DEVELOPMENT OF COMPUTER SCIENCE ONLINE AND PRELIMINARY VALIDATION OF ITS EFFICACY AS AN INSTRUCTIONAL ENVIRONMENT

by

Gregory Paul Halopoff

Bachelor of Science
University of California, Los Angeles
1982

Master of Science
University of Southern California
1986

A dissertation submitted in partial fulfillment
of the requirements for the

Doctor of Philosophy Degree in Curriculum and Instruction
Department of Curriculum and Instruction
College of Education

Graduate College
University of Nevada, Las Vegas
December 2003

UNLV
UNIVERSITY OF NEVADA LAS VEGAS

Dissertation Approval

The Graduate College
University of Nevada, Las Vegas

November 25, 2003

The Dissertation prepared by

Gregory Paul Halopoff

Entitled

Development of Computer Science Online and Preliminary Validation of its Efficacy as an Instructional Environment

is approved in partial fulfillment of the requirements for the degree of

Doctor of Philosophy in Curriculum and Instruction

Examination Committee Chair

Dean of the Graduate College

Examination Committee Member

Examination Committee Member

Graduate College Faculty Representative

ABSTRACT

Development of Computer Science Online and Preliminary Validation of its Efficacy As an Instructional Environment

By

Greg P. Halopoff

Dr. Neal Strudler, Dissertation Committee Chair
Professor of Curriculum and Instruction
University of Nevada, Las Vegas

Over the last decade, computer science has become a fragmented and misunderstood subject. Part of this can be attributed to advances in technology, which have led to increased interest in new technology departments and course offerings. Former industrial arts subjects have been absorbed into these new departments along with computer science, resulting in a less academic standing the subject once held. Furthermore, emphasis on advanced placement (AP) computer science and Java has targeted higher achieving students, resulting in declining interest and enrollment as average students show more interest in tool-based technology courses.

CS Online was developed as an instructional environment to address many issues facing computer science education. One of these is the need to rekindle interest in introductory computer science. CS Online seeks to accomplish this by offering active learning experiences set in real-world contexts. The intended outcomes are increased interest in computer science as an academic discipline, increased enrollments in related courses, and increased achievement resulting from cognitive skills growth.

The CS Online system generated data while 36 high school students solved programming problems, and questionnaires administered by the system were used to collect information about students' self-regulatory skills and experience in math and computers. In addition, qualitative data analysis of source code submitted by students was conducted to determine how students progressed through the problem solving process and the common mistakes they made.

The study revealed that students with differing levels of math and computer experience and self-regulatory skills were able to adequately complete programming problems using the system. The descriptive data on the 36 students indicated that students with high motivation seemed to outperform low motivation students in all performance measures in the study. Those who had high planning skills also seemed to outperform the low group in most of the performance measures. A similar pattern was observed in the students with high versus low math and computer skills. As the task difficulty increased, students with high planning skills seemed to require increasingly fewer attempts to complete exercises than those with lower planning skills. A qualitative analysis of problem solving revealed that students erred in syntax, logic, and then grammar – in that order. It was also shown that students spent considerable time re-running programs to observe output or to clean-up code.

Although the findings suggest that in general motivation and planning seem to be important components of learning a programming language, the current descriptive findings should be interpreted with caution. Future studies with larger sample sizes are warranted. To examine effects of self-regulation on learning and performance, other

relevant variables, such as existing computer language skills, may be included to control their effects on the performance.

Additional findings suggest that the use of hints were helpful for students with lower math skills, computer skills, and motivation. Teachers can encourage the use of hints for those who need the extra help, but can discourage their use for the more highly skilled and motivated. The findings also suggest that, based on the types of mistakes students commonly made, instruction on debugging skills should be considered to reduce the number of syntax, logic, and grammar errors. Less time spent correcting errors becomes more time spent on problem solving.

Findings from the present study can be useful for further research and development of CS Online. CS Online is currently being used by high schools in the Clark County School District in Nevada.

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

First and foremost, I want to thank to my dissertation committee chairperson, Dr. Neal Strudler, for being the mentor and guide he has been since the beginning of this project and for the last five years of my life. I also want to thank my dissertation committee members: Dr. Randy Boone, Dr. Kendall Hartley, and Dr. Eunsook Hong for their continued support throughout this project. A very special thank you belongs to Dr. Chongwei Ran, programmer extraordinaire, who invested a piece of his soul into the CS Online Web system. I'm forever grateful for my beautiful wife, Margaret, who supported me every step of the way; she is truly the other side of me. My wonderful kids Bethany, Michael, and Timmy deserve extra thanks for understanding the countless hours invested. And finally, praises to Don Reynolds and Randy Caldwell who kept my mind fresh by listening, providing unending encouragement and prayers, performing music together, and being there on some very long runs.

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

Purpose of the Study

The purpose of the study was to develop a Web-based learning system as a teaching and learning tool for introductory computer science concepts, and then perform preliminary validation of the system's efficacy as an instructional environment. More specifically, measures of student self-regulation, math experience, and computer experience would be compared to various performance measures resulting from use of the system. Common mistakes made by students while solving computer programs would also be observed.

Background

Computer science education has undergone radical change over the last ten years and, as a result, has become a fragmented and misunderstood subject (Deek & Kimmel, 1998; Tucker, 1996). Many complex factors have contributed to this problem, but some believe that, at the core, is the lack of a widely adopted high school curriculum and standards for teacher certification (Deek & Kimmel, 1999). While this is evidenced by surveys that reported continued fragmentation over the course of ten years (Stephenson, 1997; Taylor & Norris, 1988), and national and state standards have ignored computer science as an

academic discipline (CDE, 1996; NDE, 2000; NCGE, 1994; NCTM, 1989; NJDE, 1996; NRC, 1996; NYDE, 1994), other factors may be equal if not greater contributors to the problem.

To begin, the first Advanced Placement (AP) Computer Science Exam was offered in 1984 using Pascal as the programming language. The exam was then changed to C++ in May of 1999, and then changed again in 2003 to Java (College Board, 2003a). These changes have had a disastrous impact on the teacher workforce who, while not yet comfortable with C++, have suddenly found themselves facing a bigger wave of difficult and complex change:

> At present 79% of high school computer science teachers rated their current knowledge of Java as poor to fair and only 3% rated their knowledge as excellent. At the same time, 86% rated their personal need to learn Java as very important to critical and 89% indicated that they needed to do so within one year. (Stephenson, 2002, p. 2)

Java has evolved to become a complex language. With close to 50,000 public methods in the Java 2 application programming interface (API) hierarchy (Sun, 2003), teachers have been required to achieve the extremely difficult, if not impossible task of teaching the language with little to no training and support in the schools (Stephenson, 2002). While some authors have suggested that Java has achieved the same level of academic prominence Pascal reached in the 1980's (Wallace, Martin, & Lang, 1997), others have lamented the complexity of Java and other modern languages for introductory courses (Wirth, 2002). Despite evidence of dissatisfaction with Java as a teaching language (Biddle & Tempero, 1998; Hadjerrouit, 1998; Martin, 1998), Java appears more times in

paper titles accepted for the Special Interest Group in Computer Science Education (SIGSCE) annual symposium over the last eight years than all other languages combined (Roberts, 2003).

At the same time, declining interest and enrollment in high school computer science courses has become a trend over the last twenty years, with students taking more courses in business services, computer technology, graphics, computer applications, and drafting (Levesque & Hudson, 2003). In many schools, technology programs have merged with computer science, and a new focus on the *tool* aspect of technology has supplanted the once *academic* nature of the subject. This shift has occurred for three main reasons: (a) industrial arts departments are being replaced by technology departments, with industrial arts subjects being renamed as technology, (b) people with various experiences and backgrounds are brought in as teachers into technology departments, and (c) computer science has been pushed aside as the wider need for technology integration into the curriculum has increased (Deek & Kimmel, 1999). As a result, the definition of *computer science* has been broadened to include technology education subjects, producing confusion over its meaning.

It is clear to the present investigator that the true benefits of secondary computer science education have been lost among the frenzy for the latest waves of innovation and change, with a greater emphasis placed on language. What are the benefits? Research has unequivocally established that computer programming instruction improves problem solving skills (PSS) significantly enough to warrant that programming (in any language) should be included in the curriculum as an alternative for teaching problem solving in all subject areas (Casey, 1997). Such skills are vital for students to function in today's

complex society. Equally important are the cognitive skills that are gained and transferred through the application of algorithmic and logical thinking (Gesler & Kaplan, 1993; Greenburg, 1991; Jang, 1992; Martin & Hearne, 1990; McCoy, 1988; Shih & Alessi, 1994).

The time has come for educators to take responsibility for the definition and delivery of computer science education (Wirth, 2002). While efforts have already begun with standardization of a definition of computer science (Tucker, 2003) and a dream of a scaled-down version of Java (Roberts, 2003), the greater challenge lies in recasting the purpose of computer science education and making it motivational, understandable, and available to students of all ages and backgrounds. The challenge is taken by envisioning learning opportunities that redirect the emphasis away from language and complexity and balances the greater need for cognitive skills development and problem solving. Learning opportunities like these render language as an object of secondary importance (Milbrandt, 1995), and computer science education is transformed into an instructional paradigm where students acquire useful knowledge that transfers into other subject areas and real-world contexts (Wilkerson & Gijselaers, 1996). The alternative is the risk of continued decline in interest and enrollments, increased priority for technical education, and computer science becoming narrow and esoteric, reaching only the highest achievers. The present study seeks to address the above challenges by developing a Web-based learning system as a teaching and learning tool for introductory computer science education, and then performing preliminary validation of the system's efficacy as an instructional environment.

Setting

Computer science education in public and private schools in Southern Nevada has become fragmented, misunderstood, and for the most part, absorbed into newly defined technical education programs. While efforts have been made to provide training, teachers still feel isolated, unprepared to teach computer programming, and have experienced declining student interest in the subject. The sentiments shared by one teacher leader in a large high school in the Clark County School District reinforce the current situation:

> AP Computer Science has never been much of an option at my school. Unfortunately, the interest level has not been terribly high... The only problem could be selling an AP class to my principal if there are somewhere between 5-10 students... I don't think it would work very well combined with another class... I believe there is a seminar coming up regarding the AP requirements and Java curriculum on November 15... I hadn't planned on going. (B. Bogart, personal communication, October 22, 2003)

Similar conditions exist in other schools throughout the region, with some unable to offer courses or canceling programs because of difficulty in finding qualified teachers or experiencing declining student interest. The present study was established to develop and pilot the use of a hybrid instructional system to meet the challenges and renew interest in computer science as an academic discipline for students of all ages. Computer Science Online (CS Online) was conceptualized in the spring of 2002 and was subsequently designed and launched in the spring of 2003 as the development project for the pilot study. The term *hybrid* is used to describe a Web-based system that can be used for both online and classroom-based instruction. In addition, the system was concurrently used as

an instructional tool for the Methods of Teaching Computer Programming course at the University of Nevada, Las Vegas (UNLV), in which twelve graduate students participated as teaching assistants (TA) to evaluate and grade the work of high school students who participated in the study. CS Online is accessible at http://www.csonline.ccsd.net.

Theoretical Framework

The CS Online design and subsequent investigation was inspired by and built upon the Reading approach to teaching programming for high school students (Van Merrienboer & Krammer, 1987). Figure 1-1 depicts the theoretical framework for the study. The model consists of concentric shapes that circumscribe supporting bodies of research and build the foundation for effective computer-programming methodology for the purpose of the present study.

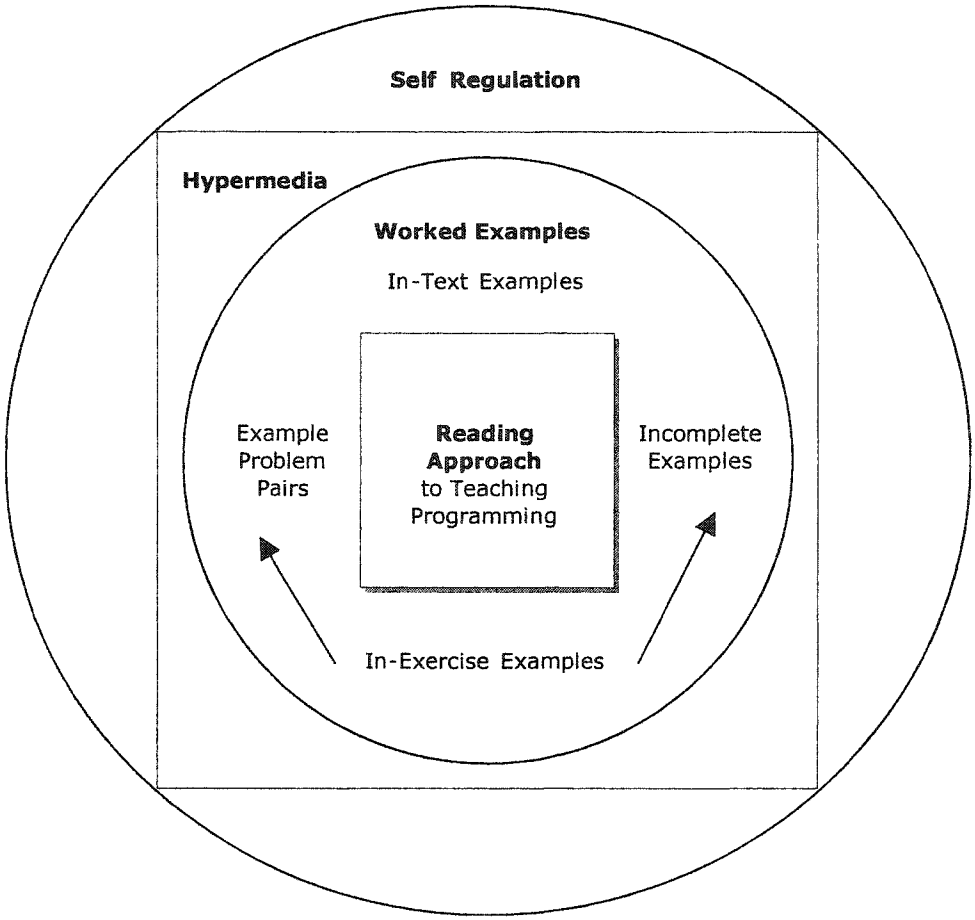


Figure 1-1. CS Online Theoretical Framework.

Beginning at the center, the Reading approach is presented as an effective method of teaching introductory programming. Because this approach is dependent on the application of worked examples, worked examples research, the next concentric shape moving outward, is then presented. Because of its relationship to Web-based learning, hypermedia research and its effect on learning is then presented. Finally, a review of self-regulation is presented as an individual learning characteristic that might be an important factor for students interested in learning programming online.

The Reading Approach

The center of the framework represents the Reading approach to teaching programming. Following a comprehensive review of teaching methodologies related to computer programming, Van Merrienboer & Krammer (1987) classified instructional methodologies into three categorical strategies: (a) the Expert Approach, (b) the Spiral approach, and (c) The Reading approach. The approaches differ in that the Expert approach presents motivational, complex problems requiring top-down solutions; the Spiral approach emphasizes acquisition of semantic and syntactical skills by mastering basic language constructs and then building; and the Reading approach recommends that students begin by understanding relatively complex solved problems and then modifying and amplifying the solutions. Following a comparison of six instructional tactics spanning each strategy, the researchers concluded that the Reading approach is superior to the other two in five out of six instructional tactics. The tactics include computer modeling, programming plans, design diagrams, worked-out examples, basic skills, and task variation.

At the heart of the Reading approach is the use of worked-out examples, referred to hereafter as worked examples. Step-1 of the Reading approach involves running working programs, observing their behavior, then evaluating their strengths and weaknesses. In Step-2, students read, run, and trace well-structured programs. In Step-3, students modify and amplify existing programs; and in Step-4, students generate completely new programs on their own.

Worked Examples

While the Reading approach was shown to be a most effective instructional method, at the core of the approach is the application of solved programming examples that students can run, modify, and amplify in support of underlying concepts. Worked examples research has provided evidence that various types of intra- and inter-example design features and the individual characteristic of self-explanation can lead to more effective learning (Atkinson, Derry, Renkl & Wortham, 2000). Self-explanation is what occurs when a student attempts to *fill in the gaps* of poorly elaborated or intentionally omitted content in worked example design (Chi, Bassok, Lewis, Reimann, & Glaser, 1989). In recent years, worked examples research has gained considerable attention and has made contributions to improved instructional design. Considering the Reading approach's dependence on worked examples, something should be known about the effectiveness of their design and delivery in various learning environments. It has been shown that worked examples are most effective when used in instructional settings that promote skills acquisition – like computer programming (Anderson, Fincham, & Douglass, 1997).

The design or structure of worked examples plays a critical role in their effectiveness in learning (Mwangi & Sweller, 1998). Design that ignores intra-example features can lead to the split attention effect, which can then degrade learning (Taramizi & Sweller, 1988). Intra-example features include integrating text and diagrams, integrating aural and visual information, and integrating steps and sub-goals (Atkinson et al., 2000). On the contrary, carefully designed worked examples can reduce or eliminate the split attention effect and result in cognitive load reduction (Cooper, 1998; Paas, 1992). Equally

important to the structure of the material used in lessons is the sequence in which that material is presented (Bruner, 1966; Glaser, 1976). Inter-example feature design focuses on several factors including the number of examples to use, how and whether examples should be varied within a lesson, how themes might be varied, and how practice and examples should be integrated. To this point, the cognitive consequences of presentation format have been more emphasized as a research discipline than how the worked examples are applied and used (Ward & Sweller, 1990).

In the present study, students were engaged with two types of worked examples: *in-text* and *in-exercise*. In-text examples were embedded in chapter sections, and in-exercise examples were coupled with section-end exercises. There were at least two in-text examples per section that the students could view, run, or modify and run at any time. Multiple examples in section content have been shown to facilitate improved learning over the use of a single example (Reed & Bolstad, 1991). In-exercise examples were available to assist with problem solutions and were presented in two general forms: (a) example-problem pairs, where the problem was associated with the in-text worked example that was most closely resembled the exercise, and (b) incomplete examples, also referred to as hints or partial solutions, which were available at three levels. Example-problem pairs have been shown to enhance skill acquisition in a most effective manner (Trafton & Reiser, 1993). Stark (1999) found that, compared to studying complete examples, incomplete examples are beneficial to produce higher levels of effective self-explanation.

To extend the current research focus on presentation format, the present study looked deeply into relationships between individual learner characteristics and dependence on

worked example use. More specifically, the study explored how students with varying levels of self-regulation, prior math experience, and prior computer experience depended on worked examples. Knowing that some students would rely heavily on examples during problem solving (Chi et al., 1989), the study sought to determine if that reliance and subsequent success (or failure) could be attributed to individual characteristics.

Cognition and Hypermedia Environments

CS Online was designed to be a hybrid system, a tool that could be used in the classroom and online. The hybrid approach is important since virtual learning has been established as the next wave in technology-based K-12 learning (WestEd, 2001). While this type of learning has gained widespread support from state and local policymakers, education researchers, and business leaders (Education Week, 2002), others are skeptical about the promises held by this approach to learning. Considering the Web as a form of hypermedia, research has yet to reveal gains achieved through the use of media or interface design (Dillon & Gabbard, 1998). Because of a lack of evidence in support of its benefit, the focus of research has shifted from the effects of media toward individual learning characteristics and learning in new technology environments. Individual characteristics like prior knowledge (Shin, Schallert, & Savenye, 1994), past experience (Lanza & Roselli, 1991), ability (Ormrod, 1999), learning style (Ormrod, 1999), and self-regulation (Zimmerman, 2000) have been shown to be important learner variables. The present study sought to determine, through measures of self-regulation, if students could successfully solve problems in the context of a hybrid environment where cognitive skills might be affected.

Self-Regulation

While it is clear that appropriate application of worked examples can improve learning in a traditional setting, it is not yet known how they might affect learning in a Web-based environment. Granted that media's effect on learning might be inconsequential, student dependence on worked examples in a Web-based environment might then be attributed more to individual learning characteristics, such as math experience, computer experience, or self-regulation. A clearer understanding of these dependencies can lead to better understanding of the constructs needed for good, scientifically-based, instructional design. It is anticipated that self-regulation will be an important factor in student learning in Web-based environments (Hartley & Bendixen, 2000; Foreman, 1990).

Self-regulation refers to self-generated thoughts, feelings, and actions that are planned and cyclically adapted to the attainment of personal goals. It entails not only behavioral skill in managing one's environment, but also the knowledge and sense to enact this skill in relevant contexts (Zimmerman, 2000). More specifically, self-regulated learning is comprised of three dimensions: meta-cognition, goal setting and monitoring one's actions (Ridley, Schutz, Glanz, & Weinstein, 1992). These dimensions can be sub-divided into: (a) self-motivation, (b) goal setting, (c) planning, (d) attention control, (e) application of learning strategies, (f) self-monitoring, and (g) self-evaluation (Ormrod, 1999).

Hong (1998) distinguishes two different classes of personality or psychological attributes that can be applied to self-regulation – trait and state constructs. State self-regulation is conceptualized as a transitory state that varies depending on situational

cognitive demands. Trait self-regulation is a performance attribute that remains relatively stable across varying cognitive demands (Hong, 2001a). While the study of both attributes is important for determining individual differences in learning and performance, the present study focused on measures of state self-regulation and student dependence on various types of worked examples.

Significance

The future of computer science education will depend on many factors, the most important of which might be a revitalization of interest in the subject (Stephenson, 1997). The design and delivery of CS Online brings a research-based learning opportunity to Southern Nevada for the purpose of equipping teachers with self-paced professional development, management tools, and rekindling interest in the subject by providing rich, motivational PBL-based learning content to students in classrooms and online. In addition, through preliminary validation of its efficacy as an instructional environment, results of the pilot study can inform the educational community of an applicable solution model in response to the issues. Furthermore, while critics and supporters of virtual learning agree that insufficient research has been conducted to determine the effectiveness of Web-based learning (Paloff & Pratt, 2001), findings from the study can inform the research community that students of varying abilities can successfully learn programming in such an environment. Finally, research involving individual learning characteristics might describe the competencies students will need to succeed. More specifically, insight might be gained into how measures of individual characteristics might describe problem solving online; including the types of learners that are likely to

succeed, the types of problem-solving strategies that are used, how much effort students are willing to expend, and how learner characteristics are related to the use of hints.

Research Questions

The present study resulted in the development of CS Online and subsequent preliminary validation of the system's efficacy as a learning environment. In particular, the study sought to answer the following five questions:

1. How do students with low versus high self-regulatory skills perform in the use of in-text worked examples, in-exercise worked examples, hints, optional exercises, and problem-solving scores?
2. How do students with low versus high math experience perform in the use of in-text worked examples, in-exercise worked examples, hints, optional exercises, and average number of attempts to solve problems, and problem-solving scores?
3. How do students with low versus high programming experience perform in the use of in-text worked examples, in-exercise worked examples, hints, optional exercises, and average number of attempts to solve problems, and problem-solving scores?
4. How do students with low versus high self-regulatory skills perform in the use of in-text worked examples, in-exercise worked examples, hints, optional exercises, average number of attempts to solve problems, and problem-solving scores as the task difficulty increases?
5. What common mistakes do students make in solving programming problems?

CHAPTER 2

REVIEW OF THE LITERATURE

The review of the literature is divided into three major parts to support the development and preliminary validation aspects of the current study. Part 1 provides the rationale for the development of the CS Online learning environment. Part 2 provides fundamental design strategies for the development of CS Online. Part 3 reviews literature related to the research questions including cognitive load reduction, worked examples, and self-regulation.

Part 1: Rationale for the Development of CS Online

Various factors have contributed to the present condition of computer science education, many of which were motivating factors for the concept and subsequent development of CS Online. These include a lack of standards for a high school curriculum and teacher certification, professional development issues, increasing growth of technical education and subsequent declining enrollments in computer science courses, increasing complexity of programming languages, accessibility to resources, and cost factors. In this section, a review of the literature related to these factors is presented in the order listed above.

Lack of Standards for a High School Curriculum and Teacher Certification

Computer science is widely accepted as an academic discipline in higher education and as a profession, but its status in secondary education is perceived quite differently. Although efforts were made in the mid 1980's by the Association for Computing Machinery (ACM) to standardize the curriculum (ACM, 1985a) and to define standards for teacher certification (ACM, 1985b), these efforts have been slow and non-systemic (Deek & Kimmel, 1999). This is evidenced by reports of the absence of a standardized or widely implemented high-school curriculum, and lack of states' adoption of teacher certification standards ten years later (Stephenson, 1997; Tucker, 1996). National and state standards have also ignored computer science as an academic discipline (CDE, 1996; NDE, 2003; NCGE, 1994; NCTM, 1989; NJDE, 1996; NRC, 1996; NYDE, 1994). As a result, computer science remains a highly fragmented and misunderstood subject (Deek & Kimmel, 1998; Tucker, 1996).

In an effort to raise awareness and focus national attention to these issues, the International Society for Technology in Education (ISTE) and the ACM have made efforts to: (a) define curriculum frameworks content standards for computer science education, (b) define teacher certification standards, (c) elevate computer science as an academic discipline in departments of education and other appropriate agencies, (d) prescribe teacher preparation programs that equip teachers with the content skills and knowledge they need for effective learning in the classroom, and (e) define provisions for re-training teachers currently in the field (ACM 1985a, 1985b, 1993; ISTE, 1992).

Professional Development Issues

Dramatic changes in technology have made it difficult for computer science teachers to receive the training they need. In fact, "no mechanisms exist to train teachers... or keep them up to date with the field." (Tucker, 1996). Evidence has suggested that while coursework still exists in some teacher preparation programs, the emphasis has shifted away from programming toward hypermedia and authoring tools, with programming only being offered amid controversy and debate (Kelley, 1994). If this trend continues, teachers will be left behind, unequipped to face the challenges of the computer science classroom. A recent case study on the state of computer science education in New Jersey found that teachers were not receiving the foundational coursework necessary for a meaningful and adequate professional development program – 57% of the surveyed teachers had not received any kind of training within five years of the survey (Deek & Kimmel, 1999). The ACM is presently conducting a nationwide survey to assess the state of computer science education and professional development needs (ACM, 2003).

Increasing Growth of Technical Education

Technology education has rapidly become a priority as new technologies have emerged, generating new interest and demand. Course offerings in subjects including productivity applications, computer technology, graphics, computer applications, drafting, multimedia, authoring, web page design networks, and distance learning have attracted students away from taking computer science courses (Deek & Kimmell, 1999; Levesque & Hudson, 2003). In many schools, technology programs have merged with computer science, and a new focus on the *tool* aspect of technology has supplanted the once *academic* nature of the subject. The shift in interest is evidenced by a declining

average number of computer programming credits earned per student. In 1998, 0.04 credits were earned per student compared to 0.13 in 1990, when computer science education reached its peak. Students in 1998 earned 0.50 credits on the average in business and computer applications compared to 0.33 in 1990 (Levesque & Hudson, 2003).

