Fall 1-1-2016

Attitudes of Pre-service Teachers Toward Computational Thinking in Education

Bekir Mugayitoglu

ATTITUDES OF PRE-SERVICE TEACHERS TOWARD COMPUTATIONAL THINKING IN EDUCATION

A Dissertation

Submitted to the School of Education

Duquesne University

In partial fulfillment of the requirements for

the degree of Doctor of Education

By

Bekir Mugayitoglu

December 2016

ATTITUDES OF PRE-SERVICE TEACHERS TOWARD COMPUTATIONAL

THINKING IN EDUCATION

By

Bekir Mugayitoglu

Approved October 24, 2016

_____	_____
Joseph Kush Professor of Instruction and Leadership in Education (Committee Chair)	David D. Carbonara Assistant Professor of Instruction and Leadership in Education (Committee Member)
_____	
Gibbs Kanyongo Associate Professor of Foundations and Leadership (Committee Member)	
_____	_____
Cindy M. Walker Dean and Professor School of Education	Misook Heo Director, Doctoral Program in Instructional Technology

ABSTRACT

THE ATTITUDES OF PRE-SERVICE TEACHERS TOWARD COMPUTATIONAL THINKING IN EDUCATION

By

Bekir Mugayitoglu

December 2016

Dissertation supervised by Dr. Joseph Kush

The purpose of the study was to examine the attitudes of pre-service teachers toward computational thinking, before and after an intervention, to convey the importance of integrating computational thinking into K-12 curricula. The two-week, course-embedded intervention introduced pre-service teachers, with varying academic specialties, to computational thinking practices and their utility. The intervention employed the Scratch programming language tool including Scratch flashcards, everyday and interdisciplinary examples of computational thinking, and unplugged activities. The findings indicated that the intervention was an effective new way to convey the value of computational thinking to all sampled pre-service teachers, no matter their academic specialties or GPAs. Further research is recommended to investigate potential increases in pre-service teachers' own computational thinking skills following from the intervention.

DEDICATION

To My Dearest Parents, Nese and Mehmet

ACKNOWLEDGEMENT

Though only my name appears on the cover of this dissertation, a great many people have contributed to the research on this dissertation.

First, I would like to express my deepest gratitude to my dissertation chair, Dr. Joseph Kush, for his excellent guidance, caring, patience, providing me with excellent atmosphere for doing research and also on-going support through communications.

I have greatly benefited from the guidance of the remainder of my dissertation committee for their insightful comments and encouragement, but also the hard question, which incented me to widen my research from various perspectives. I would like to thank to my dissertation committee members: Dr. Carbonara, and Dr. Kanyongo.

Dr. Carbonara was not only my dissertation committee member, but also tremendously helpful in completing my dissertation and developing the beginnings of my teaching academic career. He served as a teaching mentor through my doctoral program based on my dissertation topic.

I would be very remiss if I did not thanks to the generous stipend from my department allowed me to work with professors who I was research assistant with and gained experience from: Dr. Kush, Dr. Polat, Dr. Mahalingappa, and Dr. Reis.

My sincere thanks also goes to Mr. Shoop, Mr. Mckenna, Mr. Friez, and Dr. Alfieri who provided me an opportunity to join Carnegie Mellon Robotics Academy and Robomatter as an intern, and who gave access to practice about my dissertation.

Most importantly, none of this would have been possible without the love and patience of my family, I would like to thank my family: my parents; Nese and Mehmet, my brothers; Hakan and Hilmi sister; Dervisan, brother-in-law, Hilmi; sisters-in-law; Havva and Serife, nieces; Aysenur, Beyza, Elif, and Nisanur, nephews; Mehmet and Efe. My immediate family, to whom this dissertation is dedicated to, has been a constant source of love, sacrifice, support and strength all these years. I would like to express my heart-felt gratitude to my family.

Last but not least, many friends have provided much needed emotional support and encouragement. Thank you to everyone who has made this dissertation possible.

TABLE OF CONTENTS

Page

TABLE OF CONTENTS (continued)

TABLE OF CONTENTS (continued)

TABLE OF CONTENTS (continued)

LIST OF TABLES

Page

LIST OF FIGURES

Page

CHAPTER I

INTRODUCTION

Thinking, playing, and learning are the occupational activities for young learners to apply in their daily life – in school as well as outside the classroom**.** However, thinking, playing, and learning do not often happen in the traditional classroom (Papert, 2005). Programming language makes it possible for young learners to play while thinking and learning and they learn without even realizing they are learning. Learning a programming language has been shown to be one potential solution to assist students develop these skills however many pre-service teachers are not taught how to teach programming (Basawapatna, Koh, Repenning, Webb, & Marshall, 2011; Ottenbreit-Leftwich, Glazewski, Newby, & Ertmer, 2010).

Barack Obama has a statement to encourage those American youth to move quickly on programming, "Don't just play on your phone, program it". Brennan and Resnick (2012a) noted that young learners connect with computer for different reasons such as use social platforms to chat with their friends, watch various videos on YouTube, read articles on websites, and listen music, however, they do not have a chance to engage in creating and making via computer.

Definition of programming language

A programming language is a way to communicate ideas in a language between sender and receiver via codes that computer can understand such as languages that people speak to communicate with each other - English, Swahili, and Serbo-Croatian (Tipps, 1987). Computers speak multiple languages just like humans. A programming language is the way to speak to a computer with instructions that are understandable for both the computer and humans (Briggs, 2012). Programming language is the set of instructions that directs the computer hardware. It is

not the hardware, such as the wires, microchips, cards and hard drive, but the program that runs the hardware (Briggs, 2012).

Overview of Programming Language

Programming languages allow learners to create various projects such as games, animated stories, online news shows, book reports, greeting cards, music videos, science projects, tutorials, simulations, and sensor-driven art and music projects (Maloney, Resnick, Rusk, Silverman, & Eastmond, 2010). Almost all devices are we use on a daily basis are run by programming languages. If there is a lack of programming, they would completely stop or function less efficiently. Programming languages are used not just for personally owned computers but also for video game systems, cell phones, and the GPS, as well as our house devices we use everyday such as LCD TVs, remote controllers, DVD players, ovens, and refrigerators. Also, they are used for transportation devices such as car engines, traffic lights, street lamps, train signals, electronic billboards, and elevators.

Why K-12 students

Learning computer programming has been shown to have a positive impact on STEM education (Grover & Pea, 2013; Honey, Pearson, & Schweingruber, 2014). Children who learn computer-programming skills as part of a STEM curriculum have been shown to experience benefits to their education. For example, children may not understand or grasp the purpose of why they do math, as they are involved in the process of creating formulas for their projects. However, they can do just that with computer programming. Additionally, children are becoming more familiar, knowledgeable, articulated, and sophisticated about improving formal systems and are interacting with themselves and doing hands-on activity by thinking (Papert,

1980; Papert, 1993). Even for children who do not end up in STEM-related jobs, the inclusion of STEM curriculum in education will allow students to develop literacy in Science, Technology, Engineering, and Math and the critical thinking skills that are demonstrated by scientists, mathematicians, and engineers (Honey, Pearson, & Schweingruber, 2014).

Over the past decade, STEM-related (Science, Technology, Engineering, and Mathematics) jobs have increased at a rate greater than ever before (Langdon et al., 2011). STEM jobs are growing three times as fast as non-STEM jobs, with STEM workers also experiencing lower rates of unemployment. STEM jobs not only facilitate the growth of the American economy, but also provide new industries with a way to attract highly qualified workers. In a global market, STEM jobs pave the way for innovation and cutting-edge technological advances that make STEM jobs arguably the jobs of the future (Langdon et al., 2011).

Computational thinking reformulates complex and difficult problems into smaller and more manageable problems, which make it easier to solve (Wing, 2006). Computational thinking enhances human thinking by using imaginative ideas to create new things by using the computer or without computer. Computational thinking impacts many daily living skills and activities. Computational thinking is the most beneficial source to give children priceless power to invent and carry out projects with technological devices using through programming language (Papert, 1980). Computational thinking offers opportunities for students to engage in, “solving problems, designing systems, and understanding human behavior” through the same concepts as found in programming languages. It is impossible to not be affected by computational thinking while doing daily work (Voskoglou & Buckley, 2012; Wing, 2008a). Learning computational thinking

also teaches individuals problem-solving and logical thinking skills, which can generalize to many other areas, including reading and writing. However, students who are not strong problem solvers, despite having taken algebra and pre-calculus, can improve their problem-solving abilities through engaging in coding. Engaging in computational thinking is shown to increase the analytical and mathematical problem-solving ability of students (Wing, 2006).

Computer programming is best learned if introduced at a young age (Utting, Cooper, Kȍlling, Maloney, & Resnick, 2010). Children who learn programming languages at an earlier age are better at problem solving, decision-making, and computational thinking skills (Flannery, Silverman, Kazakoff, Bers, Bontá, & Resnick, 2013). Additionally, children who learn a programming language go through a similar process as those children learning a second language, with these skills leading them to become increasingly fluent with new technology. Having achieved fluency, children will better be able to express themselves and start expressing new ideas. It is paramount for coding teachers to begin teaching their students at an early age as a result of this process so closely mirroring the learning of a second language at that age. At an early age, children are becoming increasingly familiar with programming through hands-on and activities, which in turn shape the children's programming abilities. Moreover, it provides them with the foundation to explore programming language concepts, practices, and perspectives. They don't just learn the basics of programming, but become increasingly comfortable to use them and transfer these knowledge sets, knowledge, skills, and abilities to advanced programming, block-based. While learning programming, these children are also able to have fun exploring, playing, and creating their own products at early age (Wing, 2006).

Importance of Educational Programming Language

Learning a programming language provides young learners opportunities to create while expressing their thoughts, beliefs, and feelings in digital environments (Resnick et al., 2009; Wing, 2006; Wing, 2008b). With the knowledge, skills and hands-on activities of programming languages, young learners have a freedom in creating. Creativity skills develop and foster through programming language when a young learner builds various projects such as animations and simulations; designs interactive games; or makes a dynamic presentation. With hard work and practice, young learners build proficiency in their questioning skills and create projects with their self-expression. In the process of experimenting, young learners put their ideas into action and learn from their mistakes. For example, young learners are able to create individualized projects because possibilities are endless, they can create exciting things they want to program.

Young learners not only learn how to do programming, they learn other things with programming language (Resnick, 2013). For example, having young learners do programming to learn various contents such as math, science, art and other subject areas. They design games, simulations, animations, simulations or interactive stories by programming for peers that focus on the content they are learning. Similarly, utilizing the idea of programming in real life applications. As telling computer what to do, young learners can help other students learn procedures by giving peer commands (Wing, 2006). With this knowledge, young learners use computational thinking skills via concepts (sequence, loops, etc.) practices (testing, debugging, remixing, etc.), and perspectives to help them in real life. They increase their computational thinking skills via animations, simulations, dynamic and interactive content presentations, interactive stories, and games. A programming mindset will help students to tackle complex

problems by dividing them into smaller, more manageable sized units. In particular, tinkering with these activities improves their fluency of computational thinking and problem solving.

Programming is not an end by itself; students can use computational thinking to design iterative experiences, and become makers of technology products instead of consumers (Brennan & Resnick, 2013). Young people often do not engage in designing, but they like spending time on computers to watch videos, participate in social platforms to chat with their friends, and play games. Learning a programming language will empower young learners to create either their own project or software, and they will be able to create within an iterative design process (imagining, creating, playing, sharing, and reflecting). These students may no longer play games that were designed by somebody else, they would instead design the game they are interested in playing. Similarly, they would not merely watch an animation that was designed by somebody else, they will instead design their own personally interesting animations.

Why Pre-service teachers

At the present time, there are not enough teachers available to adequately teach programming languages to students (Stephenson, 2009; Tondeur, Van, Sang, Voogt, Fisser, & Ottenbreit-Leftwich, 2012). Programming languages are a part of a Computer Science major, but Computer Science graduates often prefer not to teach programming languages in the schools since it pays more to get a job in industry than it does to teach in a K-12 environment. And while most educators recognize the importance of incorporating programming or computational thinking into K-12 curricula, most classroom teachers are not adequately prepared to implement these activities. In fact, 9 out of 10 U.S. K-12 schools don't offer programming language classes (Partovi, 2015). To teach K-12 students in the beginning of their elementary school, computer programmers and software engineers are encouraged to teach them how to write and design

source code. According to the Code.org, there will be 1.4 million coding, engineering, and data mining jobs available by 2020 in the United States. Additionally, programming languages offer pre-service teachers the chance to become familiar with the essentials of programming concepts, practices, and perspectives and increases fluency with the thinking process of how to design, create, and express themselves (Kim et al., 2012). There is research to indicate that after learning how program, pre-service teachers are more knowledgeable and have more self-confidence (Al-Bow et al., 2009).

Although students have the opportunity to learn programming through technology and online resources, the importance of having teachers available cannot be overestimated (Utting, Cooper, Ko ̈lling, Maloney, & Resnick, 2010). Teachers not only teach and reinforce the fundamentals of computer programming, but also serve as catalysts to motivate, inspire, and guide students as they begin their computer programming journeys. Rather than serving as the sole educational guide, the wide variety of computer-based and iPad-based computer language programs should serve as supplemental resources in the classroom. Students all learn at different rates and through different means, which could be addressed through interactive and dynamic content taught by engaged teachers who are invested in their students. Having programming experts serving in a teaching capacity allows students access to those who have gone through the same process before them. As students work through tasks of increasing difficulty, from writing new code for their projects, to encountering and fixing bugs to run the program successfully, they need passionate and talented individuals in the field for students to learn programming on their own with online tutorials, but not everybody learns the same way and dedicated to keep at programming language. In particular, when students are required to write a new code for their projects, or encounter bugs to debug it, they subsequently might be less anxious and instead

become more passionate and dedicated. For this to occur, these students will need a teacher to show some hints or clearly explain where the problems are.

Significance of the study

This study examined the ways in which pre-service teachers understood the fundamentals of programming languages. Specifically, pre-service teachers were asked to design given hands-on and minds-on, learning activities with the goal of improving their ability to teach programming languages. Moreover, this research examined how higher education institutions provided programming course-training for pre-service teachers. It was expected that this study would assist pre-service teachers in their ability to integrate computational thinking concepts and practices into their curricula in support local and state school districts mandates.

Purpose Statement and Research Questions

The purpose of this study was to examine whether pre-service teachers attitudes and understanding would change if they were given computational thinking instruction. To this end, a computing survey was executed among pre-service teachers. Pre-service teachers were instructed in the computational thinking unit.

To address this research objective, the following research questions and hypotheses guided study:

Research Question 1: Can an embedded intervention that teaches about the importance and utility of computational thinking, change the attitudes of pre-service teachers enrolled in Instructional Technology courses?

H1: There is a statistically significant change in mean attitude scores toward computational thinking after receiving an embedded intervention on computational thinking within their Instructional Technology courses (One-way repeated measures ANOVA).

Research Question 2: Would the intervention on computational thinking affect the attitudes of pre-service teachers differently depending on their GPAs?

H1: The computational thinking intervention will show different patterns of effects that depend on whether the pre-service teachers in question have 3.5 – 4.0 range or 3.0 – 3.49 range GPAs (Split-plot repeated measures ANOVA).

Research Question 3: Are the attitude scores of pre-service teachers with STEM concentrations more subject to change after the computational thinking intervention than are the attitude scores of pre-service teachers with non-STEM concentrations?

H1: The computational thinking intervention will show different patterns of effects that depend on whether the pre-service teachers in question have STEM or non-STEM concentrations (Split-plot repeated measures ANOVA).

Research Question 4: Is the effect(s) of the computational thinking intervention on attitudes related to the gender of the pre-service teachers?

H1: The computational thinking intervention will show different patterns of effects that depend on whether the pre-service teachers in question are male or female (Split-plot repeated measures ANOVA).

Summary

Digital media offers children learning environments, like personal, real world, disciplinary and assessable. “Technology and education,” often means creating gadgets to teach the something with a little bit twist (Papert, 1980). Technology is not only a way for children to develop, but also influence and control them to apply for their projects so that they can be creator, designer, and problem-solver.

Programming and computational thinking allow them to create their own projects such as games, animations, and simulations. One reason of why there is less enrollment and diversity in CS (Computer Science) is that people believe only those people who are skilled at it could handle it (Burke & Kafai, 2010). Programming languages are difficult to learn and cope with it, but it can be fun. Most expert programmers are dedicated to learn and passionate about programming language (Hillegass & Ward, 2013).

CHAPTER II

LITERATURE REVIEW

History of Programming Languages

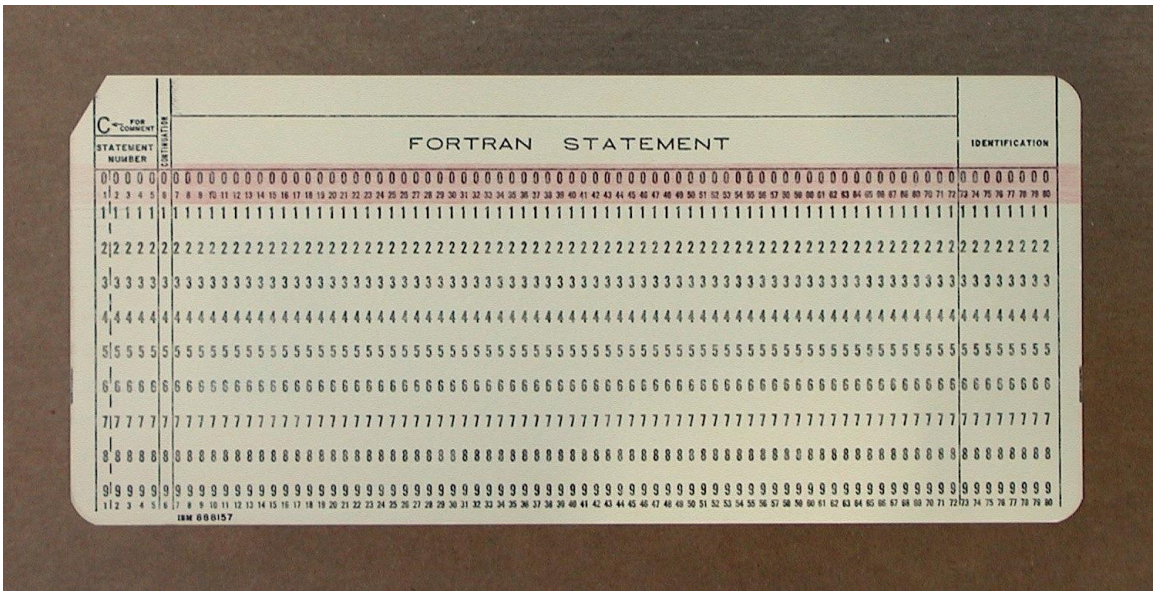
Punched card machines

Prior to the advent of modern programming languages, Herman Hollerith created the first punched card machine in the late 1880s (Driscoll, 2012; Kaur, Kumar, & Singh, 2014; Trikha, 2010). These machines were designed to encode information within each punched card for the United States government, which used punched cards for the first time for its census in 1890. The U.S. Constitution requires a census of its citizens to take place every 10 years, but it the process of using pen and paper was becoming quite difficult with the growing U.S. population and the process for conducting a comprehensive census was becoming increasingly difficult. The solution was to create a punched card system to collect the data, tabulate the count, and sort the information. Instead of the cumbersome pen-and-paper process, the new process was streamlined to be completed within a year, with punched card machines used to complete the census.

Hollerith's design became widely adopted across the country, and has served as the foundation of modern punched card technology (Allen, 1981; Elgamel & Sarrab, 2014). The first modern punched card technology began appearing in the late 1950's, beginning with the International Business Machines Corporation (IBM)'s development of the Formula Translating System (FORTRAN). FORTRAN punched card technology was specifically designed for scientific computing and was used mostly for math, science, and engineering purposes. It was particularly well-suited for scientific formulas, numerical analysis and technical applications due to its ability to express the way of complex mathematical functions similar with algorithmic

form, efficiently process mathematical equations, and incorporate complex number data type. Compared to the Hollerith system of the 1880's, FORTRAN was considered to be more efficient and easier. It provided punched cards that users could read easily with metal tabulators, because FORTRAN punched cards had rectangular holes, unlike, the round holed, Hollerith punched cards which were much more prone to reading errors. A final advantage was FORTRAN's larger storage capacity compared with the Hollerith system. FORTRAN had a storage capacity of 12 rows and 80 columns, whereas Hollerith only offered a restricted capacity of 12 rows and 24 columns.

