions from the test were divided into two groups. Group 1 includes 13 questions that were used to determine code tracing ability. Students who were able to correctly trace code 50% of the time or greater were considered to be functioning at a preoperational level, while less than 50% accuracy in code tracing indicated a sensorimotor level (Lister, 2011). Group 2 includes three questions in which students provided a written description as to the purpose of a piece of code. Students who were able to accurately state the purpose of a piece of code were considered concrete operational (Lister, 2011). Participants who scored greater than 50% on the Group 1

questions and at least 66.7% on Group 2 questions were considered to be functioning at a concrete operational level. The neo-Piagetian stage for each participant was categorized based on results in Group 1 and Group 2 questions as sensorimotor, preoperational, or concrete operational. The highest neo-Piagetian stage of formal operational was not assessed since introductory programming students are unlikely to have achieved this level (Lister, 2011).

Student Exit Survey

The Student Exit Survey included questions on computer science perceptions from the Pre-Course survey but did not repeat the demographic questions. Additional questions prompting the opinion of the participant on the delivery of the course, level of confidence and ability, and preparedness for future programming coursework were included. The survey included Likert-type responses and free responses. The free responses are qualitative in nature and are not included in the primary analysis, but were consulted to add color to the quantitative analysis in Chapter 5. A sample of the Student Exit Survey can be found in Appendix C.

Bi-Weekly Code Tracing and Coding Sprint

This was an instructor-designed assessment that was used to establish the neo-Piagetian level at which the student was performing for a given topic during the semester. The sprint contained two parts. The first part required the student to trace a computer program and to select a response of the expected output of the program. Sample problems were reviewed similar to those utilized by Teague and Lister (2014), and Kutscha (2017), and were augmented to align with topics covered in the course at a specific period. The

second portion of the sprint required the student to create a solution to a programming problem. The problems were chosen to allow the student to demonstrate knowledge from the prior two weeks of class and were designed to be completed in about 25 minutes. The sprints are included in Appendix F.

Homework

Assignments were given weekly. Each assignment consisted of at least three problems. One problem was very similar to an example presented in class. The second problem required the student to apply knowledge to a slightly different problem. The third problem required the student to combine knowledge from previous content knowledge with current content. It was expected that students would find the first problem fairly easy as it was targeted to be at the pre-operational stage. The second and third problems sought to provide opportunities for students to show a transition to concrete operational stage and to practice problem solving skills. The homework assignments are included in Appendix E.

Homework Self-Reflection Surveys

A self-reflection survey was given after each homework assignment. This survey was an instrument to help the student analyze their own performance on the assignment which is an element of cognitive apprenticeship. The survey asked for the amount of time spent on the assignment and perceived confidence level with the immediate content knowledge. The survey gathered data on self-efficacy. The instrument was piloted during a summer 2019 section of an introductory programming course. A sample of the homework self-reflection surveys is included in Appendix D.

Blackboard Adaptive Release and Metadata

Number of attempts on quizzes will be tracked for the group. Quizzes are formative and taken after a lecture on participants own time. There were 21 quizzes during the semester. Participants were allowed to take the quiz as many times as they desired with the highest scored attempt recorded in the gradebook.

Instructor Daily Diary

The instructor summarized lecture content and notable interactions with the participants. Lesson plans were written for each class meeting. The diary is included in Appendix I.

Final Exam Questions

The final exam included two portions: Post-Course Test questions and three additional questions requiring a response of a computer program. The Post-Course Test multiple-choice questions were given on paper on the last class day with a time limit of 50 minutes. These were the same as the questions on the Pre-Course Test that is included in Appendix B. The programming portion of the final was administered during final exam week with a time limit of two hours. The scores on this exam will function as a dependent variable of a continuous value. The programming portion is included as Appendix G.

DFW Rates

The number of D and F grades and the number of withdrawals were collected. This number served as a dependent variable.

Classroom Attendance

Attendance was taken at each class meeting. Attendance was a requirement toward the course grade. The course met 40 times during the semester.

Post-Course Interviews

Interviews were conducted a few weeks after the end of the semester. This qualitative data consisted of open-ended questions on the elements of the course that contributed to student success and overall thoughts regarding the format of the course. Participation in interviews from student with a variety of backgrounds and characteristics was requested and seven students agreed to participate in the individual interviews. The interviews were free flowing and directed by the students' initial responses. The interview sought to understand why the student did, or did not, prefer this course design, and what particular challenges or advantages contributed to success or lack of success. Interviews were recorded for transcription and analysis. The interview responses are qualitative in nature and are not included in the primary analysis but were consulted to add color to the quantitative analysis in Chapter 5. The questions that were asked during the interviews are included in Appendix H.

Withdrawal Interview

A request for interview via email or in-person was requested for students who withdrew from the course. The interview attempted to capture the reason the student chose to withdraw. The interview was informal and free flowing.

Data Analysis Procedures

Appropriate data analyses were conducted on collected data and are described in detail in Chapter 4.

Institutional Review Board and Permissions

Institutional Review Board approval from the University of Missouri-Kansas City and from the University of Central Missouri was obtained. Permission to conduct the study was obtained from the University of Central Missouri program administration as part of the IRB application process. Student consent for participation in the study was obtained. Participants were instructed that any data, particularly qualitative data, that the participant might consider compromising, would not be reviewed until final semester grades were submitted. Risks to participants was minimal and not greater than student participation in a normal classroom setting. Participants were able to withdraw from the study at any time without reservation or penalty.

Risks and Limitations

This study had limitations. Random selection of participants was not possible, since students were able to self-enroll in course offerings. Some variables that influenced outcomes were controlled, such as prior programming experience, prior mathematics experience, etc. A pre-test was used to establish a baseline for neo-Piagetian level and also served as the post-test at the end of the semester. Testing threat was minimal because many weeks transpired between the pre- and post- administration of the instruments. A relationship between intervention and outcome was demonstrable. Due to the quasi-experimental, one-group design, causation was not confirmable. Instrumentation was a

possible threat since grading of assignments, quizzes, and exams has potential for bias. Rubrics and training were provided to a student grader to reduce this threat. Additionally, blind grading was used. Attrition was a potential threat since this study required participation in the course. For this study, attrition was a potential threat, but also was a valuable source of data. Participants who withdrew from the course were invited to share reasons for withdrawing, and this attrition-related feedback was of value to the study. Additionally, withdrawals were generally no greater than in previous offerings of this course. The researcher deviated from the proposed methodology. A Pre-Course Self-efficacy survey was administered at the beginning of the course but was not administered at the end of the course as originally intended. Self-efficacy data was also collected, weekly, through a second instrument and provided a secondary metric for self-efficacy. External validity, or generalizability of the study is possible to other similarly situated institutions.

CHAPTER 4

ANALYSIS

This chapter presents results of the analyses used to examine the relationship between student characteristics, self-efficacy, course reflections and DFW rate and student success. The analysis proceeds from more general to individual student. First, the overall DFW rate is explored, followed by analysis of characteristics of students who completed and did not complete the course. Elements of the course that impacted student success, student characteristics, and student perceptions and confidence are then analyzed. Finally, changes in neo-Piagetian stage are explored.

RQ1

Research Question 1: How do DFW rates for the Holistic Course Delivery compare to historic DFW rates for introductory programming courses taught using Traditional Course Delivery at the institution at which the present study was conducted, and internationally?

Table 4.1 shows the DFW rates for the holistic course delivery with 13-year institutional and international averages. A total of 96 students enrolled in the introductory programming course, and 19.8% of those students did not pass the course (including withdrawals and final grades of D or F). The institutional DFW rate in the past 13 years was 29.4%, and the international average DFW rate in three studies was reported to be 33%, 32.3%, and 28% (Bennedsen & Caspersen, 2007; Watson & Li, 2014; Bennedsen &

Caspersen, 2019). Pearson's χ^2 test was conducted to evaluate whether the differences in observed rates could arise by chance. The DFW rate for the holistic course delivery was significantly lower than both the institutional and two of the international statistics.

Table 4.1
DFW Rate of Holistic Course Delivery vs. Institutional and International Averages

	N	DFW Rate	Difference	χ^2	ρ
International Mean[a]	N/A	33%	13.2%	4.48	.034*
International Mean[b]	N/A	32.3%	12.5%	4.06	.044*
International Mean[c]	N/A	28%	8.2%	1.85	.174
Institution, last 13 Academic Years	1925	29.4%	9.6%	4.11	.043*
Holistic Course Delivery	96	19.8%			

[a] Bennedsen & Caspersen, 2007
[b] Watson & Li, 2014
[c] Bennedsen & Caspersen, 2019
* ρ<.05

RQ2

Research Question 2: Does student self-efficacy with respect to Java programming and with respect to problem solving change over the course of the semester for students participating in the Holistic Course Delivery?

Student self-efficacy was self-reported in four dimensions: Perceived change in programming ability from the beginning to the end of the course, confidence in Java programming, confidence in problem solving, and preparedness for the next programming course.

In the Student Exit Survey students were asked about their programming ability at the beginning of the course and their programming ability at the end of the course (Student Exit Survey question 14). The Wilcoxon signed rank test was selected as an alternative to the t-test for comparing the paired measurements since the data for the beginning of the course were non-normal as determined by examining a histogram (shown in Figure 4.1).

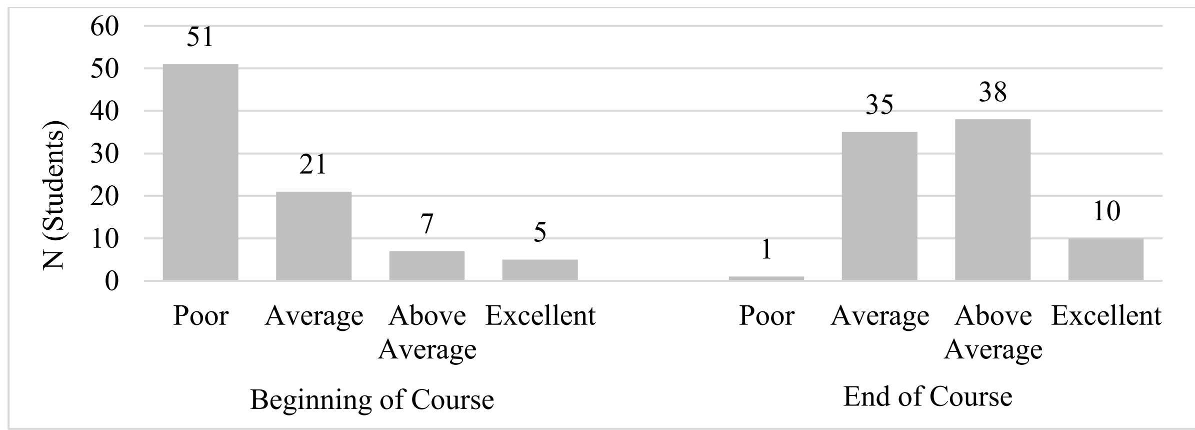

Figure 4.1
Histograms: Student Self-Reported Java Programming Confidence

Unsurprisingly, students reported significantly better programming ability at the end of the course as compared to the beginning of the semester as shown in

Table 4.2.

Table 4.2
Self-Reported Programming Ability at Start and End of Course

Self-Reported Programming Ability	N	Mean	σ	Z	ρ
At beginning of the course	84	1.60[a]	0.88	-6.87	<.001*
At the end of the course		2.68[a]	0.70		

[a] 1 = "Poor", 2 = "Average", 3="Above Average", 4="Excellent"
* <.001

Students self-reported their confidence in Java programming in the Homework Self-Reflection after completing each homework assignment (Homework Reflection Question 5). The confidence in Java programming was significantly increased over time as shown in Figure 4.2, where the prompt was, "Rate your confidence in Java programming after completing this assignment," and the responses ranged from 1 (very low) to 5 (very high).

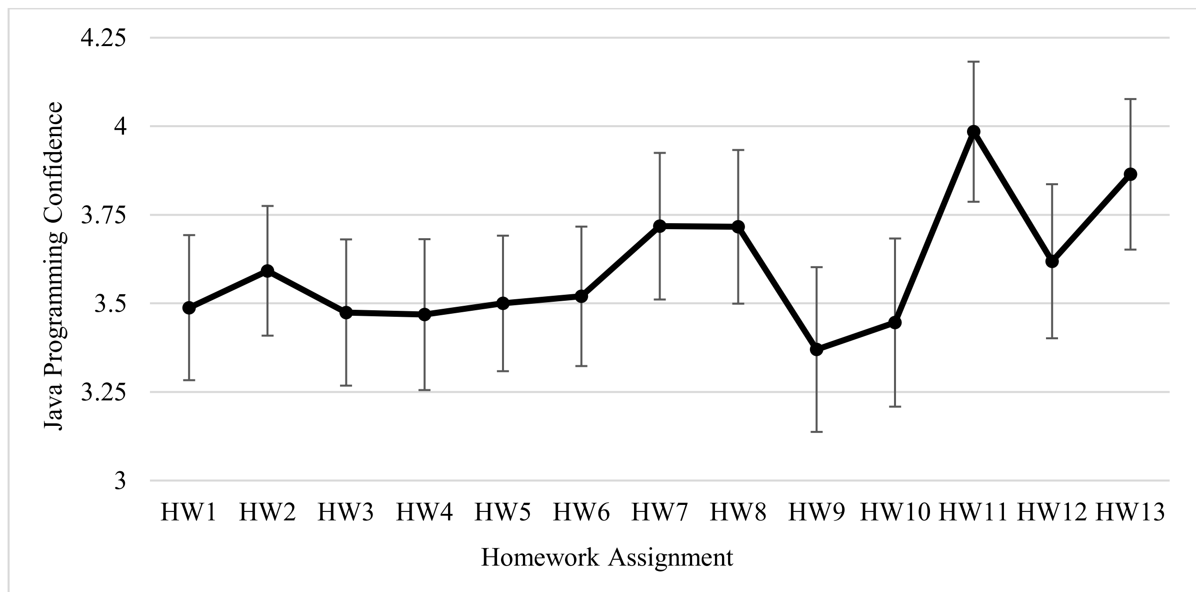

Figure 4.2
Confidence in Java Programming

Problem Solving (identifying a problem and understanding the steps needed to solve a problem) is a necessary skill to producing a working software (Morgado & Barbosa, 2012). This skill is closely related to programming, but ideally, precedes writing of code. By equipping students with techniques in developing a solution to a problem allows the student to more easily create a working software program (Loksa, et al., 2016). Students self-reported confidence in problem solving after each homework assignment. Figure 4.3 indicates a steady level of problem-solving confidence after HW2. Homework assignments progressed in complexity and inclusion of more abstraction with each subsequent assignment.

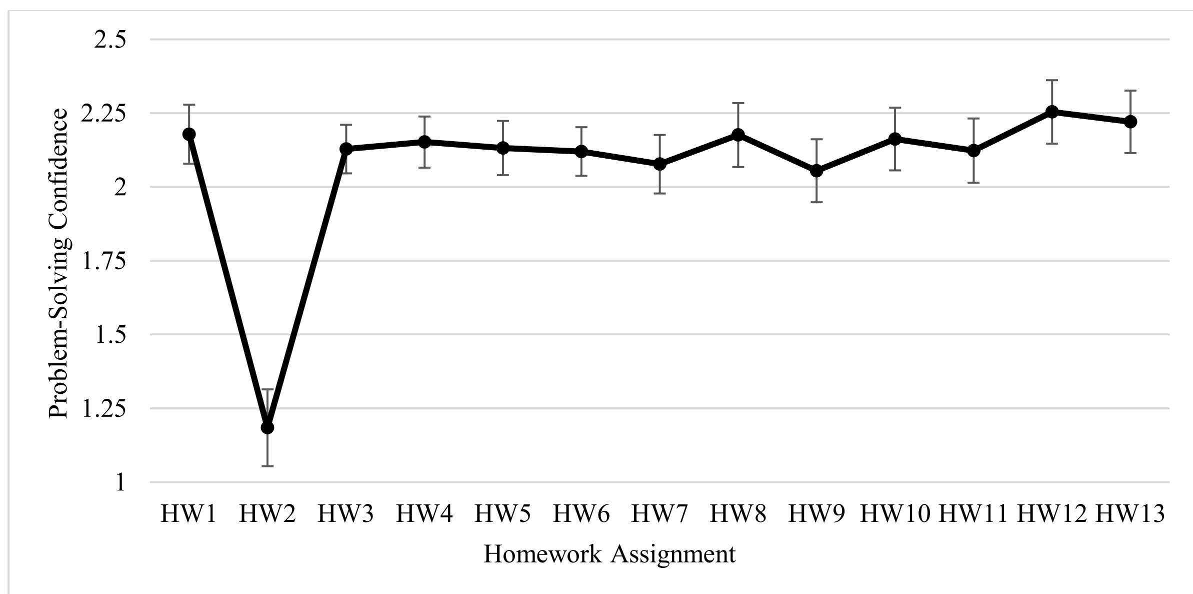

Figure 4.3
Student Confidence in Problem Solving

The Student Exit Survey asked students to rate their preparedness for code tracing and preparedness for code writing prior to each sprint (Student Exit Survey question 10). Spearman's rho correlation was selected since both variables are ranked values. It was found that students who felt better prepared for tracing also felt better prepared for writing code (r = .600, strongly positive monotonic relationship).

As another measure of confidence, students were asked to self-report whether they believed they were prepared for the follow-on course (Student Exit Survey question 12). Of the students that indicated that they intended to take the follow-on course most reported that they were "somewhat prepared" or "well prepared" to take the next course (77.6%) as shown in Table 4.3.

Table 4.3
Student Self-Reported Preparedness for the Next Programming Course

Self-Reported Preparedness	N	Percent	Cumulative
Well Prepared	32	42.1%	42.1%
Somewhat Prepared	27	35.5%	77.6%
Neutral	13	17.1%	94.7%
Somewhat Unprepared	3	3.9%	98.7%
Totally Unprepared	1	1.3%	100.0%
Total	76	100.0%	

RQ3

Research Question 3: Are there particular student characteristics associated with total points achieved in the Holistic Course Delivery?

The variable, final course grade, had a right-stacked, non-normal distribution as shown in Figure 4.4.

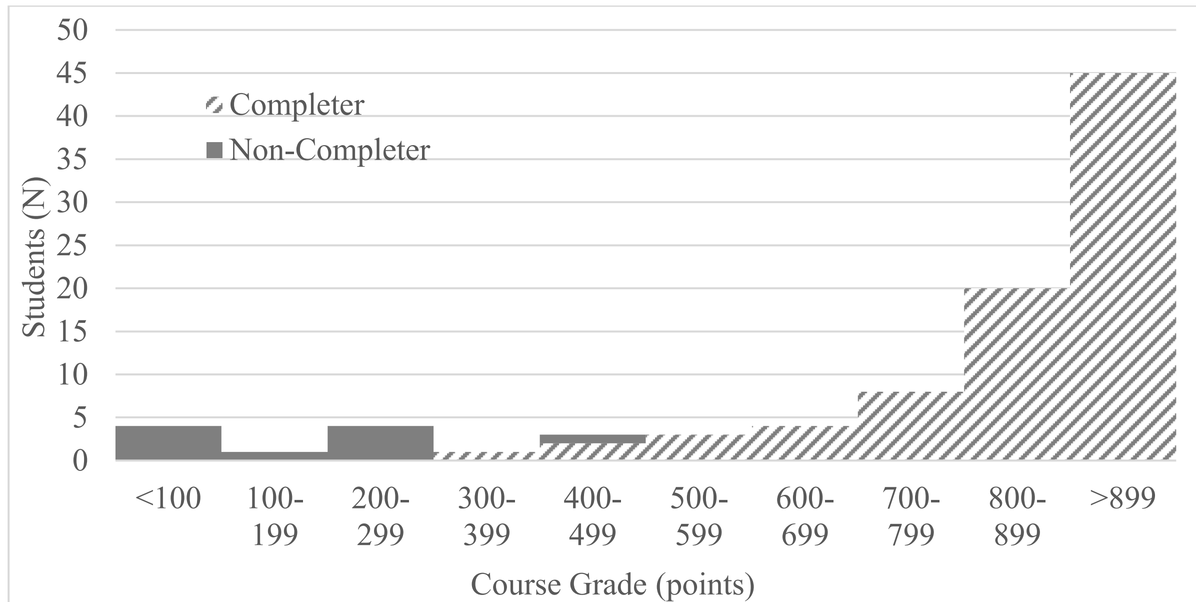

Figure 4.4
Histogram, Final Course Grade (Points out of 1000)

To determine if there was a significant relationship between the continuous variables (prior college credits, attendance, ACT Math score) and final course grade, Spearman's rho correlations were conducted. Higher attendance rate and higher ACT Math score were each correlated with higher points in the course (see Table 4.4).