In addition to dramatic changes in technology, changes in technology education have also occurred for three main reasons: (a) industrial arts departments are being replaced by technology departments, with industrial arts subjects being renamed as technology, (b) people with various experiences and backgrounds are brought in as teachers into technology departments, and (c) computer science has been pushed aside as the wider need for technology integration into the curriculum has increased (Deek & Kimmel, 1999). Despite efforts to promote computer science education and advocate teacher training programs, many schools continue to offer computer science but few students choose to take it; computer science remains an elective subject; and most computer science programs reside in math, science, technology, or business departments with teachers certified in various areas (Deek & Kimmel, 1998; Kushan, 1994).

Increasing Complexity of Languages

Long before the College Board moved the AP computer science program to Java, the language had already generated interest in the professional and computer science education communities. Java has appeared more times in professional journal article titles and papers accepted for the SIGSCE annual symposium over the last eight years than all other languages combined (Roberts, 2003). The use of Java for introductory computer science courses was evident as far back as 1998 (Stevenson & West, 1998) and

continues to gain momentum now that it has entered the AP world. Unfortunately, because of its *complexity* and *instability*, some view the language as a critical problem for introductory courses (Wirth, 2002). Roberts (2003) defines *complexity* as "the number of programming details that students must master has grown much faster than the corresponding number of high-level concepts" (p. 1). He further defines *instability* as " the languages, libraries, and tools on which introductory computer science education depends are changing more rapidly than they have in the past" (p. 1). Because of these two important factors, Java has evolved to become a complex language. With close to 50,000 public methods in the Java 2 API hierarchy (Sun Microsystems, 2003), teachers have been required to teach the language with little to no training and support in the schools (Stephenson, 2002). To remedy this problem, the present goal of the ACM Education Board is to review the Java language, APIs, and tools from the perspective of introductory computing education and to develop a smaller, more usable subset of the language for introductory computer science. JavaScript was chosen as the language for CS Online because of its resemblance to Java and ease of use.

Part 2: Basic Design Strategies for the Development of CS Online

A clear understanding of the reasons for offering computer science in high schools combined with effective methods for teaching programming can inform various instructional design strategies for delivering computer science education. This section reviews literature on the reasons for teaching computer programming followed by advantages of teaching and learning computer programming, methods of teaching programming, choice of language, and course management. These areas inform

fundamental CS Online design strategies as they apply toward increasing interest in computer science, reducing complexity in managing courses, and providing effective learning opportunities for students and teachers.

Reasons for Computer Programming Education

Radical change and apparent declining interest in computer science education beg the question of why computer science courses should continue to be offered in secondary schools. The three major reasons for offering computer programming in secondary schools reveal a wide range of influences that have shaped computer science instruction as we know it today (Goldenson, 1996). The Imperative for Educational Reform in 1983 and the National Commission on Excellence in Education report challenged Americans to embrace change through the use of technology. This report was a catalyst leading to the vocational education movement, the first reason in an effort to increase the population's technical skill level for greater opportunity in the professional work place (Campbell, 1984). Computer programming instruction was an offshoot of this proposal as part of the technology preparation agenda. The second reason is preparation for college. Many high schools provide AP computer programming courses in preparation for advanced study in college (Connolly, 1996). The third reason is an attempt to increase academic achievement in other subject areas through the promotion of analytical and creative thinking skills. It is viewed that generalization and transfer of cognitive skill growth through study of computer programming can have a dramatic impact on how students perform in other subject areas like math, science, and expository writing (Goldenson, 1996).

A fourth reason might be the increasing number of programmable applications and systems that are more accessible to users today. Examples include programming dynamic Web pages using Netscape JavaScript 1.5 or server controlled scripts like Macromedia Cold Fusion MX 6.1, Sun Java Server Pages 2.0 (JSP), Microsoft Active Server Pages 3.0 (ASP), or Hypertext Pre-Processor 4.3 (PHP); spreadsheet programming using embedded functions; and customized database application design using Microsoft Visual Basic for Applications 6.0 (VBA). Underlying multimedia-authoring tools like Macromedia Flash MX 2004 and Director MX are programming languages that, if mastered, equip the designer with an extremely high level of flexibility and functionality. The need to know programming appears to be greater now than ever.

Advantages to Teaching and Learning Programming

Much research on the benefits of programming and learning was conducted in the mid to late 1980's and early into the 1990's. The majority of research related to this topic ended in the early 1990's, closely following the peak of interest in computer science education (Levesque & Hudson, 2003). The introduction of LOGO by Papert in the late 1970s (the original version was working in 1967) launched a wave of interest in trying to find out how programming might affect the cognitive processes of students of all ages (LOGO Foundation, 2002). The bulk of the findings relates most directly to the elementary grades with some valuable information available for secondary instruction. Because of the magnitude of information available during the 1980's and the archaic nature of languages studied, this research review spans a period of 15 years – from 1987 to the present. The languages dominating this body of literature include LOGO, BASIC,

and Pascal, in descending order of importance. Surprisingly, no other languages are referenced.

The synthesis of findings from this review can be classified into four general categories and are presented in subsequent sections in the following order: (a) cognitive skills affecting programming concept acquisition, (b) programming and it's affect on cognitive skills, (c) transfer of programming skills into other academic areas, and (d) general topics of interest.

Cognitive skills affecting Programming Concept Acquisition

Relationship between cognitive science and instructional design. Human cognitive skill development has been shown to affect a student's ability to learn programming concepts. In fact, while the fields of cognitive science and instructional design have their own objects of study, they share a common interest in cognition and performance as part of instructional systems. From a case study based on experience in teaching introductory computer programming, Van Merrienboer (1990b) concluded that both sciences may reciprocally influence one another. These findings suggest that the sciences must work together to reach their common goals.

Teaching Methodology. Teaching methodology in the context of computer programming instruction can affect student cognitive development. A study comparing reflective and inquiry-based teaching practices for 2nd through 5th grade students revealed that students experiencing the reflective context developed beliefs about Logo programming practices that were tightly coupled with their performance (Lehrer & Jeong, 1999). Teaching with analogies and elaboration and placement of those analogies was

demonstrated to significantly improve concept recall for students learning a programming language (Lai & Repman, 1996).

Pre-cursors to Successful Programming. Studies have shown that programming course primers can increase learners' cognitive ability and improved performance in programming courses. Allan and Kolesar (1996) suggest a preparatory Computer Science-0 (CS 0) course to countervail conceptual weaknesses observed in novice programmers. Preparatory courses demonstrating cognitive improvement involve students with: 1) experiencing good user interfaces before being asked to design one, 2) playing with data types and round-off errors using spreadsheets, 3) understanding how an application program saves time and effort resources, and 4) developing good problem solving techniques vital to good programming practice. Miller (1988) demonstrated that pre-programming instruction involving teacher-designed graphical Logo programs and multimedia techniques, combined with modern technology, resulted in higher order of logical thinking skills such as critical thinking, problem-solving strategies, evaluation and analysis, and creativity. Baylor and Kozbe (1998) suggest the use of a Personal Intelligent Mentor (PIM) as an aid for students to develop logical and critical thinking abilities essential for problem solving in preparation for learning computer programming. The PIM they researched is a software tool that facilitates metacognitive development in the domain of solving logic word puzzles.

Specific Cognitive Skills. Many components of human cognition have been demonstrated to be required for students to perform better while learning computer programming. A study examining the relationship between field-independence, spatial visualization, logical reasoning, and direction following and initial acquisition of

programming competence suggests that individual differences be considered in all programming instruction regardless of language used and student age (Foreman, 1990). Worked examples as a cognitive load reduction effect are recommended based on findings from a study on automation and schema acquisition in learning beginning computer programming (Van Merrienboer & Paas, 1990). Automation and schema acquisition are generally considered important processes in learning cognitive skills.

Intrinsic Characteristics. Intrinsic characteristics can also play a vital role in a student's ability to learn programming skills. A study conducted by Johnson and Johnson (1992) revealed that programming competence increased as stress, neuroticism, creativity, and age increased. The study also showed that females demonstrate better computing competencies than males.

Programming and Its Affect on Cognitive Skills

Cognitive Skills Development. Problem solving skills (PSS) seems to be the predominant cognitive skill most directly impacted by learning programming. In this discussion, it will be assumed that choice of programming language is independent of cognitive skill attainment since underlying constructs like *if-then-else* do not change among languages (Sebesta, 1996). Following a summary of PSS affects, other cognitive skills, meta-cognitive skills, and potential for confusion will be addressed. The majority of research in cognitive development and programming targets students in the elementary grades, most likely because of Logo's appeal to elementary teachers and younger children. Unless noted otherwise, studies are assumed to target this age range.

Problem Solving Skills (PSS). The most researched cognitive skill affected by computer programming instruction is that of PSS. Some important outgrowths of this

research include recommendations to include computer programming (any language) in the curriculum as an alternative for teaching problem solving (Casey, 1997), include Logo to teach problem solving strategies (Swan & Black, 1993), involve the use of Lego and Logo to teach problem solving skills (Palumbo & Palumbo, 1993), and employ programming instruction to foster self-regulation, motivation, and discovery (Casey, 1997). Programming instruction and its affect on problem solving transfers into all academic areas of study. Through project-based learning, CS Online was developed to emphasize problem-solving skills through problems set in real world contexts.

Other Cognitive Skills. Various studies were conducted in the 1990's to demonstrate the effects of programming on various other cognitive skills. Logical thinking and sophisticated mathematical relationships (more of a transfer issue but included here because of logic) have been shown to be better understood by unsophisticated college students if they have some level of programming experience (Wieschenberg, 1999). Wieschenberg asserts that Math and computer programming are very similar – they both involve logical steps, which eventually result in a desired solution. In addition to logical reasoning, inductive and deductive reasoning (Kynigos, 1993), social problem solving and motivation (Suomala, 1996), cooperation (Lai, 1993), attitude toward learning (Dalton & Goodrum, 1991), conditional reasoning (Seidman, 1990), and spatial relation comprehension (Miller, 1988) were shown to improve with programming instruction.

Logo was the language used in each of the aforementioned studies with the exception of Wieschenberg's study. Object-oriented programming, a form of hypermedia authoring, has been demonstrated to effect creative thinking (Liu, 1998). This type of programming was found to "promote creative thinking in a variety of areas including the process of

sensing problems or gaps in information, forming ideas or hypotheses, testing and modifying these hypotheses, and communicating the results." (Liu, 1998, p. 27). In study to test for critical thinking skill development, three groups of high school students were tested using the Watson-Glaser Critical Thinking Appraisal: participants in a first-year BASIC class, participants in a first-year Pascal class, and above-average students in other classes who had no experience in programming. Students enrolled in both programming classes scored significantly higher than their non-programming counterparts (Jones, 1988). In summary, a meta-analysis of 65 studies on programming affect on student cognitive skills shows that students having computer programming experiences scored an average of 16 percentile points higher on various cognitive ability tests than did students who did not (Liao, 1990).

Transfer to other Subject Areas. Mathematics appears to be the subject area most impacted by cognitive skills transfer due to programming (McCoy, 1988; McCoy & Dodl, 1989; Oprea, 1988). In studies designed to measure cognitive skills transfer, it was found that groups having programming instruction scored significantly higher than the control groups in mathematical thinking skills, generalization, and understanding of variables. One study compared four groups of high school students enrolled in calculus, with two of the groups concurrently enrolled in Pascal programming. Students enrolled in Pascal programming out-performed their counterparts in their math achievement tests (Jang, 1992). In some cases, variables studied included gender, ability, socioeconomic status, prior math experience, and access to a home computer (McCoy & Dodl, 1989). Other affected subject areas include geography (Gesler & Kaplan, 1993), creative arts

(Greenburg, 1991), social studies (Martin & Hearne, 1990), and science (Shih & Alessi, 1994).

The Methods of Teaching Programming

Booth (1990) highlights three popular perspectives on teaching computer programming including computer-oriented, product-oriented, and project-based learning (PBL). PBL can also be referred to as *problem-based learning*. In the *computer-oriented* approach, programming is perceived of as an activity that focuses on the computer. Activities might involve writing programs that simulate aspects of the operating system (OS) or hardware components like a binary adder. The *product-oriented* approach tends to be more constructivist and focuses on the end goal of developing software products such as RPGs, games in general, or utility programs. The *PBL approach* treats the programming language as a matter of secondary importance, with emphasis placed on the problem to be solved and the logical steps required for its solution (Milbrandt, 1995).

Brusilovsky (1994) identifies three other general approaches to teaching programming include the incremental, mini-language, and sub-language methods. The *incremental* approach treats the language as a sequence of subsets. Each subset introduces new constructs while retaining all the constructs of the previous subsets. Each subset is also precisely defined as a complete sub-unit that can be learned or implemented without subsequent subsets. The *mini-language* approach is intended to design a small and simple language to support introductory concepts of learning programming. The development of the mini-language approach was seriously influenced by turtle graphics of Logo (Papert, 1980). The *sub-language* approach is to design a special starting subset of the full language containing several easily adaptable operations. As students master

concepts in the starting subset, additional concepts are added to build upon their knowledge base – similar to outwardly growing concentric circles.

Various techniques have been introduced to enhance students' acquisition of programming concepts. Bayman and Mayer (1988) suggest using syntactical conceptual models using the language's inherent syntax structures. Their research demonstrates that students trained in the use of language semantics and syntax, develop fewer misconceptions and perform better on problem solving. Hancock (1988) suggests two ideas that have proven valuable in teaching introductory programming. The *mental model* encourages pure conceptualization and schema development and direct translation into programming code. The *programming plan* encourages planning and documenting the program before writing any code. McCoy (1990) identifies five critical phases essential to successful computer programming: (a) general strategy, (b) planning, (c) logical thinking, (d) variables, and (e) debugging. General strategy places emphasis on high-level procedures and constructs needed to solving the problem. Planning involves sequence and hierarchy of those constructs. Logical thinking involves the writing of code to realize the solution. Variables cover the data structures used to process information and numerical calculations. Debugging is the process of getting the program to work. McCoy (1990) recommends these same strategies be used in solving complex mathematical problems. Other research suggests five common structured programming techniques applied in practitioner computer science: (a) problem definition, (b) algorithm design, (c) code writing, (d) debugging, and (e) documentation (Dalbey & Linn, 1985; Goktepe, 1985; Kurland, 1984).

Various other methods have been shown to improve students' ability to conceptualize and master difficult concepts related to computer programming. Through the use of concrete representations, collaboration in a structured laboratory environment, focused completion-type exercises, and elaboration, students are better able to comprehend and apply the concept of parameter passing (Madison, 1995). Mediational instructional strategies have been shown to foster better learning in a Logo environment (Delcros & Burns, 1993). Strategies that impact various cognitive styles have been suggested with a preferential model having greater effect on learners than a compensatory model (Van Merrienboer, 1990a). Based on ACT (Adaptive Control of Thought) theory and relevant research, Van Merrienboer and Krammer (1987) identify tactics to design programming courses based on the differences between declarative and procedural instructional approaches. Some of these tactics include the expert, spiral, and reading approaches. The Expert approach requires a top-down concept and implementation strategy involving algorithm and program design. It was the least effective of these three approaches. In the Spiral approach students were simultaneously presented with syntactic and semantic knowledge in small incremental steps. As the students mastered basic skills, program requirements progressed from simple to more complex, with design skills not required until late in the course. The most effective approach was the Reading approach. This four-step program permitted students to: (a) run pre-written programs, observe those programs' behaviors, then evaluate strengths and weaknesses, (b) hand-trace programs and predict output, (c) modify and amplify existing programs, and (d) generate their own programs. An unlimited array of creative and motivational ideas can be applied to teach individual constructs like *if* statements or iteration (Prichard, 1993; Tu & Falgout, 1995).

Examples of these might include the iterative process of randomly generating license plate or social security numbers, or artificially identifying a poker hand using nested *if* statements. The Reading Approach is the fundamental method CS Online employs for teaching computer programming.

One relatively new approach to teaching programming was introduced by computer science students at the University of Joensuu, Finland. The CANDLE model was designed to support a student locally, in her Authentic learning NeeDs, in a Light way, and through Electronic tools. What's unique about this method is that programming instruction was designed by college students to teach high school students through the Internet (Haataja, Suhonen, Sutinen, &Torvinen, 2001). This PBL approach requires students to assess the support they need to solve authentic learning problems (electronic candles). BlueJ, a visual teaching environment and language, helps students in the Candle program to understand object-oriented concepts such as objects and classes, message passing, method invocation, and parameter passing (Kolling, 2000). The Jeliot I environment allows students to write Java code in a Web text field then view the program animated after submitting. Both tools utilize a highly visual approach to teaching programming (Haajanen, Pesonius, Sutinen, Tarhio, Terasvirta, & Vannien, 1997). While many universities offer programming courses to their students through the Internet, this approach uniquely bridges the gap between secondary and higher education programming skills.

Choice of Language

The choice of language can be a difficult decision because of it's direct impact on computer programming instruction. In 1996, 442 higher education institutions reported

using one of 23 different languages for introductory computer science courses, with Pascal leading the way with 35.5% of the responses (Connolly, 1996). Secondary education has traditionally exercised similar freedom in the choice of language, despite pressure exerted by higher education on the kind of computer science instruction that should be taking place in schools (Becker & Graham, 2000). It was not too long ago that BASIC and Logo were the de facto standards for teaching programming in K-12 aged students. In support of advantages gained through new programming paradigms, Reed and Liu (1992) demonstrated that BASIC produced sub-standard attitudes and learning effects in comparison to emerging, object-oriented languages at that time like HyperTalk and C++.

Trends in Advanced Placement (AP) test design provide insight into language choice in the high schools. The first AP Computer Science Exam was offered in 1984 using Pascal. The AP Computer Science A course was implemented in September 1991 and used C++. The exam was changed from Pascal to C++ in May 1999 and was changed again in 2003 from C++ to Java in time for the May 2004 exams (College Board, 2003a). As can be seen, language standards quickly shifted from Pascal to C++ and then Java, all within the course of about ten years. Furthermore, the AP Computer Science Development Committee made a formal request in October of 2000 to the College Board to recast the AP Computer Science curriculum. The revision would include object orientation beginning with the 2003-2004 academic year (College Board, 2003b). The request was approved in November of 2000 (College Board, 2003a).

Booth (1990) discusses the impact of conceptions of programming languages on language selection and teaching methodology. The *code* perspective frames the language

as a set of instructions, commands, symbols, and constructs. This perspective leads to a more formal approach to teaching and learning. The *utility* perspective views the language as enabling the programmer to achieve certain effects. Choice of language in this case depends on achieving specific outcomes such as developing a role playing game (RPG) or multimedia effect. In this case, Visual Basic might be used to create an RPG whereas Lingo might be used produce the multimedia effect. The *communication* perspective views the language as a means of communication between the programmer and the computer. The high level language (HLL) is seen as inferior to the more perfect machine level language, which is more closely tied to the computer's hardware. The *expression* perspective views the language as a means of expressing a problem solution in such a way that the computer can have an effect. In the present study, JavaScript was chosen as the programming language because of its ease of use, relationship to web pages, resemblance to Java, and potential appeal to a wide range of users.

Management Strategies

The management of computer science instruction can be a cumbersome process that mainly involves the distribution of worked examples and evaluation of student work. Since student work is normally stored on disks, workstations, or servers; teachers are required either to work with students individually while programs are demonstrated, collect disk-based or printed hard copies of source code, or view and run programs from their own workstations. If software development tools are accessible only from school, assessment is further limited to during the school day.

There are more difficult issues to manage besides the classroom, however, and that's namely *what* to teach. Confusion over what computer science is has made it difficult for

educators to determine what the subject should encompass. Efforts have been made to develop a standard definition:

> *Computer science (CS)* is the study of computers and computational processes (known as "algorithms"), including their principals, their hardware and software designs, their applications, and their impact on society. An *algorithm* is a precise description of a solution to a computational problem. *Programming* is used to implement algorithms (Tucker, 2003, p. 2).

While definitions and standards for computer science and a curriculum are necessary, a recent survey revealed that suggested models proposed by the ACM have not received widespread recognition or implementation in the United States. Only 12 of the 70 respondents indicated they have a state-mandated computer science curriculum at the high school level, and 27 out of 70 replied that no certification is required by their states (Tucker, 2003). These difficulties translate into widespread differences among states and school districts in how course content is defined and delivered in the classroom. Although model K-12 curricula continue to be developed by the ACM, nothing has been adopted or recognized as a standard up to this point.

Part 3: Literature Related to the Research Questions

CS Online was developed fundamentally around the Reading method of teaching programming. Inherent in the Reading Approach is the use of worked examples, which serve as a cognitive load reduction technique (Paas, 1992). Since CS Online was developed as a hybrid system, the opportunity for students to learn introductory computer programming online is now available, and self-regulation might play an important role

for online learners. This section reviews literature related to elements of the research questions beginning with cognitive load reduction and followed by worked examples research and self-regulation.

Cognitive Load Reduction

There is well-established research that supports the idea that the quality of instructional design can be raised if consideration is given to the role and limitations of working memory. The corpus of this research falls into the field of cognitive load theory (Sweller, 1994). Working memory in human cognition is typically equated with consciousness, and all other cognitive functioning is hidden from view until brought into working memory (Sweller, Van Merrienboer, & Paas, 1998). Cognitive load refers to "the total amount of mental activity imposed on working memory at any instance in time. The major factor that contributes to cognitive load is the number of elements that need to be attended to" (Cooper, 1998, p. 11). Since working memory is capable of holding only seven information elements at a time (Miller, 1956), instructional design must consider efficient ways by which learners can process and store facts, large and complex interactions, and procedures. For example, the success of chess masters compared to week-end hobbyists can be attributed mainly to their long-term memory of thousands of board configurations – familiarity that came purely through experience playing the game (Simon & Gilmartin, 1973). Interestingly, the same masters were no better than any other player at reproducing random configurations with which they were not familiar. When translating this notion to the field of instructional design, instruction must facilitate domain specific knowledge acquisition, not general reasoning strategies that cannot possibly be supported by human cognitive architecture (Sweller et al., 1998).

This can be accomplished by constructing ways to organize and store information into long-term memory and reduce the load placed on working memory. "It can be argued that these two functions should constitute the primary role of education and training systems" (Sweller et al., 1998, p. 256). According to schema theory, information elements are categorized and stored into long-term memory in the manner in which they will be used (Chi, Glaser, & Rees, 1982). Long-term memory can be defined as the part of the memory system that retains information for a relatively long period of time (Ormrod, 1999). A schema, while treated as a single element in working memory, has no limits on its information capacity. Schema can also be retrieved and processed automatically – a process whereby working memory is completely bypassed. In fact, all information can be processed either consciously or automatically (Schneider & Shiffrin, 1977). Automatic processing occurs with minimal conscious effort only after extensive practice. It follows that instructional designs should consider schema automation to build task consistency from problem to problem (Van Merrienboer, 1997; Van Merrienboer, Jelsma, & Paas, 1992).

Various empirically demonstrated instructional procedures can be applied to reduce cognitive load and benefit learning when used properly. Considering the already suggested minimal effect of media on learning, the same techniques should be applicable to virtual learning environments with similar expectations of success:

1. *Goal free effect.* A problem solving strategy that employs the goal free effect induces a forward working solution path which imposes very low levels of cognitive load and facilitates learning (Ayers, 1993; Owen & Sweller, 1985).

2. *Worked example and problem completion effect.* Involves reconsidering the nature and purpose of worked examples, especially where the problem space is large. Worked examples are paired with similar un-worked or partially worked problems (Paas, 1992), giving learners the opportunity to focus specifically on one solution method at a time.
3. *Split attention effect.* This effect occurs when a learner is required to attend to independent pictorial and textual information to understand a concept. The effect is reduced or eliminated when both elements are integrated into a single source of information (Chandler & Sweller, 1991; Sweller, Chandler, Tierner, & Cooper, 1990). Additional sources of split attention include multiple sources of text (Chandler & Sweller, 1992), mixing activities such a hard copy user's guide and software tutorial (Chandler & Sweller, 1996), and attending to multiple sources of information or activities as in performing a textual or graphical search, or even pull-down menus referenced in a user's guide (Cooper, 1998).
4. *Redundancy effect.* If one source of information (pictorial or textual) is sufficient to cover a concept, then additional information (integrated or not) should be completely removed (Chandler & Sweller, 1991).
5. *Modality effect.* There is evidence supporting the idea that working memory can be expanded through sensory modalities. Mixed-mode instructional formatting presents information in ways that maximize this effect, as in pictorial information with text presented auditorially (Mousavi, Low, & Sweller, 1995).
6. *Variability effect.* Although not listed by Cooper, Sweller et al. (1998) identifies this sixth effect that, through variability of practice, encourages learners to develop

schemas that help increase the probability that they will identify similar features and distinguish between relevant and irrelevant ones.

Worked Examples

A review of cognitive load reduction research clarifies the importance of worked examples. Their importance in the current study requires further understanding of research surrounding their use, including inter- and intra- example design instruction, individual differences in example processing through self-explanations, and the impact of situational factors on worked example comprehension (Atkinson, Derry, Renkl, & Wortham, 2000). There is little doubt that worked examples are most effective when used in instructional settings that promote skills acquisition – including computer programming (Anderson, Fincham, & Douglass, 1997). Considering worked example design to be an important aspect of cognitive load reduction, a clearer understanding of worked example research will be beneficial for the design of instructional systems that are dependent on the technique – including the delivery vehicle for the present study.

The design or structure of worked examples plays a critical role in their effectiveness in learning (Mwangi & Sweller, 1998). The worked example and problem completion cognitive load reduction technique is additionally supported by a study where LISP programming students were exposed to six example-practice problem pairs, where each example was immediately followed by a similar, but not identical practice problem. A second group of students were presented all six examples immediately followed by all six practice problems. The researchers observed that, as predicted, those students who were exposed to example-problem pairs took less time and produced more accurate solutions (Trafton & Reiser, 1993). Based on these findings, the authors concluded, "the most

efficient way to present material to acquire a skill is to present an example, then a similar problem to solve immediately following" (p. 1022).

The pairing of an example with an exercise is considered to be an inter-example feature. Other inter-example features include consideration of the use of multiple examples in content, the effects of varying problem types within lessons, and the effects of themes or "surface stories" on instruction. Multiple examples in section content have been shown to facilitate improved learning over a single example (Reed & Bolstad, 1991). These authors concluded that only one additional example will improve learning, and it is not necessary to provide an example for each possible exercise or test problem. Paas and Van Merrienboer (1994) demonstrated that lessons designed with high variability in content should be accompanied by worked example instruction rather than immediate immersion into exercise solving. Quilici and Mayer (1996) demonstrated that example groups designed to emphasize structure are more effective than those that emphasize surface story. If a group of examples associated with varying concepts of mixing is based upon the making of lemonade, then the group is said to be emphasized by surface story – or the context of the examples – making lemonade. If each example takes on a unique contextual setting, then the group is said to be emphasized by structure. An example of structural emphasis might be a group of examples related to unit conversion, where each example is based on a unique context – say space exploration and pool chemistry, for example.