Figure 1. Punched card



Although FORTRAN's simplicity greatly revolutionized punched card technology, the design was not conducive to business computing because FORTRAN was not dealing with a large amount of data (Wiemer, 2011). As a result, the Common Business Oriented Language

(COBOL) system was designed to explicitly meet this need. COBOL was created in 1959 by the Conference on Data Systems Languages (CODASYL) as a simple technology with a greater ease of use than FORTRAN. COBOL is considered to be a fairly easy to learn due to it containing an English-like syntax, compared to FORTRAN's non English-like grammar which made it is difficult to learn. Additionally, COBOL was considered to be more reliable than FORTRAN, while managing a larger amount of data information. Despite both systems having similar processes and portable features, COBOL punched card machines were smaller and faster than the FORTRAN predecessor. Finally, FORTRAN punched card machines were good at handling numbers, but was not good at handling input/output like COBOL punched card machines.

Punched card machines were replaced with computers in the 1960s (Black, 2013). However, the logic behind punched card machines encouraged people to develop object-oriented programming. Although punched card machines were easier to use in the early days than pen and paper, it was frustrating for programmers for several reasons. The first concern was that spending countless hours to locate punched cards and fix bugs was a time consuming process. Programmers weren't immediately informed about the bug when a problem occurred in the sequencing, thus the problems weren't addressed in a timely way. Also, punched card machines weren't suitably efficient to store and transform a large amount of data so it was necessary to have a large amount of machine memory. In addition, punched cards were vulnerable to repeated usage and the cards could easily get bent or damaged or the punched holes could become too large for the machine to read. For all these reasons, punched card machines were gradually replaced with more contemporary computing methods, such as object-oriented programming languages.

Object-Oriented Programming Languages

Object oriented programming is a language paradigm that one or more entities interact with one another to create models based on the real world. The goal is to provide reusable solutions for complex programs (Laffra, Blake, de Mey, & Pintado, 1995; Stroustrup, 1988). Although they solve same problems, object oriented programming languages are more efficient and faster than punched card technologies (Severance, 2012). Punch card technology does not allow the user to see the commands individually, while object oriented languages are written and shown line by line. This makes the read and write functions much easier for users. Moreover, debugging is simpler than object-oriented programming. That is, the process of debugging can be frustrating in punched card machines because if even one card contains an error or is out of sequence the program will crash. Object oriented languages, in contrast, provide feedback instantly.

Figure 2. Object-oriented programming language

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

int main() {
int number, reverse = 0;
 cout<<"Input a Number to Reverse:  ";
 cin>> number;

   for( ; number!= 0 ; )
   {
      reverse = reverse * 10;
      reverse = reverse + number%10;
      number = number/10;
   }
   cout<<"New Reversed Number is:  "<<reverse;