Table 4.4
Student Continuous Characteristics and Final Course Grade

	N	r_s	ρ
Prior college credits	70	.094	.437
Attendance	85	.576	<.001**
ACT Math	64	.511	<.001**

** ρ<.001

The Mann-Whitney U test was selected to examine the relationship between the discrete student characteristics and final course grade. Mann-Whitney U was preferred over an independent t-test since the discrete variables (class rank, gender, race, major field of study, PELL Grant eligibility, and Prior experience) are not interval-scaled. After examining the categorical data it was observed that each categorical variable with more than two possible values possessed one dominant value. The multi-valued categorical variables were each therefore grouped into two dichotomous sets comprising the dominant value and the others (e.g., freshman and non-freshman, versus the original freshman, sophomore, junior, senior). The groupings, frequencies, and results of the Mann-Whitney U tests are shown in

Table 4.5.

Table 4.5
Student Discrete Characteristics and Final Course Grade

	N	Mean Points	σ	U	ρ
Class Rank	Total: 85				
Freshman	54	848.0	159.9	819.0	.869
Non-freshman	31	877.1	97.7		
Gender	Total: 84				
Male	66	842.9	138.1	316.5	.002*
Female	18	914.9	142.4		
Race	Total: 77				
White	66	862.7	139.4	303.0	.382
Non-White	11	839.2	139.9		
Major field of study	Total: 84				
Computing-focused	57	831.3	152.1	480.0	.006*
Non-computing-focused	27	915.4	94.5		
PELL grant	Total: 85				
Eligible	20	847.8	119.7	565.5	.381
Non-eligible	65	862.0	147.0		
Prior Experience	Total: 84				
Formal course	30	835.2	129.0	569.0	.024**
No formal course	54	871.2	147.3		

* ρ<.01; ** ρ<.05

There was no significant relationship between race, class rank, or PELL eligibility and points earned in the course. However, gender mattered, as females had higher course points than males (ρ=.002). Students who were part of the computing-focused majors of Computer Science, Software Engineering, and Cybersecurity earned fewer points than those who were not (ρ=.006). Students who had not taken a prior programming course performed better than those who had taken a prior course (ρ=.024).

In addition to calculating Spearman's rho correlations on the continuous variables (prior college credits, attendance, ACT Math score), Pearson's χ^2 test was used to evaluate whether the differences in pass/fail for each binary dependent variable were significant. This yielded the interesting observation that being a freshman completer of the course correlated with failing the course (grade of D or F), and in fact all the students who completed the course and failed were freshmen. These findings, summarized in Table 4.6, are discussed in more detail in Chapter 5.

Table 4.6
Course Pass/Fail and Class Rank for Course Completers

	Failed (N)	Passed (N)	χ^2	ρ
Class Rank	Total: 8	Total: 77		
Freshman	8	46	5.07	.024*
Non-freshman	0	31		

* ρ<.05

Of the eight freshmen who completed but failed the course, seven were in computing-focused major fields of study, and one was not. It is notable that students who were in computing-focused majors were more likely to be freshmen since the CS1 courses are generally encountered earliest in those programs. 68.7% of computing-focused majors in the course were freshmen, while only 53.6% of non-computing-focused majors were freshmen, as shown in Table 4.7.

Table 4.7
Class Rank and Major Field of Study

	Computing-focused	Not computing-focused
Class Rank	Total: 67	Total: 28
Freshman	46	15
Non-freshman	21	13

RQ4

Research Question 4: What course elements do students believe were helpful in the Holistic Course Delivery?

The Student Exit Survey collected data on student opinions concerning the value or merit of various elements of the Holistic Course Delivery, summarized in Table 4.8 to Table 4.14, below.

Table 4.8
End-of-Course Survey, Part 1

	N	Extremely Helpful	Helpful	Slightly Helpful	Not helpful
Watching the instructor live code was	83	65.1%	27.7%	7.2%	0.0%
Coding alongside the instructor during class was	83	77.1%	19.3%	3.6%	0.0%
Practice with problem solving before sprints was	83	66.3%	25.3%	8.4%	0.0%
Online Quizzes were	84	23.8%	56.0%	19.0%	1.2%
Homework problems were	83	53.0%	42.2%	3.6%	1.2%

Table 4.9
End-of-Course Survey, Part 2

	N	Helpful	Unhelpful	Don't Know
Soft deadlines for homework were	83	77.1%	2.4%	20.5%

Table 4.10
End-of-Course Survey, Part 3

	N	Often	Sometimes	Rarely	Never
How often did you turn in homework late?	83	16.9%	15.7%	44.6%	22.9%

Table 4.11
End-of-Course Survey, Part 4

	N	Too many	About right	Not enough
Number of problems on each assignment was	84	3.6%	89.3%	7.1%
Weekly frequency of assignments was	84	3.6%	89.3%	7.1%

Table 4.12
End-of-Course Survey, Part 5

	N	Too difficult	About right	Too easy
Difficulty of problems on each homework was	84	10.7%	85.7%	3.6%

Table 4.13
End-of-Course Survey, Part 6

	N	Sprints	Midterm exams	Don't care
Do you prefer having sprints or midterm exams?	84	90.5%	4.8%	4.8%

Table 4.14
End-of-Course Survey, Part 7

Reflections after homework…	N	Percent
Helped me to assess my own comprehension of subject matter	83	55.4%
Helped me assess my own performance on the assignment	83	49.4%
Were a good closure before moving on to the next assignment	83	54.2%
Were not of much use for me	83	26.5%
Other	83	3.6%

Homework Assignments

The first problem on each homework was a programming problem that closely followed an example presented in the classroom or in the textbook. The Homework Reflection for each assignment asked students to rate the difficulty level of the first homework problem (question 5). The possible responses were "Difficult to solve," "Completed with substantial effort and review of course materials," "Fairly easy and straightforward with some review of course material," "Easy with hardly any need to review course material," and "None of the above." The responses were re-categorized for analysis, with responses of "None of the above" treated as missing. The remaining four possible responses were dichotomized into "Difficult" and "Easy," with "Difficult to solve" and "Completed with substantial effort and review of course materials" as "Difficult," and "Fairly easy and straightforward with some review of course material" and "Easy with hardly any need to review course materials" as "Easy." Percentages were reported for the new categorizations and analyzed. The top three assignments that were perceived as "Difficult" include HW10, HW5, and HW6. The top three assignments considered "Easy" include HW1, HW2, and HW3. Student perceptions of assignments that were difficult or required substantial effort are visualized in Figure 4.5.

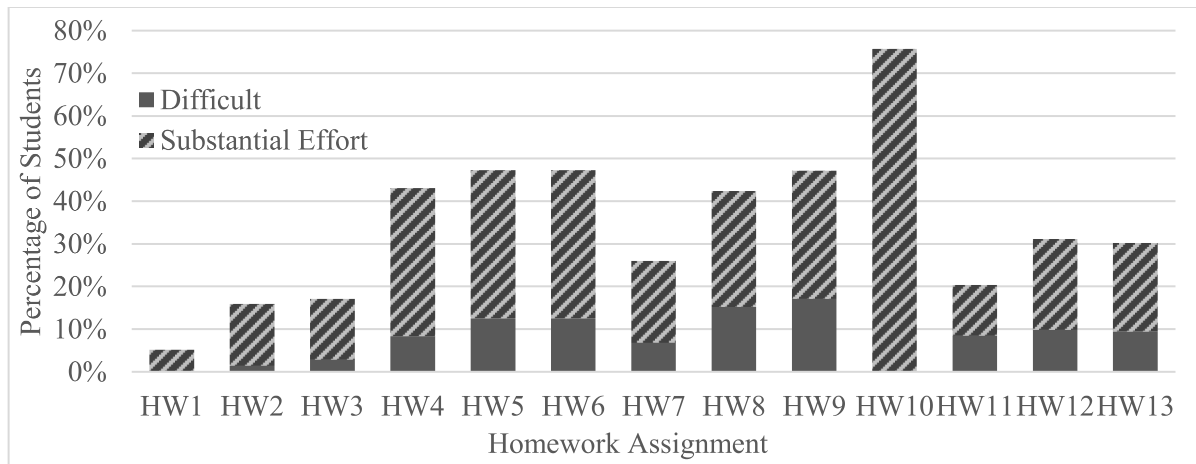

Figure 4.5
Homework First Problem Perceived as Difficult or Requiring Substantial Effort

The Student Exit Survey asked participants to rate the quantity of problems on each homework and whether the frequency of assignments was appropriate. Most students reported that the number of problems on each assignment was about right (89.2%). Most students reported that the frequency of assignments of weekly was also about right (90.4%).

Participants also self-reported the amount of time spent on each homework assignment. This information was collected through the Homework Reflection after each homework assignment (Question 3). The data were determined to be non-normal by examining the histograms which were largely left-biased. For example, the histogram for HW6 is shown in Figure 4.6.

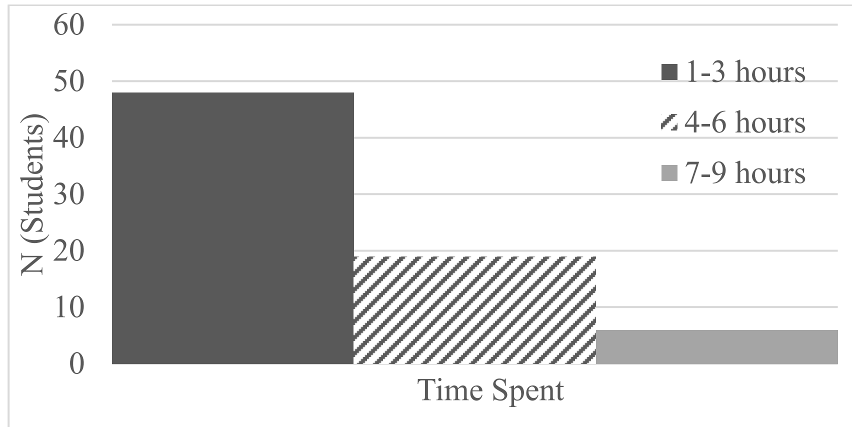

Figure 4.6
Histogram, Time Spent on HW6

Students reported that more time was spent on most assignments as the semester progressed as shown in Figure 4.7. Students reporting 1-3 hours spent on homework predominated in the early part of the semester, where a more even split between the 1-3 hours group and the 4-6 hours group is evidenced in some of the assignments in the last half of the semester. As many as three students reported spending more than nine hours on an assignment in seven of the 13 assignments. It was observed that the average number of students submitting a homework reflection decreased at the end of the semester (from an average of 73 on the first ten assignments, to an average of 60 on the last three assignments).

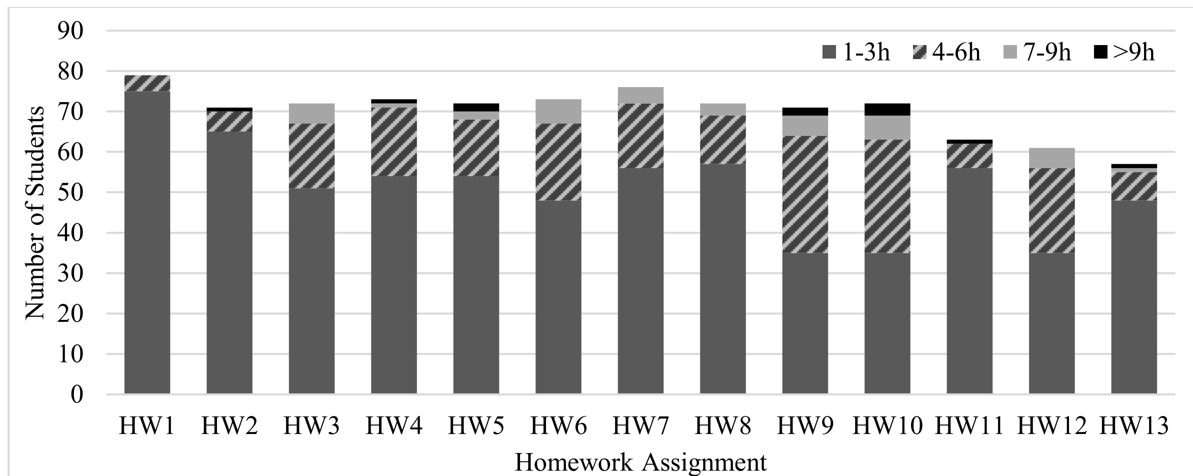

Figure 4.7
Time Spent on Homework Assignments

Formative Quizzes

Students completed formative quizzes two to three times per week that were derived from lecture meetings. Students were permitted to take a quiz as many times as they wanted with the highest scored attempt recorded in the gradebook. Since attempts were essentially unlimited, the number of quiz attempts was examined to identify if a student was randomly guessing on the quiz or making a viable attempt. When a student made four or more attempts, the effort was thought to be guessing, while one to three attempts was considered to be a knowledge-based effort. Frequencies and percentages of attempts for students were recorded for each quiz. There were 15.3% students who guessed (according to the criterion indicated above) on none of the fourteen quizzes, 15.3% students guessing on two quizzes, and 14.1% students guessing on four quizzes. Around 3.6% (N=3) students guessed on ten or more quizzes.

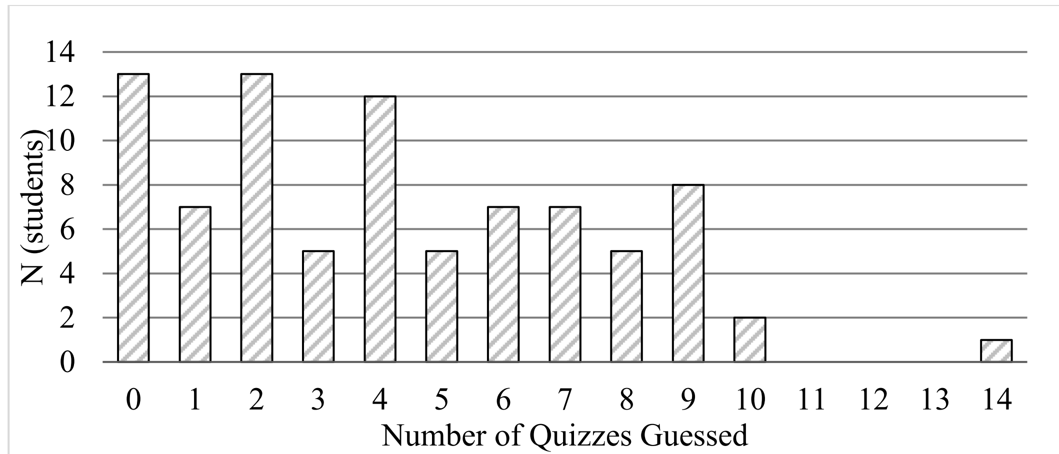

Figure 4.8
Frequency of Guessing on Quizzes

The relationship between the quiz attempts and final grade was analyzed. The number of quizzes for which a student guessed are non-normal as can be seen in Figure 4.8 (above). Due to the non-normal distribution of guessing, Spearman's rho correlation was employed. A student averaging more quiz attempts generally earned a higher final grade in the course but this positive relationship was weak (r = .279).

Code Sprints

Each of the seven Code Sprints consisted of two questions. The first question involved code tracing, in which a student examined provided code and "traced" the path through the code, determining the output that would be produced if the code had been executed on a computer. The student then selected a multiple-choice answer corresponding to the output that would have been produced. The second question involved writing a small piece of code based on knowledge content presented in the previous two weeks of instruction. The code tracing question response was scored by multiple-choice and was

therefore binary (correct/incorrect), while the programming question response was continuous. The programming question scores were not normally distributed and were right-stacked as shown (by way of example) in the histogram for HW5 in Figure 4.9.

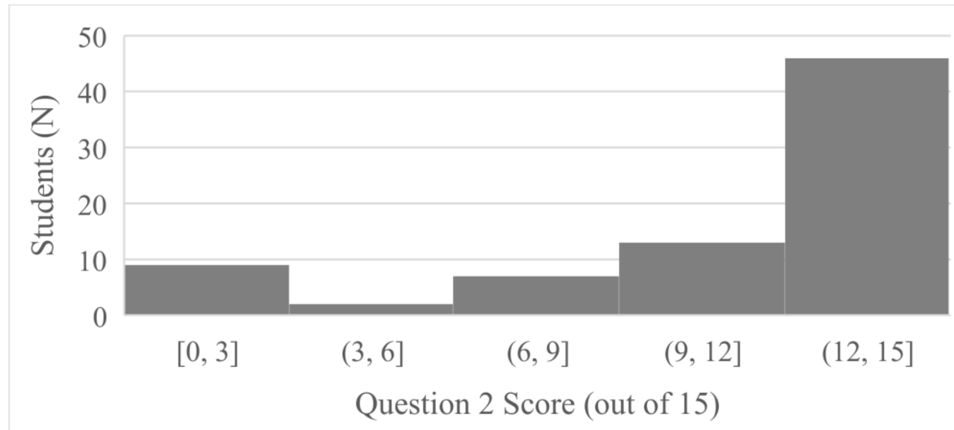

Figure 4.9
Sprint 5, Question 2 Score Histogram

Since the scores for the second question were non-normal a generalized linear model was selected, treating programming performance as an outcome. When considering the subject as a random effect, the generalized mixed linear model exhibited a better model fit than a generalized linear model (again utilizing the Akaike Information Criterion, which provides an estimate of model quality, to quantify model fitness). Thus the relationship between code tracing (question 1) and programming (question 2) was analyzed using a generalized linear mixed model with gamma regression in which the subject was a random effect, while time and code tracing performance were the fixed effects. There was no

significant relationship between code tracing and programming, though both trended lower in the second half of the course as shown in Figure 4.10.

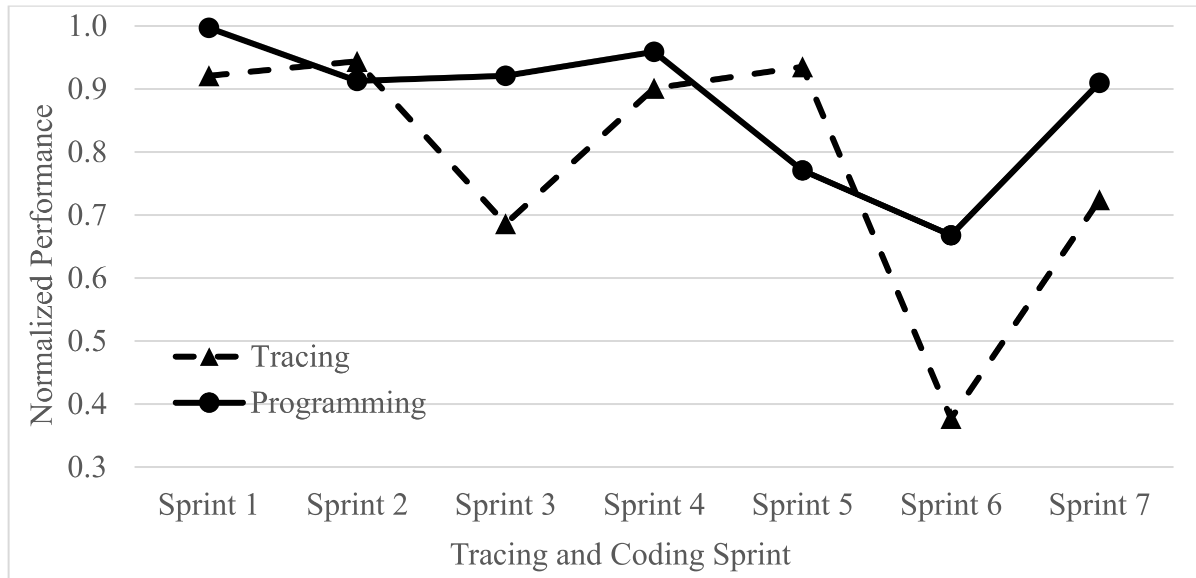

Figure 4.10
Sprint Code Tracing and Programming Performance

RQ5

Research Question 5. Does the neo-Piagetian stage of students change from the beginning to the end of the Holistic Course Delivery?