Much research has suggested that the intra-example features of worked examples also play a critical role in their effectiveness (Catrambone, 1994a ; Mwangi & Sweller, 1998; Ward & Sweller, 1990). In fact, if not constructed properly, "the structure of worked

examples may substantially compromise the benefits derived from studying them" (Mwangi & Sweller, 1998, p. 174). On the contrary, carefully designed worked examples can reduce or eliminate the split attention effect, resulting in cognitive load reduction (Cooper, 1998; Paas, 1992). Some of the more important intra-example features include integrating text and diagrams for reducing the split-attention effect (Tarmizi & Sweller, 1988), integrating aural and visual information (Mousavi, Low, & Sweller, 1995), integrating steps and sub-goals (Catrambone, 1994a, 1994b, 1995a, 1995b, 1996), and introducing incomplete examples (Stark, 1999), an important feature for the purpose of the present study. Stark (1999) found that, compared to studying complete examples, incomplete examples are beneficial to producing higher levels of effective self-explanation. *Self-explanation* occurs when a student attempts to *fill in the gaps* of poorly elaborated or intentionally omitted content in worked example design - students who self-explain will outperform students who do not (Chi et al., 1989).

The findings of worked examples research may have significant implications in constructivist learning environments where students engage in solving complex problems (Williams & Hmelo, 1998). The literature suggests that students should thoroughly review and engage in expert problem solutions before attempting to develop solutions on their own. The present study depends on research in worked examples since they reside at the core of the Reading approach and should, therefore, be designed and delivered according to principals that best define their use.

Self-Regulation and Online Learning

Self-regulation refers to self-generated thoughts, feelings, and actions that are planned and cyclically adapted to the attainment of personal goals. It entails not only

behavioral skill in managing one's environment, but also the knowledge and the sense to enact this skill in relevant contexts (Zimmerman, 2000). More specifically, self-regulated learning is comprised of these general components (Ormrod, 1999):

1. *Self-motivation.* Self-regulated learners have an intrinsic desire to attain a particular goal or perform a specific task (Zimmerman, 1995; Zimmerman & Risemberg, 1997).
2. *Goal setting.* Self-regulated learners know where they want to go and how they want to get there (Winne, 1995; Zimmerman & Bandura, 1994).
3. *Planning.* Self-regulated learners plan their time and resources to attain a specific goal or perform a task (Zimmerman & Risemberg, 1997).
4. *Attention control.* Self-regulated learners work to maximize their attention directed toward a goal or task (Winne, 1995).
5. *Application of learning strategies.* Self-regulated learners adjust learning strategies according to situation (Winne, 1995).
6. *Self-monitoring.* Self-regulated learners are capable of monitoring their own progress and adjusting learning strategies as needed to attain the goal or accomplish the task (Butler & Winne, 1995; Winne, 1995; Zimmerman & Risemberg, 1997).
7. *Self-evaluation.* Self-regulated learners can determine when they've accomplished the goal or completed the task (Schraw & Moshman, 1995; Zimmerman & Risemberg, 1997).

Students who are highly self-regulated establish high academic goals and achieve at a higher level (Schraw, 1998). In reality, relatively few students function at a high level of self-regulation, possibly due to teaching and learning paradigms imposed by traditional instructional practice (Zimmerman & Bandura, 1994). If this is the case, alternative

learning environments like the virtual classroom might unleash higher levels of currently constrained self-regulatory skills in students, which, in turn, could potentially lead to higher academic achievement.

State Self-Regulation. Although it is clear that worked examples can improve learning in a traditional setting, it is not yet known how such examples might affect learning in a Web-based environment. If media's effect on learning might be inconsequential, then student dependence on worked examples in a Web-based environment might be attributed more to individual learning characteristics, such as self-regulation. A clearer understanding of this dependence can lead to better understanding of the constructs needed for good, scientifically-based, instructional design.

Hong (1998) distinguishes two different classes of personality or psychological attributes that can be applied to self-regulation – trait and state constructs. State self-regulation is conceptualized as a transitory state that varies depending on situational cognitive demands. For example, trait self-regulation is a performance attribute that remains relatively stable across varying cognitive demands (Hong, 2001a). While the study of both attributes is important for determining individual differences in learning and performance, the present study seeks to build upon prior research to describe how state self-regulation might effect various performance measures including the use of worked examples.

Presently, no studies exist that investigate the relationship between self-regulatory skills and hypermedia environments (Hartley & Bendixen, 2000). Zeidner, Boekarts, & Pintrich (2000) offers directions and challenges for future research in this area:

1. Exploring interactions between environment and self-regulation.

2. The acquisition and transmission of self-regulatory skills.
3. Training and promotion of self-regulatory skills.
4. Examining developmental differences in self-regulatory skills.
5. Examining individual differences in self-regulatory skills.

These challenges raise some important questions that can only be answered through further research. Success or achievement in new learning environments at this point is best summarized by Hartley and Bendixen (2001), "While we may have succeeded in improving access to all, we have only succeeded in increasing access to learning for a few." (p. 24). In other words, Web learning has not yet been beneficial to the masses.

Since it is anticipated that self-regulation will play an important role in predicting student success in online courses, a review of self-regulation research might yield insight into student self-regulatory ability and how students might solve programming problems online.

CHAPTER 3

DEVELOPMENT OF CS ONLINE

The purpose of the current study was to develop a Web-based learning system as a teaching and learning tool for introductory computer science concepts, and then perform preliminary validation of the system's efficacy as an instructional environment. Approval was granted on February 20, 2003, by the Social Behavioral Sciences Institutional Review Board of the University of Nevada, Las Vegas before conducting the research. The chapter describes key design attributes of CS Online resulting from rationale and fundamental strategies described in Chapter 2. The attributes are presented within the context of environmental, pedagogical, methodological, technical, and structural design strategies. *Environmental* strategies are those that might address factors outside of teacher control including lack of standards for a high school curriculum and teacher certification, professional development issues, increasing growth of technical education, declining enrollments in computer science courses, increasing complexity of programming languages, accessibility to resources, and cost factors. *Pedagogical* strategies involve classroom and course management, scope and sequence of content, and instructional design. *Methodological* strategies involve the methods of teaching computer programming. *Technical* strategies address the choice of platform, development environment, language, and appropriate instruction and use of debugging tools.

Structural strategies are those associated with the artistic side of programming and software design.

Environmental Design Strategies and CS Online Attributes

The environmental challenges facing computer science education include the lack of standards for a high school curriculum and teacher certification, professional development issues, increasing growth of technical education and declining enrollments in computer science courses, increasing complexity of programming languages, accessibility to resources, and cost factors. While individual teachers or any one system may not possess the power to effect change in many of these areas, features inherent in CS Online can help empower teachers to overcome others. In this section, a review of those challenges most affected by CS Online design attributes is presented.

Teacher Professional Development and CS Online Design Attributes

As computer science emerged as a field of study in secondary schools, many teachers ended up teaching the subject because they were either the most knowledgeable in computers or were the first to indicate an interest. CS Online was designed to reach the many teachers who lack either content knowledge or adequate resources to effectively teach computer science. This was done by allowing teachers to function in the system both as a teacher and a student. In other words, teacher status in the system implies that in addition to managing their students, teachers can progress through content as if they were students themselves. Teachers can, therefore, use the system to learn content ahead of or alongside their students. This inherent professional development component of CS

Online can be particularly helpful for those teachers who lack extensive formal training in computer science.

Cost, Accessibility and CS Online Attributes

While national efforts to help prepare students for the AP exam are noteworthy, CS Online was designed to reach the masses, the tens of thousands of students who might never see an AP computer science course or exam, but who can benefit cognitively from learning programming. In addition, because CS Online is Web-based, the complete learning environment is accessible from anywhere that teachers and students have Internet access. The model requires no expensive software development tools or accompanying textbooks; everything is self-contained. The system was made (and continues to be made) available at no cost to all 285,000 students in the Clark County School District.

A Bridge to Technology Education

The aversion to computer science being incorporated into a broader technology education program is understandable considering the academic nature of computer science and the varying levels of inexperience technology teachers bring to the subject (Deek & Kimmel, 1999). But this trend is more likely to continue before it's reversed (Levesque & Hudson, 2003), and computer science teachers might, in the meanwhile, better serve education by adapting to rather than resisting programmatic changes. CS Online promotes a spirit of cooperation mainly because of JavaScript's natural affinity to web pages. Problem solving activities in CS Online translate directly into web pages and require knowledge of HTML for output formatting, web forms, and dynamically controlled page content. In other words, the *side effects* of CS Online are consistent with

the instructional goals of technology education programs to teach HTML and web design. In fact, Chapter-10 covers visual interface design using an advanced Web development tool, and Chapter-12 shows students how to publish their programs (pages) on the Web. In summary, CS Online can be used as a means to promote further study in technology education courses like HTML and web page design, and vice versa.

Pedagogical Design Strategies and CS Online Attributes

CS Online was designed to address several pedagogical challenges to computer science education. A discussion of its approach to classroom and course management, scope and sequence of content, and instructional design issues follows.

Classroom and Course Management

Since the management of CS Online is Web-based, teachers can view and run submitted source code from any computer that has Internet access and a browser, including their own at home. The convenience of this attribute cannot be overstated. Not only can teachers view and run the final code submitted for grading from home, but they can also view *every* attempt students make to debug and run *every* program. This *history* of problem solving opens a new window into student thinking and problem solving not found in the review of computer science education literature. From this data, teachers can identify patterns of common mistakes students make while trying to solve programming problems, giving them opportunity to improve instruction. Finally, the management component of CS Online allows teachers to provide students with immediate feedback on programming progress and reset completed problems for them to complete additional work on erroneous problems.

Content Scope and Sequence

CS Online intends to provide students with real problem solving experiences using algorithmic and logical thinking, and students write programs toward this goal. CS Online presents the most important language structures controls in a way that makes learning fun and easy for students.

Summary of CS Online Content Scope and Sequence

The course is divided into chapters and sections to keep content in small and concise chunks. Beginning with an introduction, input/output (I/O) and variables, students become familiar with the programming environment and ways to put information into and get information back from their programs. Unconditional looping is also introduced early to control I/O for array variables. Chapter-3 follows with an introduction to objects, and how to reference and use various object properties and methods in programs. CS Online emphasizes object orientation because it not only produces superior attitudes and learning effects in programming (Reed and Liu, 1992), but also offers the best way to write computer programs (Coad & Yourdon, 1993; Savitch, 2003). In Chapter-4, students construct their own objects and use them in programs, and Chapter-5 shows students how to connect visual interface components and objects. Visual interface components include images, buttons, text fields, drop-down lists, radio buttons, and checkboxes – components of standard Web forms.

By the end of Chapter-5, students have been engaged with building projects that were then expanded in subsequent chapters as new material is presented. Chapters 6 and 7 follow with decision structures, conditional iteration and expanded project functionalities.

Examples of projects include a calculator, dice roller, and a CD player simulator. The instructional benefits projects of this type can offer include opportunities to teach difficult concepts. The dice roller, for example, is used to teach *compound conditions* and *if statements* by making the program recognize suits. Suit recognition is useful in popular games like Yahtzee, Kizmet, and draw poker. Enrichment activities are provided for the more motivated to extend the basic requirements into more sophisticated solutions. The CD player simulator is a project used to teach conditional iteration through variations of random track playback sequences. Advanced topics of introductory computer science then follow in Chapters 8 and 9 with searching, sorting, and other algorithms; the application of multi-dimensioned arrays; and recursion. In Chapter-10, students learn to create their own visual interfaces using an advanced Web development tool like Macromedia Dreamweaver MX to create Web forms. By the end of Chapter-10, students are prepared to spend considerable time designing and building their own projects from scratch. CS Online provides a library of project ideas with source code and visual interfaces that students can view, run, and modify to reinforce concepts provided throughout the course and to generate ideas. Examples include a hi-lo game, a stopwatch, bingo, a scrambled word game, and other real-world projects.

Because students write programs in JavaScript (details about choice of language are provided in the *Technical Attributes* section), completed projects are Web pages that are easily published and showcased via the school Web site or anywhere else on the Web. The ability to showcase student work through the Web builds motivation and pride (DuPont, 1998). Chapter-12 provides students with instructions on how to publish their programs (pages) on the Web, and how their pages compare to Web pages in general.

The chapter also functions as a segue into study of hypertext markup language (HTML) and advanced Web design. See Table 3-1 for a summary of the scope and sequence of CS Online.

Functional Progression of Chapter and Section Content

Each chapter begins with an explanation of concepts with hyperlinks to supplementary Web sites in support of the concepts. Chapters are sub-divided into sections to keep Web pages small and concise (Lynch & Horton, 1999). The curriculum provides random access links to chapters and sections so that students could, at any time, reference content and previously solved exercises. Embedded in each section are in-text worked examples for students to read, trace, and run as often as needed. Students can also modify and re-run any example at any time. At the end of each section is a list of exercises for students to practice programming concepts. Embedded in each exercise are optional in-exercise worked examples for students to apply toward their own solutions. Appendix-A contains samples of chapter, section, and example content.

Worked examples are provided to illustrate concepts introduced in each section. The output of worked examples is viewed by clicking the link to the example. The example opens and runs in a new window. Students can then trace the program and the output to see how the solution worked, and source code could be copied and pasted into the exercise edit window.

Table 3-1

Scope and Sequence of CS Online

Chapter and Title	General Content
Chapter-1: Introduction	Introduction to the course, how to use the system, debugging tools.
Chapter-2: I/O and Variables	Input and output, variable naming convention, unconditional iteration using *for* loops.
Chapter-3: Objects	JavaScript objects including referencing object properties and methods.
Chapter-4: User-Defined Objects	Constructing objects in JavaScript.
Chapter-5: The Visual Interface	Using a visual interface with objects.
Chapter-6: Making Decisions	Decisions using *if* and *switch*, conditions.
Chapter-7: Conditional Iteration	Control using *while* loops and conditions.
Chapter-8: Advanced Topics	Multi-dimensioned arrays, recursion, and advanced parameter passing.
Chapter-9: Algorithms	Introduction to program efficiency, searching, sorting, and other popular algorithms.
Chapter-10: Web Forms and Custom Interfaces	Constructing a visual interface using a web design tool to create web forms.
Chapter-11: Projects	Analysis of projects in the project library, design of a student project.
Chapter-12: JavaScript, HTML, and Web Pages	The relationship between JavaScript and web pages. Dynamic web page design and HTML (DHTML) using JavaScript.

Exercise Sets

Section-end exercises are designed to reinforce chapter and section content and to provide students with opportunity to practice. After clicking on the link to a problem, students are presented with an option to view a related in-text worked example or three levels of in-exercise worked examples. The in-text worked example can be viewed as an example-problem pair (Trafton & Reiser, 1993), and in-exercise worked examples can be viewed as incomplete examples (Stark, 1999). For each example-problem pair, students can: (a) run the worked example and observe its behavior, (b) modify the worked example source code and run the modified program, or (c) copy, paste, and make modifications to the worked example code as their own solution.

Running exercise solutions works much in the same way as worked examples. The students have complete autonomy in the management of source code for section-end exercises. Whenever a student attempts to run a program, CS Online saves a copy of their source code in the back-end database. In fact, copies of source code for every attempt are captured for all students, chapters, and exercises. If a student wishes to revisit a submitted problem, the most currently submitted source code is presented back upon entering the exercise. The student can inform the instructor that a problem is ready for grading by clicking the 'Ready for Grading' checkbox before submitting.

If a student is having trouble solving a problem, an incomplete worked example can be displayed to assist with the programming process. Three levels of incomplete worked examples are available for each problem and generally progress as follows: (a) pseudo-code for level-1, (b) partial solution of pseudo-code for level-2, and (c) partial solution of source code for level-3. Pseudo-code is defined as:

An outline of a program, written in a form that can easily be converted into real programming statements. For example, the pseudocode for a bubble sort routine might be written:

```
while not at end of list
        compare adjacent elements
        if second is greater than first
                switch the two elements
                get the next two elements
        if elements were switched
                repeat for the entire list
```

Pseudo-code cannot be compiled nor executed, and there are no real formatting or syntax rules. It is simply one important step in producing the final code. The benefit of pseudo-code is that it enables the programmer to concentrate on the algorithms without worrying about all the syntactic details of a particular programming language. In fact, you can write pseudocode without even knowing what programming language you will use for the final implementation (Webopedia, 2003, p. 1).

If an incomplete worked example link is clicked, the source code is displayed in the exercise text box, and problem solution can progress in the same way as before – the student can make modifications and test the program. Whenever an incomplete worked example level is used, a penalty can be applied toward the total points earned for that exercise. For CS Online, a one-half point penalty was applied for each hint level used, resulting in a 1.5 point total penalty for using all three hints. The penalty can serve as an incentive for students to work harder, or conversely, as a disincentive to give up too

easily. Appendix-B contains an example of hint levels one, two, and three and the problem solution.

Methodological Design Strategies and CS Online Attributes

The CS Online learning experience is built upon the PBL instructional paradigm where students acquire *useful knowledge* that transfers into real-world contexts (Wilkerson & Gijselaers, 1996). The Reading approach to teaching programming also lies at the core of the system, where students run pre-written programs in the form of worked examples, modify and amplify those examples, and then generate programs on their own (Van Merrienboer and Krammer, 1987). An instructional goal of CS Online is as much language independence as possible, focusing more on problem solving with application to real-world contexts. This is achieved by emphasizing constructs that are common to most popular languages like Java and C++. In general, object-oriented programming concepts are introduced early and continue throughout the course, mainly since object-oriented programming has been shown to be a more effective instructional approach (Liu, 1998; Reed & Liu, 1992).

Worked Examples

CS Online engaged two types of worked examples: *in-text* and *in-exercise*. In-text examples were embedded in chapter sections, and in-exercise examples were coupled with section-end exercises. There were at least two in-text examples per section that the students could view, run, or modify and run at any time, since multiple examples in section content have been shown to facilitate improved learning over the use of a single example (Reed & Bolstad, 1991). In-exercise examples were available to assist with

problem solutions and were presented in two general forms: (a) example-problem pairs, where the problem was associated with the in-text worked example that was most closely resembled the exercise, and (b) incomplete examples, also referred to as hints or partial solutions, which were available at three levels. Example-problem pairs have been shown to enhance skill acquisition in a most effective manner (Trafton & Reiser, 1993). Stark (1999) found that, compared to studying complete examples, incomplete examples are beneficial to produce higher levels of effective self-explanation. The use of in-text and in-exercise worked examples in CS Online was completely optional for the pilot study.

Technical Design Strategies and CS Online Attributes

Choice of Development Environment

Since CS Online is Web-based, choice of platform was not an issue. CS Online could be run from any hardware platform and operating system that supports a Web browser. Internet Explorer is the recommended browser because of its ability to integrate the Microsoft Script Debugger, which is a free download that automatically launches when a JavaScript error is encountered (Microsoft, 2003). Students can either write programs within CS Online text fields, or use any other text editor or word processor, then copy and paste their programs into the system. This development suite of an editor, debugger, and run-time environment benefits schools and students in that there are no additional software costs, and programs can be written from home or any computer with Internet access.

Choice of Language

The choice of JavaScript as the programming language was not difficult because of the many benefits realized by its use. First of all, JavaScript is easy to apply and

possesses all the necessary attributes for teaching introductory computer programming concepts. It should be clarified that JavaScript and Java are not the same language. JavaScript was designed to resemble Java, and therefore, also looks a lot like C and C++. The main difference is that Java was built as a general-purpose object language, while JavaScript is intended to provide a quicker and simpler language for enhancing Web pages and servers (Google, 2003). Because of its resemblance to the other major languages, learned concepts can be easily transferred to more advanced study of computer science. In addition, JavaScript's natural affinity to Web pages made it easy for students to showcase their work, and promotes a Web-centric educational focus on HTML, Web page design, Flash, and other Internet technologies.

The biggest criticism CS Online might receive as an effective instructional environment is the choice of JavaScript as the language for teaching introductory computer science. Choice of language is one of the most important decisions educators make in planning introductory courses and inherent concepts (Stevenson & West, 1998). A critical comparison of JavaScript and Java reveals that although the two languages are concurrently similar and fundamentally different, the differences may not be dramatic enough to dismiss the simpler of the two languages as a viable alternative. First of all, computer science education leaders are already searching for a much simpler form of Java for introductory courses (Roberts, 2003). Second, because JavaScript "descends in spirit from a line of smaller, dynamically typed languages... [they] offer programming tools to a much wider audience because of their easier syntax, specialized built-in functionality, and minimal requirements for object creation" (Netscape, 2000). Third, the majority of JavaScript constructs used in CS Online are *upward compatible* with Java.

With the exception of objects and loose typing differences, language structures including literals, block, and scope; data types including strings and arrays; expressions and operators including relational, unary, assignment, and string concatenation; and control structures including if-else, switch, and while are virtually identical in appearance and use to Java. Because the study focuses on problem solving, programming language is viewed as a matter of secondary importance, with emphasis placed on the problem to be solved and the logical steps required for its solution (Milbrandt, 1995). This approach does not intend to underplay the importance of the AP exam or preparation on its behalf, but rather to promote a way of building motivational courses to attract and teach large numbers of students. Those interested in pursuing higher study can then transfer the majority of their introductory knowledge to Java.

Structural Design Strategies and CS Online Attributes

CS Online emphasizes the concepts of structured, object-oriented program design from the very beginning. These concepts include the use of self-documenting code through choice of variable names, naming convention, appropriate use of comments, use of objects and how those objects interface with one another and the outside world. Concepts attained through CS Online are transferable to other programming languages and more advanced study of computer science.

In summary, CS Online can address many of present needs of computer science education. Advances in technology have made it possible to conceptualize and implement new models that simplify instructional processes while providing access to more students through the Internet. In addition, a research-based framework for the various pedagogical

and system attributes might increase the likelihood of effective teaching and learning experiences.

CHAPTER 4

PRELIMINARY VALIDATION EFFORT

The purpose of the study was to develop a Web-based learning system as a teaching and learning tool for introductory computer science concepts, and then perform preliminary validation of the system's efficacy as an instructional environment. Approval was granted on February 20, 2003, by the Social Behavioral Sciences Institutional Review Board of the University of Nevada, Las Vegas before conducting the research. The chapter begins with the participants in the study and then follows with the instructional materials used, a table of raw and calculated measures, and the procedures employed. The chapter closes with a summary of the research questions, data sources, and analytical methods applied.

Participants

The participants were 36 students from several high schools in Southern Nevada, and 12 graduate students from the University of Nevada, Las Vegas (UNLV). The high school students' participation in the CS Online system was the main focus of the study, while the graduate students were available to evaluate and grade student work. Of the 36 high school students, 13 were female, 23 were male, and the ethnic distribution was 80% Caucasian, 11% Hispanic, and 9% Asian American with ages ranging from 13 to 18 years

old. The high school students enrolled in CS Online as an academic elective for the spring semester of 2002. Programming is offered as an elective course in the state of Nevada, and credit earned can be applied to fulfill a graduation requirement.

The UNLV graduate students were enrolled in ICG 758, the Methods of Teaching programming course, and worked through CS Online content along with the 36 high school students. Their involvement with CS Online was threefold: (a) to observe a functional implementation of the Reading approach (method) of teaching programming, (b) to directly interface with high school students engaged in learning computer programming, and (c) to evaluate submitted source code. The graduate students' interactions with the high school students were limited to answering questions and evaluating submitted work. Several UNLV graduate students were computer science teachers in CCSD who volunteered their high school students for participation in the study.

Instructional Materials

The coursework consisted of various questionnaires as described in detail in the instruments section below, 25 sections of pedagogical content including worked examples and exercises, and an exam. All worked examples, exercises, and exams required high-level thinking processes. Questionnaires and the exam included multiple-choice items, and exercises required written program solutions. There were a total of 45 possible exercises to complete.

Instructional materials consisted of chapter and section content covering introductory concepts of programming using the JavaScript programming language, in-text worked

examples embedded in section content, and chapter-end exercise sets with optional in-exercise worked examples that were optionally available. A summary of the scope and sequence of CS Online content was presented in Table 3-1.

Measures

Sixty-six measures were generated for the pilot study, 38 of which were raw data collected by the system, and 28 of which were calculated based on raw data values. CS Online was the primary data collection instrument, which generated data as students interacted with the various components of course content. Table 4-1 shows the comprehensive list of raw (R) and calculated (C) data values (or variables) described in this chapter. Variables were assigned numbers and labels for ease of reference in subsequent sections and chapters of the present study. *Type* describes whether the variable was derived from raw or calculated data, and *Freq* describes the frequency of data collection. Frequency options include: *1*, a one-time collection of the data as in a questionnaire; *Ready*, produced when a student clicked the 'Ready for Grading' checkbox before running; *Run*, when the Run button was clicked; *Hint*, when a hint was clicked; and *Click*, when an example link or button was clicked. Data for raw variables were taken directly from database tables generated by the system. Calculated variables were created based on mathematical manipulation of raw data variables. Raw data descriptions are provided first followed by detailed descriptions of each calculated variable.