   return 0;
}
```

The object-oriented, programming language movement started in the early 1960s with Simula-67, which is known as the first object oriented programming language (Perez, Jansen, & Martins, 2012). Simula-67 was designed and implemented by Dahl, Myhrhaug, and Nygaard at the Norwegian Computing Center in Oslo particularly for the creation of simulations, computer graphics, and algorithms. Simula-67 introduced object-oriented programming concepts such as classes, objects, inheritance, and dynamic binding. When Simula-67 first appeared, it was elegant, powerful and very useful for software development, but it was too slow for practical use. Also, it was not open code and considered too complicated and had limited file access. Although the original concept of object orientation was simple and inspired with Simula-67, it soon gave way for more advanced, easier to use object-oriented programming languages.

Following Simula-67, other well-known object-oriented programming languages successfully combined the object-oriented approach with procedures such as C++ (Stroustrup, 1988). Simula-67's object-oriented features were a heavy influence in the development of C++ and later Java object-oriented programming language. Although Simula-67 was a groundbreaking object oriented programming language, it was not accepted as widely as C++ in the marketplace. Class structure in Simula-67 helped organize user's code, but the memory of program was not enough, so C++ was designed to be simplified and became beneficial with increased memory of program. In addition, C++ was seen as an improvement over Simula-67 in terms of making the code easier to get right so it avoided the ambiguities and was less error prone and easier to understand since semi dependent on machine. C++ was created at Bell Laboratories by the Danish Computer Scientist, Bjarne Stroustrup in 1983 for the UNIX system environment. It was so beneficial for programmers to improve the quality of code that reusable and produced code was easier to write by them. C++ was powerful and useful language created for specific reasons such as word processors, graphics, and spreadsheets. For this reason, C++ is a well-known object oriented programming language in worldwide.

Another current, well-known, object-oriented programming language is the Java programming language that was developed in 1995 by Canadian computer scientist James Gosling at Sun Microsystems. Java combined many of the features from the object oriented languages of its time such as Simula-67 and C++ (Singh & Abraham, 2014). For example, control flow constructs are totally identical in C++ and Java. While C++ is not platform independent, Java's object oriented programming language is platform independent, meaning that the written application or algorithms written for one platform will work just as well on other

platforms, such as PC, Windows and Linux**.** Additionally, Java contains an automatic debugging module, e.g., “garbage collector” that simplifies the process of cleaning bugs. Moreover, Java contains a larger library than C++ has a lot undefined behavior than Java so in Java debugging is significantly easier than C++ because it throws errors immediately and it is easier to trace bugs. C++. Java is also currently the most widely used object oriented programming language (Viennot, Garcia, & Nieh, 2014).

Text based programming language environments made major improvements to learning programming language in comparison to punch-card technology. For example, text-based programming language provided simplicity with syntax that was similar to English-like so that programmers could easily read and program it. In addition, it was easier to access with text based programming than punched card technology because cheaper to afford so that more people had a chance to learn programming and also took up less space such as punched card machines took up a whole room. However, text-based programming language was not easy enough for non-technological people, such as beginners and novice programmers who don’t have any prior programming knowledge and experience (Maloney, Peppler, Kafai, Resnick, & Rusk, 2008). For all these reasons, text based programming language were slowly replaced with more user friendly and intuitive technological environment, such as visual based programming languages.

Visual Based Programming Languages

The Visual based programming language is a paradigm that allows programmers to create projects by dragging and dropping blocks of code onto an editing center. As the name implies, visual based programming relies on GUI (graphical user interface). Its target audience is novice programmers and most visual based languages introduces the concepts of programming using the behaviors of simple elements such as movements, turns, loops, etc. Projects can be anything, such as animated stories, greeting cards, music videos, science projects, simulations, and music projects (Maloney, Peppler, Kafai, Resnick, & Rusk, 2008).

Visual based programming language provides a more suitable and simpler environment for young learners to express their interests than text-based programming language (Cooper, Dann, & Pausch, 2003; Maloney, Peppler, Kafai, Resnick, & Rusk, 2008; Resnick, 2007). In contrast to text-based programming languages, with visual-based programming languages, users do not just write tedious lines of code. Instead, they basically snap together a block of codes**,** without worrying about unfamiliar symbols such as semicolons, brackets, and parentheses. Moreover, the visualization of event-based programming is an easier way for children to understand the importance of events than text-based programming language. For example, as a program runs, users can observe which command is being executed, because the block of code is highlighted. Additionally, text-based programming language is complex by nature, and it is often difficult to debug code after it is written. In contrast, visual-based programming language is designed to be simple, because block of codes snap together in ways that make sense. Despite this simplicity, visual based programming language is still a powerful tool (Kelleher & Pausch, 2005). Programming language concepts (sequence, loops, parallelism, events, and conditionals) and practices (experimenting, iterating, testing, and debugging) are fundamentals of any

programming language regardless of whether it is visual based or text based. For this reason, visual based programming language helps learners to develop an intuitive sense of how sequences, parallelisms, and debugging work.

Visual based programming languages were inspired by the Logo turtle robot, created by Seymour Papert, Daniel G. Bobrow, Wally Feurzeig, and Cynthia Solomon in the late 1960s. It encouraged young learners to explore their ideas visually instead of typing (Bers, 2010; Papert, 1980) and was designed to be usable by both non-programmers and beginner programmers. Logo incorporated turtle graphics and offered instructions for movement and drawing line graphics either on the screen or with a small robot called a "turtle". The underlying rationale behind Logo was to understand the turtle's motion by asking users to imagine what they would do if they were a turtle.

Figure 3. Logo and the Turtle

All visual based programming languages originated from Logo, but each language has developed its own strengths and weaknesses, while still sharing some core commonalities. Late in the 1990s, a second visual based programming language was designed for novices who have little or no prior programming background by a research team at the CMU led by Randy Pausch (Dann et al., 2012). Alice allows novice learners to create games and animations with drag-and-drop blocks in an intuitive and user-friendly environment. Alice is an interactive 3-D graphical model and terrain visual programming language environment that lets users to drag and drop graphical tiles to create programs. The graphical tiles consist of the statements for the programs. Users have the options to pick the characters that occur on the stage, and then users add various rules to each character to build its behaviors, moves, and directions. The Alice programming environment was designed for several reasons: to teach programming concepts and theory, to support object-oriented flavor, and to encourage people to do programming with storytelling for

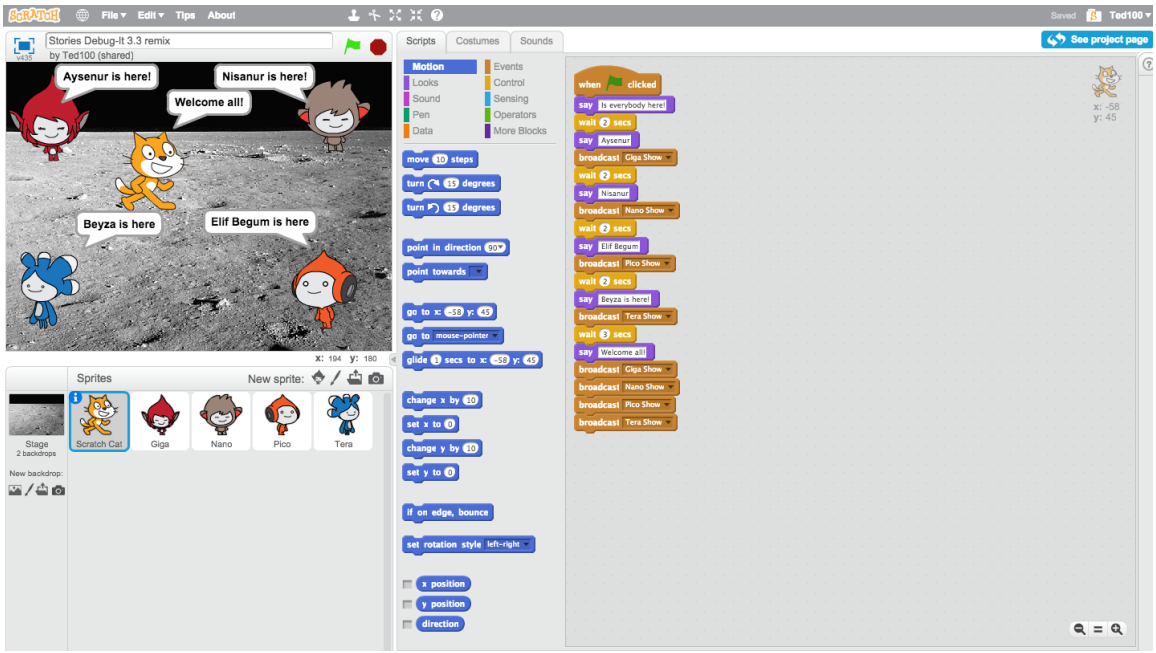
novice programmers and to complete programming practices such as sequencing, parallelism, automation, multimedia, program logic. Functions of Alice were built and inspired by object oriented programming language (Cooper, Dann, & Pausch, 2003). Logo was a powerful and fairly advanced programming language in its time, but it was often viewed as intimidating, not kid-friendly, and partly text-based so children and novice programmers were still had to worry about syntax error since spelling. In addition, Turtle was the only character, which was not interesting for some users and didn't give them a chance to choose different characters to design various projects. However, Alice made it easier and allowed users to choose the characters they wanted. Moreover, Logo had a few activities that users were limited to and it was not connected with their interests, needs and experience; for example, drawing simple geometric shapes. In contrast, Alice allowed users to do whatever they were willing to design that related with their interest and needs.

Figure 4. Alice

Scratch was created shortly after Alice. The Scratch software project was developed by the Lifelong Kindergarten group at the MIT Media Lab in 2007. Scratch was designed to foster collaborative work on a web browser platform. Accessing the platform through a web updates more projects instantly available for users so users always have the latest projects to remix. Novice programmers can use Scratch with visual block-based and drag-and-drop style to create animation stories, games, interactive presentations, music videos, and greetings.

Alice has a similar interface to Scratch, however Alice features slightly more advanced editing features and blocks of codes, so it is not easier for novice programmers and children to pick up programming concepts (Cooper, Dann, & Pausch, 2003; Resnick et al., 2009). Scratch is more widely used than Alice due to its simplified blocks, interface, and 2-D graphical environment that Scratch took from logo, and also replaced typing code style with a drag-and-drop block-based technique. Scratch is much easier to use than Alice because most novice programmers focus on 2-D, rather than 3-D graphical tools and terrain to create, import and personalize 2-D graphical tools (Burke & Kafai, 2012; Maloney, Resnick, Rusk, Silverman, & Eastmond, 2010). Moreover, Alice has not yet been translated into other languages so only English-speakers can use it, however, Scratch has been translated to around 50 different languages so that not only English speakers can learn, but non-English speakers, too (Resnick et al. 2009). Scratch allows users to share with other users, whereas Alice users can't share their projects with others since it has to be downloaded. Scratch online environment provides opportunity for users to develop sharing and socialization skills. Users can create their own projects, but also remix projects shared to the Scratch website by other users. Moreover, users make comments and answer questions to help other users.

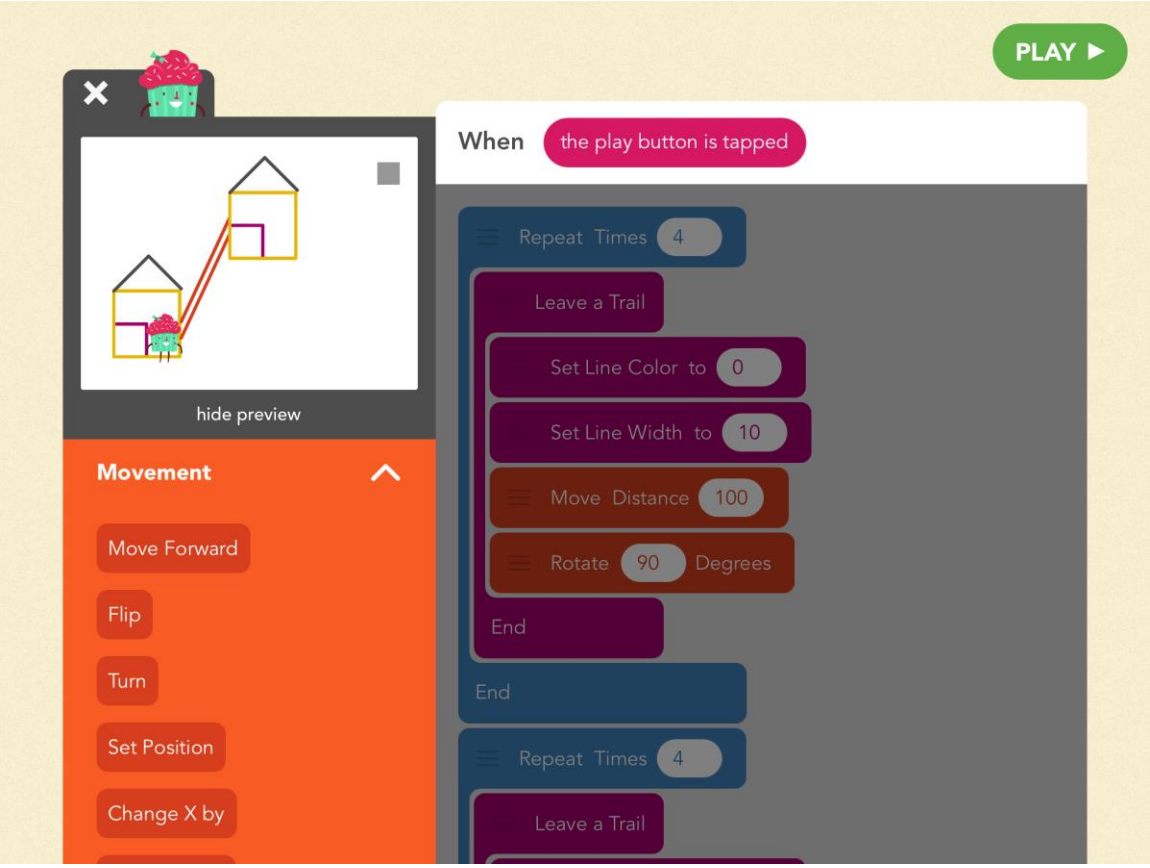
Figure 5. Scratch



Hopscotch is one of the first visual, tablet based programming languages. Hopscotch was designed in 2013 by Jocelyn Leavitt and was inspired by Scratch. The Hopscotch interface is very similar to Scratch, (e.g., Hopscotch works by dragging and dropping blocks of codes from the toolbox into the editing center) however, Hopscotch is specifically aimed at empowering and educating young males and females ages 8 to 12 them to teach how to create games and animation (Amer & Ibrahim, 2014). Hopscotch lets children share their projects within the Hopscotch community, which is an online environment where users connect with other users and write comments about projects. Hopscotch offers colorful blocks of code with which to execute a program on what is basically a blank slate so that it can be as easy or as difficult as users make it, but it also works under the assumption that they already know some programming basics.

Hopscotch smoothed the way with its kid-friendly interface, pre-built blocks, and tapping function, unlike Scratch, teaching younger children programming is difficult with computer interface since pointing and clicking are difficult for them to manipulate (Brennan, & Resnick, 2012b). Moreover, This visual based tablet based programming languages provide value for younger beginners at various stages of the learning process. Children become familiar with dragging and dropping coding blocks via various types of input, such as shaking an iPad, tapping the screen, and tilting the tablet.

Figure 6. Hopscotch



Another current tablet based programming language, ScratchJr, was developed by Tufts University as free source in 2014 (Portelance & Bers, 2015). ScrathJr allows young children between the ages of 5 to 7 to easily learn programming with a system based on Scratch. Hopscotch has many noticeable similarities to ScratchJr but also many different features. First of all, ScratchJr is highly focused on educating younger children who do not even know how to read and providing them the capability to communicate technologically in the modern world. Therefore, it is easier to use for young children with ScratchJr the basic skills for programming concepts, practices and debugging. For example, the graphic interface is very inviting and clear; the block of codes appears as colored icons that look like a jigsaw puzzle and link them together so that programs can be created. Colored icons are organized into color-coded categories such as one group of colored icons controls character looks. However, children have to know how to read in order to learn programming with Hopscotch. In addition, even though Hopscotch and ScratchJr are free to download and provide a rich selection of characters, not all characters are free in Hopscotch. For example, there are five additional characters (Mandrill, Miss Chief, Mosquito, Jeepers, and Venus) that can be purchased for $0.99 each. Unlike ScratchJr, all objects are free so that children have more objects to use they are interested in. Moreover, Hopscotch is available on iPad tablets, while ScratchJr is available on both iPad and Android tablets.

Figure 7. ScratchJr

Report on the programming language

According to the 2015 Searching for Computer Science: A Google Research Report: Access and Barriers in U.S. K-12 Education report indicates that K-12 teachers, parents, administrators, and superintendents think it is significant for students to learn programming. Students and parents also think learning to program helps them to find jobs. Ninety one percent of parents want their children to learn computer science and programming languages and approximately 66 percent of surveyed parents believe that computer science and programming should be mandatory in school, not elective or after school course. Based on U.S. Bureau of Labor Statistics data, the number of computer and mathematical jobs is expected to increase by 18% in the next 10 years. This means that 1.3 million job openings will be available by 2022.

Although K-12 teachers, parents, administrators, and superintendents agree that programming should be taught in K-12, most students don't have the opportunity to learn programming at schools in the United States for several reasons.

First of all, in-service teachers are often not qualified to teach computer science. Most K-12 in-service teachers either have not participated in a computer science coursework program or only a little bit of knowledge and experience (Google for Education, 2015; Ragonis, Hazzan, & Gal-Ezer, 2010). There are two options for qualifying to teach computer science: Earning a bachelor's degree in computer science or a relevant degree, or getting a certification. Computer science majors are qualified to teach, but they often prefer not to teach in K-12 settings as they typically get paid more at private companies as a programmer or developer. In addition, the benefits of working as a programmer are attractive since they don't have to take work home everyday and also they may receive double the pay. Some teachers are willing to get certified, but there is no path for them to apply to get certified. Thus, they don't know how to get certification for teaching computer science.

Moreover, school districts don't offer extensive training for their teachers who lack computer science skills since they don't have enough money. Therefore, teachers are not able to learn necessary computer science skills to teach their students computer science. Teachers are asked by administrators to teach programming to their students even if they aren't trained. Thus, teachers don't know how to teach programming language, they don't know what programming language to teach based on their grade level, and they don't know how to engage and motivate their students.

In addition, computer science courses are not mandatory at schools. Students are offered computer science courses as an elective or after school activity. Because computer science isn't one of the required courses for the graduation, students who take computer science don't pay the same attention as students who take core courses such as math, science, and social studies. Making computer science courses a mandatory rather than an elective or after school activity is a gateway to computer science and computer science related jobs. Additionally, mandatory computer science courses provide a great opportunity for schools to meet the STEM (science, technology, engineering, mathematics) requirements.

Next, there are not enough computers and tablets for students and teachers to access computer technology at home and school due to the shortage in budgets. In particular, poor districts don't have enough money to buy computers. Due to students not having computers and tablets, they don't have opportunities to explore programming language tools. Not only that, teachers don't have computers and tablets to access, in and out of school, to practice programming before teaching their students. Without this practice, teachers don't have a chance to create hands-on activities and see sample projects. They also don't have an opportunity to access important resources to share with their students, such as programming language flash cards, quizzes, and articles.

In addition to that, there are inequalities between students' economical situations. In particular, underrepresented groups such as women, lower-income, Hispanic and black students have less access to computer science out of school than white students. Moreover, Hispanic and black students have less opportunity to access the Internet out of the classroom setting than white students. In particular, underrepresented groups are not able to get resources, activities, and

sample projects. For these reasons, underrepresented groups are not provided with computer science out of classroom settings. Therefore, underrepresented groups are much less likely than white males to major in STEM or STEM-related fields. Women especially are underrepresented in most science, technology, engineering and mathematics majors (Google for Education, 2015; Yadav, Mayfield, Zhou, Hambrusch, & Korb, 2014).

Finally, programming is not taught as a part of computer science class in K-12 since it is not part of computer science curriculums. Most computer science classes are not taught programming, but basic computer keyboarding skills such as Microsoft word and Microsoft power point rather than programming language. Therefore, computer science curriculums focus on how to use software tools, but computer science curriculums don't focus on creating, making, and designing new projects such as animations, simulations, and games.

Learning Theories that Relate to Computational Thinking

Learning theories (behaviorism, constructivism, and constructionism) are sets of ideas to explain pedagogical approach to effectively and efficiently teach students how to think computationally while programming (Bers, 2008; Brennan, & Resnick, 2012a; Stetsenko, & Arievitch, 2004; Vygotsky, 1978; Wing, 2006). According to these theories, students automatically engage in computational thinking while programming. Moreover, these theories help teachers encourage students to use tangible programming language tools and intangible computational thinking steps. Therefore, these theories demonstrate how computational thinking and programming language tools can be used and taught to students in classroom environments.

Theory of Behaviorism

Behaviorism is a learning theory that frames learners as passive recipients of stimuli, who are responding the environmental in the process of learning (Cautili, Rosenwasser, & Hantula, 2003). Behaviorism was coined by John Watson in 1913 and then popularized by B. F. Skinner in 1948. According to behaviorism, learners begin life with a tabula rasa or blank slate. This means that the mind lacks experience, so learners have a fresh start. Behavior can be impacted by both positive and negative stimuli provided by the environment. Behaviorism is focused more on observable behavior, and minimizes the importance of intrinsic processes, such as thinking, understanding, interpreting and knowing.