Categorized results on questions from the pre-course test were used to identify the neo-Piagetian stage for each student. The same questions were repeated on the final exam and used to establish a post course neo-Piagetian stage. A set of 13 questions from the test was used to determine if a student was potentially operating at a Sensorimotor or Preoperational level. It is thought that students who can trace code with a 50% success rate are functioning a Preoperational level, and below 50% is a sensorimotor level (Lister, 2011).

Students who could successfully complete six or more questions was categorized as Preoperational, otherwise, Sensorimotor. A set of three questions was used to ascertain if a student was operating a Concrete Operational stage. Students who were able to answer two or three questions correctly were categorized as Concrete Operational. Sample questions used to determine neo-Piagetian level are displayed in Figure 4.11 and Figure 4.12, and in more detail in the appendices. Students completed the same set of questions on the final exam as were categorized a second time. A cross-tabulation of pre- and post-test neo-Piagetian stage is given in Table 4.15. There we no students who were categorized as Sensorimotor that also could have been categorized as Concrete Operational on the Pre-Test nor the Post-Test.

Table 4.15
Cross-Tabulation of Neo-Piagetian Stage, Pre- and Post-Course

	Post Stage			
Pre Stage	Sensorimotor	Preoperational	Concrete Operational	Total
Total	1	40	41	82
Sensorimotor	1	30	32	63
Preoperational	0	9	7	16
Concrete operational	0	1	2	3

After executing the following line of code:

int f = 7 + 3;

What is the value of f?

A. 3
B. 5
C. 7
D. 10
E. None of the above

Figure 4.11
Sample Question Sensorimotor/Preoperational Staging

In one sentence, state the purpose of this code, or respond Don't know.

```
if (y1 < y2){
    t = y1;
    y1 = y2;
    y2 = t;
}

if (y2 < y3){
    t = y2;
    y2 = y3;
    y3 = t;
}

if (y1 < y2){
    t = y1;
    y1 = y2;
   y2 = t;
}
```

Figure 4.12
Sample Question Concrete Operational Staging

Marginal homogeneity assesses if a significant change in a categorical value has occurred in a population between two points in time. A marginal homogeneity test was used to compare the neo-Piagetian stage of the students comprising the population at the times of the pretest and the posttest. Most students who completed the course and took the posttest (69 of 82) advanced at least one Neo-Piagetian stage. One student regressed from Concrete Operational to Preoperational. Twelve students neither regressed nor progressed but remained at the same neo-Piagetian stage: one Sensorimotor, nine Preoperational, and two Concrete Operational. Sixty-nine students advanced at least one stage, with 32 advancing two stages, from Sensorimotor to Concrete Operational. The advancement of neo-Piagetian stage was significant, as shown in Table 4.16.

Table 4.16
Improvement in Students' Neo-Piagetian Stage from Pre- to Post-Course

	Pre-course			Post-course			
Neo-Piagetian Stage	N	Mean[a]	σ	N	Mean[a]	σ	ρ
Total	82			82			
Sensorimotor	63	1.27	0.522	1	2.49	0.527	<.001*
Preoperational	16			40			
Concrete operational	3			41			

[a] 1 = Sensorimotor, 2 = Preoperational, 3 = Concrete operational
* Marginal Homogeneity, ρ<.001

CHAPTER 5

DISCUSSION

RQ1

Research Question 1: How do DFW rates for the Holistic Course Delivery compare to historic DFW rates for introductory programming courses taught using Traditional Course Delivery at the institution at which the present study was conducted, and internationally?

This is the primary question addressed by the present study. The high DFW rate in introductory computer programming courses results in a series of consequences: poor student experience, lower than optimal retention of students in computing-centric major fields of study, and ultimately inadequate supply of computing graduates to fill chronic shortages in the field. If it the DFW rate demonstrated by the Holistic Course Delivery compares favorably to the Traditional Course Delivery then it should be considered as an option for course delivery and perhaps the subject of additional study.

Three studies quantified the DFW rate for introductory programming internationally at 33%, 32.3%, and 28% (Bennedsen & Caspersen, 2007; Watson & Li, 2014; Bennedsen & Caspersen, 2019). Bennedsen and Casperson's 2007 study comprised 63 respondents with complete survey responses who were computer science educators from around the world, although two-thirds were from the United States. Bennedsen and Caspersen repeated their 2007 study in 2017 with the same metric, methods, and sources; they drew 170 respondents, again with about two-thirds from the United States. In both studies they acknowledge that

the representative nature of the universities covered by their respondent group, and by their respondent group itself, is debatable, but is nonetheless useful as an indicator. Watson and Li pursued a different methodology, performing a systematic review of the literature on introductory computer programming instruction, extracting data, and performing a statistical analysis on the resulting dataset. Their study addressed the trajectory of DFW rates over time as well as correlative factors such as geography, programming language taught, size of the class, and "grade level" of the institution (distinguishing universities from colleges and secondary schools). Both sets of researchers demonstrated that students in small classes (30 or fewer students) pass at higher rates than their counterparts in larger classes. The class sizes included in this study were on the border of this differentiation, at 33, 32, and 31 students (31, 32, and 30 after withdrawals are taken into account).

It is notable that neither Bennedsen and Casperson, nor Watson and Li, considered pedagogy. While it is unknown how introductory programming curriculum was delivered in calculating the three historic international statistics for DFW, the Holistic Course Delivery exhibits a significant improvement over the mean of between 8.2% and 13.2%.

The introductory programming courses at the university engaged in this study have a 13-year DFW rate that falls between the values established by Caspersen and Bennedsen, and Watson and Li, at 29.4%. Based on *ad-hoc* discussions with other faculty and the researcher's personal experience, the population of the introductory programming students is similar today to the 13-year demographic norms, except that there has been a modest increase in students of color and females in introductory programming courses in

approximately the last five years. The Holistic Course Delivery again demonstrates an improvement of 9.6% over the *status-quo*.

It was interesting to examine the group of students who received a grade of D or F in the course. There were five students who received a grade of D and 11 students who received a grade of F. All but four of these 16 students were on academic probation or were dismissed from the university at the end of the semester in which they took the course engaged in this study. The 12 students dismissed or on probation performed poorly in a majority of their coursework that resulted in their grade point average dropping below, or remaining below, a 2.0. It is questionable whether any realistic intervention or adaptation of the Holistic Course Delivery could have better served this group, or if it was beyond the power of the course design and delivery to address the acute need of these students. Perhaps early identification and reporting of students to university level systems of care is the only way to assist students in general academic distress since a picture of the student across all enrolled courses is required. If the 12 students who were experiencing broad academic issues were removed from the calculation of DFW rate, a realistic lower bound for DFW rate that could be reasonably attributed to the nature of the course and its content would be approximately 7%.

Bennedsen and Caspersen (2019) indicate that a 28% DFW rate is acceptable because it is lower than College Algebra DFW rates. University administrators watching program retention rates might reasonably disagree. Watson and Li similarly state that their calculated rate of 32.3% is not alarmingly high. They acknowledge, however, that improving the pass rate would improve the reputation held by introductory programming

courses and be less of a deterrent for students considering enrolling in them. Reducing the DFW rate through better course design and delivery would serve to improve the reputation of the courses and programs, improve program retention rates, and offer educators a method more likely to produce students with a firm foundation, high confidence, and strong desire to continue computing-focused studies.

RQ2

Research Question 2: Does student self-efficacy with respect to Java programming and with respect to problem solving change over the course of the semester for students participating in the Holistic Course Delivery?

This question was addressed by the present study because it has been shown that self-efficacy positively correlates to success in an introductory computer science course (Wilson and Shrock, 2001).

Self-efficacy is "people's judgments of their capabilities to organize and execute courses of action required to attain designated types of performance" (Bandura, 1986, p. 391). The ability to create working programs is intimately tied to self-efficacy. Course performance and greater self-efficacy were associated with "a well-developed and accurate mental model," and, "performance attainments on self-efficacy indicates that students need to incrementally build up a history of success at increasingly difficult tasks" (Ramalingam, LaBelle, & Wiedenbeck, 2004, p. 174). "Therefore monitoring students' level of self-efficacy is a process that is critical and necessary for educators" (Kallia & Sentence, 2019, p. 753).

It has been demonstrated that instructional design can assist in reducing emotional load and contribute to increased student self-efficacy (Kinnunen & Simon, 2010), therefore a key consideration of the Holistic Course Delivery was to architect a course content and delivery that promotes self-efficacy. While other factors are important (e.g., classroom environment), the backbone of holistic self-efficacy in the Holistic Course Delivery was exposing students to content in an order and progression carefully designed with self-efficacy as a primary goal so that students more easily establish and build upon early successes. This included code tracing examples presented in the classroom, live coding problems, and careful selection of homework problems. These were selected with a view toward building self-efficacy with progressive increases in level of challenge (further explained below).

Code tracing involves identifying the output a computer program will produce (Kumar, 2015). Code tracing is regarded as one of the first skills a student can demonstrate and informs programming (Ramalingam, LaBelle, & Wiedenbeck, 2004; Kumar, 2015). Students traced code with the instructor each lecture in the Holistic Course Delivery. It is likely that tracing code in the classroom helped students form strong mental models and reinforced syntactical knowledge.

The first programs chosen for code tracing in the classroom were simple examples of the current topic under discussion. Subsequent programming problems were more complex as students were able to absorb more nuanced and abstract elements of the topic under discussion and integrate those elements with previously held knowledge. For example, a program for tracing loops (a programming construct) for the first time would consist only of

a loop. A second example, assuming the class responded well to the first example, would involve tracing a loop which contained a method call (another programming construct). The concept of methods in Java would be, by that point in the course, fairly comfortable to the student as previously held knowledge. The combination of new and old would be the natural next progression in complexity and sophistication. This pattern of progression increased the likelihood and frequency of successes with concomitant increases in self-efficacy.

Homework problems followed a similar progression. The first problem on an assignment was parallel to an example worked together in the classroom. The objective was for the student to have an independent, initial success to build self-confidence for the subsequent work. The second problem on the assignment required the student to apply the same knowledge but to a problem not addressed previously in the classroom—a natural progression in complexity approaching a more real-world scenario, but still incremental with respect to the first problem and able to be achieved with the application of only moderate effort. The third problem on a homework activity required the student to address the current topic at a moderate level as in the second problem but combine it with and build upon previously mastered knowledge and concepts. While the first and second problems were designed to build confidence, the third problem's purpose was to stretch and exercise the student's abilities while still respecting the need to moderate cognitive load. In this way the homework assignments, like the code tracing work, were incremental and cumulative with the penultimate goal of increasing self-efficacy, and the ultimate goal of eliciting stronger student performances. "Students undertake activities that they believe they can succeed in and avoid activities they believe will exceed their abilities" (Kinnunen & Simon, 2011).

Two instruments provided insight into student self-efficacy: The Homework Reflections, administered after each homework assignment was due (13 times during the course), and the Student Exit Survey. The homework reflections captured a snapshot of student's feelings of self-efficacy with respect to Java Programming and with respect to problem solving. Self confidence in connection with Java Programming generally increased over the course of the semester while confidence in problem solving remained constant.

It is speculated that the Holistic Course Delivery's careful content progression in homework assignments, live coding and code tracing, and formative quizzes, was instrumental in building confidence in Java Programming ability. Through hands-on experience creating successful solutions in the classroom, reinforcement through structured and progressive homework assignments that overlapped and extended the Live Coding examples, and quizzes that functioned as study aids, the students were challenged, but not overwhelmed or frustrated with overly complicated and obtuse problem sets.

The problem solving confidence remained somewhat static through the semester, excepting a substantial dip at HW2. The second homework assignment was the first requiring students to create their own, novel solution to a programming problem and to document it with original pseudocode. That problem solving confidence decreased in the face of this increased challenge is unsurprising on reflection. Likewise, the level nature of problem solving confidence apart from HW2, while initially unexpected, fits a simple explanation: Early in the semester students may have experienced a failure of imagination and overestimated their problem solving ability. Nonetheless, as problems became more complex and required accumulation and application of more and more substantial

knowledge elements, problem solving confidence remained constant. In other words, the students were adequately prepared for success at each incremental step.

The Student Exit Survey assessed self-efficacy with respect to readiness for the next programming course and self-reported change in programming ability from the beginning to the end of the semester. More than three-fourths of the students who were intending to take the next programming course indicated they were "somewhat prepared" or "well prepared" for the next course and a significant increase in programming ability was reported. The Holistic Course Delivery is believed to be a viable model for introductory programming course delivery.

RQ3

Research Question 3: Are there particular student characteristics associated with total points achieved in the Holistic Course Delivery?

This secondary research question was included in the present study to better understand the student population and the relationship between characteristics and performance in the Holistic Course Delivery. Several characteristics are compared to the findings of prior studies.

Among the results, particularly interesting were the relationships between student performance and these characteristics: ACT Math score, attendance, gender, major area of study, and prior programming courses.

Higher ACT Math score was moderately correlated with higher total points in the course. This reflects previous findings in the literature that indicate mathematics ability has a positive relationship with success in computer programming since "mathematics and

programming involve the ability to understand abstract concepts in solving problems" (Owolabi, Olanipekun, & Iwerima, 2014, p. 112).

Higher course attendance was also moderately correlated with higher total points in the course. A majority of students in the course were freshmen in their first semester of college (53.1%). Incoming freshmen often have difficulty adjusting to the demands of collegiate studentship (Beaubouef & Mason, 2005), and showing up to class "offers students the opportunity to participate in class exercises and discussions, and to fully engage with the material–and that is half the battle" (Rolka & Remshagen, 2015, p. 14). The Holistic Course Delivery also expressly rewards attendance with significant hands-on practice that students found extremely valuable and an important factor in their course performance.

Students overwhelmingly indicated (96.4%) that Live Coding was Extremely helpful or Helpful and overwhelmingly indicated that Live Coding was one of the most-liked elements of the course in the qualitative Student Exit Survey questions. This finding is discussed in more detail in connection with RQ4 (below). Course delivery for introductory programming at the institution involved in this study has traditionally followed a conventional lecture format structured around content-dense PowerPoint presentations provided by the textbook publisher, and like many universities, does not have a required lab component that would afford an opportunity for hands-on practice. The PowerPoint-centric format rarely if ever left time for watching the instructor code or for hands-on live coding. Additionally, students are less engaged and interactive in lectures that rely heavily on PowerPoint-type presentation (Abernethy, 2012; Ogeyik, 2017). Students commented in the individual interviews that they, "appreciated that the class was not 'death by PowerPoint'".

Females, though a minority in the course (~20%), generally performed better in the course than males. This result is somewhat different from those found in the existing literature that suggest women generally perform similarly to males in computer programming courses (Lishinski, Yadav, Good, & Endody, 2016; Pillay & Jugoo, 2005; Akinola, 2016). Prior research has shown that underrepresented minorities perform better when they have a mentor with whom they can identify who establishes an environment of belonging (Herrmann, et al., 2016; Cotner, Ballen, Brooks, & Moore, 2011), and that females in particular, paired with a strong female mentor, are able to overcome stereotypical gender barriers (Lockwood, 2006). In this course females were an underrepresented minority and the presence of a female instructor might have positively impacted the performance of the females in the course. A recent survey of 181 Ph.D.-granting Computer Science departments in North America, while not exhaustive, found that 22.6% of Computer Science instructors are female (Zweben & Bizot, 2019). Interestingly, this is closely matched to the percentage of female students engaged in the present study (21.1%).

It has also been demonstrated that for students just entering college, females demonstrate a greater level of academic motivation and psychosocial abilities (comprising academic discipline, commitment to college, communication skills, general determination, goal striving, and study skills) than their male counterparts (Ndum, Allen, Way, & Casillas, 2018). This might have been a factor in females outperforming males in the present study, given that almost two-thirds of the students enrolled in the studied sections were incoming freshmen.

It was surprising that students whose majors were not computing-related performed better than students in computing-related major fields of study (Computer Science, Cybersecurity, and Software Engineering). The course delivered in this study is often the first course for freshmen in the computing-centric majors, while it is a later, required course for some non-computing majors. Consequently, the non-computing major group were generally more experienced students: About 69% of the students in computing-centric majors were freshmen, while about 54% of the students in non-computing majors were freshmen. That freshmen often declare a major without sufficient experience to adequately determine the fit and also tend to have less preparation in studentship than upperclassmen (Beggs, Bantham, & Taylor, 2008; Bolting, Schneider, & Muhling, 2019) may have been a factor in the surprising result. It is also worth noting that all the students who completed but failed the course (letter grade of D or F) were freshmen (N=15) and 80% of those (12 of 15) were placed on probation or dismissed from the university at the end of fall 2019, suggesting performance issues in all their coursework, not just in this course.

It is also worthy of mention that students who withdrew from the course did so because of a change of interest and major field of study. There were only three formal withdrawals from the course. One student, a sophomore, met with the instructor and indicated they had decided a computing-centric major was not suited to their interests and subsequently changed their major to a non-computing-related field of study. Another student who withdrew was a freshman who decided to change their major to a non-computing-related field early in the semester because programming was perceived to be uninteresting. The third student was also a freshman in their first semester of college and indicated that

issues of studentship led to withdrawal from the course—specifically, involvement in an extra-curricular group that led to time management issues. The student withdrew from the course but continued in the major and re-enrolled in the course in a subsequent semester.

Another surprising finding was that students without prior programming coursework performed significantly better than students who had undertaken prior programming coursework. Despite the common-sense, intuitive feeling that previous experience would be a positive indicator of success, the prior literature on this question is mixed (Ventura & Ramamurthy, 2004; Veerasamy, D'Souza, Linden, and Laakso, 2018): Ventura and Ramamurthy indicated that students with prior programing performed no better than those without prior experience, and Veerasamy, *et al.* indicated performance was better for students with prior programming experience.

This finding, and the mixed results in the literature, could be a function of the quality of prior course experience. A poor-quality prior experience of computer programming might give an inadequate impression of the difficulty of introductory programming or might develop and reinforce poor student habits. One freshman with prior high school programming coursework stated that their high school instructor never explained *why* computing topics were being covered and frequently emphasized mimicking—a "just do it this way" teaching pattern—over conceptual learning. That student did quickly grasp material in the collegiate course engaged in this study but had many misconceptions that had been formed in the high school course. The importance of quality of early programming exposure is supported by prior research that points to quality issues in pre-collegiate programming coursework. One study concluded that there is no statistically significant

advantage to students who had a high school programming course prior to coming to college in terms of their comprehension of what computer science is, and what computer scientists do; and many students were unable even to state the programming language used in the course (Copus, 2015).

RQ4

Research Question 4: What course elements do students believe were helpful in the Holistic Course Delivery?

This question was included in the present study to provide a feedback mechanism for measuring and improving the Holistic Course Delivery and for validating the principle that a holistic combination of insights from the four foundational learning theories (Neo-Piagetian, Cognitive Apprenticeship, Cognitive Load, Self-Efficacy) produces a strong course design. The course elements were intentionally implemented to create a course delivery that is informed by the neo-Piagetian, cognitive load, cognitive apprenticeship, and self-efficacy theories as detailed in CHAPTER 3, Table 3.2.

Cognitive apprenticeship is a model of instruction that "works to make thinking visible" (Collins, et al., 1989, p. 1). There are six techniques that define cognitive apprenticeship: modeling, coaching, scaffolding, articulation, reflection, and exploration. Each of the pedagogical components of the Holistic Course Delivery aligned with at least one of the six techniques. The course components that students experienced directly and could evaluate were: watching the instructor code, participating in Live Coding, pseudocoding and breaking a problem into parts, formative quizzes, homework problems, and self-reflections after homework submissions (Refer to Table 3.2, p. 48).

Students were asked on the Student Exit Survey to report on how important or helpful the various course elements were to them. A vast majority of students rated four elements as Extremely helpful or Helpful, Watching the instructor live code (92.8%), Live coding alongside the instructor (96.4%), Practice with problem solving (91.6%), and Homework problems (95.2%). Students clearly believe that these components were helpful. Online Quizzes, which were brief formative quizzes over lecture content which students could take an unlimited number of times, were perceived to be Extremely helpful or Helpful by a smaller majority of students (79.8%).

Modeling

Watching the instructor code is modeling. Students watched the instructor code or trace through a program while the instructor verbalized the thinking process. The steps used to solve the problem at hand, syntactical elements that were relevant, and potential pitfalls and errors that could have occurred were introduced in this way.