Table 4-1

Summary of Raw and Calculated Variables

Num	Variable Description	Label	Type	Freq
	The following measures are from questionnaires:			
1	Math Experience	MathScore	C	1
2	Computer Experience	CompScore	C	1
3	Trait Self-Regulation	N/A	R	1
4	Trait Self-Regulation w/Programming	N/A	R	1
	State Self-Regulation (4 separate questionnaires)		C	
5	- Section 2.7 end: planning sub-component	Planning	C	1
6	- Section 2.7 end: self-check	SelfChk	C	1
7	- Section 2.7 end: effort	Effort	C	1
8	- Section 2.7 end: self-efficacy	SelfEff	C	1
9	- Exam end: planning sub-component	Planning	C	1
10	- Exam end: self-checking	SelfChk	C	1
11	- Exam end: effort	Effort	C	1
12	- Exam end: self-efficacy	SelfEff	C	1
13	- Section 3.7 end: planning sub-component	Planning	C	1
14	- Section 3.7 end: self-checking	SelfChk	C	1
15	- Section 3.7 end: effort y	Effort	C	1
16	- Section 3.7 end: self-efficacy	SelfEff	C	1
17	- Section 4.4 end: planning sub-component	Planning	C	1
18	- Section 4.4 end: self-checking	SelfChk	C	1
19	- Section 4.4 end: effort	Effort	C	1
20	- Section 4.4 end: self-efficacy	SelfEff	C	1

Table 4-1 (Continued)

Num	Variable Description	Label	Type	Freq
	The following measures are from exercises solved:			
21	Total # of problems solved	TotSolved	R	Ready
22	% of total # of problems solved	PctSolved	C	
23	Total average score for solved problems	TotAvgSc	C	
24	# of optional problems solved	OptSolved	R	Ready
25	Average score for optional problems	OptAvgSc	C	
26	# of easy exercises solved	ESolved	R	Ready
27	Average score for easy exercises solved	EAvgSC	C	
28	# of medium exercises solved	MSolved	R	Ready
29	Average score for medium exercises solved	MAvgSC	C	
30	# of hard exercises solved	HSolved	R	Ready
31	Average score for hard exercises solved	HAvgSC	C	
	The following measures are from Submitted Attempts:			
32	Total # of submitted attempts	TotAtts	R	Run
33	Total # of exercises solved	TotSolved	R	Ready
34	Average # of submitted attempts	TotAvgAtt	C	
35	# of submitted attempts for optional exercises	OptAtts	R	Run
36	# of optional exercises solved	OptSolved	R	Ready
37	Average # of attempts for optional exercises	OptAvgAtt	C	
38	# of submitted attempts for easy exercises	EAttempts	R	Run
39	# of easy exercises solved	ESolved	R	Ready
40	Average # of attempts for easy exercises	EAvgAttempts	C	

Table 4-1 (Continued)

Num	Variable Description	Label	Type	Freq
41	# of submitted attempts for medium exercises	MAttempts	R	Run
42	# of medium exercises solved	MSolved	R	Ready
43	Average # of attempts for medium exercises	MAvgAttempts	C	
44	# of submitted attempts for medium exercises	HAttempts	R	Run
45	# of medium exercises solved	HSolved	R	
46	Average # of attempts for hard exercises	HAvgAttempts	C	Ready
	The following measures are from in-text examples:			
47	Total number of unique in-text examples visited	TotUniqInTxt	R	Click
48	Percent of in-text examples visited	PctUniqInTxt	C	
49	Total visits to in-text examples	TotInTxt	R	Click
50	Average # of visits to unique in-text examples	AvgPerUniqInTxt	C	
	The following measures are from in-exercise examples:			
51	Total number of unique in-exercise examples	TotUniqInEx	R	Click
52	Percent of in-exercise examples visited	PctUniqInEx	C	
53	Total visits to in-exercise examples	TotInEx	R	Click
54	Average # of visits to in-exercise examples	AvgPerUniqInEx	C	
	The following measures are from hints:			
55	Sum of hint levels used in all problems	HintSum	C	
56	# of problems where hints were used	HintProbs	R	Hint
57	Average hint level where hints were used	AvgHintLev	C	

Table 4-1 (Continued)

Num	Variable Description	Label	Type	Freq
58	Sum of hint levels used in optional problems	OptHintSum	C	
59	# of optional problems where hints were used	OptProbs	R	Hint
60	Average hint level in opt problems where used	AvgOptHintLev	C	
61	# of easy problems where hints were used	EHintProbs	R	Hint
62	Sum of hint levels in easy problems	EHintSum	C	
63	Average hint level in easy problems where used	EAvgHintLev	C	
64	# of medium problems where hints were used	MHintProbs	R	Hint
65	Sum of hint levels in medium problems	MHintSum	C	
66	Average hint level in medium problems where used	MAvgHintLev	C	
67	# of hard problems where hints were used	HHintProbs	R	Hint
68	Sum of hint levels in hard problems	HHintSum	C	
69	Average hint level in hard problems where used	HAvgHintLev	C	
	The following measures are from the exam:			
70	Exam score	ExamScore	C	
71	Exam percentage	ExamPct	C	

Raw Variables

For each raw and calculated variable described in this and the next section, reference is made to the variables listed in Table 4-1 using the following syntax: *Variable Mnemonic (Number)*. For example, *TotSolved(21)* refers to variable 21 in Table 4-1, the total number of problems solved. All raw variables were derived from table queries

applied to raw data generated by the system. More specifically, the Web-based data table was converted to a Microsoft Access table. Queries were then designed to extract data into workable data sets. These data sets were then transferred to Excel spreadsheets from which calculated variables were derived.

TotSolved(21). The total number of exercises (including optional) that were checked 'Ready for Grading' by the student.

OptSolved(24). The number of optional-only exercises that were checked 'Ready for Grading' by the student.

ESolved(26). The number of easy exercises that were checked 'Ready for Grading' by the student. Easy, medium, and hard level exercises were selected from within the first 20 of 45 exercises to ensure the highest rate of student completion. The five easy exercises were 2-1-1, 2-1-2, 2-1-3, 2-2-1, and 2-2-2. The five medium exercises were 2-3-1, 2-3-3, 2-4-1, 2-4-3, and 2-5-1. The five hard exercises were 2-3-2, 2-4-2, 2-6-1, 2-7-1, and 3-2-1. Exercises are identified by C-S-E notation, with C equal to chapter number, S equal to section number, and E equal to exercise number. To qualify as an easy, medium, or hard exercise, the anticipated average number of submitted attempts (calculated variable 34 of Table 3-2) was used for classification. A TotAvgAtt value of 1-5 qualified the exercise as easy. Values ranging from 6-10 were classified medium, and the 11-20 range was classified as hard.

Msolved(28). The number of medium exercises that were checked 'Ready for Grading' by the student. Refer to ESolved above for an explanation of criteria applied for classifying an exercise as easy, medium, or hard.

Hsolved(30). The number of hard exercises that were checked 'Ready for Grading' by the student. Refer to ESolved above for an explanation of criteria applied for classifying an exercise as easy, medium, or hard.

TotAttempts(32). The total number of attempts (runs) submitted for all exercises (including optional).

OptAtts(35). The total number of attempts submitted for all optional exercises.

EAttempts(38). The number of attempts submitted for easy problems.

Mattempts(41). The number of attempts submitted for medium problems.

HAttempts(41). The total number of submitted attempts for hard problems solved.

TotUniqInTxt(47). The number of unique in-text examples visited, regardless of the number of times for each. Each section provided worked examples presented as example-problem pairs.

TotInTxt(49). The total number of visits to in-text examples, including multiple visits to the same example.

TotUniqInEx(51). The number of unique in-exercise examples visited, regardless of the number of times for each.

TotInEx(43). The total number of visits to in-exercise examples, including multiple visits to the same example.

HintProbs(56). The total number of problems where *at least* hint level-1 was used. Hints were available for each problem at three levels, and the system recorded the levels at which they were requested.

OptProbs(59). The total number of optional problems where *at least* hint level-1 was used.

EhintProbs(61). The total number of easy problems where *at least* hint level-1 was used.

MhintProbs(64). The total number of medium problems where *at least* hint level-1 was used.

HhintProbs(67). The total number of hard problems where *at least* hint level-1 was used.

Calculated Variables

MathScore(1). Measure of math experience. To measure prior knowledge and achievement in mathematics, the Math Knowledge questionnaire was developed and used (Hong, 2003). The questionnaire can be found in Appendix-C. The questionnaire measured the number of math courses taken from 9 possible courses. Each positive response (Yes) yielded one point. The grade earned for each 'Yes' response yielded scores of 4 for A, 3 for B, 2 for C, 1 for D or below, and 0 for no experience in the course. The MathScore measure was calculated by the sum of positive responses (possible of 9) and the average grade (possible of 4) for courses completed. The range of values for the continuous version of this measure was 0 to 13 which were divided into four categories of poor, low, good, and high for values less than 5, 5-7.4, 7.5-9.9, and 10 and above, with low group scores ranging from 0 to 7.4 (poor to low), and high group scores ranging from 7.5 to 13 (good to high).

CompScore(2). Measure of computer experience. To measure prior knowledge and achievement in the use of computers and programming, the Computer Experience questionnaire was developed and used (Hong & Halopoff, 2003). The questionnaire can be found in Appendix-C. The questionnaire measured student experience in 13 areas of

computer use ranging from literacy to HTML and programming in various languages. The CompScore measure was calculated by the sum of positive responses. The range of values for the continuous version was 0 to 13, and these values were divided into four categories of poor, low, good, and high for values less than 5, 5-7.4, 7.5-9.9, and 10 and above, with low group scores ranging from 0 to 7.4 (poor to low), and high group scores ranging from 7.5 to 13 (good to high).

Self-regulation(5-20). Self-regulation measures were taken following the end of section 2-7 and the exam. The measures taken following sections 3-7 and 4-4 were completed by fewer students since not all students made it that far in the course. The questionnaire can be found in Appendix-C.

Planning(5, 9, 13, and 17). Derived by averaging the scores of items 1, 8, 15, 22, 29, and 33 of a 36-item questionnaire. Scores for each item ranged in value from 1 to 4, where 1 represented a response of "Not at all", 2 represented a response of "Somewhat", 3 represented a response of "Moderately so", and 4 represented a response of "Very much so."

Self-checking(6, 10, 14, and 18). Derived by averaging the scores of items 2, 9, 16, 23, 30, and 34 of the same Self-Regulation questionnaire. Scores for each item were determined in the same manner as the planning sub-component.

Effort(7, 11, 15, and 19). Derived by averaging the scores of items 3, 10, 17, 24, 31, and 35 of the same Self-Regulation questionnaire. Scores for each item were determined in the same manner as the planning sub-component.

Self-efficacy(8, 12, 16, and 20). Derived by averaging the scores of items 4, 11, 18, 25, 32, and 36 of the same Self-Regulation questionnaire. Scores for each item were determined in the same manner as the planning sub-component.

PctSolved(22). Percent of total number of problems solved. The total number of problems completed by a student divided by 45 - the total number of problems available.

TotAvgSc(23). The average score earned for problems solved – the sum of earned points divided by the number of problems completed by a student. Each exercise was worth a maximum of 10 points with a penalty of 0.5 points applied for each hint level used. Students earned a minimum of 5 points for each exercise where reasonable effort was given. Point values were assigned by teaching assistants who reviewed the problems and assigned scores.

OptAvgSc(25). The average score earned for optional exercises solved – the sum of earned points divided by the number of optional problems completed by a student. The following exercises were defined as optional: 3-2-3, 3-4-2, 3-7-3, and 4-5-2. Each optional exercise was worth a maximum of 10 points with a penalty of 0.5 points applied for each hint level used. Students earned a minimum of 5 points for each exercise where reasonable effort was given.

EAvgSC(27). The average score earned for easy problems solved – the sum of earned points divided by the number of exercises completed by a student. 5 exercises were selected to measure student performance at the easy level.

MAvgSC(29). The average score earned for medium problems solved – the sum of earned points divided by the number of exercises completed by a student. 5 exercises were selected to measure student performance at the medium level.

HAvgSC(31). The average score earned for hard problems solved – the sum of earned points divided by the number of exercises completed by a student. 5 exercises were selected to measure student performance at the hard level.

TotAvgAtt(34). Average number of submitted attempts, calculated by the sum of the number of attempts divided by the number of completed exercises. The number of attempts per completed exercise is determined by the count of attempts before the student checked the exercise as ready for grading. Ready for grading status prevented students from any further modifications, and hence, additional submitted attempts.

OptAvgAtt(37). Average number of attempts for optional exercises, calculated by the sum of the number of attempts divided by the number of completed optional exercises. The same ready-for-grading status applied to optional exercises.

EavgAttempts(40). Average number of submitted attempts for easy level exercises, calculated by the sum of the number of attempts divided by the number of completed exercises in the easy range. The same ready-for-grading status applied to easy exercises.

MAvgAttempts(43). Average number of submitted attempts for medium level exercises, calculated by the sum of the number of attempts divided by the number of completed exercises in the medium range. The same ready-for-grading status applied to medium exercises.

HavgAttempts(46). Average number of submitted attempts for hard level exercises, calculated by the sum of the number of attempts divided by the number of completed exercises in the hard range. The same ready-for-grading status applied to hard exercises.

PctUniqInTxt(64). Percent of unique in-text examples visited. 64 unique in-text examples were available throughout the entire course content. In-text examples are

workable examples embedded within the section content. A click of an example constituted a visit, and this measure is the number of in-text examples that were clicked at least once divided by the total number, 64.

AvgPerUniqInTxt(50). Average number of visits to unique in-text examples, calculated by the total number of visits (clicks) to in-text examples divided by the number of unique in-text examples visited.

AvgPerUniqInEx(54). Average number of visits to unique in-exercise examples, calculated by the total number of visits (clicks) to in-exercise examples divided by the number of unique in-exercise examples visited. In-exercise examples were available for students from within the exercise window.

HintSum(55). Total sum of hint levels used for all completed exercises. Hints were optional, and a hint was counted whenever at least hint level-1 was used. There were three hint levels available.

AvgHintLev(57). Average hint level used for each completed exercises, calculated by HintSum divided by the number of exercises where hints were used.

OptHintSum(58). Total sum of hint levels used for all completed optional exercises. Hints were optional, and a hint was counted whenever at least hint level-1 was used. There were three hint levels available.

OptAvgHintLev(60). Average hint level used for each completed optional exercise, calculated by OptHintSum divided by the number of optional exercises where hints were used.

EHintSum(62). Sum of hint levels used in easy level exercises.

MhintSum(65). Sum of hint levels used in medium level exercises.

HHintSum(68). Sum of hint levels used in hard level exercises.

EavgHintLev(63). Average hint level used in easy range exercises, calculated by EHintSum divided by the number of easy level exercises where hints were used.

MAvgHintLev(66). Average hint level used in medium range exercises, calculated by MHintSum divided by the number of medium range exercises where hints were used.

HavgHintLev(69). Average hint level used in hard range exercises, calculated by HHintSum divided by the number of hard range exercises where hints were used.

ExamScore(70). Total possible points earned for the exam. The exam was comprised of 12 programming exercises worth 5 points each. A total of 60 points were possible for this measure.

ExamPct(71). Percent of total possible points for the exam.

Source code history. For each submitted solution attempt, a copy of the source code was stored in the back-end database.

Data Sources

Questionnaires

Three questionnaires presented automatically to the students were required to be completed before students were permitted to proceed with section content. These included the math experience, computer experience, and self-regulation questionnaires that were administered according to the schedule shown in Table 4-2. See Appendix C for copies of each of these questionnaires.

Table 4-2

Schedule of Required Questionnaires

Questionnaire	When Administered
Math Knowledge	Beginning of the course
Computer Experience	Beginning of the course
State self-regulation	Following section 2.7
State self-regulation	Following section 3.7
State self-regulation	Following section 4.4
State self-regulation	Following the midterm exam

System generated data

CS Online generated or collected data automatically as students progressed through the course content. This data included all raw variables as shown in Table 4-1. Data were converted from the server into a Microsoft Access database, and then compiled into useful numbers through SQL queries.

Hand entered data

Exercise scores were hand entered into the system following student indication that the exercise was ready for grading. In some rare instances, some students requested an exercise to be *reset* so that it could be submitted again for re-grading, but the number of reset requests was negligible.

Procedure

The general procedure students followed to participate in the course involved accessing the Web site through a browser, registering for the course, awaiting an E-

Mailed password upon approval, then logging in to the system. The detailed steps are as follows:

1. Students registered for the CS Online course by accessing the Web URL http://www.csonline.ccsd.net and clicking Student Registration.
2. Computer Science instructors at the schools provided the system administrator (the present investigator) with a list of names expected to participate in the course.
3. The system administrator approved the registered students. Passwords were auto-eMailed to the students to the address they provided in the registration form.
4. After logging in to the system for the first time, students were able to review introductory information in chapter-1. Upon entering section-1 of chapter-2, students were immediately presented with the math and computer experience questionnaires.
    a. Students were required to complete all questions in the math experience questionnaire before moving on to the computer questionnaire.
    b. Students were required to complete all questions in the computer experience questionnaire before moving on to section 2-1.
5. Students proceeded to work through content chapter-by-chapter and section-by-section. Each section contained in-text worked examples and required exercises. Some sections contained optional exercises.
    a. Students were permitted to proceed to the next exercise only after submitting the current exercise by clicking the 'Ready for Grading' checkbox and then the Submit button.
    b. Students were permitted to proceed to the next section only when all required exercises in the current section were completed. This was done by clicking the

'Ready for Grading' checkbox and the Submit button following the last required exercise.

6. At the end of section 2-7, students were automatically presented with the Self-Regulation questionnaire by the system.
   a. Students were required to complete all questions in the questionnaire before they could proceed to section 3-1.
7. Midway through the 10 week time period (approximately week 6), students were presented with the exam.
   a. Students were required to complete all exercises in the exam before being allowed to continue with course content.
   b. Students were also required to complete the self-regulation questionnaire immediately following the exam before being allowed to proceed with more content.
8. At the close of the 10-week period, databases were copied and prepared for analysis through SQL queries and other calculations.

Summary of the Research Questions

Tables 4-3 through 4-7 provide a summary of the research questions, data sources, and the analytic approaches to answering the questions. The applied analytical methods were comparisons of mean frequencies and scores for each measure. Beginning with Table 4-3, the data sources used were various measures including a self-regulation questionnaire score and scores associated with the use of in-text worked examples, in-

exercise worked examples, optional exercises, hints used, attempts required to complete exercises, and exercise and exam performance.

Table 4-4 shows a summary of data and analytical methods applied to research Quesiton-2. The data sources were various measures including a math experience questionnaire score and scores associated with the use of in-text worked examples, in-exercise worked examples, optional exercises, hints used, attempts required to complete exercises, and exercise and exam performance.

Table 4-5 shows a summary of data and analytical methods applied to research Quesiton-3. The data sources were various measures including a computer experience questionnaire score and scores associated with the use of in-text worked examples, in-exercise worked examples, optional exercises, hints used, attempts required to complete exercises, and exercise and exam performance.

Table 4-6 shows a summary of data and analytical methods applied to research Quesiton-4. The data sources were various measures including self-regulation questionnaire scores and scores associated with the use of in-text worked examples, in-exercise worked examples, optional exercises, hints used, attempts required to complete exercises, and exercise performance for easy, medium, and hard-level exercises.

Table 4-7 shows a summary of data and analytical methods applied to research Quesiton-5. The data sources used were observation of submitted source code for selected exercises. The applied analytical method was a comparison of changes made by students to source code between consecutive attempts to solve exercises.

Table 4-3

Data Sources and Analyses for Research Question-1

Question	Data Sources	Analysis
(1) How do students with low and high self-regulatory skills perform in the use of in-text worked examples, in-exercise worked examples, hints, optional exercises, and problem-solving scores?	(1A) Self-regulation questionnaire given at the end of section 2-7. (1B) A count of the number of times all in-text worked examples are run for all exercises in chapters 2, 3, 4, and 5. (1C) The number of in-exercise worked examples accessed for all exercises in chapters 2, 3, 4, and 5. (1D) The number of optional exercises completed for all exercises in chapters 2, 3, 4, and 5. (1E) The average hint level used for all exercises in all chapters 2, 3, 4, and 5. (1F) The average number of attempts for each exercise in all chapters 2, 3, 4, and 5. (1G) The total score for all exercises in chapters 2, 3, 4, and 5.	Describe the mean scores of each of 15 measures for high versus low self-regulation groups using descriptive statistics. Low group self-regulation scores ranged from 0.0 to 2.5, and High group scores ranged from 2.6 to 4.0. Self-regulation score ranges applied to four sub-components of self-regulation including planning, self-checking, effort, and self-efficacy. *Measures*. Mean scores for each of the following raw and calculated data: (1-1) Planning and monitoring (meta-cognitive activity) (1-2) Self-efficacy and effort (motivation) (1-3) AvgHintLevel(57) (1-4) HintProbs (56) (1-5) AvgPerUniqInEx(54) (1-6) TotInEx(53) (1-7) TotUniqInEx(51) (1-8) AvgPerUniqInTxt(50) (1-9) TotInTxt (49) (1-10) TotUniqInTxt(47) (1-11) OptAvgAtt(37)

Table 4-3 (Continued)

Question	Data Sources	Analysis
		(1-12) OptAvgSc(25)
		(1-13) OptSolved(36)
		(1-14) TotAvgAtt(34)
		(1-15) TotAvgSc(23)
		(1-16) TotSolved(21)
		(1-17) ExamScore(70)

Table 4-4

Data Sources and Analyses for Research Question-2

Question	Data	Analysis
(2) How do students with low and high math experience perform in the use of in-text worked examples, in-exercise worked examples, hints, optional exercises, and average number of attempts to solve problems, and problem-solving scores?	(2A) Math experience questionnaire given at the beginning of the course . (2B) A count of the number of times all in-text worked examples are run for all chapters 2, 3, 4, and 5. (2C) The number of in-exercise worked examples accessed for all chapters 2, 3, 4, and 5. (2D) The number of optional exercises completed for all chapters 2, 3, 4, and 5. (2E) The average hint level used for all chapters 2, 3, 4, and 5. (2F) The average number of attempts for each exercise for all chapters 2, 3, 4, and 5. (2G) The total score for all exercises in the last sections for all chapters 2, 3, 4, and 5.	Describe the mean scores of each of 15 measures for high versus low math experience groups using descriptive statistics. Low group math scores ranged from 0.0 to 7.4, and High group scores ranged from 7.5 to 13.0. *Measures*. Mean scores for each of the following raw and calculated data: (2-1) Math experience score (2-2) AvgHintLevel(57) (2-3) HintProbs (56) (2-4) AvgPerUniqInEx(54) (2-5) TotInEx(53) (2-6) TotUniqInEx(51) (2-7) AvgPerUniqInTxt(50) (2-8) TotInTxt (49) (2-9) TotUniqInTxt(47) (2-10) OptAvgAtt(37) (2-11) OptAvgSc(25) (2-12) OptSolved(36) (2-13) TotAvgAtt(34) (2-14) TotAvgSc(23) (2-15) TotSolved(21) (2-16) ExamScore(70)

Table 4-5

Data Sources and Analyses for Research Question-3

Question	Data	Analysis
(3) How do students with low and high computer experience perform in the use of in-text worked examples, in-exercise worked examples, hints, optional exercises, and average number of attempts to solve problems, and problem-solving scores?	(3A) Computer experience questionnaire given at the beginning of the course. (3B) A count of the number of times all in-text worked examples are run for all chapters 2, 3, 4, and 5. (3C) The number of in-exercise worked examples accessed for all chapters 2, 3, 4, and 5. (3D) The number of optional exercises completed for all chapters 2, 3, 4, and 5. (3E) The average hint level used for all exercises for all chapters 2, 3, 4, and 5. (3F) The average number of attempts for each exercise in all chapters 2, 3, 4, and 5. (3G) The total score for all exercises in all chapters 2, 3, 4, and 5.	Describe the mean scores of each of 15 measures for high versus low computer experience groups using descriptive statistics. Low group computer scores ranged from 0.0 to 7.4, and High group scores ranged from 7.5 to 13.0. Measures. Mean scores for each of the following raw and calculated data: (3-1) Computer experience (3-2) AvgHintLevel(57) (3-3) HintProbs (56) (3-4) AvgPerUniqInEx(54) (3-5) TotInEx(53) (3-6) TotUniqInEx(51) (3-7) AvgPerUniqInTxt(50) (3-8) TotInTxt (49) (3-9) TotUniqInTxt(47) (3-10) OptAvgAtt(37) (3-11) OptAvgSc(25) (3-12) OptSolved(36) (3-13) TotAvgAtt(34) (3-14) TotAvgSc(23) (3-15) TotSolved(21) (3-16) ExamScore(70)

Table 4-6

Data Sources and Analyses for Research Question-4

Question	Data	Analysis
(4) How do students with low and high self-regulatory skills perform in the use of in-text worked examples, in-exercise worked examples, hints, optional exercises, average number of attempts to solve problems, and problem-solving scores as the task difficulty increases?	The operational definition of task difficulty is the average number of attempts to complete the exercise. This measure is a continuous variable ranging in difficulty from easy to medium to hard. (4A) Self-regulation questionnaire given at the end of section 2-7. (4B) Self-regulation questionnaire given at the end of the exam. (4C) A count of the number of times all in-text worked examples are run for all chapters 2, 3, 4, and 5. (4D) The number of in-exercise worked examples accessed for all chapters 2, 3, 4, and 5. (4E) The number of optional exercises completed for all chapters 2, 3, 4, and 5. (4F) The average hint level used for all exercises in all chapters 2, 3, 4, and 5.	Describe the mean scores of each of 15 measures for easy, medium, and hard level exercises for low and high self-regulation groups using descriptive statistics. Easy, medium, and hard level exercises were selected from within the first 20 of 45 exercises to ensure the highest rate of student completion. Exercises were classified according to the average number of attempts that were required by students. 1-5 attempts qualified for easy, 6-10 qualified for medium, and 11-20 qualified for hard.Self-regulation was divided into four sub-components as in Question-1 including planning, self-checking, effort, and self-efficacy; with Low group scores ranging from 0.0 to 2.5, and High group scores ranging from 2.6 to 4.0. Describe changes in mean scores for 5 measures of task difficulty. As the task difficulty progressed from easy to hard, percentage differences among low and high

Table 4-6 (Continued)

Question	Data	Analysis
	(4G) The average number of attempts for each exercise in all chapters 2, 3, 4, and 5. (4H) The total score for all exercises in all chapters 2, 3, 4, and 5.	self-regulation groups were compared to identify trends of increasing or decreasing mean score for each of the 5 measures. *Measures*. Mean scores for each of the following raw and calculated data. Percentage differences are also used to predict trends: (4-1) Planning and monitoring (meta-cognitive activity) (4-2) Self-efficacy and effort (motivation) (4-3) EAttempts(38) (4-4) EAvgAttempts(40) (4-5) EHintProbs(61) (4-6) EAvgHintLevel(63) (4-7) EAvgScore(27), (4-8) MAttempts(41) (4-9) MAvgAttempts(43) (4-10) MHintProbs(64) (4-11) MAvgHintLevel(66) (4-12) MAvgScore(30) (4-13) HAttempts(44) (4-14) HAvgAttempts(46) (4-15) HHintProbs(67) (4-16) HAvgHintLevel(69) (4-17) HAvgScore(33)

Table 4-7

Data Sources, and Analyses for Research Question-5

Question	Data	Analysis
(5) What common mistakes do students make in solving programming problems?	(5A) Observation of submitted source code for selected exercises from chapters 2, 3, 4, and 5. (5B) Requirements for students to complete worked examples and optional exercises. (5C) Findings from questions 1-4 in parts I and II above.	Qualitative evaluation of submitted source code and implications observations of common mistakes made. More specifically, perform a domain analysis of submitted source code attempts to solve problems. Identify and cluster the differences into domains to find common mistakes and possible sources of those mistakes. Analysis of descriptive statistics.

In summary, Question-1 addressed descriptive measures of student self-regulatory skills, use of various types of worked examples, and problem-solving performance based on exercise and exam scores. Questions- 2 and 3 focused on descriptive measures of student math and computer experience, the use of various types of worked examples, and problem-solving performance based on exercise and exam scores. Question-4 addressed descriptive measures of task difficulty, self-regulatory skills, the use of various types of worked examples, and problem-solving performance based on exercise and exam scores. Question-5 revealed domains in problem solving that could help inform findings from Questions 1 through 4.

CHAPTER 5

RESULTS

This chapter presents results from the pilot implementation of the development effort. Thirty-six students worked through 25 sections of content, 64 worked examples, and 45 programming exercises. Their efforts resulted in 12,436 raw data items generated by the system, 1,944 data items calculated from raw data, and 1,602 submitted source code samples for comparison. Descriptive statistical analyses were applied to identify comparative performance factors and trends. Although the pilot effort ended in mid 2003, CS Online has been put into production for the Clark County School District and continues to grow in its service to computer science education. Approximately 200 new students and teachers from ten high schools have entered the system since the close of the project.

The chapter begins with a summary of student questionnaire results, followed by findings pertaining to each of the research questions. For each of the first four research questions, results of descriptive analytical methods are presented followed by a summary of comparative performance among high and low groups. Findings for Question-5 are then presented by providing an overview of the domain analysis and a description of the resulting major and minor domains. Code comparisons are summarized in various tables for each major and minor domain.