In behaviorism, the process of learning and classroom instruction is teacher-centric and emphasizes rote memorization (Zeidler, 2002). Behaviorism doesn't encourage learners to understand concepts deeply, but rather, rewards students that give correct answers when assessed by teacher. In a behaviorist approach, students practice to avoid giving false answers on drill and practice activities during class time.

Theory of Constructivism and Constructionism

Behaviorism was replaced by constructivism in the twenty-first century because it was unable to address intangible computational thinking steps. More importantly, with constructivism, students become more active in the learning process and were taught to construct their own understanding and knowledge. In other words, constructivism encourages students to construct knowledge in their head (Alessi & Trollip, 2001).

Constructivism, which was first developed by Jean Piaget in the 1930s, asserts that learners actively construct their own learning experience, understanding, and knowledge

(Jonassen, 2000). Constructivism offers a sharp contrast with behaviorism, as the learner is more actively engaged in the learning process. With a constructivist approach, learners synthesize their own understanding and knowledge with real life experiences and reflect on them. In the long run, learners develop their own point of view, and unique interpretation of the world. Constructivism is a learning theory that builds upon learner's prior knowledge and experiences (Bednar et al., 1992; Bers, 2008; Mascolo & Fischer, 2005; Piaget & Indelder, 1969).

Constructionism is also a theory of "Learning by doing" where the learner relies on tacit knowledge, such as programing on computers, tablets, program robots (Papert, 1980; Papert & Harel, 1991; Resnick, Bruckman, & Martin, 1996). The theory of Constructionism was coined by Seymour Papert in the 1980s. This method focuses on the belief that students learn best when working on project with peers, learning from their peers, and interacting with the real world.

Constructionism brings both constructivism and tangible programming language tools into the process of constructing understanding and knowledge and then thinking computationally (Bers, 2008; Resnick, 2007; Resnick et al., 2009). In particular, constructionism empowers students in the use of programming language tools so that they can create and design artifacts based on their interests (Papert & Harel, 1991). Thus, constructionism encourages students in creative artifacts with the programming tools.

Both theories of learning believe that individuals create meaning from different experiences and previous knowledge **(**Kafai & Resnick, 1996). Constructivism and constructionism are similar learning theories, but they also have differences. The main difference between them is that Piaget believes that learning is dependent on the development of mental functions, however, Papert believes that learning is depend on the development of physical objects with hands-on activities such as programming, robotics. Hence, Piaget focused on

learning process by more mental constructions, Papert focused on learning process more by physical. According to the Papert (1980), children "learn by doing", involves collaboration and interaction between teacher and students that projects can be shared with peers and get feedback from peers as a way to build meaning. A similarity between two theories, however, is that both emphasize discovery methods of learning that let learner explore and experience projects by himself based on their interests. Moreover, students are facilitated and coached by their teacher while working on their projects instead of getting the correct answer. Therefore, teachers are not dictating their ideas, but rather discussing them with their students.

Why Constructionism?

The application of behaviorist principles in education began to wane in the twentieth century, and constructionist principles began to replace behaviorism in the twenty-first century (Duit & Treagust, 1998; Jenkins, 2000). The principles of behaviorism don't work well for computational thinking for several reasons, including the role of the teacher, the role of the student, and collaborative learning. The principles of constructionism bridge this gap for students, who can use computational thinking and develop new knowledge with coaching by teacher with their peers (Honebein, 1996; Papert, 1980; Rummel, 2008).

Behaviorism becomes teacher-oriented (Bush, 2006). In contrast, constructionism can play an important role for teachers and students in the classroom environment. According to constructionism, teaching becomes learner-oriented. Learners are active participants, not passive recipients in the process of learning, therefore offering learners a more active role in the classroom setting (Fosnot, 1996). Hence, learners are more engaged and motivated (Papert, 1993). In behaviorism, teachers dictate and lecture. Whereas, in constructionism, the teacher does not give too much information at one time, but acts as facilitator, mentor, and listener

(White-Clark, DiCarlo, & Gilchriest, 2008). In this method, the main role of teacher is coaching for learners through the process of learning (Papert, 1980; 1993). In addition to that, learners are not creative in the process of learning in behaviorism since they have no chance expressing their creativity. However, constructionism, learners have a deep understanding and know information better than behaviorism since constructionism encourages learners to try out new things, and draw conclusions (Bers, 2008; 2010). In behaviorism, learners are doing what they are told to do by teacher. The teacher provides the correct answer directly without scaffolding when students are not able to solve problems. However, in constructionism, if learners are struggling with given project, teachers don't give them the right answer directly, but scaffolding them if they have any problems or questions (Sutton, 2003). This process is known as scaffolding, which is the way that teachers help students to move from the inability to perform given project to being able to so through coaching or facilitating (Blake & Pope, 2008; Stetsenko, & Arievitch, 2004; Vygotsky, 1978). For example, a student is struggling to learn how to create a game. By working with student to teach how to add blocks of code and add a new character, the student is able to learn to create a game. Therefore, teacher let students make their projects based on their creativity, imagination, and ideas in constructionism.

Behaviorist teachers give information in front of a classroom to tell students what to do for specified project and how to design project (Shield, 2000). Each student listens and repeats what the teacher told them to do step-by-step. Projects are revised based on feedback of their teacher. Therefore, the students don't interact with peers to brainstorm and come up with the new ideas. As opposite to behaviorism, constructionism, the teacher offers opportunities for learners to work and interact with peers on a collaborative team environment (Draper, 2002; Rogoff, 1994; Slavin, 1990). Teachers encourage learners to work with peers in an authentic

environment. Therefore, learners are allowed to interact with each other, exchange big ideas, share different experiences, and construct meaningful knowledge together. In this environment, they discuss and debate, connects the dots of project parts, discover new things, and draw conclusions. For these reasons, the behaviorist approach is not really the most efficient and beneficial way to teach. However, there is one positive factor of behaviorism that is rote memorization. There are many strategies of learning, but rote memorization is the best way and easiest way to learn fundamental terms and facts. Once learners memorize fundamentals, they are able to use their information for building meaningful learning. For example, memorizing a bunch of vocabulary words, alphabet or verb forms are the fundamental concepts and terms build on four skills, speaking, writing, reading and writing for second language learners and also memorizing the basic math facts in addition, subtraction, multiplication, and division are the essential ingredients for learners to make an animation.

Computational thinking

Constructivism, constructionism, and behaviorism are the gateways for helping students learn about computational thinking concepts and steps. But most importantly, constructionism invites students to participate in creating, making, and designing while programming to think computationally (Papert & Harel, 1991). Therefore, students have an opportunity to build and develop a strong mindset. In other words, students develop a deeper understanding of computational thinking.

The term "computational thinking" was first coined by Jeannette Wing in 2006 (Wing, 2006). The meaning of “Computational thinking involves solving problems, designing systems, and understanding human behavior, by drawing on the concepts fundamental to computer science”. Brennan and Resnick (2012a) described computational thinking with dimensions of

computational thinking concepts and practices. Computational concepts are the fundamentals of computational thinking. Computational practices put computational thinking concepts together to design projects. For example, Computational thinking concepts sequence, loops, events, parallelism, and conditionals are used for projects to make program runnable with computational practices such as iterating, debugging, testing, remixing, abstracting.

Computational thinking is the new literacy technique of the twenty-first century to teach children the process of thinking abstractly. Computational thinking consists of many aspects, such as problem decomposition, pattern abstraction, and algorithm design (Google for Education, 2015; Wing, 2006). Computational thinking not only plays an important role as a fundamental part of computer science, but also influences problem solving in all disciplines such as economy, art, and engineering and in the real-life world (Bundy, 2007).

Wing, Google for Education and The Computing at School (CAS) all addressed how computational thinking should be approached. Wing created the idea of a computational thinking approach (Google for Education, 2015; Wing, 2006; Wing, 2008a). However, Google for Education and CAS builds on Wing's work in the practical world. They developed a plan to lay out the steps of computational thinking for integrating into K-12 classrooms (Google for Education, 2015). Google for Education has four basic steps that include decomposition, pattern recognition, abstraction, and algorithm design. Additionally, based on CAS, there are six basic steps that include decomposition, patterns, abstraction, algorithms, logical reasoning, and evaluation.

CAS and Google for Education are approach computational thinking steps through similar processes (Google For Education, 2015). However, the numbers of steps are different.

Critics argue that the number of steps CAS is better than Department for Education because it is more deeper. These two extra steps allow students to make a prediction of output and also review their process.

The first step of computational thinking is *decomposition*: taking a big, difficult, and complex problem and breaking it down into smaller, more manageable sub-problems. When problems are broken down into smaller pieces, the next step is *patterns*. This step allows people to identify common similarities and differences, the next step is *abstraction.* This step provides people with a way to create step-by-step techniques for solving problems. Finally, *algorithm design* provides significant instructions with a step-by-step solution for a problem and pulling out significant details to find one solution that applies multiple similar problems. *Logical reasoning* allows students to predict what the result will look like after following 4 steps. In other words, the sequence of instructions will let students know the results. *Evaluation* allows students to make sure each step of Computational thinking works well. If the evaluation doesn't show what students predicted, it allows students to restart process (Department for Education, 2013).

For example, cargo companies go to several locations to drop off goods for people. When a company has a bunch of goods that must be delivered to numerous customers, it needs to effectively and efficiently deliver them**.** *Effectively and efficiently* means finding the shortest route so that the company can travel the shortest time as well. There are too many streets, houses, offices, and so on. The first aspect of computational thinking, which is decomposition, is the first step to approach the problem. In the city, there are fifty districts, so it would be difficult to solve the problem since there are too many districts. The decomposition approach breaks the large number, fifty districts, into smaller pieces, one district, which is easier to concentrate on. But

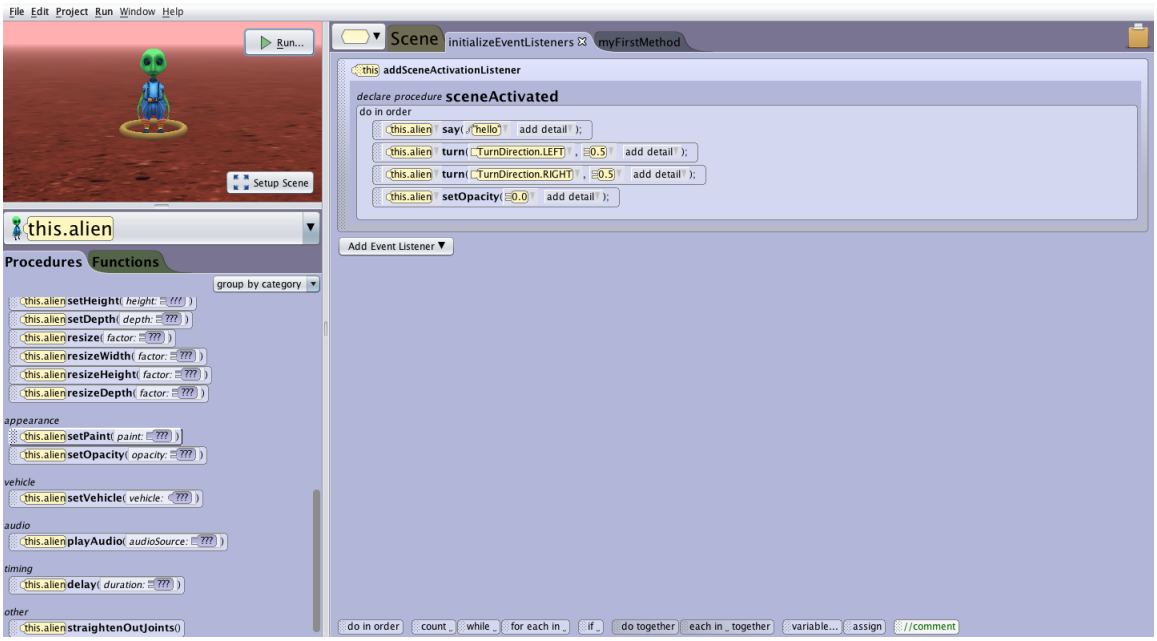
even one district has too many houses, offices, places, and so on. They need to be abstracted to ignore irrelevant details and to focus on the key parts. Next aspect of computational thinking is the creation of a series of instructions for this problem and the solving of similar problems with other districts. The next aspect of computational thinking is what an output exactly will do. The final step of computational thinking is if a series of instructions are still not working appropriately while evaluating, return back the first aspect of computational thinking, which is decomposition.

Computational thinking concepts

Sequences

A sequence is a list of code blocks that are put in a specific order to be run by a computer. As an example, the figure below, presents an Alice project and includes a list of code blocks. Each block code manipulates the alien based on the sequence. There are 4 code blocks on the list to produce the program. The first action instructs the alien to say, “Hello”, and the second block code instructs the alien to turn left. After turning left, the alien turns right. The last block code has the alien disappearing.

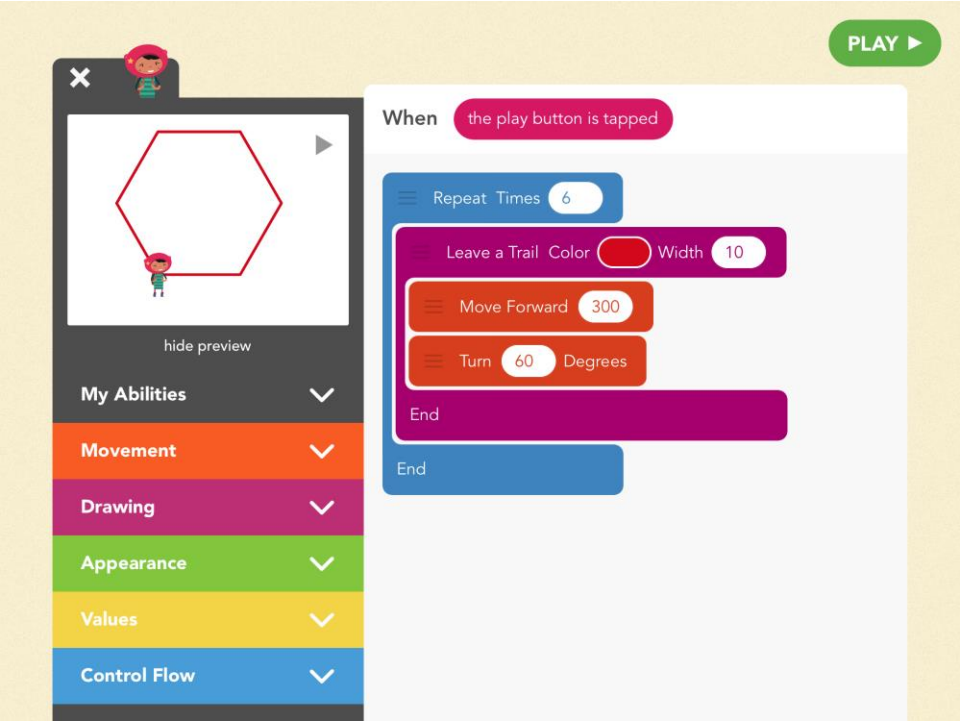
Figure 8. Alice sequences program example



Loops

A loop allows a programmed sequence of instructions to repeat multiple times. In the figure below, the project is designed by Hopscotch. The C shape is the repeat block that lets the character run the same instructions or block code stack several times based on the number in the blank box. In this example, the C loop has three blocks in which the instructions “Leave a trail color orange and 10 width”, “Move forward”, and “Turn 60 degrees repeat 6 times” occur in sequence when the play button is tapped.

Figure 9. Hopscotch loops program example



Parallelism

Parallelism allows several tasks to run at the same time. In the figure below, the project is designed by ScratchJr. There are two green flags for the same character. When the user clicks the green flag, both instructions start at the same time. Therefore, the sounds play forever while the giraffe moves 5 pixels 6 at the same time.

Figure 10. ScratchJr parallelism program example

Events

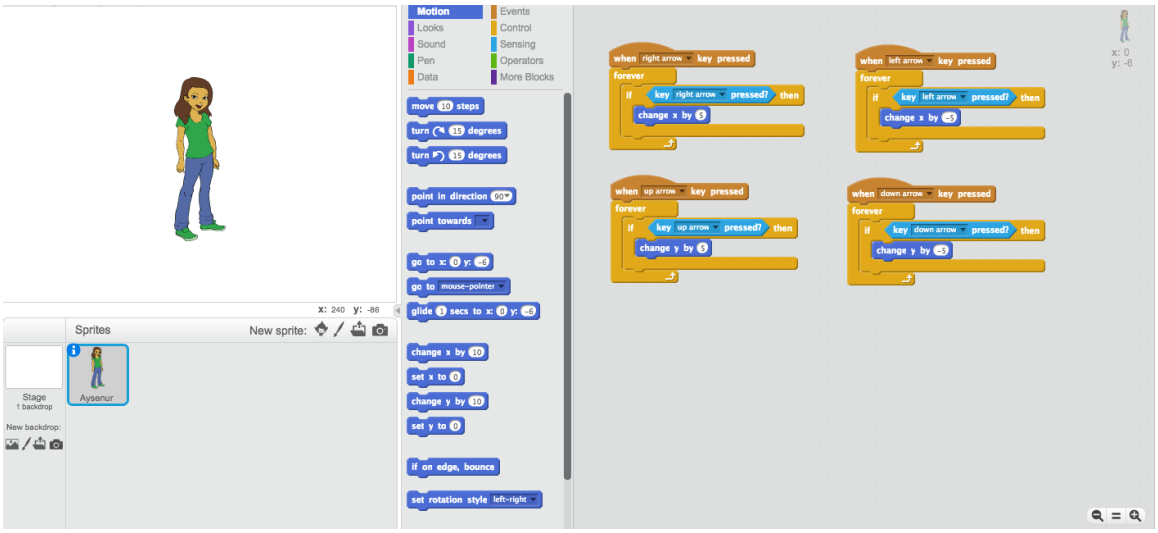
One thing starts happening because another thing is triggered. In the figure below, the project is designed by ScratchJr. When the yellow fish is tapped by the user, the yellow fish says, “Hello”. If the yellow fish is not tapped by user, there is no greeting by the yellow fish.

Figure 11. ScratchJr events program example

Conditionals

One thing occurs depending on the situations of other things. In the figure below, the project is designed by Scratch. The character has four events; when the right arrow key is pressed, when the left arrow key is pressed, when the up arrow key is pressed, and when the down arrow key is pressed. Each of them has a conditional statement which is an “if then” statement. If the user presses the right arrow, the character moves right. If the user presses the left arrow, the character moves left. If the user presses down, the character moves down. If the user presses the up arrow, the character moves up.

Figure 12. Scratch conditionals program example



Computational thinking practices

Computational thinking practices allow learners to experiment. They use computational thinking concepts to arrange a specified instruction. Computational thinking practices provide learners the opportunity to try the instruction out to see whether it works or not. Also, learners have an opportunity to debug the program since it might not be the result he wants.

Summary

I began this dissertation with a history of programming languages. In this history, computational thinking evolved recently with the research by Jeannette Wing. Computational thinking is a technique in which, students and teachers use different programming language tools such as Logo, Alice, Scratch, Hopscotch, and ScratchJr. Computational thinking pushes students to solve complex problems by working through them with a variety of strategies and steps. This dissertation was built on constructionist theory principles because computational thinking is the

best fit for this theory. This study offered significant information for pre-service teacher educators exploring computational thinking. In other words, this study could aid teacher educators who will become the models for students of tomorrow. In addition, this study provided recommendations for how institutions could provide training in computational thinking for pre-service teachers. It served as a future reference for teaching programming languages and computational thinking to pre-service teachers. The following chapters would demonstrate how this dissertation would help me advance this aspect of the education field and examined the attitudes of pre-service teachers and their understanding of computational thinking.

CHAPTER III

METHODOLOGY

Introduction

This chapter describes the methodology that was used in this study. The research questions and hypotheses are followed by a description of participants, research instruments, and procedures that were instituted to carry out the study, and the statistical procedures that were used to analyze the data. The purpose of this study was to examine pre-service teachers' attitudes toward computational thinking.

Research questions and Hypotheses

Research Question 1: Can an embedded intervention that teaches about the importance and utility of computational thinking, change the attitudes of pre-service teachers enrolled in Instructional Technology courses?

H1: There is a statistically significant change in mean attitude scores toward computational thinking after receiving an embedded intervention on computational thinking within their Instructional Technology courses (One-way repeated measures ANOVA).

Research Question 2: Would the intervention on computational thinking affect the attitudes of pre-service teachers differently depending on their GPAs?

H1: The computational thinking intervention will show different patterns of effects that depend on whether the pre-service teachers in question have 3.5 – 4.0 range or 3.0 – 3.49 range GPAs (Split-plot repeated measures ANOVA).

Research Question 3: Are the attitude scores of pre-service teachers with STEM concentrations more subject to change after the computational thinking intervention than are the attitude scores of pre-service teachers with non-STEM concentrations?

H1: The computational thinking intervention will show different patterns of effects that depend on whether the pre-service teachers in question have STEM or non-STEM concentrations (Split-plot repeated measures ANOVA).

Research Question 4: Is the effect(s) of the computational thinking intervention on attitudes related to the gender of the pre-service teachers?

H1: The computational thinking intervention will show different patterns of effects that depend on whether the pre-service teachers in question are male or female (Split-plot repeated measures ANOVA).

Participants

Participants for this study were pre-service undergraduate students, enrolled in an Instructional Technology course, within the School of Education, at a private university in the Eastern, U.S. during the spring semester of 2016. Class size was dependent upon enrollment for the semester and ranged between 15 to 20 students. Ten classes of pre-service teachers were invited to participate in the computational thinking unit.

The computational thinking instructional unit was presented to all pre-service teachers in each of the classes. Pre-service teachers who do not agree to participate in the research aspect of the project were still participated in the computational thinking unit instruction but were not