Live Coding is also modeling but includes coaching as an additional component of Cognitive Apprenticeship. Modeling and coaching have been described as a form of storytelling in which the real-world experience of the coach is “especially powerful as a source of guidance” (Bareiss & Radley, 2010, p. 163).

When asked what course elements students liked most in the open ended, optional, question in the Student Exit Survey (Question 17) and in Individual Interviews, these two course elements (watching the instructor code and live coding) were overwhelmingly liked. Student Exit Survey responses were frequently similar to, “It was hands-on and the instructor made sure nobody fell behind,” “Whenever I can do something, that’s the way

that I remember it best…doing a lot of the coding in class,” “[You] showed us what could go wrong in a multitude of different steps,” and “[You were] learning without thinking you were learning.” The free responses by the students give insight to why these two components were rated so highly. When students were given an opportunity to suggest what they would change about the course (Student Exit Survey, Question 18) there was not a single comment suggesting a change with respect to these course elements.

Classroom Environment

An objective of the Holistic Course Delivery was to create a classroom environment with a positive climate for the students: psychologically comfortable, collaborative, and uninhibited. The optimal classroom environment was thought to embody open, bilateral communication between the instructor and each student and encourage structured communication between students at appropriate times. Ultimately these would provide support and scaffolding from both the instructor and peers.

Comfort level in the computer science classroom has been shown to be a predictor of success in a course (Wilson & Shrock, 2001). A positive environment for students to feel safe admitting errors in their programs and describing the challenges they are experiencing was deemed a necessary starting point for effective coaching, correction of errors, and resolution of misconceptions, since it has been demonstrated that “comfort level in the computer science class was the best predictor of success in [the] course” (Wilson & Shrock, 2001, p.187). Some students reported in the Individual Interviews and the open-ended questions in the Student Exit Survey that they initially felt outclassed by some of their fellow students, who they presumed to have significantly more programming experience.

However, they uniformly indicated that they felt comfortable asking questions and interacting with the instructor and peers in the classroom.

For cognitive apprenticeship to be effective, communication between the teacher and learner must exist (Collins, Brown, & Holum, 1991). The Holistic Course Design sought to make communication comfortable and to ensure broad participation. Students responded often in the Student Exit Survey and Individual Interviews that they felt comfortable asking questions and were not intimidated, e.g., “The professor knew that not everyone would know how to write code” and that the course was “suited for many different kinds of students.”

Creating an environment in which the student does not feel isolated or inhibited and is shown respect enables students with a low self-efficacy or high feeling of intimidation to master concepts (Sankar, Gilmartin, & Sobel, 2015). The instructor (and sometimes a neighboring student) would assist or provide scaffolding to those experiencing errors, and the broad participation of the students in this activity was viewed by the instructor of evidence that a healthy classroom environment, conducive to effective coaching and multilateral communication, had been produced.

Code Sprints

Test anxiety can prevent a student from performing at their best (Deloatch, Bailey, & Kirlik, 2016; Adesola, Li, & Liu, 2019). Early feedback and specific feedback were shown to be important to students so that the student could change in their thinking and develop missing knowledge in time for it to still be helpful (Poulos & Mahony, 2008). Code Sprints were utilized in the course to reduce the anxiety many students experience prior to and

during high-stakes midterms. Code Sprints were conducted about every second Friday with detailed feedback and scores returned on the following Monday. Students overwhelmingly indicated that they preferred the Sprints over midterm exams (90.5%). In the Individual Interviews students cited decreased anxiety associated with multiple, lower-stakes tests and the rapid feedback loop—they had an objective measure of their mastery of the material every two weeks, versus perhaps once or twice in a semester with high-stakes midterm exam(s).

Tracing code before writing code, as practiced in the Code Sprints in this study, is analogous to reading a language prior to writing it. A surprising result was that there was no significant correlation demonstrated between the score on the code tracing and code writing components of the Code Sprints. This is counterintuitive, and various studies have previously indicated that code tracing is prerequisite and/or supplemental to code writing (Lister, et al. 2004; Hertz & Jump, 2013; Kumar, 2015; Zavala & Mendoza, 2016; Griffin, 2016). An explanation may lie in the method employed: In the present study the code tracing questions in the Code Sprints were multiple choice, while the code writing portion was graded by the instructor with a likelihood of partial credit. This disparity in metrics may have rendered the instrument less than ideal in correlating the two. While tracing and programming did not appear be related, code tracing develops an awareness of syntax and programming constructs (Kumar, 2013). Tracing code prior to starting to write code will be retained in future offerings of the course.

Problem Solving and Pseudocoding

Problem solving and pseudocoding is a form of articulation involving verbalization or demonstration of knowledge and thinking processes (McLellan, 1994). Prior to starting a Live Coding problem and in special sessions on Code Sprint days students would be presented with a problem and then given a moment to think about the steps needed to solve the problem. A list of steps would be compiled collaboratively as a class. Sometime students would collectively produce a sequence of steps to solve the problem, and other times the instructor would provide scaffolding to assist, leading the class in a minimal way so as to ensure the class collectively arrived at a viable solution. The steps as verbalized by the class were written on the whiteboard or typed as comments in a nascent Java program.

One intent and benefit of this problem solving and pseudocoding activity was that students began to recognize templates, patterns of problem types that recur from one problem to another. One of the first templates encountered in the Holistic Course Design may be characterized as, "read data, manipulate data, display results." Once students were able to internalize this pattern the activity of listing steps needed to re-use and re-apply the pattern to similar problems was anecdotally observed to be generally faster and easier. In this way problem solving skills were abstracted away prior to coding in Java as broadly recommended in the literature (Xie et al., 2019).

Pseudocoding was used throughout the entire semester as the preferred method when starting a problem. While the instructor observed that student pseudocoding skill was modest in the beginning, students were more at ease and developed stronger skills with repetition. This is reflected by the trajectory of problem solving confidence as reported by

students: On the Homework Reflection for HW2 there was a significant decrease in problem solving confidence that resolved to a consistently higher level by HW4 (see Figure 4.3, p. 61).

Formative Quizzes

Scaffolding is a component of Cognitive Apprenticeship that supports students in learning (Collins, et al., 1987). In addition to examples cited in conjunction with Live Coding and Problem Solving, the Formative Quizzes incorporated significant scaffolding. The quizzes were delivered online and Students were permitted to attempt the quizzes any number of times, with their highest score attempt recorded as the grade for that quiz. Questions that were answered incorrectly were programmed with feedback that directed the learner to the textbook or to particular slides presented during lecture that were relevant to the missed question.

The scaffolding integrated into the Formative Quizzes formed an adjunct to the instructor since correction of misconceptions was immediate and direct, permitting rapid iteration in the development of the student's mental model around a concept. It may be speculated that this had some relationship to a surprising finding around the Formative Quizzes. It is reasonable to expect that students who often guessed on quizzes would earn a lower final course score since guessing might indicate a lack of engagement of the course material during lectures or in independent study, but the data do not support this expectation. The instructor intended the quizzes to be a measure of student attentiveness and learning during lectures, but it is possible that students who took the quizzes many times might have utilized quizzes as a study opportunity, utilizing the scaffolding as a proxy for time under

the tutelage of the instructor. While the formative quizzes were likely useful as a learning implement, students did not like them as much as the hands-on Live Coding and Watching the Instructor Code.

Homework Problems

Homework problems also provided scaffolding through hints on course elements to utilize in the solution and incorporated the Cognitive Apprenticeship component of exploration. Exploration allows the student to "transfer the skill independently when faced with a novel situation" (Collins, et al., 1991). Each assignment had at least one problem that required the student solve a previously unexplored problem.

It is interesting that time to complete each homework assignment was fairly consistent (1-3 hours) from assignment to assignment even though each homework introduced a new topic while requiring the application of prior knowledge. This may be an indicator that students adequately absorbed and mastered prior knowledge, were able to focus time spent on new knowledge, and did not usually feel overwhelmed. Excepting HW10, the time to complete the assignments did not generally increase as the semester progressed and as problems began to require the incorporation of more prior knowledge.

HW10 was different in that the entire assignment was a single problem that built on prior knowledge of loops and methods, two programming constructs that are somewhat advanced for the introductory nature of the course. This assignment required the student to model a restaurant order system and calculate the cost of an order, including gratuity. It was a departure from the usual textbook-type problems of previous assignments and may have been perceived as a more substantial project than a typical homework assignment. Students

provided feedback on the Homework Reflection regarding the challenges they had with HW10 with comments such as, “methods are hard to understand,” “organizing my code was challenging,” and “using a while loop to call methods [was difficult].” The increase in perceived difficulty makes it clear that the class was not fully prepared to apply so much prior knowledge to a new and larger problem. At the request of the class, additional scaffolding (in the form of a more substantial, parallel, worked example) was introduced during class time just prior to the due date for HW10. Most of the student were ultimately successful.

Soft homework deadlines were implemented to improve classroom climate and enhance student success. Students were encouraged to submit work by due dates, but late work was accepted with no deduction or a minor deduction to encourage students who had fallen behind or had been impacted by life issues to complete the assignment. This was deemed especially important in an introductory programming course since new topics largely build upon the material in previously covered topics and assignments. The ultimate objective of Cognitive Apprenticeship is that the student be able to work independently of the instructor and produce correct solutions on their own—that the student effectively become the expert (Collins, et al., 1991). Allowing occasional late submissions reduced barriers to completing and submitting independent work, facilitating feedback and providing needed scaffolding for subsequent assignments. Students reported that they liked being able to submit work late (64 of 83, or 77.1%). It is interesting to observe that 22.9% of students reported never having taken advantage of the policy; if they are removed from consideration, 100% of the remaining students indicated the policy was Helpful.

Reflection, or the action of students pausing to look back and analyze their performance with a desire for understanding and improvement, is a component of Cognitive Apprenticeship. Students completed a Homework Reflection after each assignment (refer to Appendix D). The Homework Reflection was survey-like with an open-ended question to capture the “triumphs” or “challenges” of the week. Students believed that this activity helped them to assess their own comprehension of subject matter (55.4%), their performance on the assignment (49.4%), and was a good closure before moving on to the next assignment (54.2%). The student reflection was often paralleled by a reflection on the part of the instructor: At milestone moments in the lecture period, the instructor would pause and bring awareness to the class of how far the students had advanced in skill and knowledge during this stretch of the course. Anecdotally, this was very well received by students with palpable energy and visible smiles.

RQ5

Research Question 5. Does the neo-Piagetian stage of students change from the beginning to the end of the Holistic Course Delivery?

It has been shown that students in introductory computer science courses progress through the early neo-Piagetian stages (Lister, 2011). Measuring this progression in a CS1 course may provide stronger validation of the course design than examining final student grades.

The analysis of movement between neo-Piagetian operational levels shows that the majority of students advanced one or two levels. Little research has been published that

quantifies movement of students between neo-Piagetian levels during a semester-long introductory programming course. There are a few case studies of individual students that provide insight into how students think and what enables them to advance to a higher stage (Teague, 2015). Problem solving performance was examined at preoperational and concrete operational levels (Kozuh, Krajnc, Hadjileontiadis, & Debevc, 2018), but studies have not been conducted that analyze the semester-long progression of neo-Piagetian stages for introductory programming students. This study demonstrates that an introductory student can be staged and that progress in neo-Piagetian stages can be tracked. Educators desire that every student reach the formal operational level. However, a novice will "…tend to move to formal operational reasoning via the concrete operational level" (Lister, 2011). Therefore, it is not unexpected that a large number of students exited the course at the Preoperational or Concrete Operational levels. It is also likely that if these students continue programming coursework they will continue to grow until they reach the Functional Operational level (Lister, 2011).

The findings of this research support the idea that programming can be learned. It is not an innate ability and novice students must first be taught at a level congruent with their stage before they can grow to a higher level. All but one student advanced to a higher neo-Piagetian stage than where they began the course, demonstrating that the Holistic Course Delivery is an effective method to deliver an introductory programming course. An expert on neo-Piagetian levels with reference to novice programmers concluded that the objective of a first semester programming course should be to get the "bulk of our students to the point where they can consistently reason at the concrete operational level" (Lister, 2011, p.

17)—A noble objective, accomplished with half (41) of the students who completed the Holistic Course Delivery in connection with this study.

Conclusion

Nascent programmers are built, not born. “We don’t know the limits of good teaching. There is research evidence that we can use teaching to reduce differences that have been chalked up to genetics” (Guzdial, 2015, p. 86). The present study sought to build beginning Java programmers in a manner more effective than the Traditional Course Delivery by arranging the content and delivery in a novel, holistic manner informed by four fundamental theories: neo-Piagetian, Cognitive Apprenticeship, Cognitive Load, and Self-efficacy.

This study engaged an introductory programming course with three sections and 96 students initially enrolled. 77 students completed the course with a grade of C or better, and 19 students withdrew from the course or received a grade of D or F (collectively, DFW). This represents a DFW rate of 19.8%, versus 29.4% historically for the institution, and 28% to 33% historically, worldwide.

The all-inclusive nature of the Holistic Course Design is believed to be essential to its success. The theories were operationalized in several ways: Live coding, positive classroom environment, carefully considered content progression, and low-stakes assessments are not typically found in a traditional delivery of introductory programming. Each element was included in the course design to assist students in their learning. Students responded favorably to this design with improved DFW rate and in parting comments on the course. For institutions that allow smaller class sizes of about thirty or fewer students, the

Holistic Course Design is a design for an introductory programming course that could be effective in reducing DFW rate.

In the current context of increasing university budget pressures and more attention than ever on student retention, any mechanism that significantly reduces the historically high DFW rate in introductory programming courses, while building confident students who feel adequately prepared to advance to the next course in the sequence, will be most welcome. While the quasi-experimental design of this study cannot demonstrate causation, the clear relationship between the Holistic Course Design and student DFW rates suggests that the Holistic Course Delivery may be such a mechanism.

Future Research

Among various possibilities, two directions for future research rose to the top as promising areas that could produce results that are measurable and, more importantly, actionable in improving CS1 DFW rates.

A follow-up study on same-gender role models and success for female students may shed additional light on the significantly better performance of female students in the present study, and may also pave the way for subsequent research on the value of role model similitude in connection with other underrepresented minorities in the CS1 classroom.

Another area that a subsequent longitudinal study might address is the progression of neo-Piagetian levels in the first several CS-sequence classes. It would be valuable to understand how quickly students can progress from the Concrete Operational level to the Formal Operational level (not generally reached in CS1) and what factors influence the rate

and penetration of the progression. Does the Holistic Course Delivery have application in the second and subsequent programming-sequence courses?

A third area of interest arose from the counterintuitive connection between student prior, formal programming class experience and success in collegiate introductory programming. What are the common misconceptions? How can they be intentionally and systematically unwound to "clear the field" for subsequent development on a sound foundation?

Fourth, as the results of this study were being analyzed and concluded, the COVID-19 pandemic arose, sending students at most universities home to complete their courses online and at distance. A future study could address how the Holistic Course Delivery might be adapted and applied to an online course delivery.

Fifth, there is great variety in the languages taught in introductory programming courses. Although the literature generally indicates that language of choice, early/late objects, and other linguistic features have not had a significant effect on DFW rates, it would be interesting to examine if the Holistic Course Delivery's effect on DFW rate is orthogonal to language choice.

Finally, a quasi-experimental design in the educational setting can indicate a promising direction but cannot demonstrate causation. As such, it serves as a pilot for future work with an experimental design that would remove confounding variables (e.g., the individual instructor characteristics). Although it is difficult to imagine a context where a true experimental design could be implemented (owing the inability to randomly select participants for collegiate introductory programming), a delivery across multiple instructors

and institutions is conceivable; as is a design where each instructor delivers both the Holistic Course Delivery and the Traditional Course Delivery.

APPENDIX A

PRE-COURSE SURVEY

Initial Student Survey

* Required

1. Email address *

2. I think that a computer scientist mainly (check all that apply)

   Check all that apply.

   - [ ] Uses computers and software to solve real world problems
   - [ ] Uses programs like Microsoft Word, Excel, PowerPoint, or Photoshop to accomplish his/her work.
   - [ ] Installs and maintains computers and networks
   - [ ] Don't know

   Other: [ ] ______

3. Did you graduate from a:

   Mark only one oval.

   - ( ) Missouri public high school
   - ( ) Missouri private high school
   - ( ) Out-of-state public high school
   - ( ) Out-of-state private high school
   - ( ) Home school
   - ( ) International high school
   - ( ) Other
   - ( ) Other: ______

4. Have you (check all that apply)

Check all that apply.

- [ ] Had a formal class in Java, Scratch, C, C++, Python, or some other programming language
- [ ] Written a program outside of a class
- [ ] Had a formal class in an application like Word, PowerPoint, etc.
- [ ] Created a webpage
- [ ] Scripted in a video game

5. Have you taken one or more college-level Computer Science classes?

Mark only one oval.

- Yes *Skip to question 6*
- No *Skip to question 9*

6. If yes, how many? *

____

7. Which programming languages did you uses? *

____

8. Would you consider taking more computer science courses?

Mark only one oval.

- Yes
- No
- Maybe

9. Did your high school offer a formal computer programming class?

Mark only one oval.

- Yes *Skip to question 10*
- No *Skip to question 12*

10. Did you take it?

Mark only one oval.

- Yes
- No

11. What languages were you taught?

Check all that apply.

- C/C++/C#
- Java
- Python
- Scratch
- Other
- Don't Know

Skip to question 13

12. Would you have taken a Computer Science class had one been available?

Mark only one oval.

- Yes
- No
- Maybe

13. Which of the following do you associate with the field of Computer Science? *

Mark only one oval per row.

	Strongly Agree	Agree	Unsure	Disagree	Strongly Disagree
Cutting Edge	⬭	⬭	⬭	⬭	⬭
Anti-social	⬭	⬭	⬭	⬭	⬭
High income	⬭	⬭	⬭	⬭	⬭
Long working hours	⬭	⬭	⬭	⬭	⬭
Interesting	⬭	⬭	⬭	⬭	⬭
More for males	⬭	⬭	⬭	⬭	⬭
Boring	⬭	⬭	⬭	⬭	⬭
Sitting all day	⬭	⬭	⬭	⬭	⬭
Nerd	⬭	⬭	⬭	⬭	⬭

14. Did you take any of the following advanced placement or dual credit courses?

Check all that apply.

- [ ] AP Computer Science
- [ ] AP Science (Chemistry, Biology, Physics, etc.)
- [ ] Dual Credit Math
- [ ] AP Calculus
- [ ] Other AP
- [ ] Dual Credit Science

15. Are you planning to be certified to be a teacher?

Mark only one oval.

- Yes *Skip to question 17*
- No *Skip to question 18*

16. If teacher certification in Computer Science were available, would you be interested in becoming certified?

Mark only one oval.

- Yes
- No

17. What area do you want to be certified in?

Check all that apply.

- [ ] Secondary Math
- [ ] Elementary
- [ ] Science
- [ ] Other/Unsure

18. Are you a transfer student?

Mark only one oval.

- Yes
- No

19. Are you in the military or have you been in the military?

- Yes
- No

20. Year of Study:

Mark only one oval.

- Freshman
- Sophomore
- Junior
- Senior

21. Gender

Mark only one oval.

- Female
- Male
- Prefer not to say

22. Major (Please specify major or write undecided)

APPENDIX B

PRE- POST-COURSE TEST

Pre-course Test & Post Course Test

1. After executing the following line of code:

```
int f = 7 + 3;
```

What is the value of f ?

A. 3

B. 5

C. 7

D. 10

E. None of the above

2. Given the following two sets of code:

(1)
```
int f;
f = 2;
```

(2)
```
int f;
2 = f;
```

After the execution of which of the above parts of code will the variable f contain the value 2?

A. (1)

B. (2)

C. (1) and (2)

D. None of the above

3. Given then following code,

```
int f = 2;
f = 3;
```

What is the value of **f** after the code is executed?

A. f is 2

B. f is 2 and 3

C. f is 3

D. None of the above

4. Given the following code:

```
int f = 5;

int a = f;
```

What are the values of the variables **f** and **a** after the code is executed?

A. a is 5; f is 5

B. a is 5; f is 0

C. a is 0; f is 5

D. None of the above

5. Given the following code:

```
int a;
int b;
a = 3;
b = a;
a = 4;
```

After this code is executed, what are the values of **a** and **b**?