The 66 unique measures introduced in Table 4-1 are repeated in Table 5-1 for convenient reference throughout the chapter. Some of the measures are repeated resulting in a total of 71 measures shown in the table. Table 5-1 is a slight modification to Table 4-1 in that the overall mean scores for each unique measure are provided in place of data collection frequency. To illustrate, *TotSolved*, described as *The total number of exercises solved*, and *Raw Variable 21*, had a mean value of *29.4*. Raw variables are indicated with an 'R' in the *Raw/Calc* column, and calculated variables are indicated with a 'C'. Labels, like *TotSolved*, are provided to simplify references to the performance measures throughout the chapter.

Summary of Questionnaire Results

As seen in Table 5-1 for calculated variables 5-8, the four sub-components of self-regulation, students rated themselves with mean scores of 2.7 for planning, 2.8 for self-checking, 3.2 for effort, and 3.2 for self-efficacy on a 4-point scale. Similarly, for calculated variable 1, *MathScore*, students rated themselves with a mean score of 8.0 on a 13-point scale. Because the math questionnaire spanned experiences ranging from low-level math to advanced placement statistics, it can be seen that students with a wide range of experience levels were represented. For calculated variable 2, *CompScore*, students rated themselves with a mean score of 6.5 on a similar 13-point scale. In summary, the students in the pilot study were representative of a wide range of abilities in self-regulation, math, and computer experience.

Table 5-1

Summary of Raw and Calculated Performance Measures with Mean Scores

Num	Performance Measure Description	Label	Raw/ Calc	Mean
	The following measures are from questionnaires:			
1	Math Experience	MathScore	C	8.0
2	Computer Experience	CompScore	C	6.5
3	Trait Self-Regulation	N/A	R	
4	Trait Self-Regulation w/Programming	N/A	R	
	State Self-Regulation (4 separate questionnaires)		R	
5	- Section 2.7 end: planning sub-component	Planning	C	2.7
6	- Section 2.7 end: self-check	SelfChk	C	2.8
7	- Section 2.7 end: effort	Effort	C	3.2
8	- Section 2.7 end: self-efficacy	SelfEff	C	3.2
9	- Exam end: planning sub-component	Planning	C	2.3
10	- Exam end: self-checking	SelfChk	C	2.3
11	- Exam end: effort	Effort	C	2.8
12	- Exam end: self-efficacy	SelfEff	C	2.3
13	- Section 3.7 end: planning sub-component	Planning	C	1.5
14	- Section 3.7 end: self-checking	SelfChk	C	1.5
15	- Section 3.7 end: effort	Effort	C	1.7
16	- Section 3.7 end: self-efficacy	SelfEff	C	1.7
17	- Section 4.4 end: planning sub-component	Planning	C	0.7
18	- Section 4.4 end: self-checking	SelfChk	C	0.7
19	- Section 4.4 end: effort	Effort	C	0.8
20	- Section 4.4 end: self-efficacy	SelfEff	C	0.8

Table 5-1 (Continued)

Num	Performance Measure Description	Label	Raw/ Calc	Mean
	The following measures are from exercises solved:			
21	Total number of exercises solved	TotSolved	R	29.4
22	Percent of total number of exercises solved	PctSolved	C	0.7
23	Total average score for solved exercises	TotAvgSc	C	9.2
24	Number of optional exercises solved	OptSolved	R	1.4
25	Average score for optional exercises	OptAvgSc	C	7.2
26	Number of easy exercises solved	ESolved	R	5.0
27	Average score for easy exercises solved	EAvgSC	C	9.6
28	Number of medium exercises solved	MSolved	R	4.9
29	Average score for medium exercises solved	MAvgSC	C	9.6
30	Number of hard exercises solved	HSolved	R	4.9
31	Average score for hard exercises solved	HAvgSC	C	9.4
	The following measures are from submitted attempts:			
32	Total number of submitted attempts	TotAtts	R	356.5
33	Total # of exercises solved	TotSolved	R	29.4
34	Average number of submitted attempts	TotAvgAtt	C	11.9
35	Number of submitted attempts for optional exercises	OptAtts	R	28.9
36	Number of optional exercises solved	OptSolved	R	1.4
37	Average number of attempts for optional exercises	OptAvgAtt	C	18.6
38	Number of submitted attempts for easy exercises	EAttempts	R	17.5
39	Number of easy exercises solved	ESolved	R	5.0
40	Average Number of attempts for easy exercises	EAvgAttempts	C	3.5

Table 5-1 (Continued)

Num	Performance Measure Description	Label	Raw/ Calc	Mean
41	Number of submitted attempts for medium exercises	MAttempts	R	42.7
42	Number of medium exercises solved	MSolved	R	4.9
43	Average number of attempts for medium exercises	MAvgAttempts	C	8.7
44	Number of submitted attempts for medium exercises	HAttempts	R	67.9
45	Number of medium exercises solved	HSolved	R	4.9
46	Average number of attempts for hard exercises	HAvgAttempts	C	13.9
	The following measures are from in-text examples:			
47	Total number of unique in-text examples visited	TotUniqInTxt	R	22.5
48	Percent of in-text examples visited	PctUniqInTxt	C	0.4
49	Total visits to in-text examples	TotInTxt	R	35.7
50	Average number of visits to unique in-text examples	AvgPerUniqInTxt	C	1.5
	The following measures are from in-exercise examples:			
51	Total number of unique in-exercise examples	TotUniqInEx	R	4.3
52	Percent of in-exercise examples visited	PctUniqInEx	C	0.1
53	Total visits to in-exercise examples	TotInEx	R	11.6
54	Average number of visits to in-exercise examples	AvgPerUniqInEx	C	2.2
	The following measures are from hints:			
55	Sum of hint levels used in all exercises	HintSum	C	15.7
56	Number of exercises where hints were used	HintProbs	R	6.6
57	Average hint level where hints were used	AvgHintLev	C	1.8

Table 5-1 (Continued)

Num	Performance Measure Description	Label	Raw/ Calc	Mean
58	Sum of hint levels used in optional exercises	OptHintSum	C	0.8
59	Number of optional exercises where hints were used	OptProbs	R	0.3
60	Average hint level in opt exercises where used	AvgOptHintLev	C	0.7
61	Number of easy exercises where hints were used	EHintProbs	R	0.4
62	Sum of hint levels in easy exercises	EHintSum	C	0.8
63	Average hint level in easy exercises where used	EAvgHintLev	C	0.5
64	Number of medium exercises where hints were used	MHintProbs	R	0.9
65	Sum of hint levels in medium exercises	MHintSum	C	2.0
66	Average hint level in medium exercises where used	MAvgHintLev	C	0.8
67	Number of hard exercises where hints were used	HHintProbs	R	1.3
68	Sum of hint levels in hard exercises	HHintSum	C	2.9
69	Average hint level in hard exercises where used	HAvgHintLev	C	1.2
	The following measures are from the exam:			
70	Exam score	ExamScore	R	36.8
71	Exam percentage	ExamPct	C	0.6

Research Question Findings

Question-1: How do students with low versus high self-regulatory skills perform in the use of in-text worked examples, in-exercise worked examples, hints, optional exercises, and problem-solving scores?

Table 5-2 shows mean scores for 15 performance measures broken down according to the four components of self-regulation. Self-regulation was divided into two major categories: *metacognition* and *motivation*. Metacognition was further sub-divided into the two sub-components of *planning* and *self-checking*. Motivation was also further sub-divided into the two sub-components of *effort* and *self-efficacy*. Each self-regulation sub-component was divided into low and high score groups. Low group self-regulation scores ranged from 1.0 to 2.5, and high group self-regulation scores ranged from 2.6 to 4.0.

Fifteen performance measures were then used to gauge relative performance between the low and high score groups. These were the use of in-text worked examples [AvgPerUniqInTxt (50), TotInTxt (49), and TotUniqInTxt (47)], the use of in-exercise worked examples [AvgPerUniqInEx (54), TotInEx (53), and TotUniqInEx (51)], hint usage [AvgHintLevel (57) and HintProbs (56)], completion of optional exercises [OptAvgAtt (37), OptAvgSc (25), and OptSolved (36)], and problem solving performance [TotAvgAtt (34), TotAvgSc (23), TotSolved (21), and ExamScore (70)]. The same 15 measures were also applied to similar tables that address questions 2, 3, and 4. Comparative differences among performance measures were determined first by grouping and computing the self-regulation sub-components into two categories, low and high. Individual performance measures were then grouped according to self-regulation score groups, and the means were calculated.

Overall Findings Related to Self-Regulation

When comparing performance measures across score groups, relative performance varied among low and high score sub-groups in each of the performance measures as shown in Table 5-2. In summary, students in the high motivation score groups (effort

and self-efficacy) outperformed students in the low motivation score groups in *all* performance measures including their reliance on hints (less hints preferred), the number of attempts required to complete required and optional exercises, and exercise and exam scores. Those in the high planning score group performed at least as well or higher than students in the low planning score group in *most* objective measures, with the exception of the average hint level used and the total number of exercises completed. There was no clear performance distinction among groups with low and high self-checking scores.

Self-Regulation and Worked Examples. Beginning with the use of in-text and in-exercise worked examples, students in the high planning score group performed better by visiting fewer worked examples per exercise in total and on the average [TotInTxt (49), TotUniqInTxt (47), TotInEx (53), TotUniqInEx (51)]. High self-checking and effort score groups also required less dependence on the use of worked examples. The exception was the high self-efficacy score group, who visited more in-text and fewer in-exercise worked examples than the low score group.

Self-Regulation and the use of Hints. The second set of performance measures involved the use of hints, or AvgHintLevel (57) and HintProbs (56). Students in the low metacognition score groups used fewer hints on the average, but those in the low motivation score groups used more hints. In addition, for all four sub-components of self-regulation, students in the high score groups relied on hints in fewer exercises than the low score groups. These findings indicate that students who worked harder and believed they were capable of successfully completing the exercises relied less heavily on hints. Another finding is that students in high meta-cognition and motivation score groups depended less on the use of hints than those in the low score groups.

Self-regulation and Completion of Optional Exercises. Students in the high motivation score groups dramatically outperformed the lower score group in all three performance measures in optional exercise completion [OptAvgAtt (37), OptAvgSc (25), and OptSolved (36)]. They completed more and scored higher on all optional exercises, which explains the larger average number of attempts per exercise. These findings show that motivated students attempted and completed non-required work even through extra time and submit attempts were required above and beyond expectation.

Self-regulation and Problem Solving Performance. Effort was the only sub-component of self-regulation in which students in the high score group completed more exercises [TotSolved (21)] than students in the low score group. While students in the high motivation score groups required more attempts [TotAttempts (34)] to complete more exercises than the low score sub-groups, high metacognition score groups submitted fewer attempts for fewer completed exercises. In addition, students in the high score groups for all self-regulation sub-components outscored low score group students on the exam [ExamScore (70)].

Table 5-2

Mean Performance Measures among Low and High Group

Sub-Components of Self-Regulation

	Metacognition				Motivation			
	Planning		Self-Checking		Effort		Self-Efficacy	
	Low	High	Low	High	Low	High	Low	High
Performance Measure (Table-1 Variable)	0.0-2.5	2.6-4.0	0.0-2.5	2.6-4.0	0.0-2.5	2.6-4.0	0.0-2.5	2.6-4.0
AvgHintLevel (57)	1.8	2.0	1.8	1.9	2.3	1.8	2.2	1.2
HintProbs (56)	8.4	6.4	10.8	5.9	7.5	7.3	15.0	3.8
AvgPerUniqInEx (54)	2.4	2.1	2.4	2.1	2.1	2.2	2.6	2.6
TotInEx (53)	16.0	9.9	16.4	11.1	10.8	13.0	17.0	14.3
TotUniqInEx (51)	5.8	3.4	5.2	4.2	4.8	4.4	6.0	5.5
AvgPerUniqInTxt (50)	1.5	1.5	1.6	1.5	1.3	1.5	1.7	1.6
TotInTxt (49)	42.6	33.2	37.4	37.5	36.0	37.7	43.5	51.5
TotUniqInTxt (47)	27.5	21.2	23.7	24.2	24.3	24.0	0.4	0.5
OptAvgAtt (37)	15.0	16.3	17.9	14.8	12.8	16.1	5.6	29.5
OptAvgSc (25)	7.7	6.6	8.5	6.5	5.0	7.4	6.3	8.7
OptSolved (36)	1.4	1.6	1.2	1.6	0.5	1.6	1.3	2.2
TotAvgAtt (34)	12.4	11.2	12.6	11.3	8.6	12.2	11.3	14.4
TotAvgSc (23)	9.1	9.1	9.3	9.0	9.2	9.1	9.0	9.5
TotSolved (21)	31.4	29.9	31.9	30.0	26.8	31.1	35.0	33.0
ExamScore (70)	39.4	40.7	38.8	41.6	35.9	41.5	39.6	40.5

Question-2: How do students with low versus high math experience perform in the use of in-text worked examples, in-exercise worked examples, hints, optional exercises, and average number of attempts to solve problems, and problem-solving scores?

Table 5-3 shows mean scores for the same 15 performance measures used in Question-1. Comparative differences among performance measures were determined first by grouping and computing math experience scores into two categories: *low* and *high*, with low group math experience measures ranging from 0.0 to 7.4, and high group math experience measures ranging from 7.5 to 13.0. The low score groups included questionnaire scores in the *poor* to *low* range, and the high score group included scores in the *good* to *high* range (see description of *MathScore* in Chapter-3).

Overall Math Experience Findings

Actual measures of math experience ranged from 3.5 to 13, and 25 of the 36 students rated themselves in the *good* to *high* range with scores of 7.5 and above – this accounted for 70 percent of the participants. When comparing performance measures across score groups, relative performance varied among low and high score sub-groups in each of the descriptive measures as shown in Table 5-3. In summary, students in the high math experience score group outperformed the low score group in *all* performance measures. The high score group depended less on hints, required fewer attempts to complete required and optional exercises, and achieved higher exercise and exam scores.

Math Experience and Worked Examples. Beginning with the use of worked examples, students in the high score group visited in-text worked examples at least as many times as the low score group, but relied dramatically less on in-exercise worked examples

[AvgPerUniqInTxt (50), TotInTxt (49), TotUniqInTxt (47), AvgPerUniqInEx (54), TotInEx (53), TotUniqInEx (51)].

Math Experience and the use of Hints. While the average hint level used in both low and high score groups was close, students in the high score group required the use of hints in dramatically fewer exercises. Once again, math experience helped students in learning computer programming concepts – the higher score group required less help.

Math Experience and Completion of Optional Exercises. Similar to self-regulation, students in the high score group dramatically outperformed students in the low score group in all three performance measures in optional exercise completion [OptAvgAtt (37), OptAvgSc (25), and OptSolved (36)]. They completed more and scored higher on all optional exercises, which explains the larger average number of attempts per exercise. These findings show that math experienced students will attempt and complete non-required work at the cost of more effort.

Math Experience and Problem Solving Performance. As already explained in the above section on worked examples, students in the low score group only slightly underperformed in the number of exercises completed [TotSolved (21)], the average exercise score [TotAvgSc (23)], and the exam score [ExamScore (70)]. These findings are indicative that students with less math experience can succeed in learning introductory programming.

Table 5-3

Mean Performance Measures among Low and High Math Experience Groups

Performance Measure (Table 5-1 variable)	Low 0-7.4	High 7.5-13
AvgHintLevel (57)	2.0	1.7
HintProbs (56)	10.6	5.1
AvgPerUniqInEx (54)	2.7	2.1
TotInEx (53)	19.9	10.0
TotUniqInEx (51)	6.3	3.8
AvgPerUniqInTxt (50)	1.6	1.5
TotInTxt (49)	34.6	36.6
TotUniqInTxt (47)	20.6	23.3
OptAvgAtt (37)	13.5	20.8
OptAvgSc (25)	5.5	7.6
OptSolved (36)	0.6	1.6
TotAvgAtt (34)	14.7	11.2
TotAvgSc (23)	9.0	9.3
TotSolved (21)	27.3	29.9
ExamScore (70)	34.9	37.2

Question-3: How do students with low versus high computer experience perform in the use of in-text worked examples, in-exercise worked examples, hints, optional exercises, and average number of attempts to solve problems, and problem-solving scores?

Table 5-4 shows mean scores for the same 15 performance measures used in Questions 1 and 2. Comparative differences among performance measures were determined first by grouping and computing computer experience scores into two categories: *low* and *high*, with low group computer experience measures ranging from 0.0 to 7.4, and high group computer experience measures ranging from 7.5 to 13.0. The low score groups included questionnaire scores in the *poor* to *low* range, and the high score group included scores in the *good* to *high* range (see description of *CompScore* in Chapter-3).

Overall Computer Experience Findings. Actual measures of computer experience ranged from 3.5 to 12, and 24 of the 36 students rated themselves in the *low* to *poor* range with scores of 7.0 and below – this accounted for 67 percent of the participants. Relative performance varied among low and high score sub-groups in each of the performance measures as shown in Table 5-4. In general, students in the high score group outperformed students in the low score group in *most* objective performance measures including less dependence on hints, fewer number of attempts required to complete required and optional exercises, and higher exercise and exam scores.

Computer Experience and Worked Examples. Beginning with the use of worked examples, students in the high score group visited in-text worked examples slightly less than those in the low score group, but relied somewhat more on in-exercise worked

examples [AvgPerUniqInTxt (50), TotInTxt (49), TotUniqInTxt (47), AvgPerUniqInEx (54), TotInEx (53), TotUniqInEx (51)].

Computer Experience and the use of Hints. Students in the high score group required the use of hints in fewer exercises and used a lower hint level on the average than students in the low score group. Similar to math experience, students in the low score group relied more heavily on hints to complete the exercises. Once again, hints appear to be helpful aid for students with average to low math and computer skills.

Computer Experience and Completion of Optional Exercises. Similar to math experience, students in high score group outperformed those in the low score group in all three performance measures in optional exercise completion [OptAvgAtt (37), OptAvgSc (25), and OptSolved (36)]. Although high score group students completed more and scored higher, the differences aren't as dramatic as those in the low score group. The differences are small enough to observe that students in the high score group were not necessarily inclined to expend dramatically more effort toward solving optional exercises.

Computer Experience and Problem Solving Performance. Similar to students in the low math experience score group, students in the low computer experience score group only slightly underperformed in the mean number of attempts required per exercise [TotAvgAtt (34)] and the average exercise score [TotAvgSc (23)]. They underperformed slightly more on the total number of exercises completed [TotSolved (21)] and the exam score [ExamScore (70)].

Table 5-4

Mean Performance Measures among Low and High Computer Experience Groups

Performance Measure (Table 5-1 variable)	Low 0-7.4	High 7.5-13
AvgHintLevel (57)	1.9	1.6
HintProbs (56)	7.5	6.1
AvgPerUniqInEx (54)	2.1	2.3
TotInEx (53)	12.1	12.4
TotUniqInEx (51)	3.6	4.7
AvgPerUniqInTxt (50)	1.6	1.5
TotInTxt (49)	37.3	35.7
TotUniqInTxt (47)	23.0	22.6
OptAvgAtt (37)	17.9	19.4
OptAvgSc (25)	6.8	7.2
OptSolved (36)	1.1	1.4
TotAvgAtt (34)	11.8	12.1
TotAvgSc (23)	8.9	9.3
TotSolved (21)	26.9	30.0
ExamScore (70)	33.0	37.8

Question-4: How do students with low versus high self-regulatory skills perform in the use of in-text worked examples, in-exercise worked examples, hints, optional exercises, average number of attempts to solve problems, and problem-solving scores as the task difficulty increases?

Table 5-5 shows mean scores for 5 performance measures applied for each of three groups of exercises that increased in difficulty from *easy*, to *medium*, and then to *hard*; for a total of 15 measures. These were the total and average number of submitted attempts for easy, medium, and hard exercises [EAttempts(38), MAttempts(41), HAttempts(44), EavgAtts(40), MavgAtts(43), and HavgAtts(46)]; the total number of problems and average hint level where hints were used for easy, medium, and hard exercises [EhintProbs(61), MhintProbs(64), HhintProbs(67), EAvgHintLev(63), MAvgHintLev(66), HAvgHintLev(69)]; and the mean score achieved on exercises [EAvgSc(27), MAvgSc(29), HAvgSc(31)].

Similar to Question-1, comparative differences among performance measures were determined first by grouping and computing the self-regulation sub-components into two categories, low and high. Individual performance measures were then grouped according to self-regulation score groups, and the means were calculated. Analysis was conducted by comparing each of the five performance measures across easy, medium, and hard level exercises for each self-regulation score group. Comparisons were percentage differences (marginal differences) as the exercise difficulty increased. Increasing margins from easy to medium to hard indicated positive trends, and decreasing margins indicated negative trends.

Table 5-5

Mean Performance Measures among Low and High Group Sub-Components of Self-Regulation as Task Difficulty Increased

	Meta-cognition				Motivation			
	Planning		Self-Checking		Effort		Self-Efficacy	
Performance Measure	Low	High	Low	High	Low	High	Low	High
(Table-1 Variable)	0.0-2.5	2.6-4.0	0.0-2.5	2.6-4.0	0.0-2.5	2.6-4.0	0.0-2.5	2.6-4.0
EAttempts (38)	18.5	16.6	18.4	16.8	18.0	17.2	19.5	17.0
Mattempts (41)	49.6	40.1	48.9	41.3	44.8	43.3	56.0	41.6
HAttempts (44)	88.5	59.5	9.7	9.5	53.5	72	110.5	63.6
EavgAtts (40)	3.7	3.3	3.7	3.4	3.6	3.4	3.9	3.4
MavgAtts (43)	9.9	8.4	9.8	8.6	9.0	8.9	11.2	8.6
HavgAtts (46)	17.7	12.1	86.2	62.8	10.7	14.6	22.1	12.9
EhintProbs (61)	0.5	0.5	0.6	0.5	1.0	0.4	0.5	0.5
MhintProbs (64)	1.0	1.0	0.9	1.0	1.0	1.0	1.5	0.9
HhintProbs (67)	1.4	1.4	17.2	12.8	1.0	1.4	2.8	1.2
EavgHintLev (63)	0.8	0.4	1.0	0.3	1.8	0.4	1.0	0.5
MAvgHintLev (66)	0.7	1.0	1.6	2.6	0.5	1.0	1.0	0.9
HavgHintLev (69)	1.1	1.1	1.6	1.3	1.3	1.1	1.5	1.0
EAvgSc (27)	9.6	9.6	9.5	9.6	9.4	9.6	9.6	9.6
MAvgSc (29)	9.6	9.5	0.8	1.0	9.7	9.6	9.5	9.6
HAvgSc (31)	9.4	9.3	1.0	1.1	9.8	9.3	9.7	9.3

In Table 5-6, the percentage differences among easy, medium, and hard level performance measures are presented. Identifiable trends are indicated by groups of three numbers (easy, medium, and hard) highlighted in shades of grey. Example 4-1 below

explains Table 5-6 by using the low and high *planning* score groups across increasing difficulty of the *EAttempts*, *MAttempts*, and *HAttempts* performance measures:

Example 4-1. The low planning score group required 18.5 average attempts to complete easy exercises compared to 16.6 attempts for the high planning group. The percentage difference was computed as [-10.3%], using low planning as the reference [(18.5-16.6)/18.5]. Similar differences were computed for medium and hard level exercises, resulting in [–19.2] and [–32.8] percent differences for medium [(49.6-40.1)/49.6] and hard [(88.5-59.5)/88.5] levels, respectively. The trend is seen by observing percentage differences among the three percentages: -10.3% for easy; 19.2% for medium; and –32.8% for hard level exercises.

This trend can be interpreted as follows: *Students in the high planning score group required increasingly fewer attempts to complete exercises than those in the low planning score group as the task difficulty increased: 10.3% fewer attempts for easy, 19.2% fewer attempts for medium, and 32.8% fewer attempts for hard level exercises.*

Figures 4-1 through 4-4 illustrate these trends for the five performance measures (Attempts, AvgAttempts, HintProbs, AvgHintLevel, and AvgSc) plotted against increasing task difficulty (easy, medium, and hard). Each figure plots all trends for one of the four sub-components of self-regulation including planning, self-checking, effort, and self-efficacy. The plotted values are the percentage gains or losses shown in Table 5-6.