asked to complete either of the pre- or post-test questionnaires. The purpose of the study was explained during the first unit instruction and all pre-service teachers received consent forms indicating that their participation was entirely voluntary and would in no way influence their grade in the class. Pre-service teachers were also informed that all data collected would maintain confidentiality and anonymity.

Over the semester's computational thinking unit, pre-service teachers were instructed for a total of two 50-minute sessions. Pre-service teachers who withdrew from the Instructional Technology course during the experimental period or who did not complete the pre- and post-survey were excluded from the analyzed data.

Participants were asked to give informed written consent form before experimentation occurs (Appendix C). All participants were instructed that their participation was voluntary and that they could withdraw at any time. There was no penalty for choosing not to complete the survey. If they chose not to participate, they were informed that participation in the instructional component is a course requirement but following this computational thinking instructional unit they should return blank questionnaires and unsigned consent forms along with the remainder of the class at the completion of the allotted time. During all aspects of this procedure, the researcher was present in the classroom to provide the computational thinking instructional unit and to answer questions related to the research aspect. However, the course instructor was not present in the room.

The researcher assigned a random number to each participant. The researcher wrote numbers from 1 to number of participants in the classroom on their surveys. The researcher handed out pre-test surveys to participants and asked them to note their numbers on the part of

survey the marked “code number.” The researcher asked participants to answer the pre-test questions. When participants have finished, the researcher collected the pre-tests and the researcher gave them a two-week unit.

At the end of the second week unit, the researcher handed out post-tests and asked participants to write their code numbers on the top. The researcher reminded participants that their codes are unknown to the researcher, but they were reminded to use their same unique code on both the pre- and post-tests. The researcher asked participants to answer the post-test questions. When participants were finished, the researcher collected post-tests.

Instruments

Two surveys were administered, one survey was focusing on demographics (Appendix A) and one survey focused on pre-service teacher attitudes (Appendix B). In the first survey, pre-service teachers were asked to provide demographics information indicating their gender, race/ethnicity, age, and content area.

In the second survey, pre-service teachers were asked about their attitudes towards computational thinking; participants completed the survey twice, both before and after completing the unit. Pre-service teachers completed a single 21-question survey that was developed by Hoegh and Moskal (2009) and then later a survey was adapted by Yadav, Mayfield, Zhou, Hambrusch, and Korb (2014). This survey was used to measure teachers’ attitudes toward computational thinking. The paper-based survey contained questions based on a 5-point Likert Scale: “Strongly Agree,” “Agree,” “Neutral,” “Disagree,” and “Strongly

Disagree." The survey has produced a Cronbach's alpha internal reliability of 0.76 (Yadav, Mayfield, Zhou, Hambrusch, & Korb, 2014).

Table 1
Demographics and Attitude information

	Construct	Operational Definition	Measurement
Demographic Variables			
	Age	How old a student is.	Numeric self-report 18 to 24 year 25 to 34 years 35 to 44 years 45 to 54 years 55 to 64 years 65 to older
	Gender	What gender a student self-identifies as.	Male or Female
	Educational specialty	The student's area of focus for his or her educational training.	Self-reported specialty STEM versus non-STEM. The researcher made classification based on student's content areas. Pre-K4 Interdisciplinary English Math Science Social Studies Art or Music Other (Please specify)

	GPA	The student's cumulative average	Value provided by students
Dependent Variable			
	Attitude about computational thinking	The degree to which the survey that has five categories: Definition, Comfort, Interest, Use in classroom, and Career future use.	Likert-type scale (Interval – treated) on the 21 items (Computing attitudes scale)

Procedure

Prior to the initiation of the study, the researcher met pre-service teachers to instruct them during a two-week computational thinking unit in a required Instructional Technology course within the School of Education. The researcher synthesized a lesson plan for a two-week computational thinking unit with Google For Education Computational thinking online lecture and Scratch Computing Curriculum. The module was presented during the middle of spring semester, and computational thinking content was not introduced in the earlier lectures. Computational thinking sessions introduced pre-service teachers to an overview of computational thinking and also gave them a chance to complete hands-on activities.

The data collection was explained to pre-service teachers first during the initial face-to-face classroom meeting. The researcher made clear that all participants in the computational thinking experience would listen to lectures, participate in class discussions, and engage in hands-on activities as part of the course requirements, but only those who provided written consent forms would have their data analyzed as part of the research study.

Data collected from pre-service teachers were de-identified by the investigator, using codes and pseudonyms. Pre-service teachers were assigned codes by the researcher for the purpose of connecting pre- and post-survey results. Names were never collected from any student participant. Only the researcher had access to codes that connected individual pre-service teachers to the data.

The researcher explained the purpose of the study, the survey and informed consent form. For this study, two paper-based surveys were used. Pre-service teachers were asked to respond to the pre-survey. Participants should be able to complete the surveys in approximately 15 minutes. Participants were instructed that they can withdraw from the study at any time without penalty or loss of benefits. All data was coded with an anonymous ID to ensure anonymity and confidentiality. Participants were not put their names or any identifying information on the survey. The informed consent form was read instructor to the pre-service teachers and participants were given time to sing the agreement before proceeding to the surveys. The pre-service teachers were instructed for a total of two 50-minutes sessions consecutive weeks. At the end of the two 50-minutes sessions, pre-service teachers were asked to respond to the post-survey.

Any information obtained from this research was kept confidential. Data and results were not shared or made public in a way that indicates the identity of the individual pre-service teachers; only group outcomes were reported. Data about individual pre-service teachers were not shared with the pre-service teachers, peers or course instructors. It was expected that information gathered in research became part of a dissertation and subsequent published reports. In written descriptions and in reports of what was learned from the study, the researcher removed any information that identifies individuals.

Statistical Analysis

Descriptive statistics were gathered from the study and then analyzed with Statistical Package for Social Sciences (SPSS) Graduate Pack. The study was based on four research questions and the analysis of these included descriptive statistics.

Summary

This study examined pre-service teachers attitudes of computational thinking at a private university in the Eastern, U.S. Pre-service teachers completed surveys that examine attitudes of computational thinking. After all data has been completed, responses were examined to answer the research questions. This study of computational thinking findings will help teachers and researchers.

CHAPTER IV

RESULTS

Introduction

The purpose of this study was to examine pre-service teachers' attitudes toward computational thinking before and after an intervention that was designed to convey the importance of teaching computational thinking at the K-12 level. This chapter presents the results of the statistical analyses seeking to address the four research questions. Included in those analyses are investigations of the survey responses and pre-service teachers' demographics. Results were examined in light of the research hypotheses, and summarized for clarity.

Survey Response Rate

The survey data were collected between February 28, 2016 and April 21, 2016. There were 167 participants who completed surveys but 48 participants were removed from analyses because they did not complete all of the required surveys: the pre-survey, post-survey (immediate), or delayed post-survey. Participants who failed to answer one or two demographic items were included within overall analyses. This resulted in a total of 119 participants.

Pre-service teachers' Demographics

The first five survey questions (Appendix A) requested information regarding pre-service teachers' demographics: gender, race/ethnicity, age, content area, and cumulative GPA. Only gender, content area (concentration of study), and cumulative GPA are considered within the analyses.

Table 1 shows participants' GPAs, genders, and content area. Of the 118 respondents to the Content area item, 71 respondents from the Pre-K4 pre-service teachers, 21 respondents from the pre-service English teachers, 12 respondents from the Math pre-service teachers, and 14 respondents from the Social Studies pre-service teachers.

Table 2
Sample Sizes for the Total Population by GPA, Gender, Content Area, and Race/Ethnicity

Measure	n	
GPA	112	39 (3.0 to 3.49) 73 (3.5 to 4.0)
Gender	118	104 females, 14 males
Content area	118	83 STEM 35 non-STEM
Race/Ethnicity	117	110 White 1 African-American/Black 3 Asian/Pacific Islander 1 Hispanic/Latino 1 Multiracial 1 Native American / American Indian

Cumulative GPA Responses and Attitudes

The last question in the demographic survey asked, "What is your current cumulative GPA?" and allowed respondents to write in an answer. Because GPA is used as an indication of academic achievement, pre-service teachers with a 3.5 to 4.0 range GPA were assumed to have a more positive attitude toward computational thinking than those with a 3.0 to 3.49 range GPA. Seven participants chose not to give information about cumulative GPA.

Age and Gender Responses

There were more female participants than male participants within this sample. However, it is roughly proportionate to the numbers of each gender who pursue pre-service studies at the college. Of the respondents, 88.1% were female and 11.9% were male. One participant chose not to give information about gender. The small number of male pre-service teachers meant that analyses of gender differences would need to be considered tentatively. The entire sample reported their ages to be between 18 and 24.

Research Purpose and Results

To assess the internal consistency of survey responses, Cronbach's alpha was calculated (alpha = 0.77), which indicated a more than acceptable rate of reliability between responses. That alpha was also similar to that for the initial use of the survey (alpha = 0.76; Yadav, Mayfield, Zhou, Hambrusch, & Korb, 2014)**.**

First research hypothesis. Attitude scores were assessed using the attitudes survey and submitted to one-way repeated measures ANOVA to determine if attitude scores changed between the

times prior and immediately after the unit on computational thinking, or between the times immediately after and three weeks later.

The intervention elicited statistically significant changed in attitude pre-test, post-test, and delayed-post test, $F(2,236) = 15.175$, $p < .0005$. As a result of the computational thinking unit, it was confirmed there was a statistically significant increased in positive attitudes toward computational thinking from pre-survey to post-survey. Additionally, there was a second, statistically significant increase in positive attitudes from the post-survey to the delayed survey. Table 2 displays measures of attitudes before and after the computational thinking unit for the pre-service teachers.

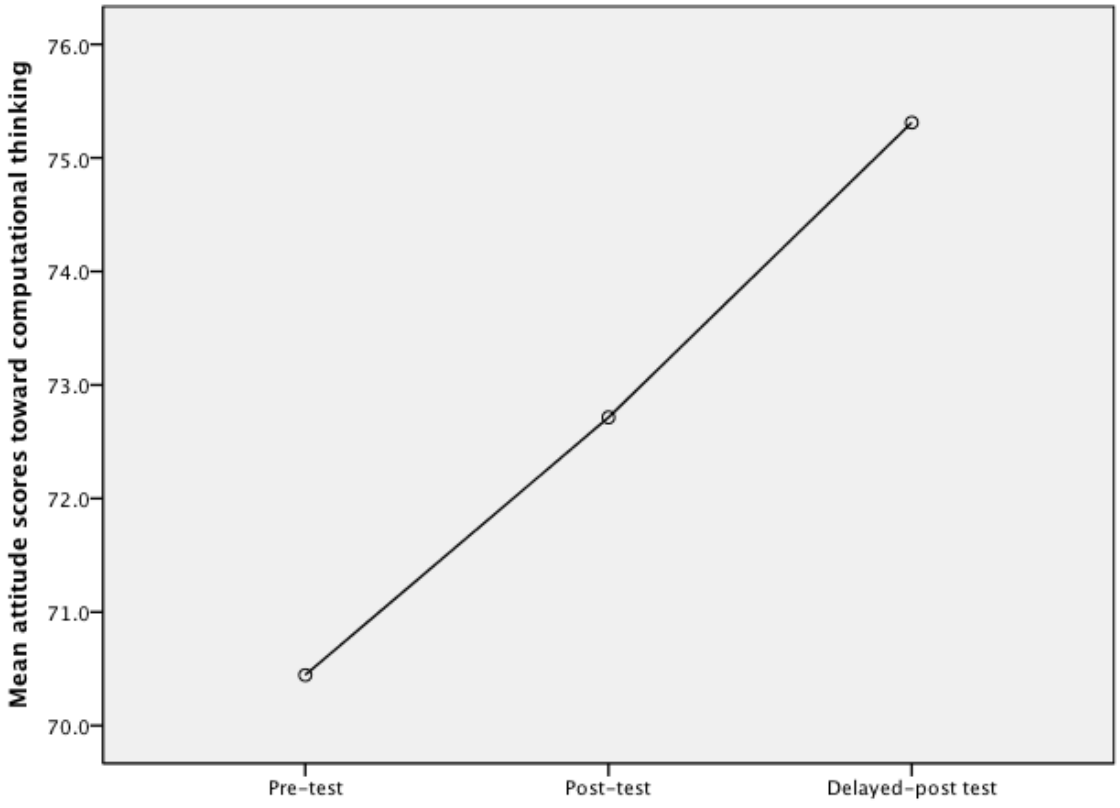
Table 3

Descriptive Statistics of Mean Scores on Attitudes Following the Computational Thinking

	Pre-test		Post-test		Delayed-post test	
	M	SD	M	SD	M	SD
Attitudes toward computational thinking	70.44	8.48	72.71	8.58	75.31	10.71

Figure 13

Changes in Attitudes Following the Computational Thinking Unit



Second research hypothesis. A series of split-plot repeated measures ANOVA were performed to determine if any significant differences between 3.5 to 4.0 range and 3.0 to 3.49 range GPAs existed among testing intervals. The intervention elicited statistically significant increased in attitude pre-test, post-test, and delayed-post test. It was confirmed that the patterns of change differed between participants with 3.5 to 4.0 range and 3.0 to 3.49 range GPAs. For statistical analysis, this measure was categorized into binary conditions (3.5 to 4.0 range and 3.0 to 3.49 range GPA). The results demonstrated that there was a positive relationship between having a 3.5 to 4.0 range cumulative GPA and a positive attitude toward computational thinking. Whereas pre-service teachers with both 3.5 to 4.0 range and 3.0 to 3.49 range GPAs increased in their attitude scores following the unit (pre-survey vs. post-survey), only teachers with 3.5 to 4.0 range GPAs continued to increase in attitudes from post-test to delayed post-test.

Table 3 presents the sums of squares, degrees of freedom, mean squares, and F-ratios for level for 3.5 to 4.0 range and 3.0 to 3.49 range GPAs. The analysis of variance (ANOVA) for Pre-Test did not reveal a significant difference [$F(1,110) = 2.83$, $p = 0.095$] between 3.5 to 4.0 range and 3.0 to 3.49 range GPAs. Nor did the ANOVA reveal a significant difference between 3.5 to 4.0 range and 3.0 to 3.49 range GPAs for Post-Test [$F(1,110) = 3.18$, $p = 0.077$] or for Delayed Post-Test [$F(1,110) = 1.33$, $p = 0.251$].

Table 4

Analysis of Variance of Pre-test, Post-test and Delayed Post-test Attitude Scores toward Computational Thinking for 3.5 to 4.0 range and 3.0 to 3.49 range GPAs

		Sum of Squares	df	Mean Square	F	Sig.	η_p^2
Pre-test	Between Groups	186.18	1	186.18	2.83	.095	
	Within Groups	7226.24	110	65.69			
	Total	7412.42	111				
Post-test	Between Groups	230.66	1	230.66	3.18	.077	
	Within Groups	7983.06	110	72.57			
	Total	8213.72	111				
Delayed Post-test	Between Groups	151.53	1	151.53	1.33	.251	
	Within Groups	12524.04	110	113.86			
	Total	12675.56	111				
Interaction (GPA*Time)		336.82	1	336.82	6.43	.013	.049
Error Corrected Total		5757.61	110	52.34			.078

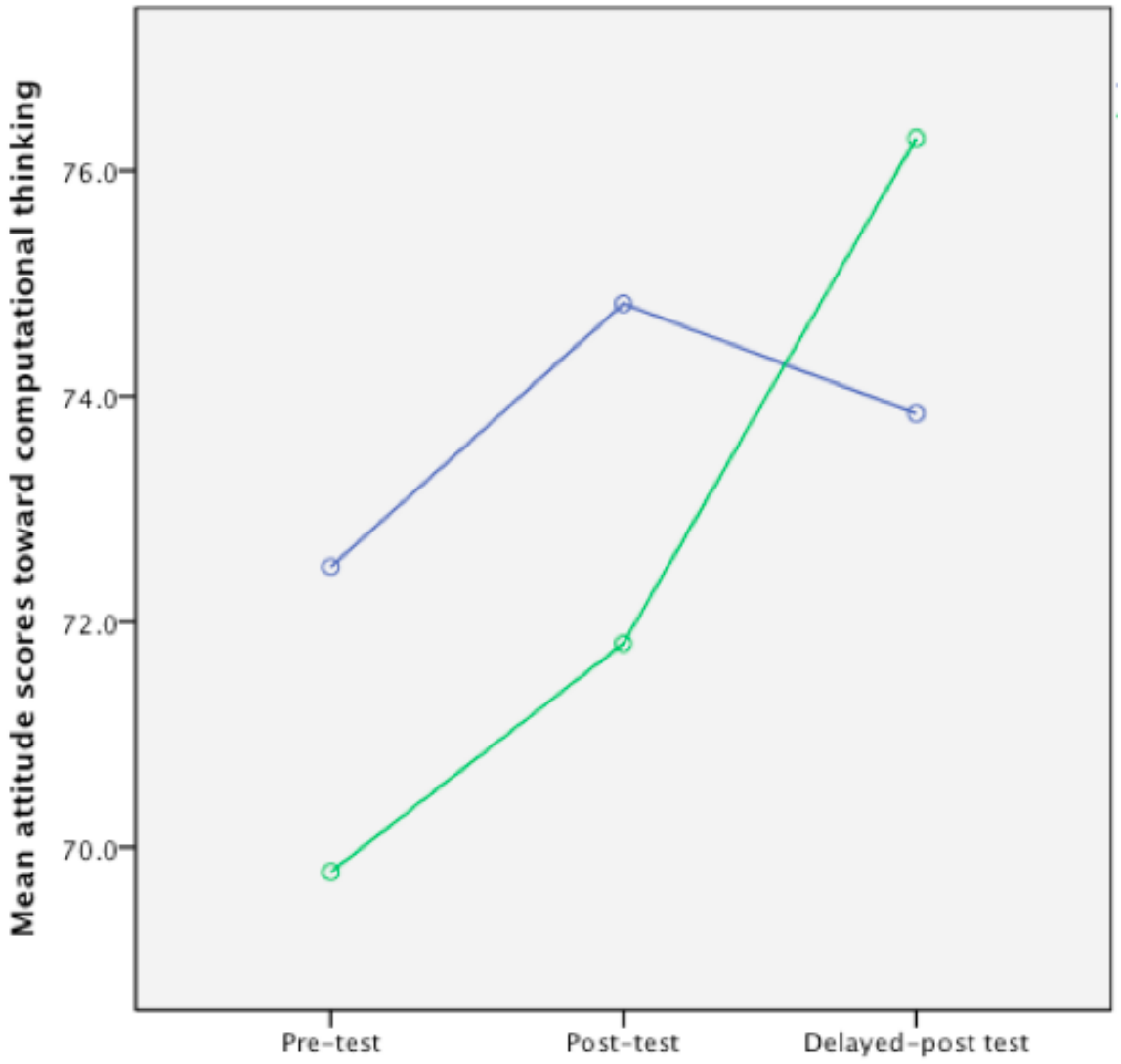
Table 5

Descriptive Statistics for 3.5 to 4.0 range and 3.0 to 3.49 range GPAs

	Pre-test		Post-test		Delayed Post-test	
	M	SD	M	SD	M	SD
3-3.49 GPA	72.48	8.69	74.82	7.90	73.84	10.72
3.5-4.0 GPA	69.78	7.77	71.80	8.82	76.28	10.64

Figure 14

Changes in Attitudes Following the Computational Thinking Unit with Respect to GPA.



Note. GPA range 3.0 – 3.49

GPA range 3.5 – 4.0

Table 6 *Pre-test, Post-test and Delayed Post-test Mean Scores of Items for* 3.5 to 4.0 range *GPAs*

Item	Pre-test	Post-test	Delayed Post-test
1	3.37	3.50	3.71
2	4.03	4.09	4.25
3	3.66	3.82	3.99
4	3.81	4.03	4.07
5	2.83	2.97	4.07
6	2.71	2.97	3.70
7	3.99	3.89	3.89
8	4.04	3.96	3.99
9	2.93	2.88	3.96
10	2.83	2.94	3.82
11	2.82	3.04	3.00
12	2.70	2.83	2.75
13	2.73	2.96	2.94
14	2.38	2.74	2.68
15	3.68	3.86	3.75
16	3.85	4.01	3.88
17	3.89	3.89	3.81
18	2.97	3.00	3.50
19	3.79	3.60	3.67
20	2.71	2.83	2.82
21	4.04	3.96	4.04

Table 7 *Pre-test, Post-test and Delayed Post-test Mean Scores of Items for* 3.0 to 3.49 range *GPAs*

Item	Pre-test	Post-test	Delayed Post-test
1	3.43	3.74	3.51
2	3.95	4.20	4.23
3	3.90	3.90	3.79
4	3.90	4.08	3.97
5	2.87	3.08	3.92
6	3.08	3.18	3.69
7	4.10	4.08	3.69
8	4.08	4.13	3.87
9	2.90	2.77	3.51
10	2.92	2.85	3.56
11	2.97	3.08	2.90
12	2.87	3.05	2.79
13	3.00	3.10	2.85
14	2.85	2.97	2.72
15	4.05	4.05	3.77
16	4.02	4.10	3.82
17	4.05	4.00	3.87
18	2.92	3.13	3.36
19	3.64	4.00	3.46
20	2.92	3.18	2.87
21	4.05	4.15	3.67

Third research hypothesis. A series of Split-plot repeated measures ANOVA were performed to determine if any significant differences between STEM and non-STEM majors existed between testing intervals. The intervention elicited statistically significant changed in attitude pre-test, post-test, and delayed-post test. It was confirmed that patterns of change differed between STEM and non-STEM majors. For statistical analysis, this measure was categorized into binary conditions (STEM or non-STEM). Whereas both STEM and non-STEM pre-service teachers increased in their attitudes from pre-survey to post-survey, only the STEM pre-service teachers increased again from post-survey to delayed post-survey.

Table 7 presents the sums of squares, degrees of freedom, mean squares, and F-ratios for level for STEM and non-STEM. The analysis of variance (ANOVA) for Pre-Test did not reveal a significant difference [$F(1,116) = 3.01$, $p = 0.085$] between STEM and non-STEM. Nor did the ANOVA reveal a significant difference between STEM and non-STEM for Post-Test [$F(1,116) = 2.60$, $p = 0.110$] or for Delayed Post-Test [$F(1,116) = 0.40$, $p = .530$].

Table 8

Analysis of Variance of Pre-Test, Post-Test and Delayed Post-Test Attitude Scores toward Computational Thinking for STEM and non-STEM

		Sum of Squares	df	Mean Square	F	Sig.	η_p^2
Pre-test	Between Groups	198.266	1	198.27	3.01	.085	
	Within Groups	7638.18	116	65.85			
	Total	7836.44	117				
Post-test	Between Groups	188.48	1	188.48	2.60	.110	
	Within Groups	8416.64	116	72.56			
	Total	8605.12	117				
Delayed Post-test	Between Groups	45.86	1	45.86	.40	.530	
	Within Groups	13430.01	116	115.78			
	Total	13475.87	117				
Interaction (Content area*Time)		217.42	1	217.42	3.96	.050	.027
Error Corrected Total		6371.90	116	54.93			.064

Table 9

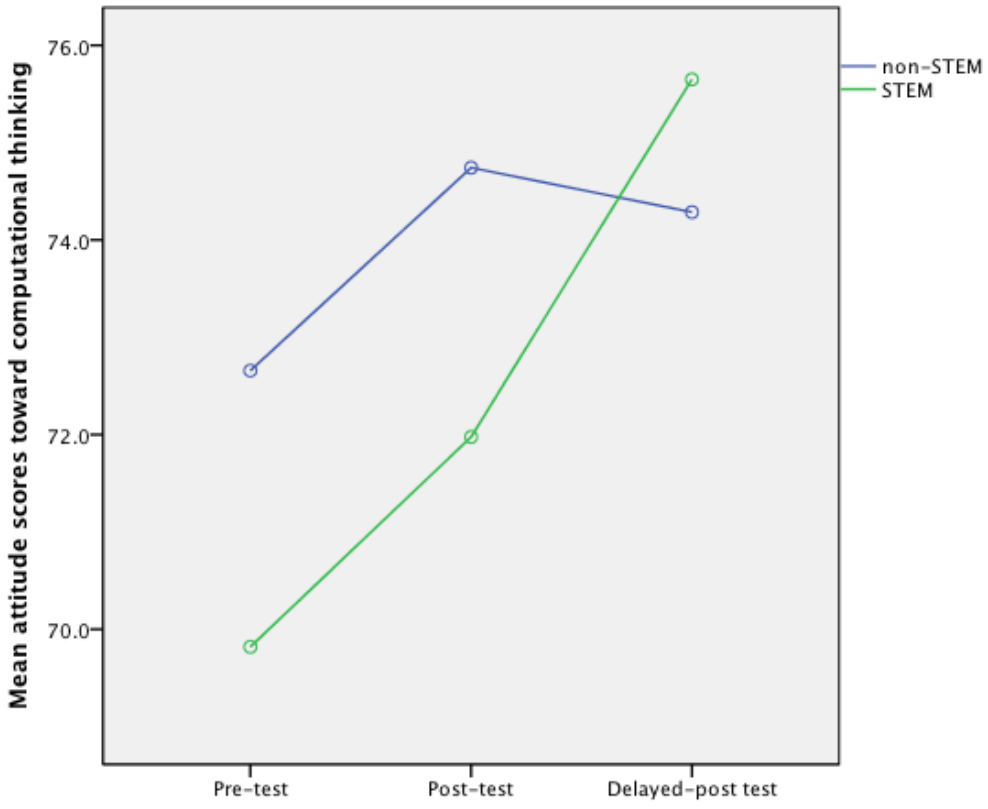
Descriptive Statistics for STEM and non-STEM

	Pre-test		Post-test		Delayed Post-test	
	M	SD	M	SD	M	SD
STEM	69.81	7.40	71.97	7.21	75.65	9.67
Non-STEM	72.65	9.62	74.74	11.03	74.28	13.00

Note. STEM = Science, Technology, Engineering, and Math

Figure 15

Changes in Attitudes Following the Computational Thinking Unit with Respect to Content Area



Note. STEM = Science, Technology, Engineering, and Math

Table 10 *Pre-test, Post-test and Delayed Post-test Mean Scores of Items for STEM*

Item	Pre-test	Post-test	Delayed Post-test
1	3.38	3.63	3.67
2	4.07	4.16	4.24
3	3.67	3.90	3.89
4	3.82	4.06	4.07
5	2.57	2.69	4.04
6	2.65	2.83	3.77
7	4.05	3.91	3.88
8	4.14	3.99	4.02
9	2.82	2.61	3.75
10	2.75	2.70	3.79
11	2.95	3.02	2.97
12	2.71	2.93	2.71
13	2.73	3.06	2.88
14	2.57	2.87	2.72
15	3.83	3.94	3.82
16	3.97	4.04	3.90
17	3.89	3.97	3.79
18	2.70	2.95	3.42
19	3.73	3.76	3.58
20	2.75	2.97	2.81
21	4.05	3.97	3.90

Note. STEM = Science, Technology, Engineering, and Math

Table 11 *Pre-test, Post-test and Delayed Post-test Mean Scores of Items for non-STEM*

Item	Pre-test	Post-test	Delayed Post-test
1	3.40	3.51	3.54
2	3.86	4.11	4.26
3	3.91	3.71	3.97
4	3.86	3.97	3.94
5	3.48	3.83	3.94
6	3.28	3.57	3.43
7	3.97	4.03	3.74
8	3.86	4.06	3.80
9	3.23	3.43	3.80
10	3.28	3.46	3.43
11	2.77	3.20	2.94
12	2.71	2.77	2.80
13	2.91	2.86	3.00
14	2.48	2.68	2.66
15	3.77	3.77	3.63
16	3.77	4.00	3.74
17	4.08	3.86	3.86
18	3.43	3.37	3.51
19	3.77	3.66	3.62
20	2.80	2.86	2.80
21	4.00	4.03	3.86

Fourth research hypothesis. A series of split-plot repeated measures ANOVAs were performed to determine if any significant differences between genders existed between testing intervals. The intervention elicited did not lead to any statistically significant changes in attitude pre-test, post-test, and delayed-post test.

With only 12 male participants, it was difficult to conclude whether differences were found between the patterns of change in respect to gender. However, those 12 male participants did not show significant increases in attitudes at post- or delayed post-surveys. For statistical analysis, this measure was categorized into binary conditions (male or female).

Table 11 presents the sums of squares, degrees of freedom, mean squares, and F-ratios for male and female. The analysis of variance (ANOVA) for Pre-Test did not reveal a significant difference [$F(1,116) = 1$, $p = 0.319$] between male and female. Nor did the ANOVA reveal a significant difference between male and female for Post-Test [$F(1,116) = 0.241$, $p = 0.624$] or for Delayed Post-Test [$F(1,116) = 1.040$, $p = 0.310$].

Table 12

Analysis of Variance of Pre-test, Post-test and Delayed Post-test Attitude Scores toward Computational Thinking for Male and Female

	Sum of Squares	df	Mean Square	F	Sig.	η_p^2
Between Groups	66.97	1	66.97	1	.319	
Within Groups	7769.47	116	66.98			
Total	7836.44	117				
Between Groups	17.87	1	17.87	.241	.624	
Within Groups	8587.25	116	74.03			
Total	8605.12	117				
Between Groups	119.76	1	119.76	1.040	.310	
Within Groups	13356.11	116	115.14			
Total	13475.87	117				
Interaction (Gender*Time)	182.92	1	182.92	3.31	.07	.020