A. a is 3; b is 3

B. a is 4; b is 4

C. a is 3; b is 4

D. a is 4; b is 3

E. None of the above

6. After which set of code will the variable f contain the value 4?

(1)

```
int f;
f = f + 4;
```

(2)

```
int f = 0;
f = f + 4;
```

A. (1)

B. (2)

C. (1) and (2)

D. None of the above

7. After the following code is executed, what is displayed on the console?

```
int width = 0;
int height = 0;
int area = width * height;
width = 4;
height = 4;
System.out.println(area);
```

A. 0

B. 4

C. 8

D. 16

E. None of the above

8. After the following code is executed, what are the values of a, b, c, d, and e?

```
int a = 0;
int b;
int c = 10 + 5;
int d = 23;
int e = 4;
b = c;
c = a;
a = b;
e = c + 3;
d = c;
c = d;
```

Answer:

a is ____

b is ____

c is ____

d is ____

e is ____

9. You are given the variables a, b, and c that have been properly declared as integers and initialized. Which triplet of code will result in a and b being swapped? Assume code will execute top to bottom.

A.
```
c = a;
b = a;
a = c;
```

B.
```
c = a;
a = b;
b = c;
```

C.
```
c = b;
a = b;
b = c;
```

D. None of the above

10. After the following coded is executed, what is displayed on the console?

```
int a = 9;
if (a == 10){

    System.out.print("first ");

}

System.out.print("second ");

a = 10;
```

A. first

B. second

C. first second

D. None of the above

11. If the following code is executed, what is the output when the user enters a value of 6 at the prompt?

```
Scanner input = new Scanner(System.in);
System.out.print("Enter an integer: ");
int num = input.nextInt();
if (num > 5){
    System.out.print("Big Number! ");
}
System.out.print("Small Number! ");
```

A. Big Number!

B. Small Number!

C. Big Number! Small Number!

D. Nothing is displayed

12. In one sentence, state the purpose of this code, or respond Don't know.

```
if (y1 < y2){
    t = y1;
    y1 = y2;
    y2 = t;
}

if (y2 < y3){
    t = y2;
    y2 = y3;
    y3 = t;
}

if (y1 < y2){
    t = y1;
    y1 = y2;
   y2 = t;
}
```

Your response:

13. In one sentence state the purpose of the following code. Do NOT give a line-by-line description of what the code does. Instead, tell the purpose of the code. If you do not know, respond "Don't know".

```
if (a > b){
   if (b > c){
       System.out.print(c);
   } else {
       System.out.print(b);
   }
}else if (a > c){
       System.out.print(c);
} else {
    System.out.print(a);
}
```

Your response:

14. What is the output of the following code?

```
int counter = 0;
while (counter < 2){
      System.out.print("A");
      counter = counter + 1;
      System.out.print("B ");
}
```

A. AB AB A

B. AB AB

C. AB AB AB

D. AB A

E. None of the above

15. What is the output from executing the following program?

```
public class RandomTest{
    public static void main(String[] args){
        int a = 5;
        int b = 6;
        mystery(a);
    }

    public static void mystery(int b){
         System.out.println(b);
    }
}//RandomTest
```

A. 5

B. 6

C. 11

D. None of the above

16. State in one sentence the purpose of the following code. If you do not know, state "Don't know".

```
int x;
int[] w = {3,2};
if (w[0] > w[1]){
    x = w[0];
    w[0] = w[1];
    w[1] = x;
}
```

Your response:

17. What is the output of the following code?

```
public class RandomTest{
   public static void main(String[] args){
      int[] a = {1, 2, 3};
      int[] b = {4, 5, 6};
      mystery(a);
      System.out.print(a[0] + "  " + b [0]);
   }

   public static void mystery(int[] b){
      b[0] = 7;
   }
}//end RandomTest
```

A. 1 4

B. 7 4

C. 4 7

D. 7 7

E. None of the above

APPENDIX C

POST-COURSE SURVEY

Student Exit Survey Fall 2019

Please complete this survey for 30 points. Your responses will not be reviewed until after final grades are submitted, so please feel free to respond honestly. It has been a joy to work with you this semester and I wish you the very best for your future.
* Required

1. Email address *

____

Course Delivery

2. *

Mark only one oval per row.

	Extremely helpful	Helpful	Slightly helpful	Not helpful
Watching the instructor live code was	⬭	⬭	⬭	⬭
Coding along side the instructor during class was	⬭	⬭	⬭	⬭
Practice with problem solving before Sprints was	⬭	⬭	⬭	⬭

Online Quizzes

3. *

Mark only one oval per row.

	Extremely helpful	Helpful	Slightly helpful	Unhelpful
Online quizzes were	⬭	⬭	⬭	⬭

4. *

Mark only one oval per row.

	Always	Sometimes	Never
How often did you seek out knowledge for questions that you missed on quizzes prior to retaking the quiz.	⬭	⬭	⬭

Homework

5.

Mark only one oval per row.

	Extremely helpful	Helpful	Slightly helpful	Unhelpful
Homework problems were	⬭	⬭	⬭	⬭

6. *

Mark only one oval per row.

	About right	Too many	Not enough
Number of problems on each assignment was	⬭	⬭	⬭
Frequency (weekly) of assignments was	⬭	⬭	⬭

7. *

Mark only one oval per row.

	Too difficult	About Right	Too easy
Difficulty of problems on each homework	⬭	⬭	⬭

8. Reflections after homework (check all that apply) *

Check all that apply.

- [ ] Helped me to assess my own comprehension of subject matter
- [ ] Helped me assess my own performance on the assignment
- [ ] Were a good closure before moving onto the next assignment
- [ ] Were not of much use for me
- [ ] Other

Sprints

9. Do you prefer having sprints or midterm exams? *

Mark only one oval.

- ( ) Sprints
- ( ) Midterm Exams
- ( ) Don't care

10. *

Mark only one oval per row.

	Always	Usually	Rarely	Never
Did you feel prepared for the code tracing question on the sprint?	( )	( )	( )	( )
Did you feel prepared for the code writing portion of the sprint?	( )	( )	( )	( )

Future plans

11. Are you planning to take CS 1110 Computer Programming II *

Mark only one oval.

- Yes *Skip to question 12*
- No *Skip to question 14*
- Don't know *Skip to question 12*

Skip to question 14

Preparation for future

12. How prepared do you feel to take CS 1110 Computer Programming II? *

Mark only one oval.

	1	2	3	4	5	
Totally unprepared	○	○	○	○	○	Well prepared

13. If you feel unprepared, what are your concerns?

____

Programming Ability

14. *

Mark only one oval per row.

	Poor	Average	Above Average	Excellent
I feel that my programming ability at the beginning of the semester was				
I feel that my programming ability today is				
I feel that my ability to break a problem into steps is				

Due Dates, etc.

15. *

Mark only one oval per row.

	Never	Rarely	Sometimes	Often
How often did you turn in homework late?				

16. *

Mark only one oval per row.

	Help	Don't know	Unhelpful
Soft deadlines for late work were....			

Parting Thoughts...

17. What did like about this course?

18. If you were the instructor, what would you change?

19. If you were the instructor, what would you keep the same?

20. Please share any thoughts you have about your experience in this course, either positive or negative.

21. I think that a computer scientist mainly (check all that apply)

Check all that apply.

- [ ] Uses computers and software to solve real world problems
- [ ] Uses programs like Microsoft Word, Excel, PowerPoint, or Photoshop to accomplish his/her work.
- [ ] Installs and maintains computers and networks
- [ ] Don't know

Other: [ ] ____________________

22. Which of the following do you associate with the field of Computer Science? *

Mark only one oval per row.

	Strongly Agree	Agree	Unsure	Disagree	Strongly Disagree
Cutting Edge					
Anti-social					
High income					
Long working hours					
Interesting					
More for males					
Boring					
Sitting all day					
Nerd					

APPENDIX D

HOMEWORK REFLECTION

Homework Reflection HWX

Complete this from after you have submitted your homework assignment *
Required

1. Email address *

____

2. How much time did you spend on the assignment? *

Mark only one oval.

- ( ) 1-3 hours
- ( ) 4-6 hours
- ( ) 7-9 hours
- ( ) more than 9 hours

3. Please select level of use of the following resources. *

Mark only one oval per row.

	None	Some	Heavy use
Internet	( )	( )	( )
Classmate/Peer	( )	( )	( )
Tutor	( )	( )	( )
Instructor	( )	( )	( )
Class resources	( )	( )	( )

4. 4. Rate your confidence in Java programming after completing this assignment. *

Mark only one oval.

	1	2	3	4	5	
Extremely unsure	⬭	⬭	⬭	⬭	⬭	Extremely sure

5. The first problem on the assignment was *

Mark only one oval.

- ⬭ Difficult to solve
- ⬭ Completed with substantial effort and review of course materials
- ⬭ Fairly easy and straightforward with some review of course material
- ⬭ Easy with hardly any need to review course materials.
- ⬭ None of the above

6. Overall, the homework problems were useful to practice and to learn the concepts in this course? *

Mark only one oval.

	1	2	3	4	5	
Strongly Disagree	⬭	⬭	⬭	⬭	⬭	Agree

7. List your Challenges and Triumphs from this assignment. If you had nothing to respond, reply "None" *

8. Do you feel that you are getting stronger in programming each week? *

Mark only one oval.

- ⬭ Yes
- ⬭ No
- ⬭ Maybe

9. Do you believe that your are getting stronger at solving problems each week? *

Mark only one oval.

- Yes
- No
- Maybe

10. Please list any concerns you have regarding this class.

This content is neither created nor endorsed by Google.

Google Forms

APPENDIX E

HOMEWORK ASSIGNMENTS

CS1100 Homework 1 Due Date: Wed 9/04/2019, 11:59 pm Total points: 24 points + 5 points reflection

Instructions: Programs are due by 11:59 pm on the due date via Blackboard. **Please remember to properly indent your code, include appropriate comments and the required header. Deductions will be made for improperly formatted programs.**

Complete the reflection after you submit your assignment. There is a link in the homework folder to the reflection. You will receive an email confirmation of the submission but no indication will appear in the gradebook until I manually add the completion grade.

A header is comments at the top of your file with your name and brief description of the problem.

Programs must compile and run for full credit.

Please submit your 3 .java files through the link for HW1 on Blackboard. You will submit each file individually. Please do not zip your work.

1. (8 points) Create a Java program that prints your name, your major, and your favorite food. For example, for Belinda, the program would display:

   Belinda
   Computer Science
   Lasagna

2. (8 points) Problem 1.3 from the textbook. Write a program that displays the following pattern as shown in the text:

```
    J    A     V   V    A
    J   A A     V V    A A
 J  J  AAAAA     VV   AAAAA
 J  J A     A    V   A     A
```

3. (8 points)

   Write a program that displays the sum of 5 and 10.

   The output from running the program will be:

   15

CS1100 Homework 2 Due Date: 9/11/2019, 11:59 pm Total points: 24 points + 5 points reflection

Instructions: Programs are due by 11:59 pm on the due date via Blackboard. **Please remember to properly indent your code, include appropriate comments and the required header. Deductions will be made for improperly formatted programs.**

A header is comments at the top of your file with your name and brief description of the problem. Programs must compile and run for full credit.

Please submit your 3 .java files through the link for HW2 on Blackboard. You will submit each file individually. Please do not zip your work. You will need to submit pseudocode for Problem 3 through the link HW2-Psuedo

1. (7 points) (*Area and perimeter of a rectangle*)

   Write a program that prompts and reads user input for the width and height of a rectangle. You program will calculate and display the area and perimeter of the rectangle. The values for width and height should be treated a a number with a decimal point. Area and perimeter are calculated as shown below.

   area = width × height

   perimeter = (2 x width) + (2 x height)

   The output from running the program with a width of 5.5 and height of 2.2 will be:

   Please enter the width: 5.5

   Please enter the height: 2.2

   (you will likely have more to the right of the decimal point, and that is ok.)

   Area is 12.100000
   Perimeter is 15.4

2. (7 points) Create a Java program that will calculate the volume and area of a sphere. To calculate the area, use this formula

   area = $4 \pi r^2$

   $$\text{volume} = \frac{4}{3}\pi r^3$$

   - Your program will need to read a value for the radius (r) for the console. Radius should be a double.
   - Create a constant for π with the value 3.14159.

   For example, if the user enters a value of 5.5 for radius, your program should display

Area is 380.13239

Volume is 696.909381666666

3. (10 points) (*2 points for submitting pseudocode prior to Java code & 8 points Java program*)

   You are in a job where you earn a tip for service. You need a program that will calculate how much to add to the bill after calculating gratuity. Write the pseudocode that would solve this problem and submit to HW2-Psuedo on Blackboard prior to writing any Java code. Your program will read the subtotal and the gratuity rate from the user, computes the gratuity and total and displays the results to the user.

   For example, if the user enters 10 for subtotal and 15 for gratuity rate, the program displays $1.5 for gratuity and $11.5 as total. Don't worry that the format of the output is "$X.X" rather than "$X.XX".

   To calculate a tip, calculate subtotal * rate / 100

   To calculate the total bill, use the above calculation and add to the subtotal.

   Sample run:

   Enter the subtotal or the bill 10

   Enter the gratuity rate 15

   The amount for gratuity is $1.5

   The total final bill is $11.5

CS1100 Homework 3 Due Date: 9/18/2019, 11:59 pm Total points: 24 points

+ 5 points for Reflection

Instructions: Programs are due by 11:59 pm on the due date via Blackboard. **Please remember to properly indent your code, include appropriate comments and the required header. Deductions will be made for improperly formatted programs.**

A header is comments at the top of your file with your name and brief description of the problem. Programs must compile and run for full credit.

Please submit your 4 .java files through the link for HW3 on Blackboard. You will submit each file individually. Please do not zip your work.

1. (6 points) Create a Java program that takes an integer value entered from the console and displays whether the number is even or odd. (Hint: use % operator and if-statement)
   For example, the use enters 53 at the prompt:
   Please enter an integer => 53
   The program will displays,
   53 is odd.

2. (8 points) Problem 3.19 from the textbook.
   Write a program that will read in three integer values from the console that represent the sides of a triangle. Display whether the sides form a legal triangle. For a triangle to be legal, the sum of each pair of sides must be greater than the third side. If your triangle is "legal", display the perimeter. If the triangle is "illegal" display a message to the user that the sides to not form a valid triangle.

   Please enter 3 sides of a triangle: 1 2 3

   The sides 1 2 3 make an illegal triangle.

   Please enter the 3 sides of a triangle: 3 4 5
   The perimeter is 12

Problem 3 – We will cover this material on Monday, 9/17.

3. (10 points) Problem 3.17 from the textbook, with modification.
   Rock, Paper, Scissors. Write a program that plays the popular scissor-rock-paper game. (A scissor can cut paper, a rock can crush scissor, and a paper can cover rock). The program randomly generates a number 0, 1, or 2 representing the

scissor, rock, and paper. The program prompts the user to enter a number 0, 1, or 2 and displays a message indicating whether the user or the computer wins, loses, or ties. See the sample output, as provided in your textbook. (See section 3.7 for instructions on how to generate a random number.)

Submit your pseudo-code to HW3-Pseudo prior to coding. (2 points)

CS1100 Homework 4 Due Date: 9/25/2019, 11:59 pm Total points: 24 points

+5 points reflection

Instructions: Programs are due by 11:59 pm on the due date via Blackboard. **Please remember to properly indent your code, include appropriate comments and the required header. Deductions will be made for improperly formatted programs.**

A header is comments at the top of your file with your name and brief description of the problem. Programs must compile and run for full credit.

Please submit your 2 .java files through the link for HW4 on Blackboard. You will submit each file individually. Please do not zip your work.

1. (12 points) Problem 3.17 from the textbook, this time with a **Switch** statement. Rock, Paper, Scissors. Write a program that plays the popular scissor-rock-paper game. (A scissor can cut paper, a rock can crush scissor, and a paper can cover rock). The program randomly generates a number 0, 1, or 2 representing the scissor, rock, and paper. The program prompts the user to enter a number 0, 1, or 2 and displays a message indicating whether the user or the computer wins, loses, or ties. See the sample output, as provided in your textbook. (See section 3.7 for instructions on how to generate a random number.)

   Instead of the if…else version from HW3, use a Switch statement. You must use a switch statement to receive credit. No pseudo-code submission is required.

2. (12 points) Extend the NumberPalindrome example to read a 4-digit number and determine if the value entered is a palindrome.

   Examples of palindromes include: 1221 3443

   Examples that are not a palindrome 1231 3453

   Hint: you will need to add code to isolate the 1000's, 100's, 10's and 1's digits. You will need a variable for each place.

   Sample run:
   Please enter a 4-digit number: 1221

   1221 is a palindrome.

CS 1100 Homework 5 Due Date: 10/2/2019, 11:59 pm Total points: 24 points

+5 points Reflection

Instructions: Programs are due by 11:59 pm on the due date via Blackboard. **Please remember to properly indent your code, include appropriate comments and the required header. Deductions will be made for improperly formatted programs.**

A header is comments at the top of your file with your name and brief description of the problem. Programs must compile and run for full credit.

Please submit your 2 .java files through the link for HW5 on Blackboard. You will submit each file individually. Please do not zip your work.

1. (10) Write a Java program that will read three integers from the user and display the numbers in ascending order. Your input and output should be similar to the following.

    Please enter 3 integers: 55 33 44

    The numbers in order are: 33 44 55

2. (14 points) Problem 4.15 from the textbook.
    The international standard letter/number mapping found on the telephone is shown below:

image from https://www.dcode.fr/phone-keypad-cipher

Write a program that prompts the user to enter a lowercase or uppercase letter and display its corresponding number. For a non-letter input, display "invalid input". (Hint: if the user does not enter 'A' –'Z' or 'a'-'z', then the input is invalid.)

This is a great problem to practice the switch-statement.

Sample from three different runs:

```
Enter a letter: T

The corresponding number is 8


Enter a letter: r

The corresponding number is 7


Enter a letter: %

% is invalid input
```

CS 1100 Homework 6 Due Date: 10/9/2019, 11:59 pm Total points: 24 points

+ 5 points reflection

Instructions: Programs are due by 11:59 pm on the due date via Blackboard. **Please remember to properly indent your code, include appropriate comments and the required header. Deductions will be made for improperly formatted programs. A header is comments at the top of your file with your name and brief description of the problem. Programs must compile and run for full credit. Please submit your .java files through the link for HW6 on Blackboard. You will submit each file individually. Please do not zip your work.**

1. (8 points) Problem 4.25 from the textbook.
   (*Generate vehicle plate numbers*) Assume that a vehicle plate number consists of three uppercase letters followed by four digits. Write a program to generate a plate number.
   Hint: to generate a random capital letter – generate a number between 65 and 90, inclusively. To generate the code for a number, generate a number between 48 and 57, inclusively. You will need to convert the random number to a displayable ASCII character. See Section 4.3.1 and 4.3.3 in the text for details on how to covert between the code for a character and the displayed character.

   Examples of license plate: ABC1234 ZBG5387

   If you need a hint on generating a range of numbers, please refer to RandomNumbers.java that we did in class:

```
//  Generating a number from any range...
//Let's generate a random number between
// min and max, inclusively
// How about a number between 20 and 100, inclusively.
int max = 100;  //biggest value to generate
int min = 20;   //smallest value to generate
int range = max - min + 1;  //number of values in range

int rand = (int)(Math.random() * range) + min;
System.out.println("rand is " + rand);
```

   Also, recall that

```
int rand = (int)(Math.random() * range) + min;  //min is code for A and range is 26
char c = (char) rand;
System.out.println(c);
```

   Will display the character represented by rand in the ASCII table. You can generate a random number in the appropriate range and assign to a variable of type char by casting rand to a char.

2. (8 points) Write a program the reads a string from the user that represents dollars and cents. The user will enter the dollar-sign and amount, $XXX.XX. The user will enter the $ sign and decimal point as part of the string. The output should be how many dollars and how many cents are represented by the amount entered.
   For example:

   Please enter an amount: $35.46

   The output will be

   There are 35 dollars and 46 cents.

   Hint: You will want to use the indexOf and substring methods to solve this problem.