Table 5-6

Percentage Difference in Performance Measures among Sub-Components of Self-Regulation as Task Difficulty Increased

	Meta-cognition		Motivation	
	Planning	Self-Checking	Effort	Self-Efficacy
Performance Measure (Table-1 Variable)	Percentage Difference	Percentage Difference	Percentage Difference	Percentage Difference
EAttempts (38)	-10.3	-8.6	-4.4	-12.8
Mattempts (41)	-19.2	-15.5	-3.3	-25.7
HAttempts (44)	-32.8	-2.1	-34.6	42.4
EavgAtts (40)	-10.8	-8.1	-5.6	-12.8
MavgAtts (43)	-15.2	-12.2	-1.1	-23.2
HavgAtts (46)	-31.6	-27.1	36.4	-41.6
EhintProbs (61)	0.0	-16.7	-60.0	0.0
MhintProbs (64)	0.0	11.1	0.0	-40.0
HhintProbs (67)	0.0	-25.6	40.0	-57.1
EavgHintLev (63)	-50.0	-70	-77.8	50.0
MAvgHintLev (66)	42.9	62.5	100.0	-10.0
HavgHintLev (69)	0.0	-18.8	-15.4	-33.3
EAvgSc (27)	0.0	1.1	2.1	0.0
MAvgSc (29)	-1.0	25.0	-1.0	1.1
HAvgSc (31)	-1.1	-10.0	-5.1	-4.1

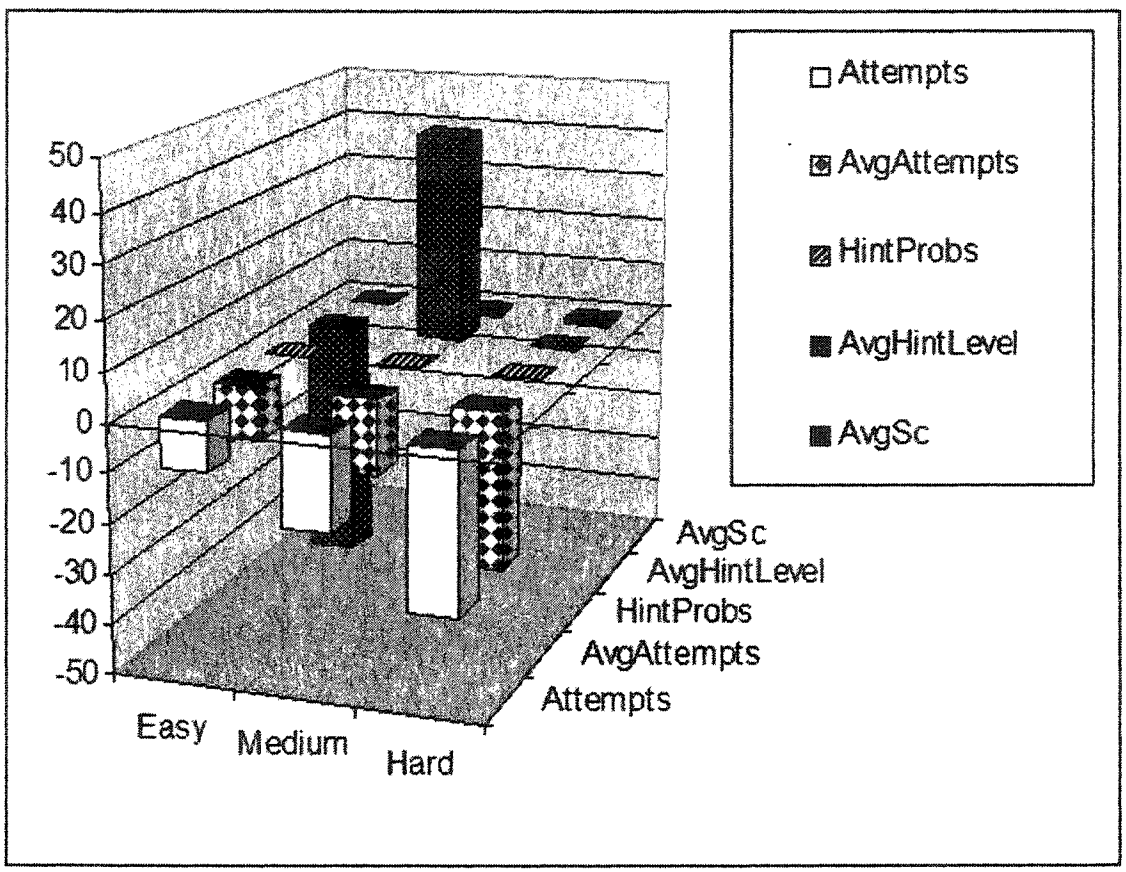


Figure 5-1. Percentage Differences in Five Performance Measures as theTask Difficulty Increased for the Planning Sub-component of Self-Regulation

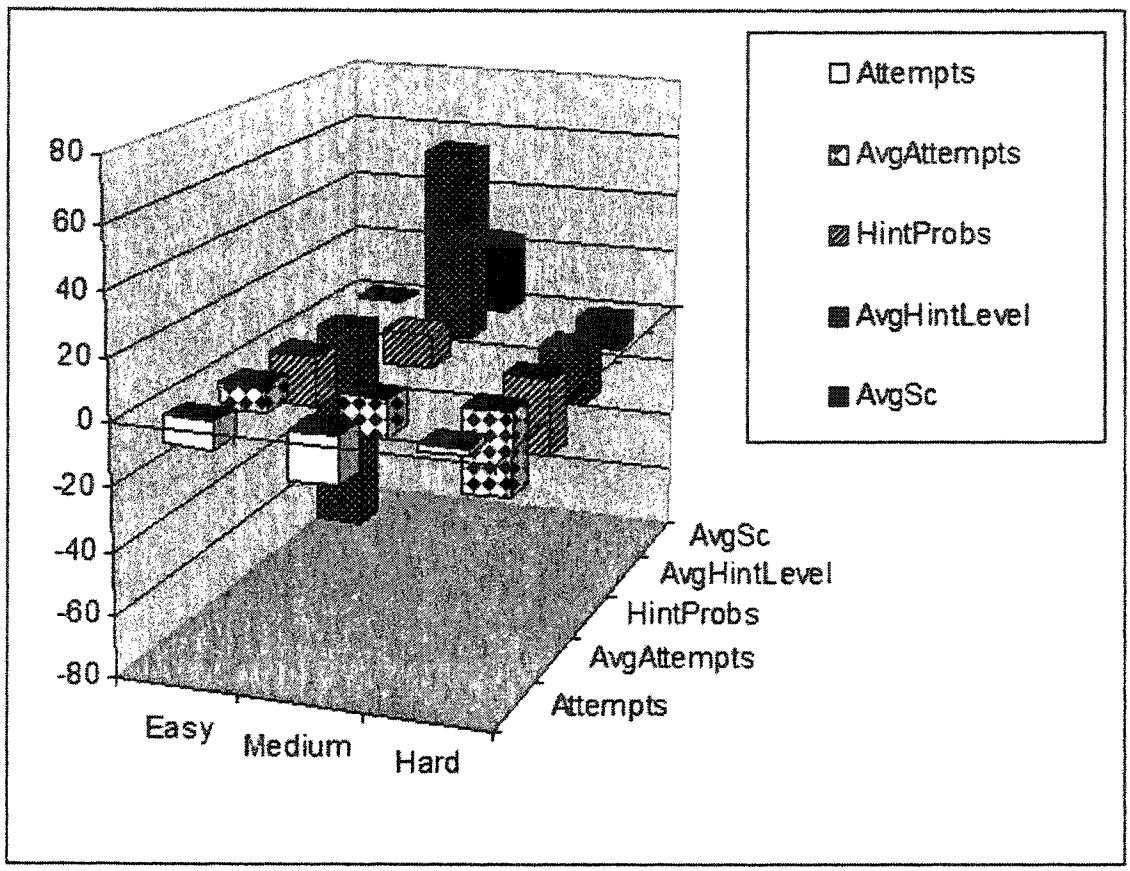


Figure 5-2. Percentage Differences in Five Performance Measures as the Task Difficulty Increased for the Self-Checking Sub-component of Self-Regulation

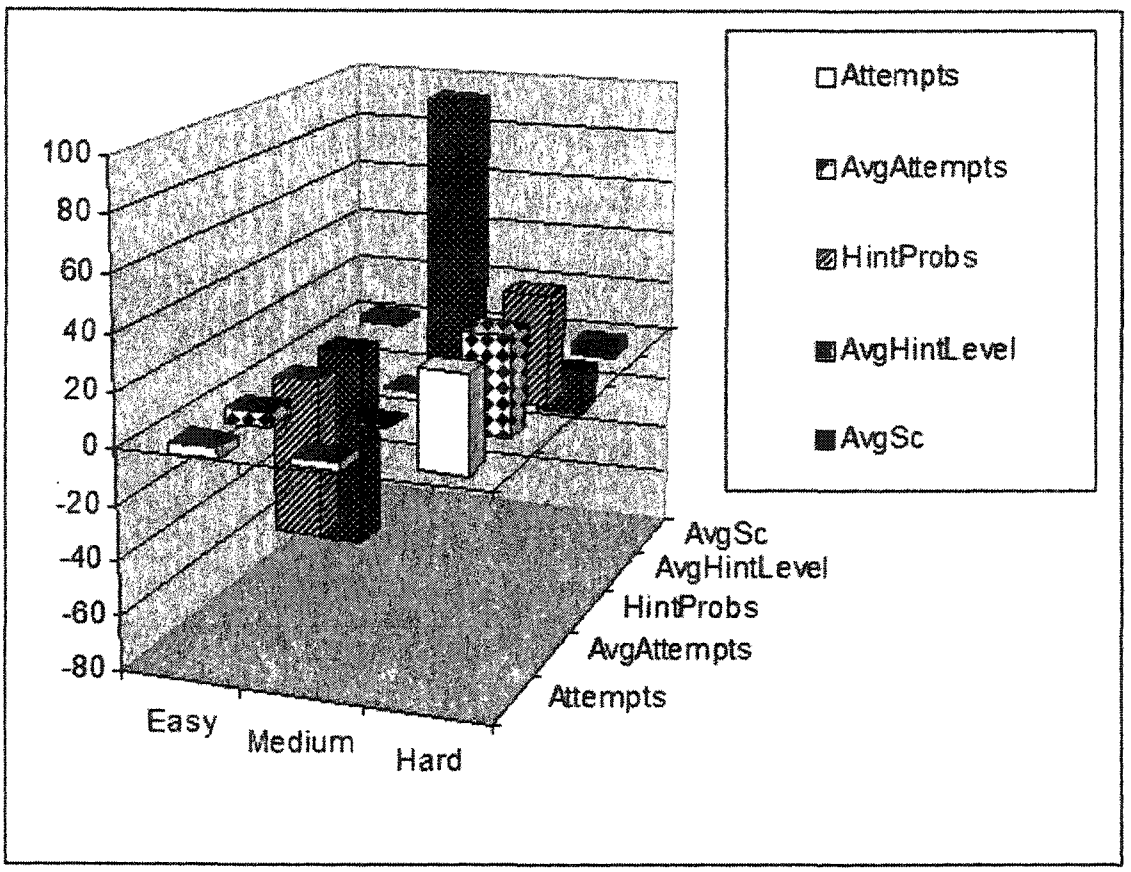


Figure 5-3. Percentage Differences in Five Performance Measures as the Task Difficulty Increased for the Effort Component of Self-Regulation

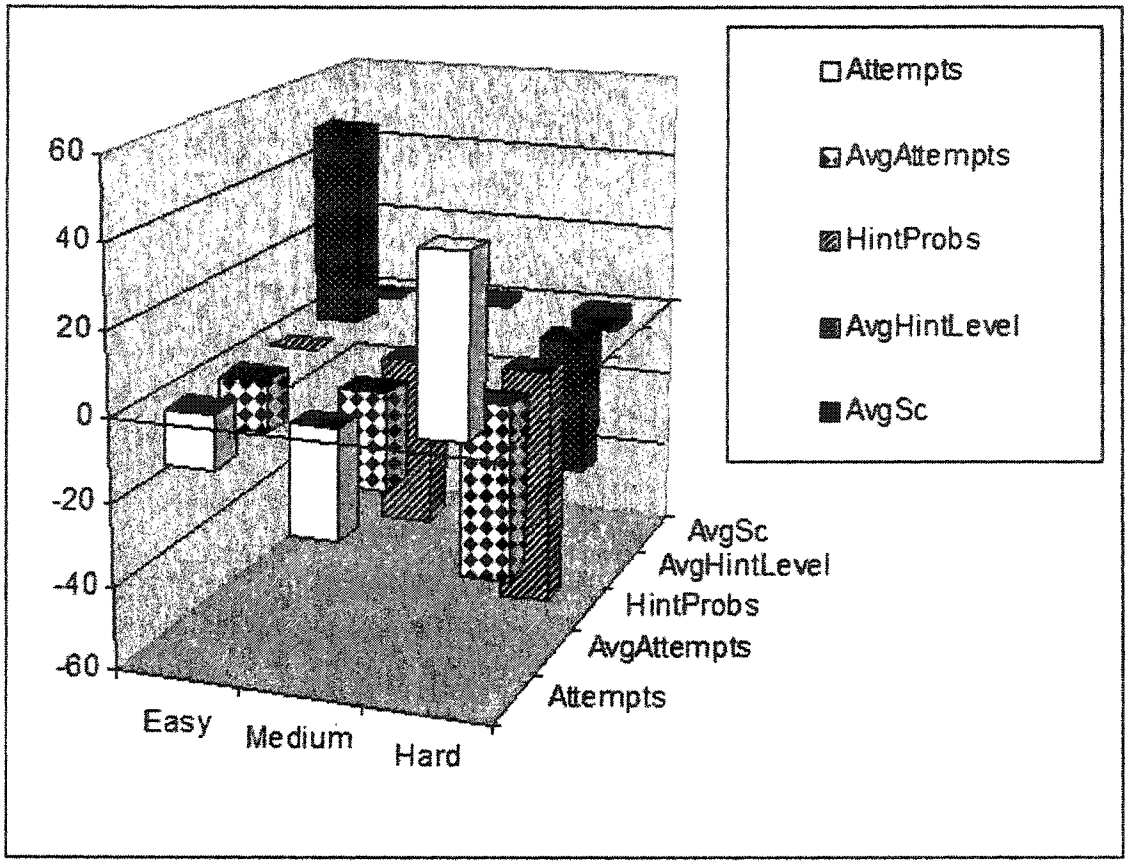


Figure 5-4. Percentage Differences in Five Performance Measures as the Task Difficulty Increased for the Self-Efficacy Component of Self-Regulation

Overall Findings Related to Self-Regulation and Increasing Task Difficulty

A total of ten identifiable trends emerged from the analysis depicted in Table 5-6. Seven of these appeared in the effort and self-efficacy sub-components of self-regulation. Two of the remaining three appeared in planning, and the last appeared in self-checking. It appears that effort had the greatest impact on performance factors as the task difficulty increased.

Attempts to Complete Exercises and Increasing Task Difficulty. This was the performance factor most greatly impacted by self-regulation and increasing task

difficulty. In all four sub-components of self-regulation, trends in MavgAtts emerged, and trends in Attempts emerged in two of the components. With the exception of effort, students in the high score groups for planning, self-checking, and self-efficacy increasingly submitted fewer attempts to complete exercises as the task difficulty increased. Conversely, those in the high effort score group increasingly submitted more attempts as the task difficulty increased. Students in the high planning score group increasingly submitted fewer total attempts as the task difficulty increased, but those in the high effort score group submitted more total attempts. These findings show that students in the high effort score group increasingly tried harder as the task difficulty increased.

The Use of Hints and Increasing Task Difficulty. Two identifiable trends emerged from the use of hints as the task difficulty increased. The first trend showed that students in the high effort score group tended to depend on hints more than the lower score group as the task difficulty increased. The second trend showed an opposite effect for students in the high self-efficacy score group. These students relied less on the use of hints as the task difficulty increased in both the number of exercises where hints were used, and the average hint level reached in each of those exercises.

Exercise Score and Increasing Task Difficulty. One identifiable trend emerged from the average score on exercises (AvgSc). Students in the high effort score group scored relatively lower on exercises as the task difficulty increased. Interestingly, while this same group increasingly submitted more attempts and depended more on hints, they also score lower.

Question-5: What common mistakes do students make in solving programming problems?

To answer the question, insight into what students do between corrections to programming errors and subsequent runs is needed. In a classroom setting where students are writing programs using a traditional development environment, changes to source code might not be tracked by the instructor or the development software, resulting in potentially hundreds of unknown changes and runs within the scope of a single class period. Working within the online system environment, an ethnographic record of every exercise run was captured by the system for every student and every exercise, thus empowering the present investigator to apply the verbatim principle of qualitative research (Spradley, 1980).

Data from ten exercises ranging from medium to hard difficulty were selected to observe changes made to source code between subsequent run attempts. Medium to hard level problems were selected because of the wider range of attempts to complete each exercise. The selected exercises and descriptive statistical information are shown in Table 5-7. This section provides an overview of findings in the order of how the domain analysis was conducted and for each of seven major identified domains.

Domain Analysis

The first step in the analysis process was to identify major domains, or categories of common changes made to source code between successive runs (each run constituted submitted source code into a database). This was done by observing source code from contiguous runs and identifying the major changes that occurred between the runs. The second step involved isolating the comparison to a sub-domain of the major domain if necessary. Four of the major domains were divided into sub-domains based on observed

data. The third step involved identifying the dominant change – or the most likely reason for the change. Based on these findings, the fourth step was to identify common problems that students made in the process of programming.

First Major Domain: No Change

The most common domain was *No Change*, where the source code between adjacent runs was identical. This domain accounted for 23% of all program runs, and could be explained with further analysis of inherent sub-domains. While it's difficult to know exactly why no change occurred, inferences could be made based on time intervals between successive runs. Since the system clock resolution was limited to one second, successive runs with an identical time stamp might infer an erroneous double or triple-click of the mouse on the single-click submit button. These accounted for 6% of all *No Change* runs. Another inference made from time stamp observation was a relatively short span of 3 to 5 seconds, which might imply the program generated no output, or appeared to not run, and the student ran it again just to be sure. These accounted for 11% of all *No Change* runs. Most other *No Change* runs were mainly unknowable and accounted for 83% of all those runs. Since students were enabled to re-visit and run previously solved exercises at any time, many of these might be explained by student interest in seeing or showing their work *in action*. The only attributable error in this domain would be unfamiliarity with the programming environment, thus leading to unnecessary multiple clicks of the mouse.

Second Major Domain: Syntax Errors

The second most common domain was *Syntax Errors*, which accounted for 18% of all program runs. Because of the complexity of programming languages and the myriad of syntax rules (and therefore potential errors), sub-domains were identified to simplify the list to the most common. Syntax errors mainly resulted from a misapplication of the rules while structuring a program statement. While it was shown to be difficult to know why the rules were misapplied, misuse of parenthesis, function parameters, quotes, and curly brackets together accounted for 31% of all syntax errors. The other, more detailed errors accounted for the remaining 69%. Like good writing, good programming will result from knowing the rules and plenty of practice. In the case of syntax errors, it's likely that these will always be the most common type of error – at least for beginning programmers.

Third Major Domain: Clean Up

The third most common domain was *Clean Up*, or the process of modifying code to be more readable or self-documenting, which accounted for 17% of all observed program runs. Like *No Change*, *Clean Up* does not qualify as a programming error, but does indicate that students are interested in how their code looks and reads. Removal of test code also falls into this domain.

Fourth Major Domain: Logic Change

The fourth major domain was *Logic Change*, which accounted for 14% of all observed program runs. *Logic Change* was defined as a correction to a logic error or bug in the program. Most programs with logic errors will run, but will also produce erroneous output or results. The most common logic errors observed involved changing loop

counter variable bounds, which were incorrectly initialized. Most other logic errors are very difficult to ascertain, mainly because of the level of analysis required to observe a program that runs but doesn't produce expected output. Up to this point, syntax and logic errors combined accounted for 32% of modified programming errors.

Fifth Major Domain: Build Upon

The fifth major domain, which accounted for 11% of all runs, was *Build Upon*. Similar to *No Change* and *Clean Up*, *Build Upon* did not contribute to programming errors, but to improvements or enhancements in the source code over previous runs. While errors might have been present in the augmented code, the dominant intent was to build upon the previous code.

Sixth Major Domain: Sudden Change

The sixth major domain was *Sudden Change*, which accounted for 8% of all program runs. *Sudden Change* is interesting in that it doesn't reflect any type of programming error, but rather a paradigmatic shift in the problem solving process. Because of the nature of such a change, further analysis was conducted to try to determine the reason for the change – resulting in the sub-domains of plagiarism, application of hints or examples, return to previous code, or other.

The use of hints or worked examples, which accounted for 19% or all sudden changes, was obvious because of familiarity with the example and hint source code. In most cases, students reached a dead-end, gave up, and resorted to seeking help. Hints were the only form of available help until all three levels had been exhausted.

Plagiarism, or 17% or all sudden changes, was determined based on a radical change in source code with no reference to hints, examples, or writing style established by

previous work. Considering the online environment in which students worked, plagiarism accounted for only 1.36% of source code changes overall. This statistic was encouraging considering the ease with which source text could be E-Mailed or distributed through a shared server.

Another promising statistic was the rate at which students returned to source code from a previous attempt, or 15% of sudden changes. In other words, students were just about as willing to go back and start over as they were willing to plagiarize, or 1.2% of the time. The majority of Sudden Change observations were attributed to students writing code offline, the copying and pasting that code into the online system. These accounted for 49% of all sudden change runs, and were determined by long time intervals between runs. This type of code writing was sometimes encouraged if Internet connectivity was unreliable.

Seventh Major Domain: Grammar

The seventh major domain was *Grammar*, or 5% of all runs. It was decided to separate grammar from syntax since a misspelled variable name or case sensitivity mistake was not related to erroneous statement structure. Seventy-eight percent of all grammar errors resulted from misspelled variable names, 17% were misspelled reserved words, and 5% were attributed to errors in case sensitivity. Because variable names can be quite long, and capitalization of words within the name was required, errors of this nature were likely.

Table 5-7

Frequencies of Exercises selected for Qualitative Analysis of Common Mistakes

Exercise	Difficulty	Total Attempts	Average Attempts
2-7-1	H	436	12.5
3-3-2	H	315	12.6
3-3-3	M	144	6.9
3-3-4	H	257	13.5
3-3-5	H	202	11.2
3-5-1	H	386	13.8
3-5-2	H	993	36.8
3-5-3	H	726	26.9
3-7-1	M	203	9.7
3-7-2	H	258	12.3

A domain analysis was performed by visually comparing source code from all pairs of submitted attempts. This translated to 1602 comparisons of source code submitted by 36 students over the course of 10 weeks. As with any analysis, some of the domains were apparent while others were tacit, thus requiring the use of inferences to extract meaning. For example, a drastic change in source code between attempts might infer a complete start over, copy and paste from an example, or plagiarism. Analytic terms and their descriptions related to computer programming were selected to identify major domains are shown in Table 5-8.

Table 5-8

Analytic Terms for Major Domains Related to Changes in Student Source Code

Major Domain	Description
Build upon	Student added to or built upon previous code
Clean-up	Formatting of source code to be more self-descriptive or self-commenting
Comment	Add, delete, or move a comment
Dissection	Code is split and expanded to add more functionality
Grammar	The structuring of major code segments such as object definitions
Logic	The code obeys syntax rules but the output is incorrect or unexpected
No change	Code remained the same
Sudden change	Code changed significantly enough to be considered completely different from the previous attempt
Syntax	Change in a statement's syntax

Domain analysis was limited because it could describe, in some cases, what *appeared* to be the most significant change. Since many instances of change could be made between pairs of attempts, the most likely or dominant change was recorded. For example, a single attempt might include a combination of *Build upon*, correction of a *Syntax* error, and a *Logic change* to improve upon a previous problem. If the dominant change was the correction of a syntax error, that domain was recorded. This rule was established to help make the task of analyzing thousands of source code pairs feasible in a limited amount of time. The major domain analyses for each exercise are presented in Figures 4-2 through 4-11 followed by a summary of all exercises involved in Figure 5-15. Refer to Appendixes D and E for examples of source code comparison.

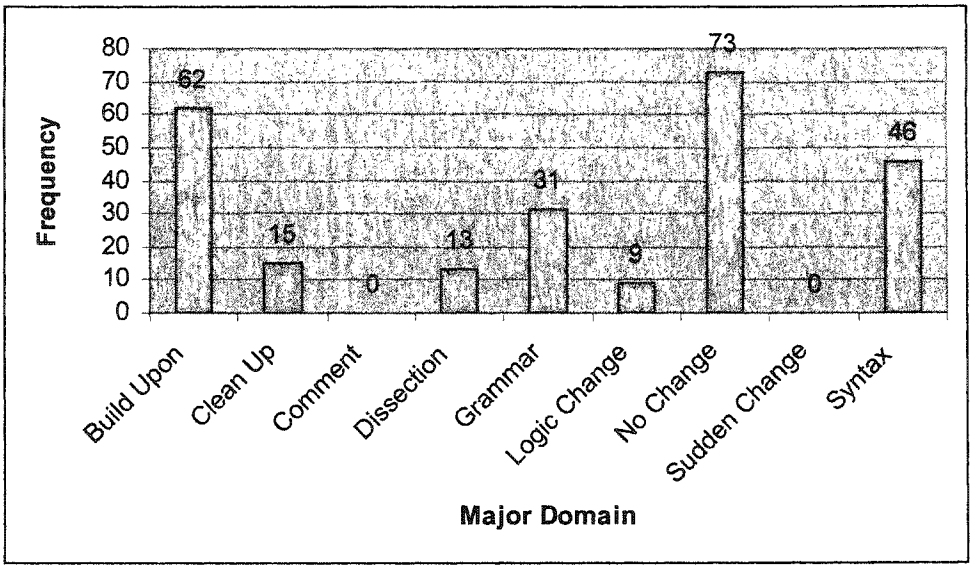


Figure 5-5. Exercise 2-7-1 Major Domain Frequency Distribution.

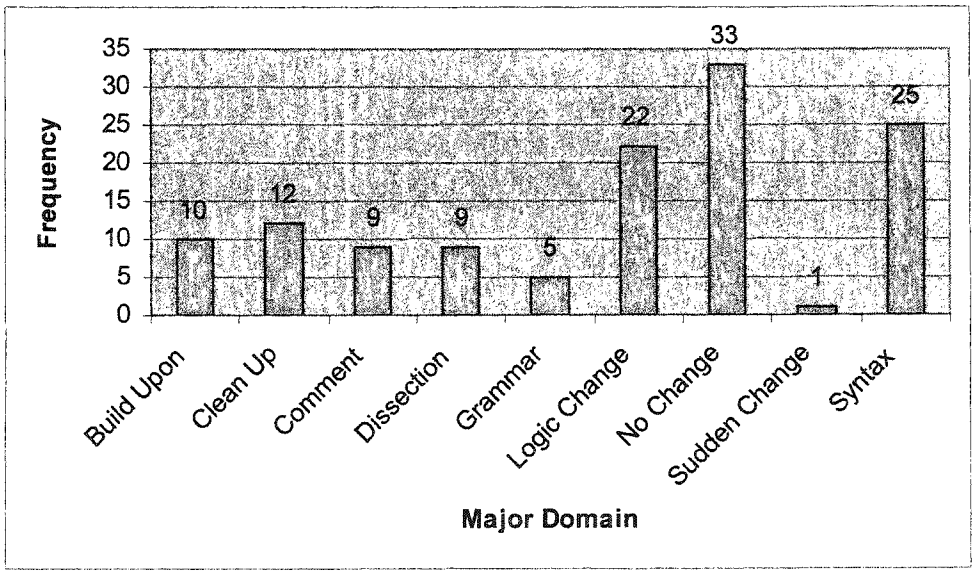


Figure 5-6. Exercise 3-3-2 Major Domain Frequency Distribution.

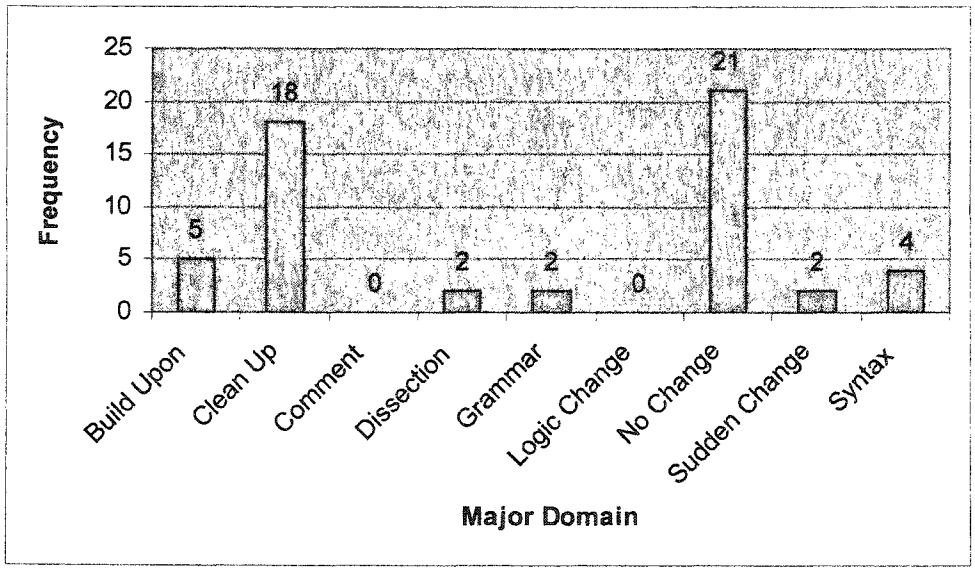


Figure 5-7. Exercise 3-3-3 Major Domain Frequency Distribution.