Error	6406.41	116	55.23			
Corrected Total						.016

Table 13

Descriptive Statistics for Mean Scores Male and Female

	Pre-test		Post-test		Delayed Post-test	
	M	SD	M	SD	M	SD
Male	72.71	5.68	73.85	8.88	72.50	9.23
Female	70.38	8.44	72.65	8.56	75.61	10.90

Figure 16

Changes in Attitudes over the Course of the Computational Thinking Unit with Respect to Gender

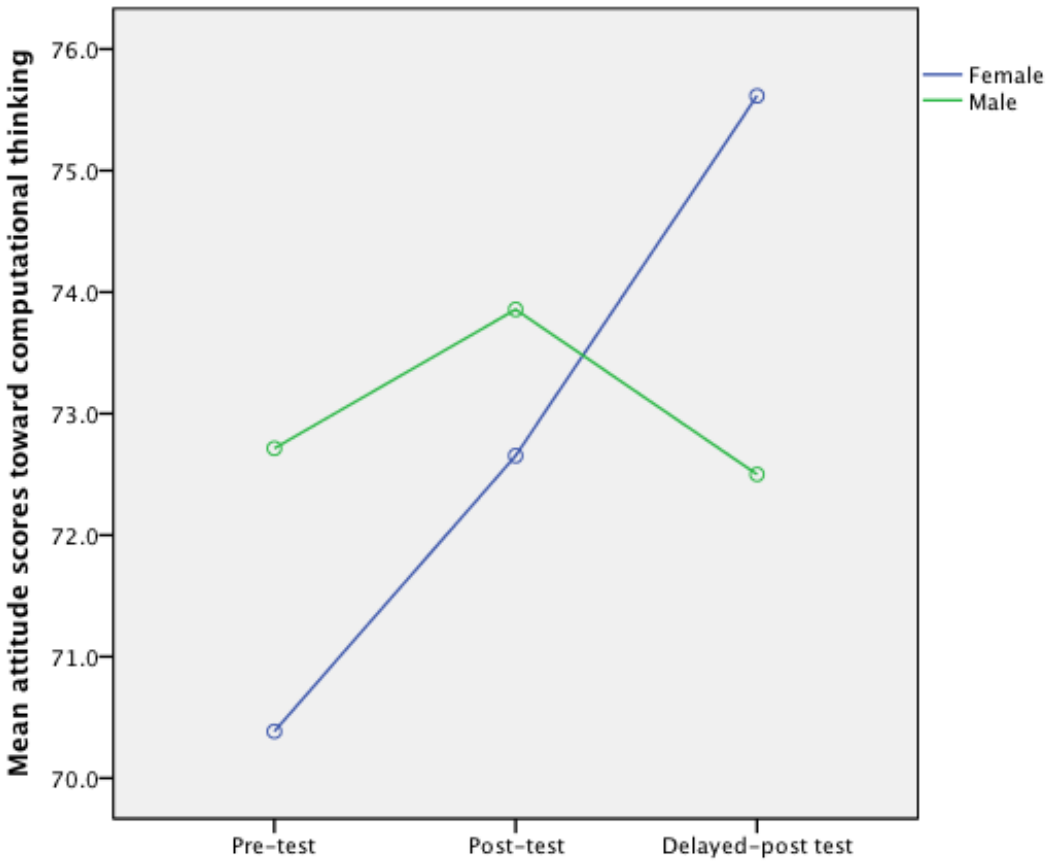


Table 14 *Pre-test, Post-test and Delayed Post-test Mean Scores of Items*

Item	Pre-test	Post-test	Delayed Post-test
1	3.38	3.59	3.64
2	3.99	4.14	4.25
3	3.73	3.84	3.92
4	3.81	4.03	4.04
5	2.82	3.01	4.01
6	2.83	3.04	3.67
7	4.01	3.94	3.84
8	4.05	4.00	3.95
9	2.93	2.85	3.76
10	2.89	2.91	3.68
11	2.89	3.09	2.96
12	2.70	2.87	2.73
13	2.77	2.98	2.90
14	2.52	2.79	2.69
15	3.79	3.89	3.76
16	3.89	4.01	3.86
17	3.93	3.93	3.81
18	2.89	3.07	3.45
19	3.73	3.72	3.59
20	2.76	2.92	2.80
21	4.02	3.99	3.89

Summary

This chapter presented statistical analyses of the data collected during a computational thinking unit in an Instructional Technology course. The data were collected via paper-based survey with a return rate of 71%. The survey measured pre-service teachers' attitudes towards computational thinking, and also asked them to supply demographic information.

A summary of pre-service teachers' demographics indicated that a majority of the respondents were female and white. Respondents further indicated various content areas of focus (Pre-K4, Interdisciplinary, English, Math, Science, Social Studies, Art or Music, and Other).

The results of the study demonstrated that the computational thinking unit's intervention increased the attitudes of pre-service teachers both immediately afterward and again after a three-week interval. This was generally true of both STEM and non-STEM teachers with both 3.5 to 4.0 range and 3.0 to 3.49 range GPAs. However, there was not a secondary increased in attitudes for those with non-STEM content areas nor 3.0 to 3.49 range GPAs. The small sample of male pre-service teachers made it difficult to determine whether they follow the same trends.

CHAPTER V

DISCUSSION

Introduction

The purpose of this chapter is to summarize and discuss the results of this study of pre-service teacher perceptions of an intervention aimed at improving their attitudes toward computational thinking in education. Moreover, this chapter presents the findings as they relate to previous research literature, important conclusions, and limitations. Finally, the chapter provides recommendations for further research.

Summary of the Procedure

Within the study, pre-service teachers were invited to participate in an instructional unit on computational thinking. Undergraduate, pre-service teachers first completed demographic information and attitude surveys during a regularly scheduled class within their School of Education curriculum. This first attitude survey (pre-test) asked teachers for their familiarity with computational thinking. After two subsequent, 50-minute training sessions, participants completed a second attitude survey (post-test) and then again following a three-week lapse, completed the third and final attitude survey (delayed post-test).

Summary of the Findings

The current investigation was motivated by four research questions. Each of the four research questions is presented below, along with a summary of the findings.

Research Question One: Can an embedded intervention that teaches about the importance and utility of computational thinking, change the attitudes of pre-service teachers enrolled in Instructional Technology courses?

The first research question examined pre-service teachers' attitudes toward computational thinking before and after an intervention. Analyses of the responses from the attitude survey indicated that the intervention was effective in changing the attitudes of the aggregate group or pre-service teachers toward computational thinking. Specifically, attitude increased from pre-test to post-test and then again from post-test to delayed post-test. All pre-service teachers started out at about the same level of attitude generally positive toward computational thinking and ended with increased attitudes mostly positive toward computational thinking.

Previous research has found that pre-service teachers' attitudes toward computational thinking increased following in-service training. Moreover, the current study provided preliminary evidence that this increase may extend beyond the completion of the instructional training period. The current findings extended research by Harmbrusch and colleagues (2009), who found benefits when computational thinking was integrated into a variety of subject areas for pre-service teachers. In addition, in this study, the computational thinking instructional unit included Scratch programming language tool, examples of Scratch flash cards, computational thinking examples, and unplugged activities to all pre-service teachers. This is important because

these examples and activities allowed teachers to develop their own computational thinking skills, as well as gain experience in implementing computational thinking concepts and practices through the use of the Scratch programming language tool.

Hands-on activities in the current study were aligned with the theory of constructionism to teach students how to think computationally (Bers, 2008; Brennan, & Resnick, 2012a; Wing, 2006). According to constructionism, students-centered learning in computational thinking while doing hands-on activities (Papert, 1980; 1993) Additionally, pre-service teachers were becoming familiar, knowledgeable, articulated, and sophisticated about improving computational thinking concepts, practices and perspectives and are interacting with peers and doing hands-on activity by thinking. To date, however, research identifying hands-on activities based on computational thinking has not previously been completed in a pre-service education program with the programming language tool Scratch**.** Therefore, the findings demonstrated a new way to teach computational thinking with using a programming language tool Scratch to pre-service teachers who may not have recognized the value of computational thinking before. Moreover, the constructionist design of this unit empowered pre-service teachers in their use of computational thinking by fostering their own computational thinking skills through activities that could also be adapted to meet the needs of their own classrooms. As a result, this theory demonstrates how computational thinking can be used and taught to students in classroom environments.

Research Question Two: Would the intervention on computational thinking affect the attitudes of pre-service teachers differently depending on their GPAs?

The second research question examined pre-service teachers' patterns of attitude change and whether they differed between participants with 3.5 to 4.0 range and 3.0 to 3.49 range GPAs. In this study, analyses revealed that pre-service teachers with both 3.0 to 3.49 range and 3.5 to 4.0 range GPA showed patterns of attitude increases following the unit. They had generally positive attitude that increased to mostly positive attitudes, and investigation of this research question confirmed that the intervention, as intended, increased their attitudes toward computational thinking.

While past studies have investigated pre-service teachers' attitudes toward computational thinking units, no similar research has been conducted to investigate the influence of having a 3.5 to 4.0 range and 3.0 to 3.49 range GPA (Hoegh & Moskal, 2009; Wing, 2006; Wing, 2008b; Yadav, Mayfield, Zhou, Hambrusch, & Korb, 2014). In this study, analyses revealed that 3.5 to 4.0 range and 3.0 to 3.49 range GPAs pre-service teachers showed patterns of attitude increases from generally positive attitude scores to mostly positive attitude scores following the instructional unit. The findings revealed that the instructional unit increased the attitudes of all pre-service teachers, regardless of their GPAs. Moreover, this study found that the instructional unit worked for all of pre-service teachers equally.

In this study, the developed computational thinking unit included and relied upon pedagogy derived from constructionism theory, such as project-based learning, which encouraged the inclusion of 21st century skills within the activities. According to constructionism, teaching becomes learner-oriented and that was the format of the unit. Learners

were actively involved in hands-on activities, and not passively listening in order to learn. That format offered learners a more active role in the classroom setting (Fosnot, 1996). Additionally, pre-service teachers could engage with the materials and in the activities without much, if any, prior knowledge of broader subject matters or even of computational thinking. And so, it is understandable that pre-service teachers would equally be affected by the intervention, regardless of their GPAs. Constructionism prescribes hands-on activities and real life experiences within the classroom. Those types of hands-on activities and real life experiences within the unit could be engaged in and recognized as important by even novice or lower-performing pre-service teachers.

Questions 15 and 16 asked specifically about that process. These items are important because how teachers conceptualize what it means and what is required to integrate computational thinking skills can positively or negatively affect their attitudes. Question 2 asked about whether computer applications are necessary to teach computational thinking. This item is important because teachers might feel reluctant to learn and introduce a new application within their classrooms. Asked to respond on a five-point Likert scale, Question 15 posed the following statement: Computational thinking can be incorporated in the classroom by using computers within the lesson plan. Question 16 posed the following statement: Computational thinking can be incorporated into the classroom by allowing students to problem solve. Similarly, Question 2 posed the following statement: Computational thinking involves thinking logically to solve problems. On all three of these items, pre-service teachers' responses to these questions became increasingly similar following the intervention, whereas pre-surveys showed some potential differences in opinion between those with 3.5 to 4.0 range and 3.0 to 3.49 range GPAs. This is evidence that the intervention helped to define what it means to integrate computational thinking

into a K-12 curriculum. It does not mean adding computers or applications to already challenging classroom schedules. After pre-service teachers with both 3.5 to 4.0 range and 3.0 to 3.49 range GPAs understood that, attitude scores increased for both groups.

Research Question Three: Are the attitude scores of pre-service teachers with STEM concentrations more subject to change after the computational thinking intervention than are the attitude scores of pre-service teachers with non-STEM concentrations?

Unlike previous research, this study examined attitudes by pre-service teachers separately based upon whether their degrees would be in STEM or non-STEM areas of instruction. In this study, analyses revealed that STEM and non-STEM pre-service raided their generally positive attitude scores to mostly positive attitude scores following the instructional unit. The intervention elicited statistically significant changes in attitude pre-test, post-test, and delayed-post test. Although for the most part, the current study produced expected results, one area of the study produced different findings for STEM and non-STEM pre-service teachers. One of the most important findings of the current study was the result of investigating whether the instructional unit would be as effective for non-STEM pre-service teachers as for STEM pre-service teachers. The findings revealed that the instructional unit increased the attitudes of all pre-service teachers, regardless of their concentrations (STEM or non-STEM). Because of the computational thinking required within STEM fields, it might be supposed that STEM pre-service teachers would have more positive attitudes toward computational thinking than non-STEM pre-service teachers have, but both groups raided their generally positive attitude scores to mostly positive attitude scores following the instructional unit.

The current study provides evidence that both STEM and non-STEM pre-service teachers can be trained to recognize the value of computational thinking practices, and that a two-week computational thinking unit is enough to do it. It might be better to train pre-service teachers in computational thinking during their academic work in order for them to be effective in teaching computational thinking, than it is to train them later in their careers. This is because, early in their careers, they would then know how to engage students with hands-on activities in fun and meaningful ways that could promote computational skills and practices. Importantly, it would be worth investigating whether there is a link between the teacher's understanding of, and attitude toward, computational thinking and the attitudes adopted by her/his students.

Constructionism can play a significant role for STEM and non-STEM teachers and their future students in the classroom setting. It is important to train STEM and non-STEM pre-service teachers to get enough knowledge of computational thinking and how to integrate it into for their curriculum, and incorporate it into their classroom activities (Yadav, Mayfield, Zhou, Hambrusch, & Korb, 2014). It is likely the inclusion of hands-on activities that allowed both STEM and non-STEM teachers to be recognize the importance of computational thinking concepts, practices, and perspectives.

Constructionism prescribes hands-on activities like those in the current computational thinking unit. Those hands-on activities are more common within STEM subjects than in non-STEM subjects (Bers, 2010; Resnick et al., 2009). So, STEM pre-service teachers might benefit more than non-STEM pre-service teachers from this computational thinking that employs such hands-on activities. However, the current findings revealed that it is not the case, that is, both STEM and non-STEM pre-service teachers were positively influenced by computational thinking unit.

STEM and non-STEM pre-service teachers were asked to react to a statement about their attitudes toward computational thinking before and after an intervention. In the current study, the intervention was a computational thinking unit. During the computational thinking unit, participants were given opportunities to do hands-on activities with Scratch programming application Questions 2 and 3 of the survey asked specifically about the definition of computational thinking to make sure the participants understood exactly what computational thinking means. Pre-service teachers responded to a twenty-one-item survey before and after an intervention. Question 2 asked them to respond "Strongly agree," "Disagree," "Neutral," "Agree," "Strongly agree" to the following: "Computational thinking involves thinking logically to solve problems," and Question 2 asked them to respond to the following: "Computational thinking involves using computers to solve problems." Before the intervention, both STEM and non-STEM pre-service teachers responded with less agreement on the definition of computational thinking, however after the intervention, both groups responded with more agreement.

Question 10 of the survey asked specifically about the participants' comfort with computational thinking. It indicates how comfortable STEM and non-STEM pre-service teachers are with using computer applications in the classroom. The question asked them to rate (using the aforementioned scale of "Strongly agree," "Disagree," "Neutral," "Agree," "Strongly agree") the following statement: "I doubt that I have the skills to solve problems by using computer application." Before the intervention, both STEM and non-STEM pre-service teachers responded with less agreement on their comfort with computational thinking and after the intervention, they responded with more agreement on their comfort with computational thinking.

Research Question Four: Are changes in attitude following the unit related to the gender of the pre-service teacher?

Findings revealed that there were no differences between the male and female pre-service teachers in this study. However, there were significantly fewer male participants than there were female participants. The predominance of women in this study is roughly proportionate to the numbers of each gender who pursue pre-service studies at the college. Of course, the small number of male pre-service teachers requires the finding of the lack of gender differences to be considered tentatively.

Female pre-service teachers are role models since in the theory of constructionism they teach but they learn with and from the children. Role models are fundamental in K-12 students life, they are seeing their teachers of STEM or STEM-related field. However, Fewer girls are involved in programming and other types of computational thinking. It is probably true that if there are more female role models, more girls will get involved in this type of thinking (Google for Education, 2015; Google-Gallup, 2005; Yadav, Mayfield, Zhou, Hambrusch, & Korb, 2014).

Although fewer women currently work in the computer science field, there were more female students than male pre-service teachers in my study. It is important to note that most of the pre-service teachers in the classroom were PreK-4 teachers so that this research also indicates the importance of early age. In particular, early usage has been shown to increase success in future computing and STEM classes (Cohoon & Aspray, 2006).

Teachers need to be better taught on how to teach computational thinking to their students. In particular, female pre-service teachers' attitudes toward computational thinking play a major role in how best to teach computational thinking for prospective students (Yadav, Mayfield, Zhou, Hambrusch, & Korb, 2014; Zhao et al, 2001). Teachers play a critical role in

each of these areas as they work to maximize learning outcomes in their students by motivating and engaging them in computational thinking. An exploration of the role of the teacher in promoting computational thinking among students is therefore an important step towards building science literacy in our youth. If they have positive attitudes, they have the power to influence children at an early age on issues regarding computational thinking and programming. This study should be replicated with more male students, in order to provide generalized results for a larger pre-service teacher population than the community represented in this study. In particular, underrepresented groups are not able to get resources, activities, and sample projects.

Limitations

There are several limitations regarding the current study. One limitation is that there were fewer male pre-service teachers than female pre-service teachers, which is not surprising, because more women than men are education majors. It would be interesting to repeat this study with a larger number of male participants, if not equal numbers.

Another potential limitation of this study was that it incorporated only a two-week computational thinking unit intended to convey the importance of teaching computational thinking. Thus, the only dependent measure were pre-service teachers' attitudes toward computational thinking. Additional measures should have been planned during pre-unit and post-unit assessments to investigate learning specifically. Did these pre-service teachers also increase in their own computational thinking, in their abilities to use the programming language of Scratch, and in their content knowledge of computer science more generally? To sufficiently foster such learning, a longer exposure would help pre-service teachers to understand more about computational thinking and would give pre-service teachers more practice with the programming

language tool Scratch. This study has shown that a brief intervention can help teachers to recognize the importance of computational thinking. But a similar unit could focus more on teaching computational thinking concepts and skills to pre-service teachers. Future research should examine how best to teach pre-service teachers about computational thinking concepts and skills in shorter units like this study's that do not require an entire course.

Broader implementation of this unit is also a concern. This study was of a unit implemented at a single university by a single instructor. Although it was implemented in several class sections with different primary instructors and there is not a reason to believe that the pre-service teachers enrolled at this university are different from pre-service teachers at other universities, it was a limitation to only have a single instructor for the unit. This study should be replicated at other universities or in other contexts and with a variety of instructors in order to provide more reliably generalizable results. It also would add to the validity of the study because university classrooms are often taught by more than one instructor.