3. (8 points)
   Write a program to display the multiplication facts for a number. The user will enter a value and then a table will be displayed showing the multiplication facts for that value.

```
Please enter a number: 3

Multiplication table for 3
Multiplier    Result
--------------------------------
1             3
2             6
3             9
4             12
5             15
6             18
7             21
8             24
9             27
10            30
Press any key to continue . . .
```

   Please use a while-loop to solve this problem.

   Hint: The header (column names and the ----‘s) will be printed prior to entering the while loop.

CS 1100 Homework 7 Due Date: 10/16/2019, 11:59 pm Total points: 24 points +5 points reflection

Instructions: Programs are due by 11:59 pm on the due date via Blackboard. **Please remember to properly indent your code, include appropriate comments and the required header. Deductions will be made for improperly formatted programs. A header is comments at the top of your file with your name and brief description of the problem. Programs must compile and run for full credit. Please submit your .java files through the link for HW7 on Blackboard. You will submit each file individually. Please do not zip your work.**

1. (8 points)
   Revisit of HW6 – Problem 3, now with a for-loop. Write a program to display the multiplication facts for a number. The user will enter a value and then a table will be displayed showing the multiplication facts for that value.

   ```
   Please enter a number: 3

   Multiplication table for 3
   Multiplier    Result
   --------------------------------
   1             3
   2             6
   3             9
   4             12
   5             15
   6             18
   7             21
   8             24
   9             27
   10            30
   Press any key to continue . . .
   ```

   Use a for-loop to solve this problem.

   Hint: The header (column names and the ----'s) will be printed prior to entering the while loop.

2. (8 points) Problem 5.3 from the textbook.
   (*Conversion from kilograms to pounds*) Write a program that displays the following table (note **1** kilogram is **2.2** pounds):

```
Kilograms    Pounds
------------------------
1             2.2
3             6.6
5            11.0
7            15.4
9            19.8
11           24.2
13           28.6
15           33.0
17           37.4
19           41.8
Press any key to continue . . .
```

Please use a for-loop to solve this problem. Additionally, to format your output into the nice columns, please use printf and formatters. See 4.6 on how to format. Table 4.12 explanation of %10.f is a good hint.

Hint: The header (column names and the ----'s) will be printed prior to entering the for loop.

3. (8 points) Write a program that reads integers greater than zero, and displays the average and sum of the numbers entered. The program will continue to prompt for numbers until the user enters a -1, to indicate termination of input. A sample run is a follows:

```
Please enter a values and terminate with -1: 10
Please enter a values and terminate with -1: 3
Please enter a values and terminate with -1: 4
Please enter a values and terminate with -1: -1
The sum is 17
The average is 5.666666666666667
Press any key to continue . . .
```

Hint: Recall that to obtain a double value from integer division, you will need to cast the result to a double.

i.e. 5/3 is a result of 1, but, (double)5/3 is a result of 1.6666666666666667

Hint: You will need to use a **sentinel** controlled **while** loop in your solution.

CS 1100 Homework 8 Due Date: 10/24/2019, 11:59 pm Total points: 24 points +5 points Reflection

Instructions: Programs are due by 11:59 pm on the due date via Blackboard. **Please remember to properly indent your code, include appropriate comments and the required header. Deductions will be made for improperly formatted programs. A header is comments at the top of your file with your name and brief description of the problem. Programs must compile and run for full credit. You will submit each file individually. Please do not zip your work.**

1. (12 points) Write a program that reads 10 integers and displays the largest value that was entered. Use a for-loop, do…while loop, or while loop -- your choice.

   Enter 10 values:

   10 15 25 5 4 12 33 20 22 12

   The largest value is: 33

   Press any key to continue . . .

2. (12 points) Write a program using two nested for loops to display the following:

```
10 11 12 13 14 15 16 17 18 19
20 21 22 23 24 25 26 27 28 29
30 31 32 33 34 35 36 37 38 39
40 41 42 43 44 45 46 47 48 49
50 51 52 53 54 55 56 57 58 59
60 61 62 63 64 65 66 67 68 69
70 71 72 73 74 75 76 77 78 79
```

CS 1100 Homework 9 Due Date: 10/31/2019, 11:59 pm Total points: 24 points

+5 points reflecton

Instructions: Programs are due by 11:59 pm on the due date via Blackboard. **Please remember to properly indent your code, include appropriate comments and the required header. Deductions will be made for improperly formatted programs. A header is comments at the top of your file with your name and brief description of the problem. Programs must compile and run for full credit. Please submit your 3 .java files through the link for HW9 on Blackboard. You will submit each file individually. Do not zip your work.**

1. (8 points) Problem 6.5 from the textbook (You sorted three numbers in a previous assignment and can reuse that solution, now in a method.)
   Write a method with the following header to display three number in increasing order:

   ```
   public static void displaySortedNumbers(int num1, int
       num2, int num3)
   ```

   You will need to include a main that prompts the user for three values and makes a call to (or invokes) the method displaySortedNumbers.

2. (8 points) Problem 6.17 from the textbook
   (Display matrix of 0s and 1s) Write a method that displays an n-by-n matrix using the following header:

   ```
   public static void printMatrix(int n)
   ```

   You will prompt the user for the value n, and then invoke the method printMatrix. Each element is 0 or 1, will be generated randomly. A sample run is as follows:

   Enter n: 3

   0 1 0
   0 0 0
   1 1 1

   note: since the 0's and 1's are generated randomly, your output will vary, but in this case there will be 3 rows by 3 columns of output. You will need to use nested loops in your solution.

3. (8 points) problem 6.21 from the textbook

The international standard letter/number mapping for telephones is given in problem Exercise 4.15 (which we did in an earlier assignment). Write a method that returns a number, given an uppercase letter, as follows:

public static int getNumber(char uppercaseLetter)

You will prompt the user to enter a phone number as a string. The input number may contain letters. The program translates a letter (uppercase or lowercase) to a digit and leaves all other letters intact. Here are sample input and output:

Enter a string: 1-800-Flowers
1-800-3569377

Enter a string: 1800flowers
18003569377

Hint: You solved the problem of going from a character to number on a keypad in a previous homework. Turn this code into method getNumber. Your main will need to prompt and read the string and then loop from the beginning of the string to the end. If the character is a a number or non-alpha symbol, output to the console. If the character is alpha, call getNumber and then display the value that is returned.

CS 1100 Homework 10 Due Date: 11/7/2019, 11:59 pm Total points: 24 points

+5 points reflection

Instructions: Programs are due by 11:59 pm on the due date via Blackboard. **Please remember to properly indent your code, include appropriate comments and the required header. Deductions will be made for improperly formatted programs. A header is comments at the top of your file with your name and brief description of the problem. Programs must compile and run for full credit. Please submit your .java file through the link for HW10 on Blackboard. You will submit each file individually. Do not zip your work. Don't forget to complete the survey.**

This program models a restaurant ordering system. You will need to have the following methods:

public static int displayMenu()

> **This method will display the menu items and price per item. I had 5 items, you can add items or feel free to have different food items. You need an option so that the customer can indicate that ordering is done. This method will read the choice from the customer and return that number.**

public static double getQuantityAndCost(int x)

> **This method will prompt the user for the quantity for the menu choice and then multiply by the appropriate amount. For example if the user had selected Tacos, then x would be passed in as the value 2 and the return value for this method would be 2 * 4.00.**

public static double AddTipAmount(double sum)

> **This method will prompt the user for a tip amount. The method receives sum which is the total on the bill after the user has selected Done ordering. For example the method is called with a sum of 22.00 and the user indicate 10 percent for the tip, the method should return 22.00 + 2.20, or 24.20.**

The menu will be displayed and get the customer's item, quantity and cost for will be calculated until the user has selected the Done option.

After Done option is selected, ask the user for a tip amount. Then finally display the total bill.

<u>**Sample run below**</u>

Welcome to Belinda's Restaurant. Please make your choices and select 6 when you are finished.

Menu
1 : Pizza $2.50
2 : Tacos $4.00
3 : Lasagna $5.00
4 : Sushi $3.00
5 : Fish $7.00
6 : Done ordering

Please enter your choice: 2

Please enter quantity: 2

Menu
1 : Pizza $2.50
2 : Tacos $4.00
3 : Lasagna $5.00
4 : Sushi $3.00
5 : Fish $7.00
6 : Done ordering

Please enter your choice: 5

Please enter quantity: 2

Menu
1 : Pizza $2.50
2 : Tacos $4.00
3 : Lasagna $5.00
4 : Sushi $3.00
5 : Fish $7.00
6 : Done ordering
Please enter your choice: 6

Enter tip amount as % i.e. 15: 10

The total bill is $24.20

CS 1100 Homework 11 Due Date: 11/17/2019, 11:59 pm Total points: 24 points

+5 points reflection

Instructions: Programs are due by 11:59 pm on the due date via Blackboard. **Please remember to properly indent your code, include appropriate comments and the required header. Deductions will be made for improperly formatted programs. A header is comments at the top of your file with your name and brief description of the problem. Programs must compile and run for full credit. Please submit your .java file through the link for HW11 on Blackboard. You will submit each file individually. Do not zip your work. Don't forget to complete the survey.**

Problem 1 (12 points)

This program is practice on the concept of method overloadoing. See Section 6.8 Overloading Methods for more information.

Write a program that contains the following overloaded methods:

public static void add(int n1, int n2) //Will display the sum of n1 and n2

public static void add(String s1, String s2) //Will display the concatenation of s1 and s2

public static void add(int n1, int n2, int n3) //Will display the sum of n1, n2, and n3

You need to complete each method and write a main that invokes each method. You may choose to hardwire values to pass to the method, or read input.

Sample run:

Sum of 2 3 is 5
dog added to cat is dogcat
Sum of 2 3 4 is 9
Press any key to continue . . .

Problem 2 (12 points)

Download and open the file HW11_Problem2.java. Your task is to improve the solution by modularizing, or moving chunks of code into methods. Your program should include 3 methods and produce the same output as the sample run shown below. Have your main read in all input.

Sample run:

Enter height and width of a rectangle: 5 10
The area is 50
The perimeter is 30

```
Enter height and width of a rectangle: 2 4
The area is 8
The perimeter is 12

Enter height and width of a rectangle: 25 35
Width is greater than height
Press any key to continue . . .
```

CS 1100 Homework 12 Due Date: 12/01/2019, 11:59 pm Total points: 24 points +5 points Reflection

Instructions: Programs are due by 11:59 pm on the due date via Blackboard. **Please remember to properly indent your code, include appropriate comments and the required header. Deductions will be made for improperly formatted programs. A header is comments at the top of your file with your name and brief description of the problem. Programs must compile and run for full credit. Please submit your .java file on Blackboard.**

Problem 1 (8 points) Write a program that declares an array to hold 12 integers. The contents of the array should be the values 2,4,6,8,10,12,14,16,18,20,22,24. Use a for-loop to initialize the contents of the array to these values. After initializing the array to these values, the program should display the contents of the array.

Suggested output:

```
The contents of the array is:
Index          Value
0                2
1                4
```

Etc…..

Problem 2 *7.1 (*Assign grades*) (8 points) Write a program that reads student scores, gets the best score, and then assigns grades based on the following scheme:

- Grade is A if score is $\geq$ best -10;
- Grade is B if score is $\geq$ best -20;
- Grade is C if score is $\geq$ best -30;
- Grade is D if score is $\geq$ best -40;
- Grade is F otherwise.

The program prompts the user to enter the total number of students, then prompts the user to enter all of the scores, and concludes by displaying the grades. Use an array of int to store the input. The array size will depend on the number of students the user enters. Here is a sample run:

Enter number of students: 7

Enter 7 scores: 97 85 88 35 72 79 82

Student 0 score is 97.0 and grade is A
Student 1 score is 85.0 and grade is B
Student 2 score is 88.0 and grade is A
Student 3 score is 35.0 and grade is F
Student 4 score is 72.0 and grade is C

Student 5 score is 79.0 and grade is B
Student 6 score is 82.0 and grade is B

Problem 3 7.3 Modified (8 points) (*Count occurrence of numbers*) Write a program that reads the integers between **1 and 10** and counts the occurrences of each. Assume the input ends with **0**. Use an array to keep track of occurrences. Here is a sample run of the program:

Enter the integers between 1 and 10: 1 3 5 4 2 1 7 8 4 4 2 1 0

1 occurs 3 times

2 occurs 2 times

3 occurs 1 time

4 occurs 3 times

5 occurs 1 time

7 occurs 1 time

8 occurs 1 time

CS 1100 Homework 13 Due Date: 12/06/2019, 11:59 pm Total points: 24 points +5 points Reflection

Instructions: Programs are due by 11:59 pm on the due date via Blackboard. **Please remember to properly indent your code, include appropriate comments and the required header. Deductions will be made for improperly formatted programs. A header is comments at the top of your file with your name and brief description of the problem. Programs must compile and run for full credit. Please submit your .java file on Blackboard.**

Write a program that:

1. Declares and array capable of holding **4** integers. Only values > 0 can populate the array.
2. Loop prompting the user to enter values to insert into the array until they enter a -1.
3. You will need to check to see if the array is full before inserting the value in the next available spot in the array.
4. If the array is full you will need to invoke the method, public static int[] expandArray(int[ ] array) that will double the size of the array and copy in the contents of the array.
5. Include the method public static void displayArray(int[ ] array) that will display the contents of the array.
6. Your main will contain the loop that prompts user to enter a value or -1 to indicate to quit. If a value greater than 0 is enter, insert the value into the next available slot in the array. You need to check to see if there is room in the array prior to inserting. After the user enters -1 and the loop terminates, display the contents of the array by calling displayArray. Hint: your main will want to keep track of the next available index into the array.
7. When testing your program try to enter 10 values.

Sample run:

Please enter values to insert into the array or -1 to quit.

5
15
4
12
3
6
8
22
16
11
-1

The contents of the array:

Index	Value
0	5
1	15
2	4
3	12
4	3
5	6
6	8
7	22
8	16
9	11

APPENDIX F

CODING SPRINTS

Sprint 1

Problem 1

What output does the following program produce?

```
public class CodeTrace_1{
    public static void main(String[ ] args){
        System.out.println("The result is " + (3 +7));
    }
}
```

A. The result is 10

B. The result is 37

C. 10

D. The result is (3 + 4)

Problem 2

Write a Java program (and submit through an attachment) that will display:

```
3
2
1
Blast Off!
```

Sprint 2

Problem 1:
What is the output of the following statements?

```
int num = 0;
int x = num + 3;
if (x > 0)
    System.out.print("x is greater than 0");
else if (x < 0)
    System.out.print("x is less than 0");
else
    System.out.print("x equals 0");
```

A. x equals 0
B. x is less than 0
C. x is greater than 0
D. No output is produced

Problem 2

Download the skeleton file CodeSprint2A.java and follow the instructions presented in the comments.

Submit your .java file (not your .class file) **CodeSprint2A.java**

```
//This program reads a number that represents the number
//of items meals purchased at a restaurant. The program displays
//the appropriate reward based on number of meals purchase.

//The following rules apply:
// 1 <= meals < 3 meals receive a free drink
//4 <= meals <= 7 meals receive a free dessert
//greater than 7 meals, receive a free meal.


import java.util.Scanner;

public class CodeSprint2A {
  public static void main(String[] args) {
    Scanner input = new Scanner(System.in);

    // Prompt the user to enter weight in pounds
    System.out.print("Enter number of meals purchased: ");
    int numMeals = input.nextInt();

        //Add the code to display the appropriate reward.

  }
}
```

Sprint 3

Problem 1

What is the output of the following code snippet?

```
int a = 2;
int b = 5;

a = a + b;
b = a - b;
a = a - b;

System.out.println("a = " + a + "\n b = " + b);
```

A. a=2
   b=5

B. a= 5
   b = 2

C. a = 5
   b = 5

D. a = 2
   b = 2

Problem 2

CodeSprint3.java Write a program that prompts the user to enter a String. The program will display the first character and whether the input is a least 3 characters long.
A sample run:

```
Please enter a string: Apple


The first character is A
 Apple is longer than 3 characters.
Press any key to continue . . .
```

Another sample run:

```
Please enter a string: go
The first character is g
go is NOT longer than 3 characters.
Press any key to continue . .
```

Please attach your Java file for grading.

```
//Write a program that prompts the user to
//enter a String and displays the first character
//and
//displays whether the string is a least 3 characters long.

import java.util.Scanner;

public class CodeSprint3{
        public static void main(String[] args){
                Scanner input = new Scanner(System.in);

                System.out.print("Please enter a string: ");
                String str = input.nextLine();

                //Display the first character of str


                //Display whether str has greater than two characters.

        }
}
```

Sprint 4

Problem 1

What is the output of this code snippet?

```
for (int i = 10; i <= 30; i+=5){
 System.out.print(i + " ");
}
```

A. 10 15 20 25 30

B. 10 20 30 40

C. 10 15 20 25 30 35

D. 5 10 15 20 25 30 35

Problem 2

Write program that displays the following:

3 9 15 21 27 33

You must use a while loop

Here is a skeleton file to start with, if you wish. **CodeSprint4_9AM.java**

```
//Write program that displays the following:
//3 9 15 21 27 33
//You must use a while loop

//There is not input for this program
public class CodeSprint4_9AM{
        public static void main(String[] args){


        }//end main
}//end class CodeSprint4_9AM
```

Sprint 5

Problem 1

What is the output of the following program:

```
public class CodeTrace5{
 public static void main(String[] args){
 int x = 5;
 int y = 10;

     int z = doubleIt(x,y);
 System.out.println(z);
 }//end main

 public static int doubleIt(int x, int y){
 return x * 2;
 }
}//end class CodeTrace5
```

A. 5
B. 10
C. 20
D. 50

Problem 2

Download CodeSprint5.java. **CodeSprint5.java** Instructions are contained in the comments of the code.
Two different sample runs:

```
Please enter width: 5
Please enter height: 4
The area is 20
This is a rectangle.


Please enter width: 5
Please enter height: 5
The area is 25
This is a square.
```

```
//CodeSprint5
//This program reads two integers from the user that represent
//two sets of adjacent sides of a four sided shape.
//Think  two sides of a square or  a rectangle....
//The program will display the area of the shape.
//The program will display whether the shape is
//a square or rectangle.
```

```
//You need to complete the two methods and complete the
//main to invoke the methods.

import java.util.Scanner;

public class CodeSprint5{
        public static void main(String[] args){
                Scanner input = new Scanner(System.in);

                //Prompt and read the width
                System.out.print("Please enter width: ");
                int width = input.nextInt();

                //Prompt and read the height
                System.out.print("Please enter height: ");
                int height = input.nextInt();

                //Call method to calculate the area
                //and display the result from the method call.



                //invoke the method to display whether this is
                //a square of a rectangle.


        }//end main

        //method calculateArea will receive a value
        //for height and width of a 4-sided shape and return
        //the area which is height times width
        //public static int calculateArea(int h, int w)
    //PUT METHOD HERE



        //method squareOrRectangle will receive a value for height
        //and width and will compare the height and width to display
        //whether the two sides make a square or a rectangle.
        //A square has all sides the same length.
        //Rectangles have sides that are not the same length.
        //This method does not return a value, just displays
        // "square" or "rectangle"
        //public static void squareOrRectangle(int h, int w)
        //PUT METHOD HERE


}//end class CodeSprint5
```

Sprint 6

Problem 1

What is the output of the following program?

```
public class CodeTrace6_9AM{

   public static void main(String[] args){

     int x = 5;
     int y = 10;

     mystery(x,y);

     System.out.println("x is " + x);
     System.out.println("y is " + y);
    }//end main

     public static void mystery(int x, int y){

           int temp;
           temp = x;
           x = y;
           y = temp;
      }
    }//end class CodeTrace6_9AM
```

A. x is 5
   y is 10
B. x is 10
   y is 5
C. x is 10
   y is 10
D. x is 5
   y is 5

Problem 2

Download the attached skeleton file and follow the instructions included in the comments.

Sprint6_9AM.java Upload your completed file here.

```
import java.util.Scanner;

public class Sprint6_9AM{
        public static void main(String[] args){
                Scanner input = new Scanner(System.in);
```

```
		System.out.print("Please enter a number: ");
		int num = input.nextInt();

		//invoke a method that will display whether num
		//is a multiple of 3.   Recall that num % 3 can
		//be used to determine if num is equally divisble by 3.
	}
}//end Sprint6_9AM
```

Sprint 7

Problem 1

What is the output of the following program:

```
public class CodeTrace7{

    public static void main(String[] args){

        int[] array = {1,2,3};
        mystery(array[0], array[1]);
        System.out.print(array[0] + " " + array[1]);

    } //end main

    public static void mystery(int x, int y){
        x = 5;
        y = 6;
    }
}//end CodeTrace7
```

A. 1 2
B. 2 1
C. 5 6
D. 6 5

Problem 2

Down load the following file. Instructions are included in the comments.