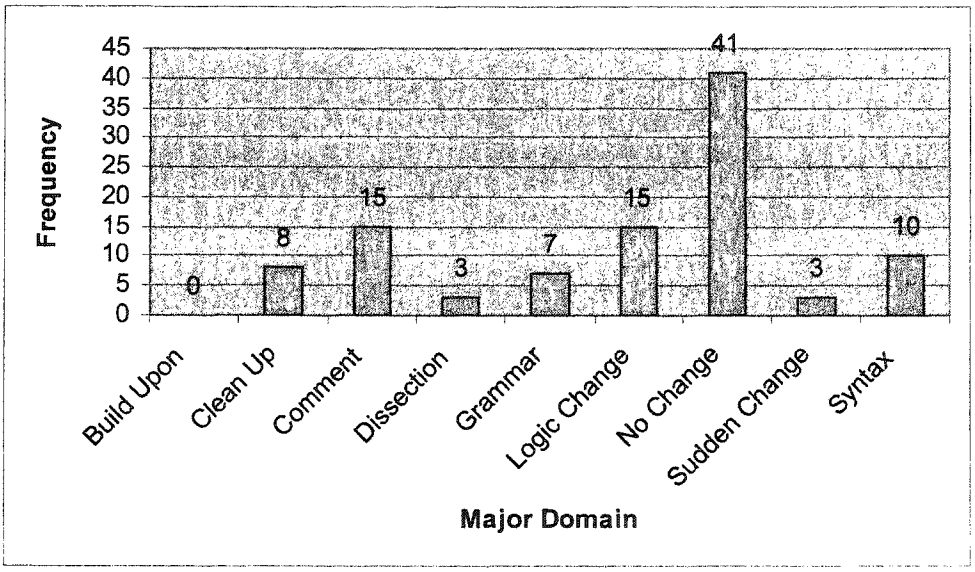


Figure 5-8. Exercise 3-3-4 Major Domain Frequency Distribution.

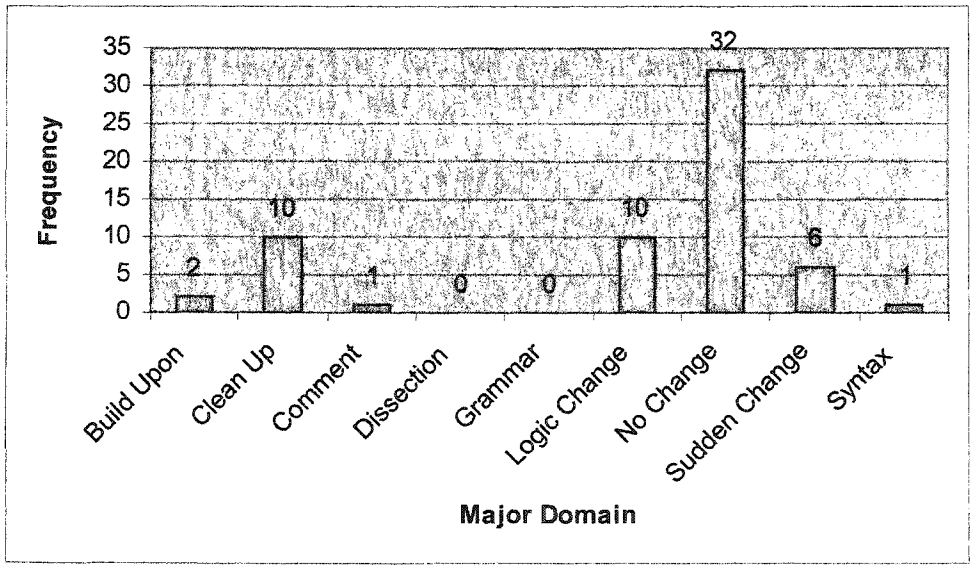


Figure 5-9. Exercise 3-3-5 Major Domain Frequency Distribution.

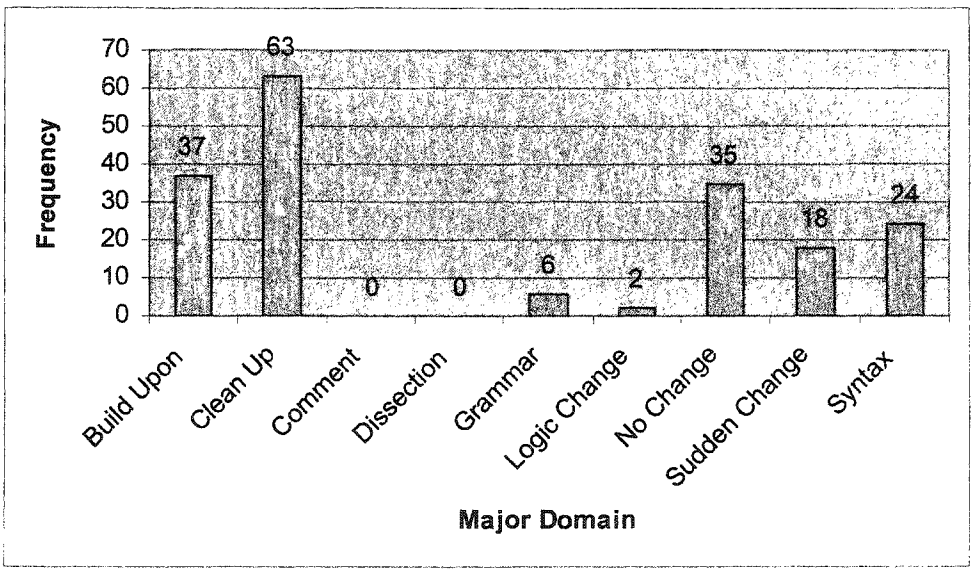


Figure 5-10. Exercise 3-5-1 Major Domain Frequency Distribution.

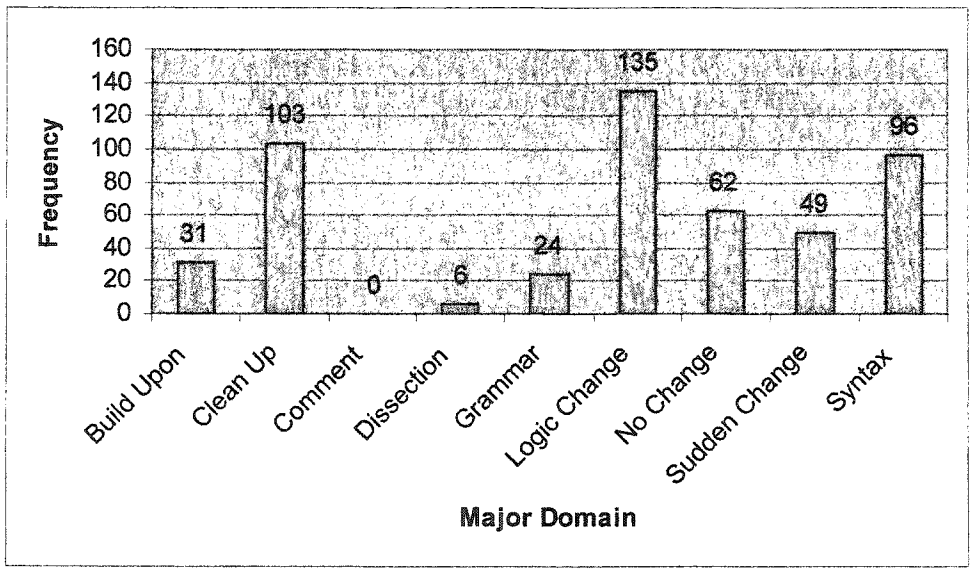


Figure 5-11. Exercise 3-5-2 Major Domain Frequency Distribution.

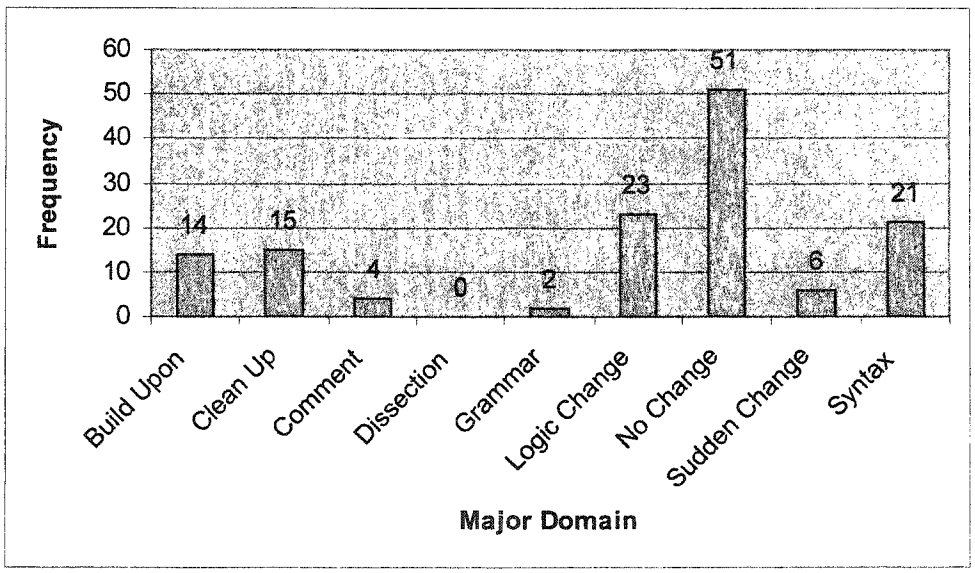


Figure 5-12. Exercise 3-5-3 Major Domain Frequency Distribution.

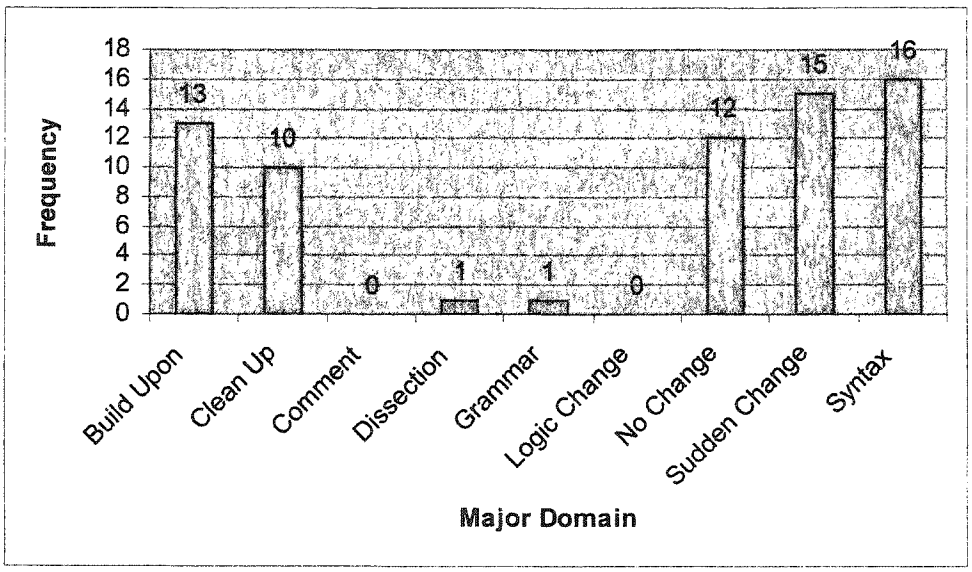


Figure 5-13. Exercise 3-7-1 Major Domain Frequency Distribution.

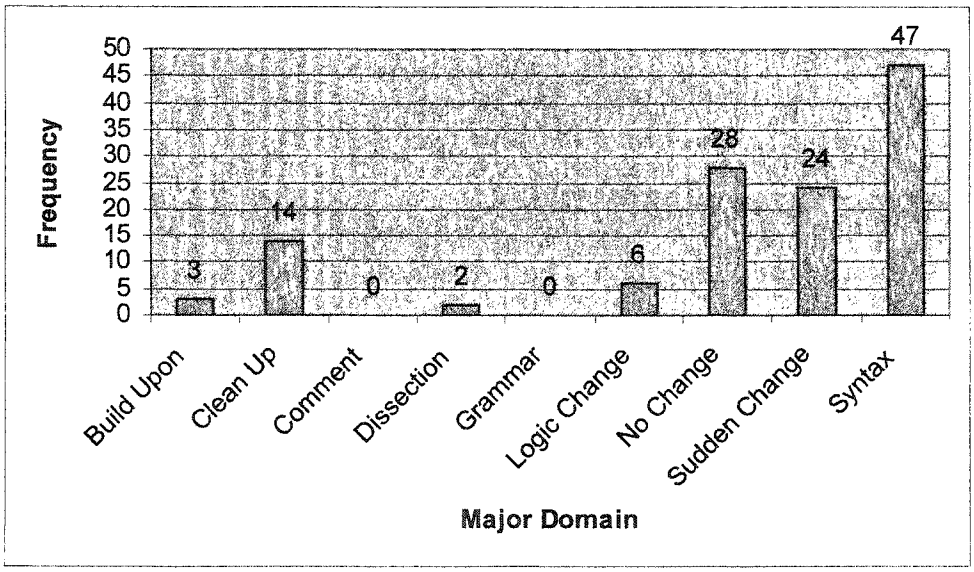


Figure 5-14. Exercise 3-7-2 Major Domain Frequency Distribution

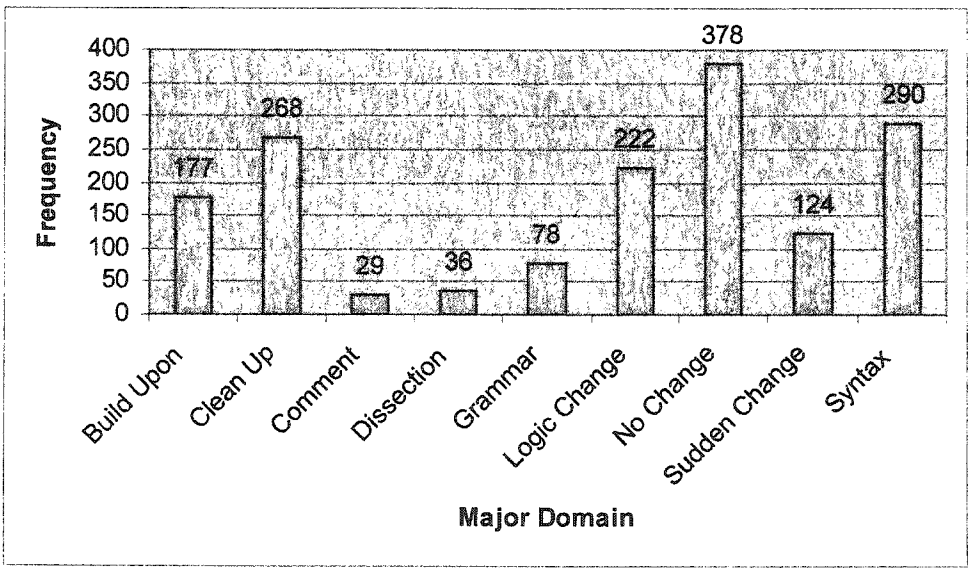


Figure 5-15. Summary of Major Domain Frequency Distributions.

Sub-domains

Four of the nine major domains were further divided into minor sub-domains. These include *Grammar*, *No change*, *Sudden change*, and *Syntax*. If the cause for a change could be inferred, a sub-domain was noted to clarify that inference. For example, a sudden change might take place for various reasons including a desire to start over, use of an in-text or in-exercise example, or use of borrowed code. In many cases there was no way to know the exact reason for the dramatic change. Plagiarism (borrowed code) was inferred if the change was attributable to the use of hints and the programming style was dramatically different from previously observed code. The major domains and their sub-domains are listed in Table 5-9 with descriptions for each.

Table 5-9

Sub-domains of Major Domains of Changes Students Make to Source Code

Major Domain	Minor Sub-domain	Description
Grammar		The structuring of major code segments such as object definitions
	Misspell	Variable or statement was misspelled
	Case sensitivity	Correction was made to account for case sensitivity in the language
	Variable name	Change in the use of variable because it was misspelled or the wrong variable was used
No change		Code remained the same
	Did it run?	Unsure about whether the program ran, the exercise was submitted again
	Desire to start over	Complete re-start of the solution
	Double/triple click	Double or triple clicking a submit button when only one click is necessary. Multiple copies were submitted as a result.
	Debug console clear	Following the first submit, the debug console was full of error messages from previous runs. The console was cleared and the program was run again.
	Other	Any other unexplainable submittal where the source code did not change.
Sudden change		Code changed significantly enough to be considered completely different from the previous attempt

Table 5-9 (Continued)

Major Domain	Minor Sub-domain	Description
	Plagiarism	Code taken from an external source that's pasted over the previous submitted code
	Hint or example	Source code from a hint, in-text example, or in-exercise example was copied.
	Previous code	Code from a previous run (that was saved somewhere in a text file) was copied, pasted back, and run
	Other	Any other unexplainable dramatic change in source code
Syntax		Change in a statement's syntax
	( ) Parenthesis	Incorrect pairing or use of parenthesis
	{ } Brackets	Incorrect pairing or use of curly brackets
	[ ] Brackets	Incorrect pairing or use of square brackets
	Misuse of quotes	Quotes used to delimit strings or in objects and other definitions was misused
	Function parameter	Parameters or arguments were mis-applied
	Language confusion	Order of operations error or incorrect application of math functions
	General confusion	Incorrect use of order of operations, loop counters, logical combinations, etc.
	Other	Any other syntax error

A summary of sub-domain frequency distributions for each of the major domains is provided in Figures 4-16 through 4-19.

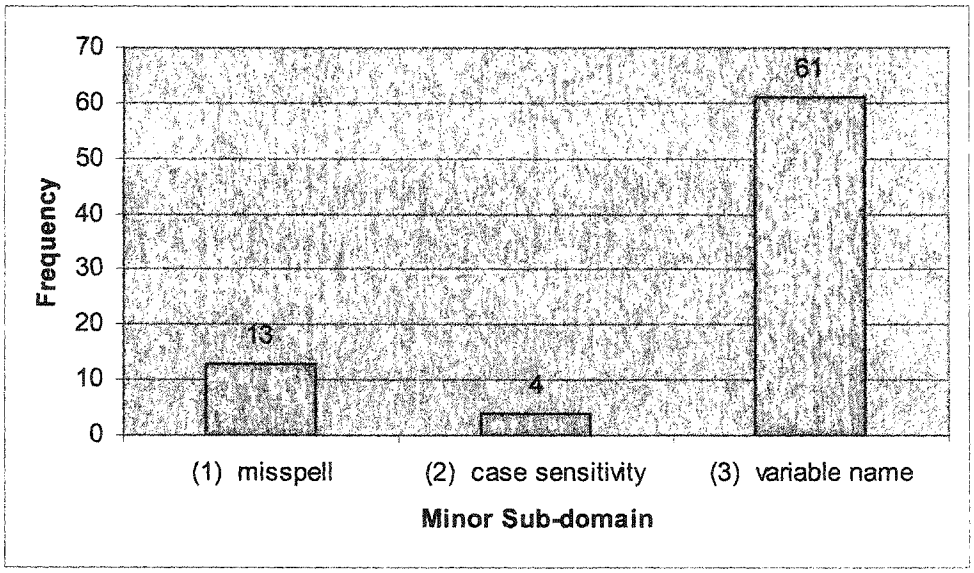


Figure 5-16. Summary of the Grammar Sub-domain Frequency Distributions.

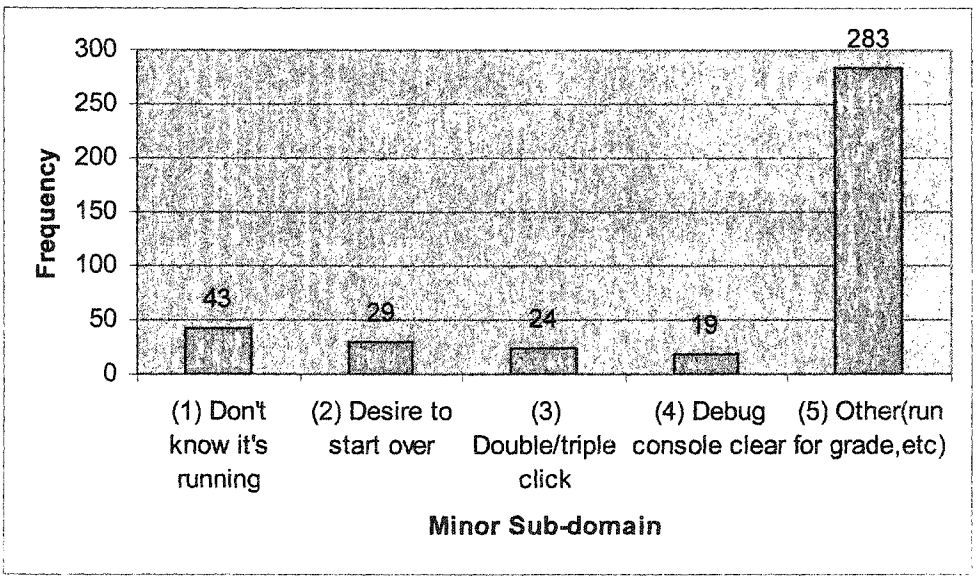


Figure 5-17. Summary of the No-change Sub-domain Frequency Distributions.

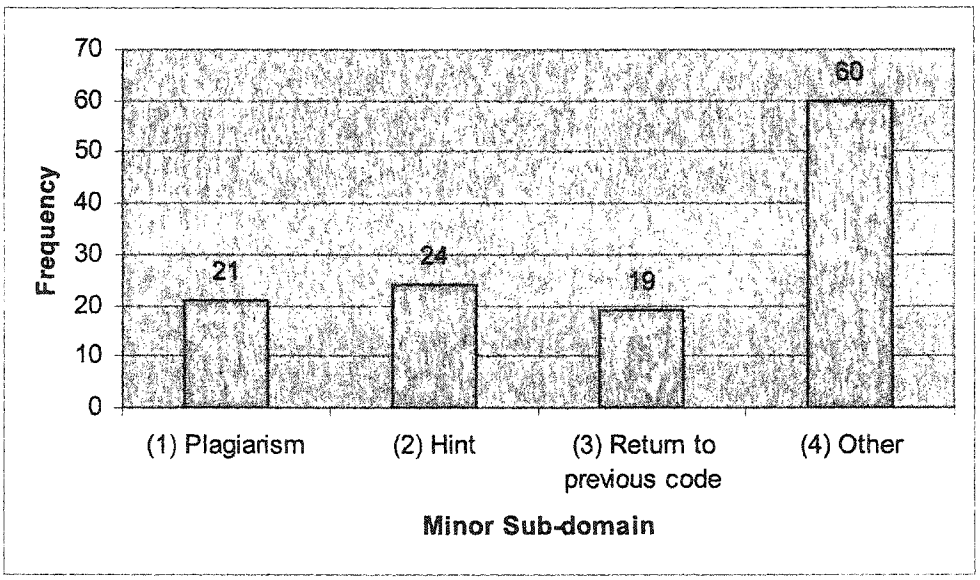


Figure 5-18. Summary of the Sudden Change Sub-domain Frequency Distributions.

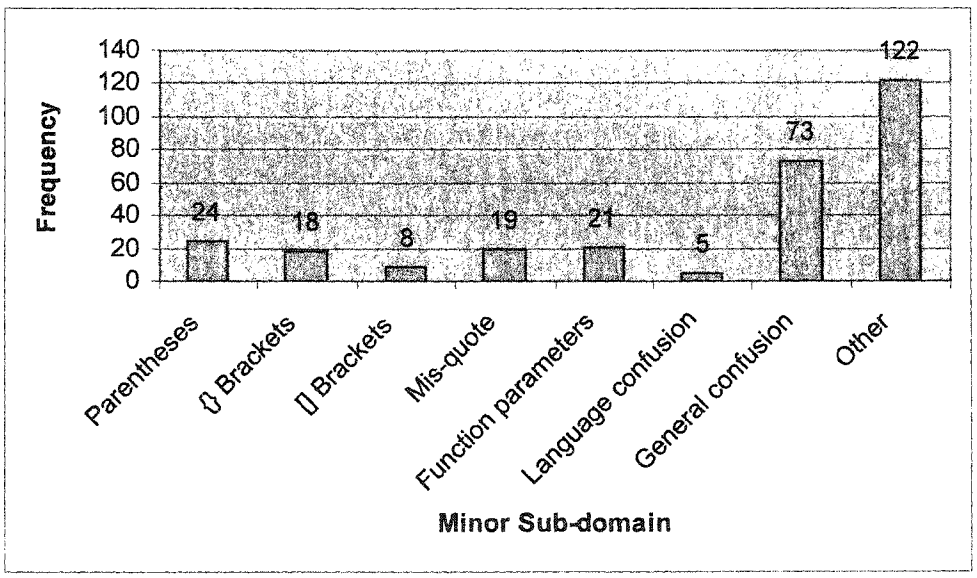


Figure 5-19. Summary of the Syntax Sub-domain Frequency Distributions

In summary, an analysis of code samples between runs showed that students not only fixed errors between runs, they also spent considerable time re-running programs (no change), cleaning-up code, and building upon previous code. Insight into these behaviors

makes it possible to more clearly understand learning patterns while solving programming problems. The most common mistakes made were syntax, logic, and grammar, in that order. Findings varied among individual exercises to the extent that no discernable patterns emerged. Many reasons for changes to source code were unknowable because of limitations imposed on the analysis. The limitations were imposed mainly because of the complexity of the programming language and the many ways mistakes can be made. Changes to source code that were obvious or discernable within the analysis limits were those that contributed to the statistics.

CHAPTER 6

DISCUSSION

This chapter is divided into five parts. Part 1 provides a discussion of the major results presented in Chapter 4. Part 2 presents the implications for practice resulting from the study. Part 3 presents the implications for research. Part 4 addresses the overall efficacy of CS Online as an instructional environment, and Part 5 provides the limitations inherent in the present study. Part 6 provides concluding remarks.

Part 1: Discussion of Results

This section presents a discussion of results based on student performance in the use of in-text and in-exercise worked examples, hint usage, completion of optional exercises, and problem solving performance. Due to the small sample size, only descriptive statistics were provided. Thus, the discussion of the findings only describes the performance of the 36 students involved in this current project. The discussion begins with self-regulation and student performance including findings related to increasing task difficulty, and then continues with math experience, student performance, computer experience, and student performance. The section closes with a discussion of common mistakes made by students while solving programming problems. Findings from the study can be used for further development of CS Online and future research.

Self-Regulation and Student Performance

Self-regulation seems to affect student performance in the use of CS Online, but given enough time, students with lower measures might succeed in completing course content with acceptable scores. It was shown that students who were more highly motivated outperformed the lower motivation group in *all* objective performance measures. These findings are consistent with research that shows students who are highly self-motivated establish high academic goals and achieve at a higher level (Schraw, 1998). In addition, those with higher planning performed at least as well or higher than lower planners in *most* objective measures, with the exception of the average hint level used and the total number of exercises completed. To begin, student dependence on worked examples in CS Online reinforces research that their use is paramount to effective programming instruction (Van Merrienboer & Krammer, 1987) and as a skills acquisition system (Anderson, Fincham, & Douglass, 1997). It appears that worked examples can be important to *all* students regardless of their level of self-regulation with the exception, however, of planning. Planning has already been determined to be an essential component required for effective programming instruction (McCoy, 1990). By emphasizing sound program design through planning, students will need to reference in-text examples less frequently and, thereby, direct their attention more toward synthesizing their own solutions.