Recommendations for Future Research

To gain a better understanding of how students progress when they have sustained explicit training in the use of programming languages and computational thinking, a study of longer duration is recommended. More than two weeks of the computational thinking unit could provide more examples and activities to pre-service teachers.

Additionally, more time could be spent helping teachers to adopt best practices for integrating computational thinking into their teaching by having them draft hypothetical lesson plans in a variety of subject matter domains. Before teaching in their field, pre-service teachers

would then be thoroughly familiar with computational thinking through hands-on activities and ready to integrate such skills into their classrooms. The current findings suggest that the longer unit should foster more positive attitudes in pre-service teachers but that would need to be investigated further.

This study should be replicated with a larger male students population, in order to provide generalized results for a larger pre-service teacher population than the community represented in this study. Most importantly, it is uncertain whether any differences between male and female pre-service teachers should be expected, or if the findings from this predominantly female sample can be assumed to be true of male pre-service teachers as well. A larger, more male-inclusive sample would help to answer that question.

It would also be helpful to include in-service teachers in order to compare their attitudes about the computational thinking with those of the pre-service teachers. Are pre-service teachers more open to new content, like computational thinking, and in-service teachers are more often reluctant to introduce new content? Or, is it that this unit makes a compelling case for computational thinking's integration into classes and therefore would increase attitudes toward computational thinking in both pre-service and in-service teachers? Additionally, it would be interesting to know whether this unit could be expanded to train both pre-service and in-service teachers to effectively teach computational thinking.

Future studies could also address the impact of computational thinking on students learning about different programming language tool. This study focused primarily on pre-service teachers and their use of the programming language tool Scratch. However, there are many programming language tools that are now being used within classrooms (e.g., ScratchJr, Alice,

etc.). This study might be replicated with ScratchJr because many Pre-K- 4 pre-service teachers are beginning to use ScratchJr. Alternatively, this study could be replicated with Alice for those secondary pre-service teachers. It would be important to see whether attitudes increase with both easier programming languages (ScratchJr) and more challenging programming languages (Alice).

A final recommendation would be to examine pre-service teachers in more qualitative ways than changes in their attitudes towards computational thinking. It would be interesting to administer assessments of critical thinking, problem solving, resourcefulness, teaching style preferences, and other measures that could give a more holistic profile of the pre-service teachers involved. Is there a range of profiles that predicts better integration/adaptation of computational thinking into classrooms, or is it that a teacher needs particular characteristics to successfully implement those changes? An investigation of a wider variety of teachers and these sorts of measures might begin to answer those questions. If teachers' attitudes toward computational thinking factor largely into the motivations for successful integration of computational thinking skills into classrooms, then this study has already found a short unit that is effective for instilling more open and positive attitudes.

Summary

The overall purpose of the study was to help pre-service teachers learn about computational thinking and how it differs from computer science. Moreover, pre-service teachers increased their awareness and attitudes of computational thinking, explore examples of computational thinking integrated into their subject areas, and experiment with examples of

computational thinking integrated activities for their subject areas with Scratch programming language tool.

This final chapter presented discussion, recommendations, and limitations of the study conducted for this dissertation. This was the first known study that used a computational thinking unit, which includes Scratch programming language, Scratch flash cards, debugging activities and Harvard CS50 online lecture unplugged activity. The present investigation expanded the existing research base by using computational thinking unit.

REFERENCES

Al-Bow, M., Austin, D., Edgington, J., Fajardo, R., Fishburn, J., Lara, C., Leutengger, S., & Meyer, S. (2009). Using game creation for teaching computer programming to high school students and teachers. *SIGCSE Bulletin*, 41(3), 104-108. ACM.

Alessi, S. M., & Trollip, S. R. (2001). *Multimedia for learning: Methods and development* (3rd ed.). Boston, MA: Allyn & Bacon.

Allen, F. E. (1981). The history of language processor technology in IBM. *IBM Journal of Research and Development*, 25(5), 535-548.

Amer, H., & Ibrahim, W. (2014). Using the iPad as a pedagogical tool to enhance the learning experience for novice programming students. *Proceeding of the Global Engineering Education Conference (EDUCON)*, 178-183. doi: 10.1109/EDUCON.2014.6826087

Basawapatna, A., Koh, K. H., Repenning, A., Webb, D. C., & Marshall, K. S. (2011). Recognizing computational thinking patterns. In *ACM Technical Symposium on Computer Science Education* (pp. 245-250). Dallas, TX: ACM Press.

Bednar, A. K., Cunningham, D., Duffy, T. M., & Perry, J. D. (1992). Theory into practice: How do we link? In T.M. Duffy, & D. H. Jonassen (Eds.), *Constructionism and the Technology of Instruction* (pp. 17-34). Hillsdale, NJ: Lawrence Erlbaum Associates Inc.

Bers, M. (2008). *Blocks to robots: Learning with technology in the early childhood classroom*. New York, NY: Teacher's College Press.

Bers, M. U. (2010). The TangibleK Robotics Program: Applied Computational Thinking for Young Children. *Early Childhood Research & Practice*, 12(2).

Black, A. P. (2013). Object-oriented programming: Some history, and challenges for the next fifty years. *Information and Computation*, 231, 3-20.

Blake, B., & Pope, T. (2008). Developmental psychology: Incorporating Piaget's and Vygotsky's theories in classrooms. *Journal of Cross-Disciplinary Perspectives in Education*, 1(1), 59–67.

Brennan, K. (2011). *Creative Computing: A design-based introduction to computational thinking.* Retrieved May, 2012, from http://scratched.media.mit.edu/sites/default/files/CurriculumGuide-v20110923.pdf

Brennan, K., & Resnick, M. (2012a). *New frameworks for studying and assessing the development of computational thinking.* Paper presented at the 2012 Annual Meeting of the American Educational Research Association, Vancouver, Canada.

Brennan, K., & Resnick, M. (2012b). *Using artifact-based interviews to study the development of computational thinking in interactive media design*. Paper presented at the 2012 Annual Meeting of the American Educational Research Association, Vancouver, Canada.

Brennan, K., & Resnick, M. (2013). Imagining, creating, playing, sharing, reflecting: How online community supports young people as designers of interactive media. In N. Lavigne, & C. Mouza (Eds.), *Emerging technologies for the classroom: A learning sciences perspective* (pp. 253-268). New York, NY: Springer.

Brennan, K., Balch, C., & Chung, M. (2014). *Creative Computing*. Cambridge, MA: Harvard Graduate School of Education.

Briggs, J.R., (2012). *Python For Children: A Playful Introduction to Programming*. San Francisco, CA: No Starch Press.

Bundy, A. (2007). Computational thinking is pervasive. *Journal of Scientific and Practical Computing*, 1(2), 67-69.

Burke, Q., & Kafai, Y.B. (2010). Programming & storytelling: Opportunities for learning about coding & composition. In *Proceedings of the 9th International Conference on Interaction Design and Children* (pp. 348-351). New York, NY: ACM.

Burke, Q., & Kafai, Y.B. (2012). The writers' workshop for youth programmers: digital storytelling with Scratch in middle school classrooms. In *Proceedings of the 43rd ACM Technical Symposium on Computer Science Education* (pp. 433-438). Raleigh, NC: ACM.

Bush, G. (2006). Learning about learning: From theories to trends. *Teacher Librarian*, 34(2), 14-18.

Cautili, J., Rosenwasser, B., & Hantula, D. (2003). Behavioral science as the art of the 21st century philosophical similarities between B.F. Skinner's radical behaviorism and postmodern science. *The Behavior Analyst Today*, 4(2), 225.

Cohoon, J.M., & Aspray, W. (Eds.). (2006). *Women and Information Technology: Research on Underrepresentation*. Cambridge, MA: MIT Press.

Cooper, S., Dann, W., & Pausch, R. (2003). Teaching objects-first in introductory computer science. *ACM SIGCSE Bulletin*, 35(1), 191-195.

Draper, R. J. (2002). School mathematics reform, constructivism, and literacy: A case for literacy instruction in the reform-oriented math classroom. *Journal of Adolescent and Adult Literacy*, 45, 520-529.

Driscoll, K. (2012). From Punched Cards to "Big Data": A Social History of Database Populism. *Communication +1*, 1(1), 4.

Duit, R., & Treagust, D. (1998). Learning in science: From behaviorism towards social constructivism and beyond. In B. Fraser, & K. Tobin (Eds.), *International handbook of science education* (pp. 3-16). Dordrecht, The Netherlands: Kluwer Academic Publishers.

Elgamel, L., & Sarrab, M. (2014). Selection of Programming Languages for Developing Distributed Systems. *World Applied Sciences Journal*, 31(10), 1791-1803.

Flannery, L.P., Silverman, B., Kazakoff, E.R., Bers, M.U., Bontá, P., & Resnick, M. (2013). Designing ScratchJr: Support for early childhood learning through computer programming. In *Proceedings of the 12th International Conference on Interaction Design and Children*. New York, NY: ACM.

Fosnot, C.T. (1996). Constructivism: A psychological theory of learning. In C.T. Fosnot (Ed.), *Constructivism: Theory, Perspectives, and Practice.* New York, NY: Teachers College Press.

Google For Education. (2015). *Exploring Computational Thinking.* Retrieved from https://www.google.com/edu/resources/programs/exploring-computational-thinking/

Google-Gallup. (2005). *Searching for Computer Science: Access and Barriers in U.S. K-12 Education.* Retrieved from http://services.google.com/fh/files/misc/searching-for-computer-science_report.pdf

Grover, S., & Pea, R. (2013). Computational Thinking in K-12: A review of the state of the field. *Educational Researcher*, 42(1), 38-43.

Hillegass, A., & Ward, M. (2013). *Objective-C Programming: The Big Nerd Ranch Guide*. Atlanta, GA: Big Nerd Ranch.

Hoegh, A., & Moskal, B. M. (2009). Examining science and engineering students' attitudes toward computer science. *Proceeding of the 39th IEEE Frontiers in Education Conference,* 1-6. doi: 10.1109/FIE.2009.5350836

Honebein, P.C. (1996). Seven goals for the design of constructivist learning environments. In B. Wilson (Ed.), *Constructivist learning environments* (pp. 11-24). Englewood Cliffs, NJ: Educational Technology.

Honey, M., Pearson, G., & Schweingruber, H. (2014). *STEM integration in K-12 education: Status, prospects, and an agenda for research*. Washington, D.C.: National Academic Press.

Jenkins, E. W. (2000). Constructivism in school science education: Powerful model or the most dangerous intellectual tendency? *Science & Education*, 9, 599-610.

Jonassen, D. H. (2000). Toward a design theory of problem solving. *Educational Technology Research and Development*, 48(4), 63-85.

Kafai, Y. & Resnick, M. (Eds.). (1996). *Constructivism in practice: Designing, thinking, and learning in a digital world.* Mahwah, NJ: Lawrence Erlbaum.

Kaur, R., Kumar, P., & Singh, R. P. (2014). A Journey of digital storage from punch cards to cloud. *IOSR Journal of Engineering*, 4(3), 36-41.

Kelleher, C., & Pausch, R. (2005). Lowering the barriers to programming: A taxonomy of programming environments and languages for novice programmers. *ACM Computing Surveys (CSUR)*, 37(2), 83-137.

Kim, H., Choi, H., Han, J., So, H. (2012). Enhancing teachers' ICT capacity for the 21st century learning environment: three cases of teacher education in Korea. *Australasian Journal of Education Technology (AJET)*, 28(6), 965-982.

Laffra, C., Blake, E. H., de Mey, V., & Pintado, X. (Eds.). (1995). *Object-oriented programming for graphics*. Berlin, Heidelberg: Springer-Verlag.

Langdon, D., McKittrick, G., Beede, D., Khan, B., & Doms, M. (2011). *STEM: Good jobs now and for the future* (ESA Issue Brief #03-11). Retrieved from U.S. Department of Commerce Economics and Statistics Administration website: http://www.esa.doc.gov/Reports/stem-good-jobs-now-and-future

Maloney, J. H., Peppler, K., Kafai, Y.B., Resnick, M. & Rusk, N. (2008). Programming by choice: Urban youth learning programming with Scratch. In *Proceedings of the 39th SIGCSE Technical Symposium on Computer Science Education* (pp. 367-371). New York, NY: ACM.

Maloney, J., Resnick, M., Rusk, N., Silverman, B., & Eastmond, E. (2010). The programming language and environment. *ACM Transactions on Computing Education (TOCE)*, 10(4), 1-15.

Mascolo, M. F., & Fischer, K. W. (2005). Constructivist theories. In *Cambridge encyclopedia of child development* (pp. 49-63). Cambridge, UK: Cambridge University Press.

Ottenbreit-Leftwich, A. T., Glazewski, K. D., Newby, T. J., & Ertmer, P. A. (2010). Teacher value beliefs associated with using technology: Addressing professional and student needs. *Computers & Education*, 55(3), 1321-1335.

Papert, S. (1980). *Mindstorms: Children, computers, and powerful ideas*. New York, NY: Basic Books.

Papert, S. (1993). *The children's machine: Rethinking school in the age of the computer.* New York, NY: Basic Books.

Papert, S. (2005). Teaching children thinking. *Contemporary Issues in Technology and Teacher Education*, 5(3), 353-365.

Papert, S., & Harel, I. (1991). Situating constructionism. *Constructionism*, 36, 1-11.

Partovi, H. (2015). A comprehensive effort to expand access and diversity in computer science. *ACM Inroads*, 6(3), 67-72.

Perez, R. E., Jansen, P. W., & Martins, J. R. (2012). pyOpt: a Python-based object-oriented framework for nonlinear constrained optimization. *Structural and Multidisciplinary Optimization*, 45(1), 101-118.

Piaget, J., & Inhelder, B. (1969). *The psychology of the child*. New York, NY: Basic Books.

Portelance, D. J., & Bers, M. U. (2015). Code and tell: Assessing young children's learning of computational thinking using peer video interviews with ScratchJr. In *Proceedings of the 14th International Conference on Interaction Design and Children* (pp. 271-274). New York, NY: ACM.

Ragonis, N., Hazzan, O., & Gal-Ezer, J. (2010). A survey of computer science teacher preparation programs in Israel tells us: Computer science deserves a designated high school teacher preparation! In *Proceedings of the 41st ACM Technical Symposium on Computer Science Education* (pp. 401-405). New York, NY: ACM.

Resnick, M., Bruckman, A., & Martin, F. (1996). Pianos not stereos: Creating computational construction kits. *Interactions*, 3(5), 40-50.

Resnick, M. (2007). All I really need to know (about creative thinking) I learned (by studying how children learn) in kindergarten. In *Proceedings of the 6th ACM SIGCHI Conference on Creativity & Cognition*. New York, NY: ACM.

Resnick, M., Maloney, J., Monroy-Hernandez, A., Rusk, N., Eastmond, E., Brennan, K., … Kafai, Y. (2009). Scratch: Programming for all. *Communications of the ACM*, 52(11), 60-67.

Resnick, M. (2013). Learn to code, code to learn. *EdSurge*. Retrieved from https://www.edsurge.com/news/2013-05-08-learn-to-code-code-to-learn

Rogoff, B. (1994). Developing understanding of the idea of communities of learners. *Mind, Culture, and Activity*, 1, 209-229.

Rummel, E. (2008). Constructing cognition. *American Scientist*, 96(1), 80-82.

Severance, C. (2012). Bertrand Meyer: Software Engineering and the Eiffel Programming Language. *Computer*, 45(9), 6-8.

Shield, G. (2000). A critical appraisal of learning technology using information and communication technologies. *Journal of Technology Studies,* 26(1), 71-79.

Singh, J., & Abraham, J. (2014). JAVA Improvised Approach to Java. *International Journal of Research*, 1(10), 1138-1144.

Slavin, R. E. (1990). Research on cooperative learning: Consensus and controversy. *Educational leadership*, 47(4), 52-54.

Stephenson, C. (2009). IT is a pivotal time for K-12 computer science. *Communications of the ACM,* 52(12), 5-5.

Stetsenko, A., & Arievitch, I. M. (2004). Vygotskian collaborative project of social transformation: History, politics, and practice in knowledge construction. *The International Journal of Critical Psychology,* 12(4), 58-80.

Stroustrup, B. (1988). What is object-oriented programming? *Software, IEEE,* 5(3), 10-20.

Sutton, M. J. (2003). Problem representation, understanding, and learning transfer: Implications for technology education research. *Journal of Industrial Teacher Education*, 40(4), 47-63.

Tipps, S. (1987). *Beginning with Logo: Terrapin Version.* Prentice-Hall. Englewood Cliffs, NJ.

Tondeur, J., Van Braak, J., Sang, G., Voogt, J., Fisser, P., & Ottenbreit-Leftwich, A. (2012). Preparing pre-service teachers to integrate technology in education: A synthesis of qualitative evidence. *Computers & Education*, 59(1), 134-144.

Trikha, B. (2010). A Journey from floppy disk to cloud storage. *International Journal on Computer Science and Engineering*, 2(4), 1449-1452.

Utting, I., Cooper, S., Kölling, M., Maloney, J., & Resnick, M. (2010). Alice Greenfoot, and Scratch—A discussion. *ACM Transactions on Computing Education (TOCE)*, 10(4), 17.

Viennot, N., Garcia, E., & Nieh, J. (2014). A measurement study of Google Play. In *The 2014 ACM International Conference on Measurement and Modeling of Computer Systems* (pp. 221-233). New York, NY: ACM.

Voskoglou, M. G., & Buckley, S. (2012). Problem solving and computational thinking in a learning environment. *Egyptian Computer Science Journal*, 36(4), 28-45.

Vygotsky, L. (1978). Interaction between learning and development. *Readings on the Development of Children*, *23*(3), 34-41.

White-Clark, R., DiCarlo, M., & Gilcgriest, S. N. (2008). "Guide on the side": An instructional approach to meet mathematics standards. *The High School Journal*, 9(14), 40-44.

Wiemer, S. (2011). Computer history and the movement of business simulations. *Proceedings of the 2011 DIGRA International Conference*, 1-7. Retrieved from http://www.digra.org/wp-content/uploads/digital-library/11310.52587.pdf

Wing, J. M. (2006). Computational thinking. *Communications of the ACM*, 49(3), 33-35.

Wing, J. M. (2008a). Computational thinking and thinking about computing. *Philosophical Transactions of the Royal Society*, 366, 3717-3725.

Wing, J. M. (2008b). Five deep questions in computing. *Communications of the ACM*, 51(1), 58-60.

Yadav, A., Mayfield, C., Zhou, N., Hambrusch, S., & Korb, J. T. (2014). Computational thinking in elementary and secondary teacher education. *ACM Transactions on Computing Education (TOCE)*, 14(1), 5.

Zeidler, D. L. (2002). Dancing with maggots and saints: Visions for subject matter knowledge, pedagogical knowledge, and pedagogical content knowledge in science teacher education reform. *Journal of Science Teacher Education*, 13(1), 27-42.

APPENDIX A

Demographics

Gender

------- Male

------- Female

Race/Ethnicity

------- African American/Black

------- Asian/Pacific Islander

------- Hispanic/Latino

------- Multiracial

------- Native American / American Indian

------- White

------- Not listed (please specify)

Age

------- 18 to 24 years

------- 25 to 34 years

------- 35 to 44 years

------- 45 to 54 years

------- 55 to 64 years

------- 65 to older

Content area

------- Pre-K4

------- Interdisciplinary

------- English

------- Math

------- Science

------- Social Studies

------- Art or Music

------- Other (Please specify)

What is your current cumulative GPA?

APPENDIX B

Pre-survey & Post-survey

Questions	Strongly Disagree (1)	Disagree (2)	Neutral (3)	Agree (4)	Strongly Agree (5)
Computational thinking is understanding how computers work	(1)	(2)	(3)	(4)	(5)
Computational thinking involves thinking logically to solve problems	(1)	(2)	(3)	(4)	(5)
Computational thinking involves using computers to solve problems	(1)	(2)	(3)	(4)	(5)
Computational thinking involves abstracting general principles and applying them to other situations	(1)	(2)	(3)	(4)	(5)
I do not think it is possible to apply computing knowledge to solve other problems	(1)	(2)	(3)	(4)	(5)

I am not comfortable with learning computing concepts	(1)	(2)	(3)	(4)	(5)
I can achieve good grades (C or better) in computing courses	(1)	(2)	(3)	(4)	(5)
I can learn to understand computing concepts	(1)	(2)	(3)	(4)	(5)
I do not use computing skills in my daily life	(1)	(2)	(3)	(4)	(5)
I doubt that I have the skills to solve problems by using computer application	(1)	(2)	(3)	(4)	(5)