CodeSprint7_9AM.java Submit your solution through this link.

```
public class CodeSprint7_9AM{
        public static void main(String[] args){

                //declare an array of integers to hold 10 integers


                //initialize the contents of the array to
                //0 10 20 30 40 50 60 70 80 90
                //by using a for loop.


            //Display the contents of the array at indexes 0,2,4,6,8
            //You can use a loop or not to solve this part.

        }//end main
}//end CodeSprint7_9AM
```

APPENDIX G

FINAL EXAM PROGRAMMING QUESTIONS

CS 1100 Final Exam Programming portion   Fall 2019

Name: ____

Directions: Create solutions to each of these problems. You will submit a separate solution for each problem to Blackboard. Partial credit will be given when possible. Please include comments if you are needing partial credit. Formatting and commenting is not required. You may use your textbook, but no other reference material.

1. **(40 points)** Write a program where the main reads in three integers and then invokes a method that displays whether the three integers are equal.

   public static void isEqual(int x, int y, int z)

   A sample run,

   Enter 3 integers: 3 10 15

   The three values are not equal.

   Another sample run,

   Enter 3 integers: 5 5 5

   The three values are equal.

2. (40 points) Book Club Points

Serendipity Booksellers has a book club that awards points to its customers based on the number of books purchased each month. The points are awarded as follows:

- If a customer purchases 0 books, he or she earns 0 points.
- If a customer purchases 1 book, he or she earns 5 points.
- If a customer purchases 2 books, he or she earns 15 points.
- If a customer purchases 3 books, he or she earns 30 points.
- If a customer purchases 4 or more books, he or she earns 60 points.

Write a program that asks the user to enter the number of books that he or she has purchased this month and then displays the number of points awarded.

A Sample Run…

Please enter number of books purchased: 3

You have earned 30 points.

3. (40 points)

Write a program that do the following (in the order listed):

1. Declare and array of integers capable of holding 5 integers.
2. Read in 5 integer values from the console and store them in the array.
3. Change the value at index 2 by doubling the value.
4. Change the value at index 4 with the sum of the values at index 1 and 3.
5. Call a method public static void displayArray(int[ ] numbers) that will display the contents of the array.
6. The method displayArray must use a loop to display the values.

```
Sample Run…

Enter 5 integers: 3 5 4 2 1

The contents of the array is:

Index   Content

0       3
1       5
2       8
3       2
4       7
```

APPENDIX H

QUALITATIVE INTERVIEW QUESTIONS

Qualitative Interview Questions

The following questions will be a starting point for discussions about the course and the student's experience in the course.

1. Did you accomplish your objectives for this course?
2. Did you feel prepared to tackle each programming assignment and coding sprint?
3. Do you believe that the way this course was delivered helped you? Why or why not.
4. What specific elements in the course did you find helpful?
5. If you were teaching this course, what would you do differently?

APPENDIX I

CURRICULUM DAILY OUTLINE

Date	Topic	Slides presented	Live Coding	Assignments
8/19/2019	Introduction to CS1100			Complete Pre-course Surveys & Pretest by Tuesday 11:59 pm. See Blackboard for details.
8/21/2019	What is a Computer? Chapter 1 Sections 1.0-1.4; HelloWorld	Chapter 1 slide 3-5; 9-16	Students type HelloWorld.java from printed paper into Textpad, Compile and Run.	Complete Q1 by 9/1, 11:59 pm
8/23/2019	The basics of a Java program Chapter 1 Sections 1.7, 1.8	Chapter 1 slide 18; 25-27; 31-43	Walk-thru and Revision of HelloWorld.java to include more print statements and dislaying a simple mathematical expression (5+3)	Complete Q2 by Mon. 8/26, 11:59 pm am HW1 Posted
8/26/2019	Chapter 1 The concept of syntax, runtime errors, logic errors; formatting; Sections 1.9, 1.10 Ch. 2 What is an Algorithm?, Variables	Chapter 1 slides 44-53 Chapter 2 slide 4	Revisit HelloWorld.java to demonstrate syntax errors and compiler messages	**Quiz 3 posted, due Wed. 11:59 pm.**
8/28/2019	Chapter 2 Writing a Simple Program, Reading Input from the Console, Identifiers, Variables, Assignment statement, 2.1, 2.3, 2.4, 2.5, 2.6,	Chapter 2 slides 5-10	Board exercise on how to do a load of laundry Write a program that displays area of circle with radius of 20. ComputerArea.java Write a program that takes radius as input and display area of circle. (With pseudocoded steps to solve the problem) ComputerAreaWithConsoleInput.java	
8/30/2019	Chapter 2 Constants, Numeric data types and operations 2.7, 2.8, 2.9	Chapter 2 slides 11-24	Write a program that reads two integers as input and displays (A+B), (A-B), (A*B), A/B)	**Quiz 3 posted after class - due 9/4, 11:59 pm; Quiz 4 posted**
9/2/2019	**University Holiday, NO CLASSES**			
9/4/2019	Chapter 2 2.9, 2.10, 2.11, 2.13, 2.14	Chapter 2 slides 26-44	Code walkthru Chapter2Examples.java Write a program that calculates the average of 3 numbers. Numbers are doubles read as input. Write a program that takes a number representing seconds and displays the number of minutes and seconds (division and modulo)	HW1 DUE TONIGHT 11:59 pm; Quiz 5 posted
9/6/2019	Sprint Day		In-class programming assignment - Write a program that prompts user for distance, mpg, and price per gallon and displays cost. (pseudo coding followed by Java coding)	Coding & Code Tracing Sprint #1
9/9/2019	Wrap-up Chapter 2 Chapter 3 Introduction of Boolean logic and concept of an expression and relational operators 3.2	Chapter 2 slides 44-45 Chapter 3 slides 1-18	Code walk thru and revision MathPractice.java (casting numeric types)	Wrap up CH 2 & start CH 3
9/11/2019	Chapter 3 Recap of Boolean logic Introduction to IF and If..else statements In-class practice with IF; 3.3, 3.4	Chapter 3 slides 19-27	Code walk thru SimpleIfDemo.java CompareTwoNums.java CompareTwoNumsIfElse.java CompoundIf.java	HW2 Due; Quiz 6 posted after class
9/13/2019	No Class - at a conference			
9/16/2019	Chapter 3 Nested if, Dangling else, common pitfalls with if…else, Math.random() 3.5, 3.6, 3.7		walk thru and revision to RandomNumbers.java walk thru SubtractionQuiz.java	In class practice with random numbers and if...else
9/18/2019	Chapter 3 In-class coding and live coding of switch statement		ComputeBMI.java DailyPlanner.java walk thru ChineseZodiac.java	HW3 DUE; Wrap on CH 3; Switch stmt; In-Class coding of 4 problems; Quiz 7 posted
9/20/2019	Sprint Day		OneTensHundreds.java	Coding & Code Tracing Sprint #2
9/23/2019	Chapter 4 -Math functions, char datatype 4.2, 4.3	Chapter 4 slides 1-22	GuessNumber.java DessertCalculator.java (recap of switch) Code walkthru: MathFunctionExamples.java, MoreMathFuntions.java, ComputeAngles.java, CharacterExamples.java	
9/25/2019	Chapter 4 -Char data type continued and String datatype 4.3, 4.4 primitive datatypes vs. String type and memory storage	Chapter 4 slides 23-38	Code walkthru StringExamples1.java,OrderTowCities.java Live coding StringPractice.java	**HW4 DUE Quiz 8 posted**
9/27/2019	Chapter 4 - InClass Coding		Code walkthru StringExamples2.java Live coding MajorStatus.java, MonthDays.java	
9/30/2019	Chapter 4 WrapUp; printf; operator precedence, 4.6	Chapter 3 slide 73 Chapter 4 slide 40-42; 44	Code walkthru StringExamples2.java Common mistakes examples (; at end of if, dangling else, switch w/o default, printf errors) PrintFpractice.java Code walkthru FormatDemo.java	
10/2/2019	Chapter 5 Loops, motivation, while loop, 5.2, 5.3, 5.4, 5.5	Chapter 5 slides 2-15	Code walkthru PrintWelcome.java, PrintWelcomeLoop.java, RepeatAddictionQuiz.java, SentinelValue.java Live coding GuessNumberLoop.java	HW 5 DUE & Quiz 8 due 11:59 pm
10/4/2019	Sprint Day		In-class programming sort 3 numbers from smallest to largest; sort three strings from smallest to largest	Coding & Code Tracing Sprint #3
10/7/2019	Chapter 5 Crazy about While loops		Live coding RockPaperScissorsLoop.java, PrintWithWhile.java, CircleAreas.java	**Quiz 9 and 10 posted, SOFT Due Dates**
10/9/2019	Chapter 5 for-loops, 5.7	Chapter 5 slides 22-34	Code walkthru ForLoopFun.java, Live coding HeadsTailsLoop.java	**HW 6 DUE**
10/11/2019	Chapter 5 Converting from while to for and viceversa		Live coding Problem 10112019B.java, Problem10112019A.java, modification of LicensePlate.java to use loop	

10/14/2019	No class			NO Face-2-Face class
10/16/2019	Chapter 5 Do..While loop, 5.6, code tracing loops		Code walkthru DoWhileExample.java, LoopFun.java and LoopFun2.java	HW7 DUE
10/18/2019	Sprint Day		In-Class problem solving and coding ReverseString.java	**Coding & Code Tracing Sprint #4**
10/21/2019	Chapter 5 Nested loops; break & continue, 5.9, 5.12, 5.13	Chapter 5 slides 39, 46-48	Code walkthru MultiplicatonTable.java, TestBreak.java, TestContinue.java, Palindrome.java Live coding TestContinueNums.java	
10/23/2019	Chapter 6 Motivation for Methods, syntax,invoking,, 6.2, 6.3, 6.4	Chapter 6 slides 1-9	Code walkthru Motivation.java Live coding MethodFun.java	**HW8 DUE 10/24, 11:59 pm Thursday**
10/25/2019	Chapter 6 return type void, non-void return type, no argument method, method with arguments,	Chapter 6 slides 10-12	Live coding MethodFun.java, continued	Quiz 11,12, &13 posted Due 11/15
10/28/2019	Chapter 6 Pass by value,	Chapter 6 slides 13-24	Code walkthru TestVoidMethod.java, TestReturnGradeMethod.java, Increment.java, TestPassByValue.java Live coding CircleMethods.java	Quiz 14 posted Due 11/3
10/30/2019	Chapter 6 Scope, Abstraction 6.8, 6.6, 6.9, 6.11	Chapter 6 slides 25-29	Live coding MethodPractice1.java	HW9 Due Friday 11/1
11/1/2019	Sprint Day - no class; sprint online during class period			**Coding & Code Tracing Sprint #5 online; NO FACE-2-FACE Class**
11/4/2019	Chapter 6 Overloading, Scope 6.8, 6.9	Chapter 6 slides 33-41	Code walkthru TestMethodOverloading.java	
11/6/2019	Chapter 6 Scope, 6.9		Live coding ScopePractice.java	**HW 10 Due 11/11**
11/8/2019	Chapter 6		Live Coding PasswordChecker.java	Start CH 7. HW 10 DUE 11/11 11:59 pm
11/11/2019	Chapter 7 Intro One-Dimensional Arrays	Chapter 7 slides 1-11	Live Coding ArrayOfNumbers.java	
11/13/2019	Chapter 7 Looping and Arrays	Chapter 7 slides 31-36	Worksheet on Array Live Coding FindBigSmall.java AnalyzeNumbers.java	**HW 11 Due 11/17**
11/15/2019	Chapter 7		Problem Solving - Eight Ball Game	Coding & Code Tracing Sprint #6
11/18/2019	Chapter 7 Arrays and Methods	Chapter 7 Slides 12-15; 48-51;	Live Coding RollDice.java PassArrays.java	
11/20/2019	Chapter 7 Arrays and Methods	Chapter 7 Slides 55; 57;108	TestPassArray.java CountLettersInArray.java	**HW 12 Due 12/01**
11/22/2019	Chapter 7		InClass Problem Solving ExpandArray with int, with String	Coding & Code Tracing Sprint #7
11/25 - 11/29	**Thanksgiving Break**			
12/2/2019	Chapter 7		Recapping Arrays	
12/4/2019	Review for the Final Exam		Review for Final Exam	HW 13 Due 12/6
12/6/2019	Written portion of Final Exam In Class			
12/9/2019	**10 AM Section Final 1-3 pm**			
12/11/2019	**9 AM Section Final 8-10 am 11 AM Section Final Exam 1-3 pm**			
12/13/2019				

APPENDIX J

COURSE SYLLABUS

School of Computer Science and Mathematics
College of Health, Science, and Technology
University of Central Missouri
Course Syllabus

3 Credit Hours
3 Contact Hours

Course: CS 1100 - Computer Programming I
Semester: Fall 2019
Class Time: MWF 09:00 – 09:50 a.m. CRN 10692
MWF 10:00 – 10:50 a.m. CRN 10150
MWF 11:00 – 11:50 a.m. CRN 10898
Classroom: HUM 110

Instructor: Belinda Copus
Email: copus@ucmo.edu
Office Hours: MW 1:30 – 3 pm*,
Tues. 2 – 3 p.m.
Fri. 1 – 2 pm* & by appointment
Office: WCM 206C
Phone: 660-543-4354

*There will be times when I am called into a meeting during office hours on MWF, alternate times will be announced, or you may request an appointment.

Textbook : *Introduction to Java Programming, 11h edition*, by Daniel Liang, Pearson, 2017 (10th edition is fine too!)

Prerequisite: None.

Description: An introduction to computer programming in the structured programming paradigm using a modern high-level programming language. Topics include foundational programming concepts, data types, variables, operators, selections, loops, methods, and arrays.

Objectives:
1. Develop and analyze algorithms to solve problems.
2. Write programs to solve various problems.
3. Understand and use recursion.

Course Content Outline:
1. Introduction to Computers, Programs and Java
2. Elementary Programming
3. Selections
4. Mathematical Functions, Characters, and Strings
5. Loops
6. Methods
7. Single-Dimensional Array
8. Multidimensional Array
9. Recursion

ABET Outcomes:
Computer Science and Cybersecurity
SO2 - Design, implement, and evaluate a computing-based solution to meet a given set of computing requirements in the context of the program's discipline.

Software Engineering
SE SO1 - an ability to identify, formulate, and solve complex engineering problems by applying principles of engineering, science, and mathematics.
SE SO3 - an ability to communicate effectively with a range of audiences.

SE SO4 - an ability to recognize ethical and professional responsibilities in engineering situations and make informed judgments, which must consider the impact of engineering solutions in global, economic, environmental, and societal contexts.
SE SO6 - an ability to develop and conduct appropriate experimentation, analyze and interpret data, and use engineering judgment to draw conclusions.
SE SO7 - an ability to acquire and apply new knowledge as needed, using appropriate learning strategies.

Course Format This class is a lecture format with opportunity for hands-on in-class practice exercises. There will be online quizzes that must be completed outside of class and prior to the next lecture. These will usually occur after Monday and Wednesday lectures. In addition, you will need to complete outside homework with self-reflections, in-class programming exercises, and a final exam. There will be extra resources for content knowledge posted on Blackboard for your use.

Course Guidelines:

1. You are expected to attend all lectures. **Three** unexcused absences will be allowed during the semester. More than three unexcused absences will result in a deduction from the participation component of this course. Advance arrangements for unavoidable absence(s) will be made when possible. Contact me by email prior to the class you would miss.
2. You are expected to complete all programming assignments by the designated due date. Work is considered late if it is not submitted by the assigned deadline. Work submitted after the assigned deadline will receive a 10% deduction for one day late and, 20% deduction for two days late, 30% deduction for three days late, and will not be accepted after 3 days. Prior arrangement must be made with me and will only be allowed for extraordinary circumstances.
3. Please silence all cellphones and do not text, receive calls, or make calls during class.

Grading This course totals to a maximum 1000 points.
Homework (377 points) There will be 13 homework assignments worth approximately 29 points each. Each homework will consists of programming problems of 24 points and a self-reflection of 5 points. Homework must be submitted according to homework submission instructions to be eligible for full credit. Homework will be submitted through Blackboard. Self-reflections will be through a google link listed with the homework posting on Blackboard. Please see Course Guideline #2 regarding late policy.

Online Quizzes (160 points) There will be 20 online quizzes, 1-2 per week, in Blackboard, worth 8 points each. The quiz will cover material from the lecture. The quiz can be re-taken multiple times, until mastery of content is demonstrated. Quizzes for the week must be completed with an 85% or better in order to begin homework problems for the week.

Coding & Code Tracing Sprint (175 points) There will be 7 tracing and coding problems completed every other Friday during class. Each sprint is worth 25 points. The sprint will cover material from the current two-week period of content.

Final Exam (175 points) The final exam will be worth 175 points and will be comprehensive.

Attendance/Participation (53 points) There will be up to 53 points subjectively given toward expected attendance and class participation.

Misc. points (60 points): There will be pre-course and post course surveys to complete. Submission of the survey earns full marks. Each survey set (pre- post-) is worth 30 points.

Grading Scale (1000 points possible) A: 900-1000, B: 800-899, C: 700-799, D: 600-699, F: 0-599

Other Information and Policies

1. **Your UCM email account will be used, frequently, by the instructor, to communicate messages. It is your responsibility to check this account regularly.**
2. Class notes and assignments will be posted by email and on Blackboard. It is the responsibility of the student to frequently check their email and Blackboard for course changes and updates.
3. The assigned textbook is required, either physical or digital.
4. Completion of all homework is encouraged. Please submit your homework even if it is late.
5. Students with documented disabilities who are seeking academic accommodations should contact the Office of Accessibility Services, Union 222, 660.543.4421.
6. Advanced arrangement for unavoidable absences should be made whenever possible. Neither absence nor notification of absence relieves you of the responsibility of meeting all course requirements.
7. Make-up exams will be given only for valid excuses with proof and have to be completed within 5 days of the regular exams. A make-up exam will be different from the regular exam.
8. Homework is to be done independently unless otherwise directed.
9. Individual homework grading sheet will be shared via Google with the student and grade posted to the gradebook in Blackboard.
10. During tests, no communication tools are allowed. **Any form of academic dishonesty will be dealt with according to the guidelines found in the UCM Student Planner/Handbook or at http://www.ucmo.edu/student/documents/honest.pdf.**

REFERENCES

Abernethy, M. (2012). Reducing “Death by PowerPoint”. *Journal of Teaching and Learning with Technology*, 1(1), 63. Retrieved from https://scholarworks.iu.edu/journals/index.php/jotlt/article/view/2044

Adams, J. C. (2014). *Computing is the safe STEM career choice today*. Retrieved from https://cacm.acm.org/blogs/blog-cacm/180053-computing-is-the-safe-stem-career-choice-today/fulltext#

Adesola, S., Li, Y., Liu, X. (2019). Effect of emotions on student learning strategies. *ICEIT 2019: Proceedings of the 2019 8th International Conference on Educational and Information Technology*, 153-156.

Akaike, H. (1973). Information theory and an extension of the maximum likelihood principle. *Proceedings of the 2nd International Symposium of Information Theory*, 267-281.

Akinola, S. (2015). Computer programming skill and gender difference: an empirical study. *American Journal of Scientific and Industrial Research*, *7*(1), 1-9.

Bandura, A. (1986). *Social foundations of thought and action*. Englewood Cliffs, NJ.: Prentice Hall.

Bandura, A. (2008). An agentic perspective on positive psychology. In S. Lopez (Editor), *Positive psychology: Exploring the best in people*, Vol. 1, (p. 167-196). Westport, CT: Praeger Publishers.

Bareiss, R., & Radley, M. (2010). Coaching via cognitive apprenticeship. *Proceedings of the 41st ACM Technical Symposium on Computer Science Education*, 162-166.

Beaubouef, T., & Mason, J. (2005). Why the high attrition rate for computer science students: Some Thoughts and Observations. *SIGCSE Bulletin, 37*(2), 103-106.