I think computer science is boring	(1)	(2)	(3)	(4)	(5)
The challenge of solving problems using computer science appeals to me	(1)	(2)	(3)	(4)	(5)
I think computer science is interesting	(1)	(2)	(3)	(4)	(5)

I will voluntarily take computing courses if I were given the opportunity	(1)	(2)	(3)	(4)	(5)
Computational thinking can be incorporated in the classroom by using computers in the lesson plan	(1)	(2)	(3)	(4)	(5)
Computational thinking can be incorporated in the classroom by allowing students to problem solve	(1)	(2)	(3)	(4)	(5)
Knowledge of computing will allow me to secure a better job	(1)	(2)	(3)	(4)	(5)
My career goals do not require that I learn computing skills	(1)	(2)	(3)	(4)	(5)
I expect that learning computing skills will help me to achieve my career goals	(1)	(2)	(3)	(4)	(5)

I hope that my future career will require the use of computing concepts	(1)	(2)	(3)	(4)	(5)
Having background knowledge and understanding of computer science is valuable in and of itself	(1)	(2)	(3)	(4)	(5)

(Yadav, Mayfield, Zhou, Hambrusch, & Korb, 2014)

APPENDIX C

Letter of consent

Duquesne University IRB
Protocol #2016-01-3
Approved: 2-17-2016
Exempt Study

DUQUESNE UNIVERSITY
600 FORBES AVENUE ♦ PITTSBURGH, PA 15282

CONSENT TO PARTICIPATE IN A RESEARCH STUDY

TITLE: Attitudes of Pre-service Teachers Toward Computational Thinking in Education.

INVESTIGATOR: Bekir Mugayitoglu, M.B.A.
Ed.D. Candidate in Instructional Technology & Leadership

mugayit635@duq.edu

ADVISOR: Joseph Kush, Ph.D.
Professor
School of Education
Instruction and Leadership in Education
327C Fisher Hall
600 Forbes Avenue
Pittsburgh, PA 15282
412-396-1151
kush@duq.edu

SOURCE OF SUPPORT: This study is being performed as partial fulfillment of the requirements for the doctoral degree in the School of Education at Duquesne University

PURPOSE: You are being asked to participate in a research project that seeks to investigate pre-service teachers' attitudes toward computational thinking in a traditional classroom. As a part of this class you will be required to participate in two modules that will introduce you to computational thinking. The purpose of this research study is to collect data examining pre-service teachers' attitudes toward computational thinking with these teaching modules.

Duquesne University IRB
Protocol #2016-01-3
Approved: 2-17-2016
Exempt Study

To participate in this study, you must be:

- At least 18 years old.
- A Duquesne University student.
- Able to read and type on a keyboard.
- Enrolled in an Instructional Technology course.

PARTICIPANT PROCEDURES:

Specifically, you are being asked to participate in the research study by completing a pre-test and post-test questionnaire that asks you to rate your learning and your attitude toward the topic of computational thinking. The pre/post test will take approximately 15 minutes each to complete. In addition, you will be contacted three weeks following the lesson on computational thinking and asked to complete the survey a final time.

Demographic information and computing attitude assessment will be assessed through a paper-based survey during your regularly scheduled class. You will take a pre-test prior to learning about computational thinking, and a post-test following the units on computational thinking, and then once again three weeks later.

These are the only requests that will be made of you.

RISKS AND BENEFITS:

There are no risks anticipated greater than those encountered in everyday life. There are no direct benefits, other than the knowledge gained from the learning computational thinking and programming language, and helping instructors determine their usefulness in curriculum.

COMPENSATION:

You will not be compensated in any way; however, participation in the project will require no monetary cost to you.

CONFIDENTIALITY:

Your name will never appear on any survey or research instruments. No identification of you or your institution will be made in the data analysis. You may want to explain the coding system here as a way to

reassure them they will not be tracked. All data will be kept secure and only be accessible to the research team and academic advisors at Duquesne University. Your response(s) will only appear in statistical data summaries. Any data collected will be kept for five years following the completion of the research.

RIGHT TO WITHDRAW: You are under no obligation to participate in this study. Your participation in this research will not affect your classroom grades. The research is not related to your academic progress in your program, and there will be no consequences for withdrawing. You are free to withdraw your consent to participate at any time. If you choose to withdraw your participation after the modules are completed, simply provide the number that was issued to you on the date of the study to the researcher. The course instructor will not know whether or not you participated.

SUMMARY OF RESULTS: A summary of the results of this research will be supplied to you, at no cost, upon request.

VOLUNTARY CONSENT: I have read the above statements and understand what is being requested of me. I also understand that my participation is voluntary and that I am free to withdraw my consent at any time, for any reason. On these terms, I certify that I am willing to participate in this research project.

I understand that should I have any further questions about my participation in this study, I may call the researcher, Bekir Mugayitoglu (412-580-4680), or his doctoral advisor, Dr. Joseph Kush (412-396-1151). For questions related to human subjects in research, you may contact Dr. Linda Goodfellow, Chair of the Duquesne University Institutional Review Board (412-396-1886).

_____ _____

Participant's Signature Date

_____ _____

Researcher's Signature Date

APPENDIX D

Module 1 Lesson Plan

(50 minutes)

Lesson Plan adapted from Google for Education (Google for Education, 2015) and Scratch Computing Curriculum (Brennan, 2011), and also Scratch website (https://scratch.mit.edu/).

Lesson Description

The goal of this unit is to help pre-service teachers learn about computational thinking, how it differs from computer science and how it can be integrated into a variety of subject areas. Upon completion of this unit, they will increase their awareness of computational thinking, explore examples of computational thinking integrated into their subject areas, and experiment with examples of computational thinking integrated activities for their subject areas with Scratch programming language tool.

Learning objectives

By the end of this session, pre-service teachers will be able to:

- Explore computational thinking.
- Explore Scratch programming language tool.
- Identify potential benefits of utilizing computational thinking.
- Understand computational thinking concepts: (Sequence, loops, events, parallelism, and conditionals).
- Understand computational thinking practices (debugging and abstracting).
- Understand computational thinking steps (Decomposition, pattern recognition, abstraction, and algorithm design).

Materials and Preparation

Overhead projector to display presentation.

Power point

20 PC Desktop Computers

Time Frame (50 minutes)

Introduction	5 minutes
Computational thinking steps, concepts, and practices	15 minutes
Video(s)	10 minutes (CS50,
Week 0, continued)	
(https://www.youtube.com/watch?v=KUB-aJXquUA	
(Steve Jobs on programming)	
https://www.youtube.com/watch?v=5Z1gfgM7kzo	
Scratch programming language tool	15 minutes
Closing	5 minutes

Introduction – 5 minutes

Beginning unit with introductory lesson that overviews important aspects of computational thinking and programming language tool, Scratch. I will explain the definition of computational thinking concepts, practices, and steps as well definition of programming language. This focuses mainly on computational thinking and Scratch programming language tool.

Computational thinking steps and concepts – 15 minutes

Pre-service teachers are presented with a description of computational thinking, computational thinking steps and concepts.

Explain four computational thinking steps:

- Decomposition: Taking a big, difficult, and complex problem and breaking it down into smaller, more manageable sub-problems.
- Pattern recognition: Identifying common similarities and differences
- Abstraction: Creating step-by-step techniques for solving problems
- Algorithm design: Providing significant instructions with a step-by-step solution for a problem and pulling out significant details to find one solution that applies multiple similar problems.

Explain computational thinking concepts:

- Sequence: A sequence is a list of code blocks that are put in a specific order to be run by a computer.
- Loops: A loop allows a programmed sequence of instructions to repeat multiple times.
- Parallelism: Parallelism allows several tasks to run at the same time.
- Events: One thing starts happening because another thing is triggered
- Conditionals: One thing occurs depending on the situations of other things.

Videos – 10 minutes

Demonstrate to them the video from CS50, Week 0, continued 19:10 to 26:10 with this URL (https://www.youtube.com/watch?v=KUB-aJXquUA) and (https://www.youtube.com/watch?v=5Z1gfgM7kzo). Explain that pre-service teachers are going to learn what programming language through real life example.

Scratch programming language tool – 15 minutes

Pre-service teachers will learn Scratch programming language tool for programming (https://scratch.mit.edu/projects/editor/). Demonstrate them how to use Scratch programming language tool.

Show them how to:

- Add new objects
- Use toolbox
- Add block of codes
- Drag and drop a new block on editing center
- Add new background.

Closing – 5 minutes

If time permits, pre-service teachers ask questions. I will summarize the main points of the lesson. I will conclude the lesson only by summarizing the main points, but also by previewing the next lesson. I will explain how does the topic relate to the one that is coming.

APPENDIX E

Module 2 Lesson Plan

(50 minutes)

Lesson Plan adapted from Google for Education (Google for Education, 2015), Scratch Computing Curriculum (Brennan, 2011), and also Scratch website (https://scratch.mit.edu/).

Learning Description

I will give the pre-service teachers clear instructions on what they will have to do at the hands-on activities and how to complete the lesson. By the end of this session, students will be able to understand computational thinking: Computational thinking concepts, practices, and steps. In addition to, pre-service teachers will know how to use Scratch programming language tool. Hands-on activities will require pre-service teachers to use Scratch to implement computational thinking concepts, practices and steps.

Learning objectives

By the end of this session, pre-service teachers will be able to:

- Synthesize various resources that support their computational creation.
- Evaluate possibilities for their own Scratch computational creation.
- Effectively apply scratch programming language tool to use computational thinking.
- Implement computational thinking concepts and practices through Scratch programming language tool.

Materials and Preparation

Overhead projector to display the presentation.

One desktop PC per student

Handouts

- Scratch Cards (Appendix F) (https://scratch.mit.edu/help/cards/)
- Scratch Debugging activities (Appendix E)

Time Frame (50 minutes)

Introduction	5 minutes
Hands-on activities 5 flash cards (Scratch)	20 minutes
Hands-on activities 5 debugging activities	20 minutes
Closing	5 minutes

Introduction – 5 minutes

Beginning the unit to explain what kind of hands-on activities they will do. I will walk around the class checking on their progress and also make sure pre-service teachers are able to access Scratch programming language tool. Each pre-service teacher will be responsible for assigned programming. I will help pre-service teachers who are struggling with programming. In addition to helping them, I will be assisting. The pre-service students will work on their projects and then share their projects voluntarily with their peers to display on the projector.

Teachers will also acquire the necessary skills to scaffold students in open-ended exploration. I will describe and introduce the range of projects they will be able to create by using Scratch programming language tool.

Hands-on activities 5 flash cards (Scratch) – 20 minutes

Have pre-service teachers create a new project and begin the programming process with 5 flash cards (Appendix F). Explain to pre-service teachers that they are going to use Scratch

programming language tool. Distribute Scratch card handouts. Tell pre-service students to create their projects based on Scratch cards. Scratch programming language tool will allow pre-service teachers experiencing programming language, imagining possibilities as well as creating projects such as: video games, simulations, animations, interactive presentations, and interactive stories. Scratch cards will scaffold pre-service teachers to create these projects. These Scratch cards developed from MIT Scratch team and published at Scratch website (https://scratch.mit.edu/help/cards/).

Hands-on activities 5 debugging activities - 20 minutes

Pre-service teachers will debug with assigned 5 Scratch debugging programs (Appendix E). Pre-service teachers Students will participate in follow-up discussion in which he/she explains how he/she solved debugging activity. What strategies he/she used to debug it.

DEBUG IT! 1.1 https://scratch.mit.edu/projects/10437040/
DEBUG IT! 1.2 https://scratch.mit.edu/projects/10437249/
DEBUG IT! 1.3 https://scratch.mit.edu/projects/10437366/
DEBUG IT! 1.4 https://scratch.mit.edu/projects/10437439/
DEBUG IT! 1.5 https://scratch.mit.edu/projects/10437476/

Closing – 5 minutes

If time permits, I will encourage pre-service teachers to share their projects with peers. In addition, I will lead each pre-service teacher in creating a tangible expression of his or her understanding.

APPENDIX F

Scratch Debugging activities

DEBUG IT!

HELP! CAN YOU DEBUG THESE FIVE SCRATCH PROGRAMS?

In this activity, you will investigate what is going awry and find a solution for each of the five Debug It! challenges.

START HERE

- ❑ Go to the Unit 1 Debug It! studio: http://scratch.mit.edu/studios/475483
- ❑ Test and debug each of the five debugging challenges in the studio.
- ❑ Write down your solution or remix the buggy program with your solution.

❑ **DEBUG IT! 1.1** http://scratch.mit.edu/projects/10437040

When the green flag is clicked, both Gabe and Scratch Cat should start dancing. But only Scratch Cat starts dancing! How do we fix the program?

❑ **DEBUG IT! 1.2** http://scratch.mit.edu/projects/10437249

In this project, when the green flag is clicked, the Scratch Cat should start on the left side of the stage, say something about being on the left side, glide to the right side of the stage, and say something about being on the right side. It works the first time the green flag is clicked, but not again. How do we fix the program?

❑ **DEBUG IT! 1.3** http://scratch.mit.edu/projects/10437266

The Scratch Cat should do a flip when the space key is pressed. But when the space key is pressed, nothing happens! How do we fix the program?

❑ **DEBUG IT! 1.4** http://scratch.mit.edu/projects/10437439

In this project, the Scratch Cat should pace back and forth across the stage, when it is clicked. But the Scratch Cat is flipping out – and is walking upside down! How do we fix the program?

❑ **DEBUG IT! 1.5** http://scratch.mit.edu/projects/10437476

In this project, when the green flag is clicked, the Scratch Cat should say "Meow, meow, meow!" in a speech bubble and as a sound. But the speech bubble happens before the sound – and the Scratch Cat only makes one "Meow" sound! How do we fix the program?

FEELING STUCK?

THAT'S OKAY! TRY THESE THINGS...

- ❑ Make a list of possible bugs in the program.
- ❑ Keep track of your work! This can be a useful reminder of what you have already tried and point you toward what to try next.
- ❑ Share and compare your problem finding and problem solving approaches with a neighbor until you find something that works for you!

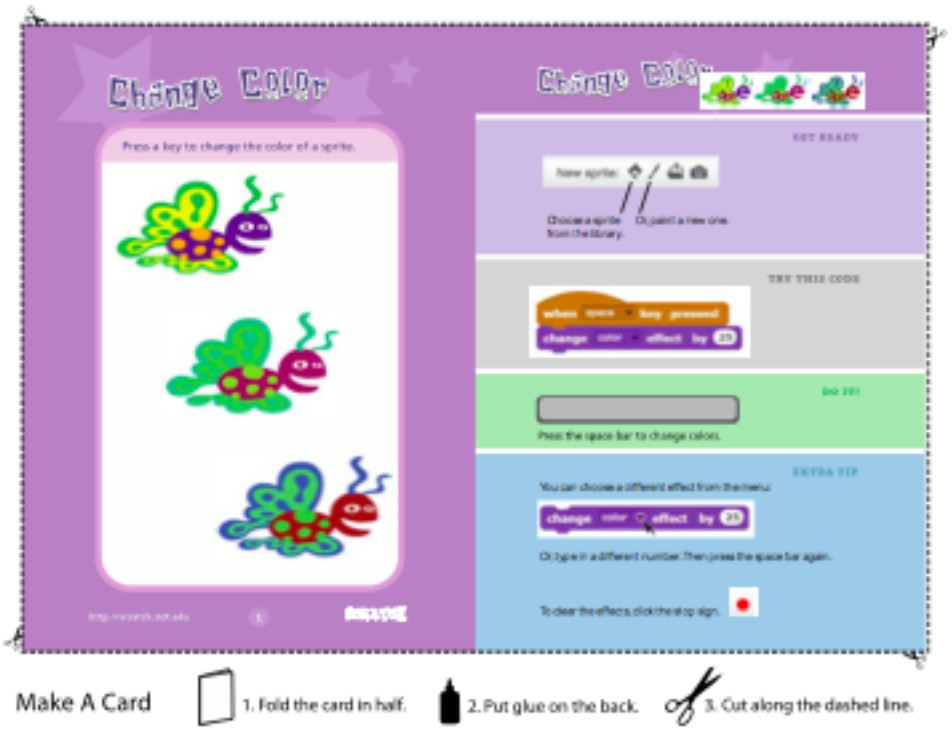
FINISHED?

- + Discuss your testing and debugging practices with a partner. Make note of the similarities and differences in your strategies.
- + Add code commentary by right clicking on blocks in your scripts. This can help others understand different parts of your program!
- + Help a neighbor!

(Brennan, Balch, & Chung, 2014)

APPENDIX G

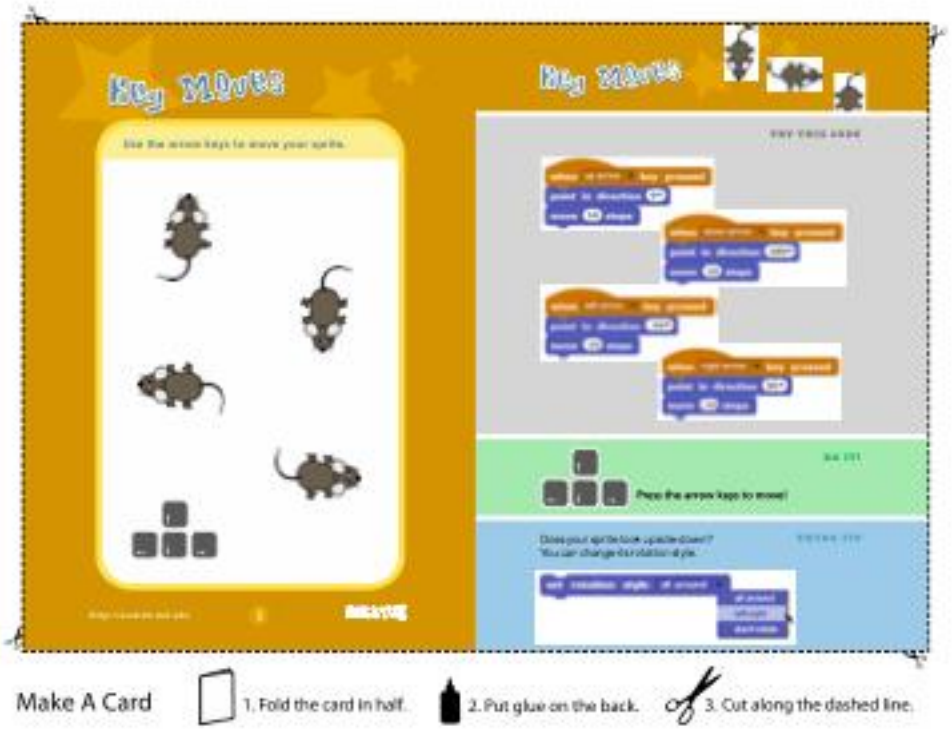
Scratch Cards



Make A Card 1. Fold the card in half. 2. Put glue on the back. 3. Cut along the dashed line.

https://scratch.mit.edu/help/cards/

https://scratch.mit.edu/help/cards/



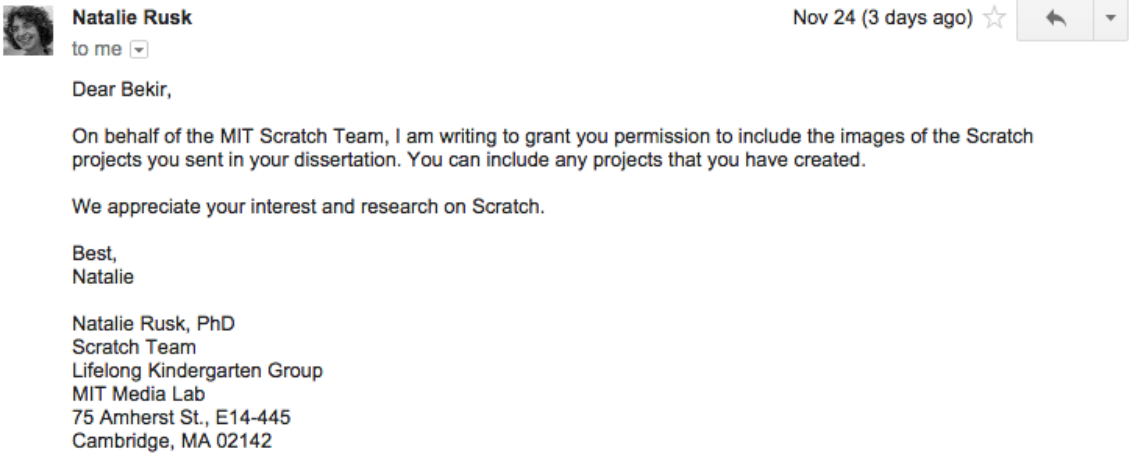
https://scratch.mit.edu/help/cards/

https://scratch.mit.edu/help/cards/

https://scratch.mit.edu/help/cards/

APPENDIX H

Image Permissions



Natalie Rusk Nov 24 (3 days ago)

to me

Dear Bekir,

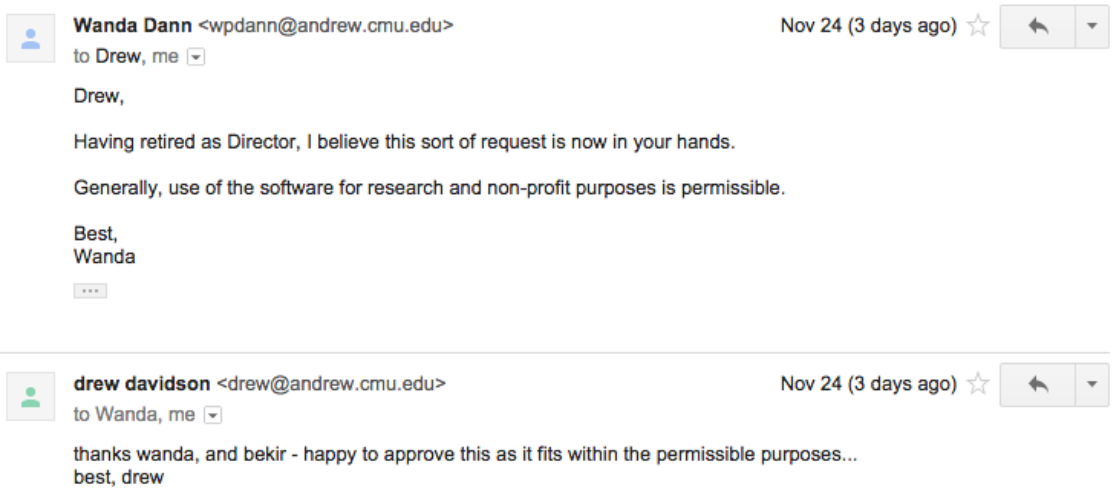
On behalf of the MIT Scratch Team, I am writing to grant you permission to include the images of the Scratch projects you sent in your dissertation. You can include any projects that you have created.

We appreciate your interest and research on Scratch.

Best,
Natalie

Natalie Rusk, PhD
Scratch Team
Lifelong Kindergarten Group
MIT Media Lab
75 Amherst St., E14-445
Cambridge, MA 02142

Permission to use Scratch from Dr. Rusk



Wanda Dann <wpdann@andrew.cmu.edu> Nov 24 (3 days ago)

to Drew, me

Drew,

Having retired as Director, I believe this sort of request is now in your hands.

Generally, use of the software for research and non-profit purposes is permissible.

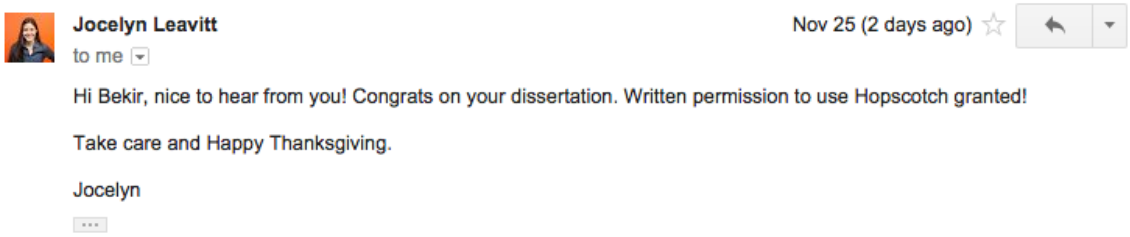
Best,
Wanda

drew davidson <drew@andrew.cmu.edu> Nov 24 (3 days ago)

to Wanda, me

thanks wanda, and bekir - happy to approve this as it fits within the permissible purposes...
best, drew

Permission to use Alice from Mr. Davidson



Jocelyn Leavitt Nov 25 (2 days ago)

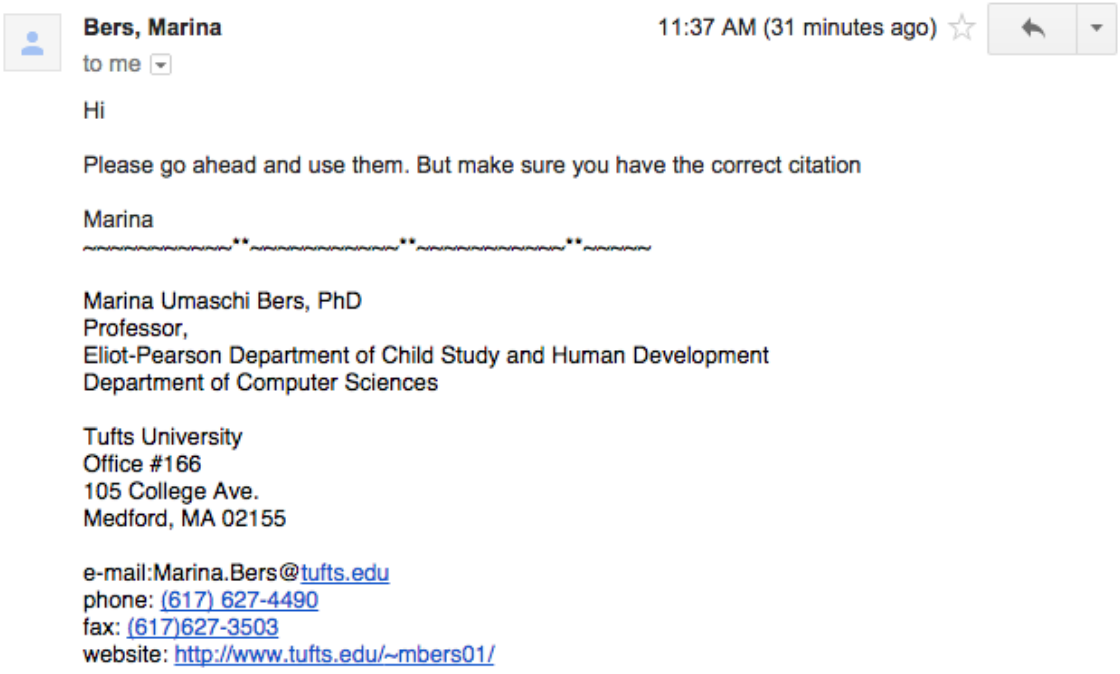
to me

Hi Bekir, nice to hear from you! Congrats on your dissertation. Written permission to use Hopscotch granted!

Take care and Happy Thanksgiving.

Jocelyn

Permission to use Hopscotch from Mrs. Leavitt



Bers, Marina 11:37 AM (31 minutes ago)

to me

Hi

Please go ahead and use them. But make sure you have the correct citation

Marina

Marina Umaschi Bers, PhD
Professor,
Eliot-Pearson Department of Child Study and Human Development
Department of Computer Sciences

Tufts University
Office #166
105 College Ave.
Medford, MA 02155

e-mail:Marina.Bers@tufts.edu
phone: (617) 627-4490
fax: (617)627-3503
website: http://www.tufts.edu/~mbers01/

Permission to use ScratchJr from Dr. Bers

APPENDIX I

Survey Permission

ASSOCIATION FOR COMPUTING MACHINERY, INC. LICENSE TERMS AND CONDITIONS

Nov 07, 2016

This Agreement between Bekir Mugayitoglu ("You") and Association for Computing Machinery, Inc. ("Association for Computing Machinery, Inc.") consists of your license details and the terms and conditions provided by Association for Computing Machinery, Inc. and Copyright Clearance Center.

All payments must be made in full to CCC. For payment instructions, please see information listed at the bottom of this form.

License Number	3983720299775
License date	Nov 07, 2016
Licensed Content Publisher	Association for Computing Machinery, Inc.
Licensed Content Publication	ACM Transactions on Computing Education (TOCE)
Licensed Content Title	Computational Thinking in Elementary and Secondary Teacher Education
Licensed Content Author	Aman Yadav, et al
Licensed Content Date	Mar 1, 2014
Licensed Content Volume Number	14
Licensed Content Issue Number	1
Volume number	14
Issue number	1
Type of Use	Thesis/Dissertation
Requestor type	Academic
Format	Print and electronic
Portion	Excerpt (< 250 words)
Will you be translating?	No
Order reference number	
Title of your thesis/dissertation	ATTITUDES OF PRE-SERVICE TEACHERS TOWARD COMPUTATIONAL THINKING IN EDUCATION
Expected completion date	Dec 2016
Estimated size (pages)	144

Permission to use Survey from ACM

Definition of terms

STEM: Acronym for the fields of science, technology, engineering, and math.

Pre-service teacher: College student who is training to teach, classes provided to student-teachers before they have any teaching responsibility.

STEM pre-service teachers: Math, Science, and Computer teacher.

Non-STEM pre-service teachers: Special Education, Pre-K4, Language Art, Music, Art, and Social Studies/History teacher.

Computational thinking: Computational thinking enhances human thinking by using imaginative ideas to create new things by using the computer or without computer.

Programming language: An artificial language used to write instructions that a computer or tablet can understand to do programmer wants.

iPad-based language program: Any programming language applications whose action of delivery is an iPad.

Computer-based language program: Any programming language tools whose action of delivery is a computer.

Digital media: Computerized tools such as data, animations text, graphics, audio, and video that can be transferable and publishable a computer through Internet.