Becker, B., Murray, C., Tao, T., Song, C., McCartney, R., & Sanders, K. (2018). Fix the First, Ignore the Rest: Dealing with Multiple Compiler Error Messages. *SIGCSE '18 Proceedings of the 49th ACM Technical Symposium on Computer Science Education*, 634-639.

Beggs, J., Bantham, J., & Taylor, S. (2008). Distinguishing the factors influencing college students' choice of major. *College Student Journal*, *42*(2), 381-394.

Bennedsen, J., & Caspersen, M. (2007). Failure rates in introductory programming. *ACM SIGCSE Bulletin*, *39*(2), 32-36.

Bennedsen, J., & Caspersen, M. (2019). Failure rates in introductory programming – 12 years later. *ACM Inroads*, *10*(2), 30-36.

Bergersen, G., & Gustafsson, J. (2011). Programming skill, knowledge, and working memory among professional software developers from investment theory perspective. *Journal of Individual Differences*, *32*(4), 201-209.

Bolting, G., Schneider, Y., Muhling, A. (2019). It's like computers speak a different language. *Proceedings of the 19th Kolli Calling International Conference on Computing Education*, 1-5.

Bouvier, D., Lovellette, E., Matta, J., Bedour, A., Becker, B., Craig, M., Jackova, J., McCartney, R., Sanders, K., & Zarb, M. (2016). Novice programmers and the problem description effect. *ITiCSE '16: Proceedings of the 2016 ITiCSE Working Group Reports*, 103-118.

Brown, N., & Altadmri, A. (2017). Novice Java programming mistakes: Large-scale data vs. educator beliefs, *ACM Transaction on Computing Education*, *17*(2), 1-21.

Bruner, J. (1960). *The process of education*. Cambridge, MA: Harvard University Press, retrieved from http://edci770.pbworks.com/w/file/fetch/45494576/Bruner_Processes_of_Education.pdf

Caspersen, M., & Bennedsen, J. (2007). Instructional design of a programming course – A learning theoretic approach. *ICER '07 Proceedings of the Third International Workshop on Computing Education Research*, 111-122.

Christ, T. (2014). Scientific-based research and randomized controlled trials, the "gold" standard? Alternative paradigms and mixed methodologies. *Qualitative Inquiry*, *20*(1), 72-80.

Collins, A., Brown, J., & Newman, S. (1987). *Cognitive apprenticeship: Teaching the craft of reading, writing, and mathematics*. Center for the Study of Reading Technical Reports. Retrieved from

https://www.ideals.illinois.edu/bitstream/handle/2142/17958/ctrstreadtechrepv01987i00403_opt.pdf?sequence

Collins, A., Brown, J, & Holum, A. (1991). Cognitive apprenticeship: Making thinking visible. *American Educator*, *6*, 38-46.

Code.org (2019). Retrieved from https://code.org/about

Corbett, C., & Hill, C. (2015). *Solving the equation: The variables for women's success in engineering and computing*. American Association of University Women. https://www.aauw.org/app/uploads/2020/03/Solving-the-Equation-report-nsa.pdf

Corney, M., Teague, D., Ahadi, A., & Lister, R. (2012). Some empirical results for Neo-Piagetian reasoning in novice programmers and the relationship to code explanation questions. *Proceedings of the Fourteenth Australasian Computing Education Conference*, 77-86.

Copus, B. (2015). Computer science and the hiring gap: Understanding why the gap exists. Unpublished manuscript, School of Computer Science and Mathematics, University of Central Missouri, Warrensburg, Missouri.

Cotner, S, Ballen, C., Brooks, D., & Moore, R. (2011). Instructor gender and student confidence in the sciences: a need for more role models? *Journal of College Science Teaching*, *40*(5), 96-101.

Deloatch, R., Bailey, B., & Kirlik, A. (2016). Measuring effects of modality on perceived test anxiety for computer programming exams. *SIGCSE '16: Proceedings of the 47th ACM Technical Symposium on Computer Science Education*, 291-296. doi: 10.1145/2839509.2844604

Espinosa, L. (2015, March). Where are the women in STEM? *Higher Education Today*. Retrieved from http://higheredtoday.org

CS Education Coalition (2016). An open letter to U.S. Senate: Every Student in America Should Have this Opportunity [web log post]. Retrieved July 7, 2020, https://www.change.org/p/offer-computer-science-in-our-public-schools-csforall

Fisher, K., & Kenny, S. (1986). The environmental conditions for discontinuities in the development of abstractions. In R. Mines & K. Kitchener (Eds.), *Adult cognitive development: Methods and models* (pp. 57-75). New York, NY: Praeger.

Griffin, J. (2016). Learning by taking apart: deconstructing code by reading, tracing, and debugging. *SIGITE '16: Proceedings of the 17th Annual Conference on Information Technology Education*, 148-153.

Guzdial, M. (2015). *Learner-centered design of computing education: Research on computing for everyone.* Williston, VT: Morgan & Claypool. doi:10.2200/s00684ed1v01y201511hci033

Hawi, N. (2010). Causal attributions of success and failure made by undergraduate students in an introductory-level computer programming course. *Computers & Education*, *54*, 1127-1136.

Herrmann, S., Adelman, R., Bodford, J., Graudejus, O., Okun, M., & Kwan, V. (2016). The effects of a female role model on academic performance and persistence of women in STEM courses. *Basic and Applied Social Psychology*, *38*(5), 258-268.

Hertz, M. (2010). What do CS1 and CS2 mean? Investigating differences in the early courses. *SIGCSE '10 Proceedings of the 41st ACM Technical Symposium on Computer Science Education*, 199-203.

Hertz, M. & Jump, M. (2013). Trace-based teaching in early programming courses. *SIGCSE '13 Proceedings of the 44th ACM Technical Symposium on Computer Science Education*, 561-566.

Hiebert, J., Carpenter, T. Fennema, E., Fuson, K., Human, P., Murray, H., Olivier, A. & Wearne, D. (1996). Problem solving as a basis for reform in curriculum and instruction: The case of mathematics. *Educational Researcher*, *25*(4), 12-21.

Horton, D. & Craig, M. (2015). Drop, fail, pass, continue: Persistence in CS1 and beyond in traditional and inverted delivery. *SIGCSE '15 Proceedings of the 2015 ACM SIGCSE Technical Symposium on Computer Science Education*, 235-240.

Kaczmarczyk, L., Petrick, E., East, J., & Herman, G. (2010). Identifying student misconceptions of programming. *SIGCSE '10: Proceedings of the 41st ACM Technical Symposium on Computer Science Education*, 107-111.

Kallia, M., & Sentance, S. (2018). Are boys more confident than girls? The role of calibration and students' self-efficacy in programming tasks and computer science. *WiPSCE '18: Proceedings of the 13th Workshop in Primary and Secondary Computing Education,* 1-4.

Kallia, M., & Sentance, S. (2019). Learning to use functions: The relationship between misconceptions and self-efficacy. *SIGCSE '19 Proceedings of the 50th ACM Technical Symposium on Computer Science Education*, 752–758.

Kinnunen, P., & Simon, B. (2010). Experiencing programming assignments in CS1: The emotional toll. *ICER '10: Proceedings of the Sixth International Workshop on Computing Education Research*, 77-85.

Kinnunen, P., & Simon, B. (2011). CS majors' self-efficacy perceptions in CS1: Results in light of social cognitive theory. *ICER '11 Proceedings of the Seventh International Workshop on Computing Education Research*, 19-26.

Knight, C. & Sutton, R. (2004). Neo-Piagetian theory and research: enhancing pedagogical practice for educators of adults. *London Review of Education*, *2*(1), 47- 60.

Kozuh, I., Krajnc, R., Hadjileontiadis, L., & Debevc, M. (2018). Assessment of problem solving ability in novice programmers. *PLoS ONE*, *13*(9), retrieved from https://doi.org/10.1371/journal.pone.0201919

Kramer, J. (2007). Is abstraction the key to computing? *Communications of the ACM*, *50*(4), 36-42.

Kumar, A. (2013). A study of the influence of code-tracing problems on code-writing skills. *ITiCSE '13: Proceedings of the 18th ACM Conference on Innovation and Technology in Computer Science Education*, 183-188.

Kumar, A. (2015). Solving code-tracing problems and its effects on code-writing skills pertaining to program semantics. *ITiCSE '15: Proceedings of the 2015 ACM Conference on Innovation and Technology in Computer Science Education*, 315-319.

Kutscha, T. (2017). *Towards a Practical Application of the Neo-Piagetian Theory for Novice Programmers*. (Master's thesis, Radboud University Nijmegen). Retrieved from https://www.ru.nl/publish/pages/769526/timkutschamasterthesis.pdf

Lishinski, A., Yadav, A., Good, J., & Endody, R. (2016). Learning to program: gender differences and interactive effects of students' motivation, goals, and self-efficacy on performance. *ICER '16: Proceedings of the 2016 ACM Conference on International Computing Education Research*, 211-220.

Lister, R., Adams, E., Fitzgerald, S., Fone, W., Hamer, J., Lindholm, M., McCartney, R., Mostrom, R., Sanders, K., Seppala, O., Simon, B., & Thomas, L. (2004). A multi-national study of reading and tracing skills in novice programmers. *SIGCSE Bulletin 36*(4), 119-150.

Lister, R. (2011). Concrete and other Neo-Piagetian forms of reasoning in the novice programmer. *ACE '11 Proceedings of the Thirteenth Australasian Computing Education Conference*, 114, 9-18.

Lockwood, P. (2006). Someone like me can be successful: Do college students need same-gender role models? *Psychology of Women Quarterly*, *30*, 36-46.

Loksa, D., Ko, A., Jernigan, W., Oleson, A., Mendez, C., & Burnett, M. (2016). Programming, problem solving, and self-awareness: Effects of explicit guidance. *CHI '16: Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*, 1449-1461.

Mascolo, M. (2015). Neo-Piagetian Theories of Cognitive Development. In J. Wright (Ed.), *International Encyclopedia of the Social & Behavioral Sciences* (pp. 501-510). Retrieved from https://www-sciencedirect-com.proxy.library.umkc.edu/science/article/pii/B9780080970868230973

McCracken, M., Almstrum, V., Diaz, D., Guzdial, M., Hagan, D., Yifat, B., … Wilusz, T. (2001). A multi-national, multi-institutional study of assessment of programming skills of first-year CS students. *ACM SIGCSE Bulletin*, *33*(4), 125-140.

McDowell, C., Werner, L., Bullock, H., & Fernald, J. (2006). Pair programming improves student retention, confidence, and programming quality. *Communications of the ACM*, *49*(8), p. 90-95.

McGill, T. & Volet, S. (1997). A conceptual framework for analyzing students' knowledge of programming. *Journal of Research on Computing in Education*, *3*, 276-298.

McLellan, H. (1994). Situated learning: continuing the conversation. *Educational Technology*, *34*, 7-8.

Merriam, S., Caffarella, R., & Baumgartner, L. (2007). *Learning in adulthood: A comprehensive guide*. San Francisco, California: Jossey-Bass.

Misa, T. (Ed.). (2017). *Communities of Computing: Computer Science and society in the ACM*. Association for Computer Machinery and Morgan & Claypool. doi: 10.1145/2973856

Morgado, C., & Barbosa, F. (2012). A structured approach to problem solving in CS1. *ITiCSE '12: Proceedings of the 17th Annual Conference on Innovation and Technology in Computer Science Education*, 399.

National Academy of Sciences, Engineering, and Medicine. (2018). *Accessing and responding to the growth of computer science undergraduate enrollments*. Washington, DC.: The National Academies Press. retrieved from https://doi.org/10.17226/24926

National Science Foundation, National Center for Science and Engineering Statistics. (2017). *Women, minorities, and persons with disabilities in science and engineering: 2017.* Special Report NSF 17-310. Arlington, VA. Available at retrieved from www.nsf.gov/statistics/wmpd/.

Ndum, E., Allen, J., Way, J., & Casillas, A. (2018). Explaining gender gaps in English composition and college algebra in college: The mediating role of psychosocial factors. *Journal of Advanced Academics*. *29*(1), 56-88.

Ogeyik, M. (2017). The effectiveness of PowerPoint presentation and conventional lecture on pedagogical content knowledge attainment. *Innovations in Education and Teaching International, 54*(5), 503-510.

Owolabi, J., Olanipekun, P., & Iwerima, J. (2014). Mathematics ability and anxiety, computer and programming anxieties, age and gender as determinants of achievement in basic programming. *GSTF International Journal of Computing*. *3*(4), 109-114.

Poulos, A., & Mahony, M. (2008). Effectiveness of feedback: the students' perspective. *Assessment & Evaluation in Higher Education*, *33*(2), 143-154.

Petersen, A., Craig, M., Campbell, J., & Tafliovich, A. (2016). Revisiting why students drop CS1. *Koli Calling '16 Proceedings of the 16th Koli Calling International Conference on Computing Education*, 71-80.

Pillay, N., & Jugoo, V. (2005). An investigation into student characteristics affecting novice programming performance. *Inroads – The SIGCSE Bulletin*, *37*(4), 107-110.

Qian, Y., & Lehman, J. (2017). Students' misconceptions and other difficulties in introductory programing: A literature review. *ACM Transactions on Computing Education*, *18*(1), 1-24.

Ramalingam, V., & Wiedenbeck, S. (1998). Development and validation of scores on a computer programming self-efficacy scale and group analyses of novice programmer self-efficacy. *Journal of Educational Computing Research*, *19*(4), 367-381.

Ramalingam, V., LaBelle, D., & Wiedenbeck, S. (2004). Self-efficacy and mental models in learning to program. *ITISCS '04: Proceedings of the 9th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education*, 171-175.

Rolka, C. & Remshagen, A. (2015). Showing up is half the battle: assessing different contextualized learning tools to increase the performance in introductory computer science courses. *International Journal for the Scholarship of Teaching and Learning*, *9*(1), article 10.

Rubio, M., Romero-Zaliz, R., Manoso, C., & Madrid, A. (2015). Closing the gender gap in an introductory programming course. *Computers & Education*, *82*, 409-420.

Sankar, P., Gilmartin, J., & Sobel, M. (2015). An examination of belongingness and confidence among female computer science students. *SIGCAS Computers & Society*, *45*(2), 7-10.

Sax, L., Lehman, K., & Zavala, C. (2017). Examining the enrollment growth: Non-CS majors in CS1 courses. *SIGCSE '17 Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education*, 513-518.

Sharmain, S., Zingaro, D., Zhang, L., & Brett, C. (2019). Impact of open-ended assignments on student self-efficacy in CS1. *CompEd '19*, 215-221.

Soloway, E., Ehrlich, K., Bonar, J., & Greenspan, J. (1982). What do novices know about programming? In A. Badre & Shneiderman (Eds.), *Directions in Human-Computer Interactions*, (pp. 27-54). Norwood, NJ: Ablex.

Stachel, J., Marghitu, D., Brahim, T., Sims, R, Reynolds, L., & Czelusniak, V. (2013). Managing cognitive load in introductory programming courses: A cognitive aware scaffolding tool. *Transactions on the SDPS: Journal of Integrated Design and Process Science*, *17*(1), 37-54.

Sweller, J. (1988). Cognitive load during problem solving: Effects on learning. *Cognitive Science, 12*, 257-285.

Teague, D., Corney, M., Ahadi, A. & Lister, R. (2013). A qualitative think aloud study of the early Neo-Piagetian stages of reasoning in novice programmers. *Proceedings of the 15th Australasian Computing Education Conference*, *136*, 87-95.

Teague, D., & Lister, R. (2014). Programming: Reading, writing, and reversing. *ITICSE '14 Proceedings of the 2014 Conference on Innovation & Technology in Computer Science Education*, 285-290.

Teague, D. (2015). Neo-Piagetian theory and the novice programmer (Doctoral thesis, Queensland University of Technology, Queensland, Australia). Retrieved from https://eprints.qut.edu.au/86690/1/Donna_Teague_Thesis.pdf

Turner, A. (1991). Computing curricula 1991. *Communications of the ACM*, *34*(6), 68-84.

United States Department of Education, National Center for Education Statistics. (2013). *STEM attrition: College students' path into and out of STEM fields*. Retrieved from https://nces.ed.gov/pubs2014/2014001rev.pdf

United States Department of Labor, Bureau of Labor Statistics. (2018). *Occupational Outlook Handbook*. Retrieved from https://www.bls.gov/ooh/computer-and-information-technology/home.htm

VanDeGrift, T., Bouvier, D., Chen, T., Lewandowski, G., McCartney, R., & Simon, B. (2010). Commonsense computing (episode 6): Logic is harder than pie. *Koli Calling '10: Proceedings of the 10th Koli Calling International Conference on Computing Education Research*, 76-85.

Veerasamy, A., D'Souza, D., & Laakso, M. (2016). Identifying novice student programming misconceptions and errors from summative assessments. *Journal of Educational Technology*, *45*(1), 50-73.

Veerasamy, A., D'Souza, D., Linden, R., & Laakso, M. (2018). The impact of prior programming knowledge on lecture attendance and final exam. *Journal of Educational Research*, *56*(2), 226-253.

Ventura, P. and Ramamurthy, B. (2004). Wanted: CS1 students. No experience required. *SIGCSE '04 Proceedings of the 35th SIGCSE Technical Symposium on Computer Science Education*. *36*(1), 240-244.

Vihavainen, A., Paksula, M., & Luukkainen, M. (2011). Extreme apprenticeship method in teaching programming for beginners. *SIGCSE 2011: Proceedings of the 42nd ACM Technical Symposium on Computer Science Education*, 93-98.

Vygotsky, L. (1978). *Mind in society: The development of higher psychological processes*. Cambridge, MA: Harvard University Press, retrieved from http://ouleft.org/wp-content/uploads/Vygotsky-Mind-in-Society.pdf

Watson, C. and Li, F. (2014). Failure rates in introductory programming revisited. *ITiCSE '14 Proceedings of the 2014 Conference on Innovation & Technology in Computer Science Education*, 39-44.

Watson, C., Li, F., & Godwin, J. (2014). No tests required: comparing traditional and dynamic predictors of programming success. *SIGCSE '14 Proceedings of the 45th ACM Technical Symposium on Computer Science Education*, 469-474.

Wilson, B., & Shrock, S. (2001). Contributing to success in an introductory computer science course: A study of twelve factors. *SIGCSE '01 Proceedings of the Thirty-Second SIGCSE Technical Symposium on Computer Science Education*, 184-188.

Xie, B., Dastyni, L., Nelson, G., Davidson, M., Dong, D., Kwik, H., Li, M., & Ko, A. (2019). A theory of instruction for introductory programming skills. *Computer Science Education*, *29*, (2-3), 205-253. DOI: 10.1080/08993408.2019.1565235

Zavala, L, & Mendoza, B. (2016). Precursor skills to writing code. *Journal of Computing Sciences in Colleges*, *32*(3), 149-156.

Zweben, S., & Bizot, B. (2019). *2019 Taulbee Survey: Total undergraduate CS enrollment rises again, but with fewer new majors; doctoral degree production recovers from last year's dip*. Retrieved from Computer Research Association website: https://cra.org/wp-content/uploads/2020/05/2019-Taulbee-Survey.pdf

VITA

Belinda Joan Copus was born in 1970 in Dallas, Texas. She graduated from Plano East Senior High School, Plano, Texas, in 1988. She attended the University of Texas at Austin and earned a degree of Bachelor of Science in Computer Science in 1993.

Ms. Copus worked many years in the telecommunications industry as a software engineer and co-launched a start-up company which produced workstation graphics subsystems for industrial applications. She later earned a Master of Science degree in Computer Science in May 2014.

Ms. Copus joined the faculty in the School of Computer Science and Mathematics at the University of Central Missouri in 2013 and currently serves as Assistant Professor. She has served as the program coordinator for Computer Science and Software Engineering programs since August 2017. In addition to teaching and program administration she serves on several committees within her department, college, and university, including Assessment, College Curriculum, and the Institutional Review Board.

Copus began work toward a Ph.D. in Curriculum and Instruction and Computer Science in 2015. Upon completion of her degree requirements, Ms. Copus plans to continue her career and service in higher education, to pursue research interests, and to continue to work with young students to introduce and develop interest in the field of Computer Science.

Ms. Copus is a member of the Association for Computing Machinery, Upsilon Pi Epsilon, Computer Science Teachers Association, Consortium for Computing Sciences in

Colleges, National Academic Advising Association, and Missouri Academic Advising Association.