The use of hints was especially important in the pilot study since students were required to rely on them as first order assistance. The same will likely be true in online settings where teachers may not be as accessible as in a traditional classroom. Knowing that building students' self-confidence and encouraging them to try harder can result in

successful completion of assigned work with no grade penalties (assuming a penalty for hint usage). Students can try harder by learning to master the art of debugging – which lies at the core of good programming practice. Debugging can also be viewed as the mechanism by which algorithmic and logical design is transformed into a functional product. In addition, the use of hints can be an effective tool to reach the less motivated.

Optional problems appear to be a very useful tool to provide additional learning opportunities to students who tend to finish their work quickly. In the case of CS Online, findings showed that students who were more highly motivated were those who sought out and completed additional work. The practical importance of these findings is that in addition to expanding project functional requirements as enrichment opportunities, optional exercises should also be incorporated as enrichment for the more motivated. These can come in the form of additional projects or standalone exercises.

Since the mean exercise and exam scores were comparatively similar among self-regulation sub-groups, it appears that students who try hard can succeed in learning introductory computer programming, regardless of their mathematical or technical background. This finding is important, knowing that students with various math and computer backgrounds can learn algorithmic and logical thinking in the context of a well-designed learning environment. It also appears that a lower number of completed exercises is not indicative of a lower level of achievement, but rather a reflection of higher quality in planning and self-checking that takes place during program design (Hancock, 1988).

Self-Regulation, Increasing Task Difficulty, and Student Performance

The first trends found were in the planning, self-checking, and self-efficacy sub-components of self-regulation with the total and average number of attempts required to complete exercises as the task difficulty increased. Students in high self-regulatory skill groups required fewer attempts to solve increasingly difficult exercises than the lower score group. It appears that self-regulation can be an important learner characteristic that can dramatically affect programming skills, especially when the exercise difficulty increases (Anderson, Fincham, & Douglass, 1997). Training in self-regulatory skills acquisition might require more time on the front-end of the problem solving process, but will, in effect, serve to build more efficient problem solvers in the long run.

Similarly, two salient trends emerged from the use of hints as the task difficulty increased. The first trend showed that students who exerted higher effort tended to depend on hints more than the lower score group as the task difficulty increased. The second trend showed an opposite effect for students with high self-efficacy. These students relied less on the use of hints as the task difficulty increased in both the number of exercises where hints were used, and the average hint level reached in each of those exercises. These trends might indicate that although both are related to motivation, higher self-efficacy can have an opposite effect from effort when the task difficulty increases. Thus, students with higher confidence in their ability to tackle harder problems might need less help than those who are motivated hard workers. This is important in that confidence building might be more important in the long run than just encouraging hard work.

Math Experience and Student Performance

Students in the high math group outperformed the low score group students in all performance measures of the pilot study. These findings indicate that students with more math experience may perform better in programming classes; findings that are consistent with other related research (McCoy & Dodl, 1989). Students interested in learning computer programming can be encouraged to take math classes either as pre-requisites or concurrently. However, since students in the lower math experience group also showed success by completing slightly fewer exercises and scoring slightly less on the exam and exercise scores, math should not have to be pre-required for introductory computer science. The reverse might also be just as important. Since learning computer programming improves math skills (Wieschenberg, 1999), students should be encouraged to take introductory programming in preparation for more advanced math classes. Students in the low math group depended more highly on the use of hints. It might be worthwhile to try defining hint usage according to pre-determined math experience. In other words, students in the low math experience group might be encouraged to use hints with little or no penalty.

Similar to the use of hints, students in the low math group completed fewer optional exercises. It seems practical that students in the high math group, like those in the high self-regulation group, should be provided optional exercises as enrichment. Because students of all math experience scores were able to perform adequately on exercise and exam scores, students of all backgrounds and interests should be encouraged to study computer science not for the purpose of learning programming, but for developing

problem solving skills which are vital for success in other subject areas and life in general (Casey, 1997).

Computer Experience and Student Performance

In general, students in the high computer experience group outperformed those in the low score group in *most* performance measures including less dependence on hints, fewer number of attempts required to complete required and optional exercises, and higher exercise and exam scores. It appears from these findings that students with computer experience might be more likely to jump to the exercises with less attention paid to in-text worked examples. This behavior provides evidence of the importance of research that suggests worked examples most closely related to the exercise should be developed and directly paired with that exercise (Paas, 1992).

Similar to self-regulation and math, students in the high math experience group required the use of hints in fewer exercises and used a lower hint level on the average. It follows that students in the low computer group might be allowed to use hints with a reduced or eliminated penalty, or that in-exercise hints could be designed to more closely resemble the paired exercise for those students. Similarly, high score group computer experience students outperformed low score group students in all measures of optional exercise use, but not as dramatically as those in the high math group. It appears that students with a stronger computer background might not necessarily excel in the use of enrichment activities.

Students in both low and high scoring computer experience groups were very close in their average scores on exercises and the exam. These findings indicate that students with less computer experience might be able to succeed almost equally as well as those in the

high experience group in learning introductory programming in CS Online. The findings also reinforce that students can succeed and should be encouraged to study computer science for the many benefits it offers.

Common Mistakes made while Solving Computer Programming Exercises

Writing a computer program is quite similar to writing an essay or a report; the revision process involves proofreading, isolating and correcting errors, and then observing for correctness and formatting. The cycle might need to be repeated many times before the product is complete or sufficiently ready for approval or grading. In the case of the pilot study, students weren't required to perfect programs, but to make them work according to requirements at a sufficient level of development effort. The result of their work was a compilation of thousands of source code samples that could be compared and analyzed to gain insight into how the students solved programming problems. The analysis led to an understanding of the common mistakes made in the process.

Findings from the pilot study imply that students tended to err on the side of syntax, logic, and grammar, in that order. For syntax, it appears that instructional methods similar to those used to teach written language constructs could be applied to help reduce error frequency. With fewer syntax errors, students could direct more attention toward higher-level problem solving efforts. Students should use a good debugging tool to help recognize and correct syntax errors while program lines are being typed. By tracing a program line-by-line, the debugger is also the best tool to isolate and correct logic errors. The importance of learning a debugger cannot be over-emphasized; it is the best tool for observing the inner-workings of a computer program. In addition to the debugger,

students should be instructed how to embed extra test code into their programs as a second tier of debugging capability. Embedded code can display extra information or perform intermediate calculations between lines where errors are believed to exist. In summary, instructional strategies should include use of a debugger, embedded test code, and debugging skills.

Students should become familiar with the programming environment from the very beginning of the learning experience. Lack of familiarity with the environment will not only lead to more errors, but also to increased inability to trap and eliminate errors efficiently. In the CS Online learning environment, all activities took place in a browser where programs were run when a web form button was clicked. Lack of familiarity with web forms and objects might have contributed to many unnecessary multiple runs. Students should also be encouraged to run, re-run, and trace their programs to see where improvements can be made. In the case of an online environment, instructional modules should provide students with a clear description of the how the interface is to be properly used.

Students worked hard to solve problems as evidenced by the rate at which they built their code and the number of sudden changes to hints, worked examples, and previously run code. These findings imply that students should be instructed *when* and *how* to start over. To be more specific, there might be ways students can learn to gracefully turn away from the current wrong path and start over in a different direction.

Part 2: Implications for Practice

Implications for Computer Science Teachers

When used in a traditional classroom or as an online course, CS Online becomes a dynamic textbook, the development environment, and the classroom manager. As a dynamic textbook, material in chapter sections can be used to prepare lesson plans, presentations, and demonstrations. Through demonstration, teachers can trace and amplify worked examples to provide further clarification of concepts. As the development environment, students can run programs from classroom computers with access to the Internet or from home. As a management tool, the difficulties normally associated with managing student work are handled by the system.

With cumbersome management issues set aside, teachers can focus more clearly on the important aspects of teaching and learning computer science. Beginning with self-regulation, the study showed that, in general, students in the high planning, effort, and self-efficacy score groups outperformed their low score group counterparts. Thus, teachers should consider teaching planning skills according to research-based methodologies (Bayman & Mayer, 1988; Dalbey & Linn, 1985; Goktepe, 1985; Hancock, 1988; Kurland, 1984; McCoy, 1990). Because planning takes times, teachers should not expect students to complete as many exercises, although higher performance on scores, fewer attempts, and less dependence on hints can be expected, especially when the task difficulty increases.

Motivation (effort and self-efficacy) was also seen to be an important individual characteristic in the study. Students in the high motivation groups not only outperformed low score group students, they also excelled in completing additional, non-required work.

CS Online content appears to be inherently motivational, designed to interest all students including those who lack intrinsic motivation. This is evidenced by a high number of completed exercises in both low and high effort groups. In addition, by permitting students to work at their own pace through the content from the classroom or from home, learning becomes individualized, and students take more responsibility for their learning. Flexibility in this type of learning can be very motivational in itself (Martin, 1997).

The various forms of worked examples including in-text, in-exercise, and hints were valuable tools during the learning process in the current study. This was evidenced by reliance on worked examples by students of various experiences. Teachers should encourage the use of hints for students with low math and computer skills and the less motivated, but should discourage their use for those with more experience. This can be accomplished through the use of a penalty disincentive or through a Version 2.0 feature that will allow teachers to enable or disable hints for each student.

Knowing the types of mistakes students most commonly made during the pilot study, teachers can prevent many of those mistakes by finding ways to emphasize language syntax and good logic. By reducing time wasted by inefficient debugging methods, students can move forward at a more rapid pace, keeping them motivated and interested in learning the next topic.

Implications for Decision Makers

Evidenced by findings from the current study, students with a wide range of math experience, computer experience, and self-regulatory skills can succeed in introductory computer science. The study showed that students in the low score groups for math experience, computer experience, and self-regulation performed only marginally below

the high score group students; indicating that enough time through self-paced study can lead to success. The increasingly esoteric nature of AP computer science and emphasis on Java will only serve to alienate more students, giving them incentive to favor less academic technology education courses. Introductory computer science should be promoted to students of all ages as an *academic* subject for the myriad cognitive benefits it provides, and the transfer of these skills into all other academic areas of study (Casey, 1997; Goldenson, 1996).

Part 3: Implications for Research

CS Online creates opportunity for research because of the vast amount of data it generates while students engage in learning activities. In addition, any number of questionnaires can be designed and placed for students to complete at designated times such as at the next login, the beginning of a chapter, or following completion of all required exercises in a section. The goal of this section is to provide an overview of the types of research that can be conducted in light of findings from the pilot study and data generation capabilities inherent in CS Online. The section begins with self-regulation and follows with math experience, computer experience, and then general research possibilities.

Implications for Research on Self-Regulation and CS Online

Findings from the pilot study reinforced research that shows students who were highly self-regulated established high academic goals and achieved at a higher level (Schraw, 1998). Anticipating the importance of self-regulation in online learning, more can be done to further understand how this individual characteristic affects student

learning when controlling for math and computer experience. To begin, the pilot study recorded various student interactions with worked examples including the number of visits per exercise. The system did not, however, record how those worked examples were modified or amplified before each run. By adding a feature to collect this information, further research can reveal the importance of worked examples and their design for various types of self-regulated learners. Research could also determine the *sequence* of worked example use, or the order in which worked examples were visited with respect to submitted attempts. Patterns of worked example use might predict performance factors such as the average number of attempts required or average score. In addition, because hints in the pilot study were a type of worked example that resulted in a penalty for use, future research could explore the effects self-regulation might have on hint usage without a penalty.

In the areas of performance on exercises and the exam, future research on the completion of optional exercises might investigate the dependence on worked examples and the use of hints. In other words, how much *more* do students depend on worked examples and hints when solving enrichment or optional exercises - or how hard do they try knowing the non-essential nature of the work? Qualitative studies might examine possible ways of delivering self-regulation study skills instruction in the context of CS Online. And future research could also examine trends more closely to try to find relationships between components of self-regulation, increasing task difficulty, and reliance on help to solve problems.

Implications for Research on Math and Computer Experience and CS Online

Research has established that experience in computer programming leads to increased achievement in mathematics (McCoy, 1988; McCoy & Dodl, 1989; Oprea, 1988). To be sure, future research might investigate the relationship between students having completed introductory computer programming and performance in Algebra and other math-related subjects. If introductory programming is offered to early middle school grades, then a higher success rate in 8th grade algebra might result. Future research might also compare the effects of reducing or eliminating penalties for hint usage for low math and computer experience students only, giving them additional opportunity to succeed among higher achieving students. In addition, research might explore the relationship between self-regulation, math experience, and student performance in completing optional exercises. Since these two measures appeared to be related, which is more significant in contributing to the achievement effect? More research might explore the relationship between math experience, computer experience, and inverse dependence on in-text and in-exercise worked examples. In other words, why do students with math experience rely more heavily on in-text worked examples while students with computer experience rely more on in-exercise worked examples? Findings from this research can explain worked example types that benefit students of various backgrounds, especially in a highly constructivist learning environment (Williams & Hmelo, 1998). Future research might also explore the relationship between specific math and computer experiences and less dependence on hint usage. More specifically, which math and computer experiences have the greatest effect on hint usage in learning introductory programming? Finally, future research might explore the relationship between computer experience, math

experience, and student performance in completing optional exercises. Since mathematics and computer programming involve similar processes (Wieschenberg, 1999), which contributes more significantly toward logical reasoning, inductive, deductive reasoning?

Implications for Research on Common Mistakes Students Made

The debugging process is one that is of considerable importance to learning programming. Although debugger use was not a topic of the CS Online pilot study, future research could explore the use of debuggers in instructional design. More specifically, how can debuggers be designed to be as instructionally viable as they are practical? In addition, future research can explore the comparative effects of development environment on students' ability to efficiently grasp and applying programming concepts. In other words, are there any benefits or detriments to using a web-based learning environment over traditional software development tools? Finally, research into when and why students quit and start over might inform improved instructional design tactics to help students avoid programming themselves into dead-ends. Finally, research can also attempt to identify relationships between worked example design and placement within instructional modules, and effects on sudden changes to source code.

In general, future studies could explore the amount of time expended between successive program runs, and first and final submittals for all exercises. In other words, research should seek to find if there is a relationship between the amount of time taken to complete an exercise and performance measures used in the present study. Additional research could also investigate whether time constraints imposed on chapter sections might impact student performance factors, or whether self-regulatory skills would play a

more significant role. If more sophisticated code comparison tools were available, future research might also determine more accurately the nature of differences between runs – which would lend clearer insight into the steps students take while solving programming problems. Finally, research can determine how well students can transition from JavaScript to Java as they program from introductory to advanced computer science study.

Part 4: The Efficacy of CS Online as an Instructional Environment

The CS Online pilot study provided insights into how students learn introductory computer programming concepts in the context of a new learning environment. Advances in technology, more specifically Web-based applications, have made it possible to not only deliver educational opportunities to a wider array of learners, but to observe learning in ways previously not possible. New understanding gained through data collection and observation can help inform the educational community by making recommendations for improved instructional practices and design.

Self-regulation was chosen as a measure for observation because of its anticipated importance for learners in online settings. In addition, it was hoped that the pilot study would uncover new evidence into how self-regulation affects learners in various ways, especially in a Web-based environment where little is known about how this individual characteristic affects learning (Hartley & Bendixen, 2000). It was discovered that self-regulation played an important role in student performance in CS Online, mainly in the areas of motivation and planning.

Findings from the pilot study amplify the importance of well-designed worked examples of various types, including hints at all levels, in-text examples, and in-exercise examples to help those who are weak in planning and less motivated. In-exercise examples should be designed to more closely match the exercises they support. Hints and in-exercise examples can remain as they were since a closely matched in-exercise example might negate the importance of the hint. In other words, if an in-exercise worked example is nearly identical to the paired exercise, why would hints be needed at all?

Students completed, on the average, 29.4 out of 45 possible exercises, or 67% of available exercises. Compared to low score group students, students in the high math group completed 3 additional exercises, those in the high computer experience score roup completed 4 additional exercises, and those in the high effort group completed close to 4.5 additional exercises. Students also earned an average of 9.2 out of 10 possible points for each exercise completed. The relatively high average score might be attributed to a lack of time limits imposed on the content, thereby enabling students to take as much time as needed to get their solutions ready. In summary, while certain individual characteristics led to better performance, students in low score groups for those characteristics were still able to successfully learn introductory computer programming through the use of CS Online. These findings imply that the CS Online model for introductory computer science can be educationally beneficial to students of with a wide range of previous experience.

Part 5: Limitations of the Study

CS Online was built upon research-based principles to maximize the benefits of what works for the improvement of computer science instruction, and for further research in the field of computer science education. As with any software development project, CS Online is incomplete and has already accumulated a list of new features for version 2.0. Some of the limitations in the current system translated into limitations for the pilot study.

To begin, several students suggested that in-exercise examples should more closely *resemble* the exercise they're paired with. The in-exercise examples actually used were those from section content that were most similar to the exercise. Because hint usage resulted in a penalty, the students were looking for a cost-free way to get to answers through a relatively identical paired example. Second, the in-text worked examples that were provided were the minimum necessary to get the project off the ground in time for the pilot study. While the multiple examples provide in each section were sufficient for instruction (Reed & Bolstad, 1991), more examples would only be of greater benefit for content comprehension. Finally, thirty-six students were not a large enough sample size to perform inferential tests of data generated by CS Online. Future research could use data collected to perform tests of this nature from a much larger sample size to provide insight into relationships between learner characteristics and performance measures.

Another limitation included unlimited time allowed for completing work. Given an unlimited amount of time to complete exercises in a self-paced environment, problem-solving performance might have been affected. CS Online provides the ability to impose time limitations on individual sections and all the exercises therein. A final limitation was

use of the Netscape Navigator 7.0 debugger to debug and fix programming errors. Although the debugger was sufficient for the current study, better debuggers like the Microsoft Script Debugger might have an effect on better debugging and ultimately problem solving.

With regard to common mistakes made by students, many changes to source code were unknowable because of the complexities inherent in programming languages and the many ways mistakes can be made. Because of this complexity, limits had to be placed on the types of mistakes to look for, thereby limiting the major and minor domains that emerged.

Part 6: Concluding Remarks

The CS Online pilot study showed that students of various self-regulatory skills, math experiences, and computer backgrounds could succeed in learning introductory computer programming concepts. Many students of all ages can benefit from computer science instruction, and introductory courses can be developed to lead the way to motivational and meaningful learning experiences. These experiences may, in turn, result in increased enrollments and a renewed interest in this challenging subject.

APPENDIX A

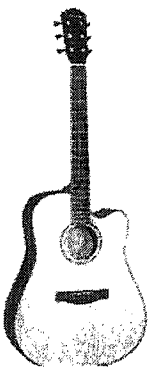
EXPANDED TABLE OF CONTENTS, SECTION CONTENT, EXAMPLES, AND EXERCISES

Building Your Own Objects

Other languages handle object construction differently, but the fundamental conc
master these concepts throughout the rest of this chapter, programming in the re
lot easier, and programming objects in other languages will be easier too.

Creating and using an object is very simple - only a three-step process.

1. Define the object and it's properties by writing a **constructor function**.
2. Define the methods for the object with separate functions
3. Create an instance of the object with new (as you did with pre-defined Jav
   is called instantiation. Whenver an object is instantiated, the properties rec
   them in the constructor function. These a known as **default properties**.



Example 1

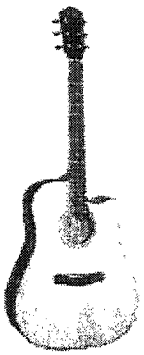
```
// This first example illustrates the three main steps
// The object is an acoustic guitar with 6 strings, t
//      to use a capo.

// ================================================
// The entire collection of functions between these t

// Step-1. Define the object and it's properties by u

function guitar() {
    this.string1 = 'E';
```

Building Your Own Objects

Other languages handle object construction differently, but the fundamental concepts rer can master these concepts throughout the rest of this chapter, programming in the rema be a lot easier, and programming objects in other languages will be easier too.

Creating and using an object is very simple - only a three-step process.

1. Define the object and it's properties by writing a **constructor function.**
2. Define the methods for the object with separate functions
3. Create an instance of the object with new (as you did with pre-defined JavaScript c is called instantiation. Whenver an object is instantiated, the properties receive the them in the constructor function. These a known as **default properties.**



Example 1

```
// This first example illustrates the three main steps to bu
// The object is an acoustic guitar with 6 strings, twelve f
//      to use a capo.

// ============================================================
// The entire collection of functions between these two line

// Step-1. Define the object and it's properties by using a

function guitar() {
     this.string1 = 'E';
     this.string2 = 'B';
     this.string3 = 'G';
     this.string4 = 'D';
     this.string5 = 'A';
     this.string6 = 'E';
}

// Step-2. Define the methods for the object using separate
//     We will start defining methods in example-2.

// ============================================================

// Step-3. Create an instance of the object and use it in a

var guitar1 = new guitar();
document.write('String-1 is a(n) ' + guitar1.string1 + '<br>
document.write('String-2 is a(n) ' + guitar1.string2 + '<br>
document.write('String-3 is a(n) ' + guitar1.string3 + '<br>
document.write('String-4 is a(n) ' + guitar1.string4 + '<br>
```

```
document.write('String-5 is a(n) ' + guitar1.string5 + '<br>
document.write('String-6 is a(n) ' + guitar1.string6 + '<br>
```

Try Example

Steps 1 and 2 define the object, step 3 is for instantiating and using the object in a prog
guitar object has six properties and no methods (yet). The properties are string1 ... strin
note assigned to the open string. An open string note is what you hear when the string is
pressing on any frets.

The word this refers to the object itself. In other words, this.string1 = 'E' can be read
this object will receive a default value of 'E' for every instantiation.

Let's make this object a little more interesting by adding frets to the guitar. Whenever a
string is plucked, you will hear a higher pitch. For each fret from 1 to 12, the pitch is rais
example, the open note for string1 is 'E', but when fret 1 is pressed, the note becomes a
twelve additional notes to each string using an array for each fret value:

Example 2

```
// This example illustrates the addition of frets to the guitar object.

// ============================================================================
// The entire collection of functions between these two lines defines the c

// Step-1. Define the object and it's properties by using a constructor fur

function guitar() {
     this.string1 = new Array('E','F','F#','G','G#','A',
          'A#','B','C','C#','D','D#','E');
     this.string2 = new Array('B','C','C#','D','D#','E',
          'F','F#','G','G#','A','A#','B');
     this.string3 = new Array('G','G#','A','A#','B','C',
          'C#','D','D#','E','F','F#','G');
     this.string4 = new Array('D','D#','E','F','F#','G',
          'G#','A','A#','B','C','C#','D');
     this.string5 = new Array('A','A#','B','C','C#','D',
          'D#','E','F','F#','G','G#','A');
     this.string6 = new Array('E','F','F#','G','G#','A',
          'A#','B','C','C#','D','D#','E');
     this.fret = 0;
}

// Step-2. Define the methods for the object using separate functions
//     We'll start defining methods in example-3.

// ============================================================================

// Step-3. Create an instance of the object and use it in a program.

var guitar1 = new guitar();
var fret_num = prompt('Enter a fret number (0-12): ','0');
fret_num = fret_num * 1;
```

```
document.write('When fret-' + fret_num + ' is pressed, the notes on all str
document.write('String-1 is a(n) ' + guitar1.string1[fret_num] + '<br>');
document.write('String-2 is a(n) ' + guitar1.string2[fret_num] + '<br>');
document.write('String-3 is a(n) ' + guitar1.string3[fret_num] + '<br>');
document.write('String-4 is a(n) ' + guitar1.string4[fret_num] + '<br>');
document.write('String-5 is a(n) ' + guitar1.string5[fret_num] + '<br>');
document.write('String-6 is a(n) ' + guitar1.string6[fret_num] + '<br>');
```

Try Example

Whenever you instantiate a new Array() object in JavaScript, you can assign values to t placing then in parenthesis and separating them by commas - as shown in the above exa this.string1[0] is assigned the open string note of 'E', this.string[1] is assigned 'F', ar

In example-2, all the notes are shown as either natural or sharp. In music, all notes can name. For example, D# is the same note as Eb. What happens if you want the notes to l flatted names in place of the sharps? We'll need to add a method that changes the string flatted note names:

Example 3

```
// This example illustrates the addition of methods to the guitar object.

// ==========================================================================
// The entire collection of functions between these two lines defines the o

// Step-1. Define the object and it's properties by using a constructor fur

function guitar() {
     this.string1 = new Array('E','F','F#','G','G#','A',
          'A#','B','C','C#','D','D#','E');
     this.string2 = new Array('B','C','C#','D','D#','E',
          'F','F#','G','G#','A','A#','B');
     this.string3 = new Array('G','G#','A','A#','B','C',
          'C#','D','D#','E','F','F#','G');
     this.string4 = new Array('D','D#','E','F','F#','G',
          'G#','A','A#','B','C','C#','D');
     this.string5 = new Array('A','A#','B','C','C#','D',
          'D#','E','F','F#','G','G#','A');
     this.string6 = new Array('E','F','F#','G','G#','A',
          'A#','B','C','C#','D','D#','E');
     this.fret = 0;
     this.changeToFlats = changeToFlats;
     this.displayAllNotes = displayAllNotes;
}

// Step-2. Define the methods for the object using separate functions
// The 1st method changes the property values to contain flatted notes
// The 2nd method display all notes staring with a user prompted fret numbe

function changeToFlats() {
     this.string1[2] = 'Gb'; this.string1[4] = 'Ab'; this.string1[6] = 'Bb'
          this.string1[9] = 'Db'; this.string1[11] = 'Eb';
     this.string2[2] = 'Db'; this.string2[4] = 'Eb'; this.string2[6] = 'Gb'
```

```
        this.string2[9] = 'Ab'; this.string2[11] = 'Bb';
    this.string3[1] = 'Ab'; this.string3[3] = 'Bb'; this.string3[6] = 'Db'
        this.string3[8] = 'Eb'; this.string3[11] = 'Gb';
    this.string4[1] = 'Eb'; this.string4[4] = 'Gb'; this.string4[6] = 'Ab'
        this.string4[8] = 'Bb'; this.string4[11] = 'Db';
    this.string5[1] = 'Bb'; this.string5[4] = 'Db'; this.string5[6] = 'Eb'
        this.string5[9] = 'Gb'; this.string5[11] = 'Ab';
    this.string6[2] = 'Gb'; this.string6[4] = 'Ab'; this.string6[6] = 'Bb'
        this.string6[9] = 'Db'; this.string6[11] = 'Eb';
}

function displayAllNotes(fret_start) {
    document.write('Starting from fret-' + fret_start +
        ', all notes on strings 6 5 4 3 2 1 are: <br><br>');
    for (var i = fret_start; i <= 12; i++) {
        document.write('Fret-' + i + ': ' + this.string6[i] + ' ');
        document.write(this.string5[i] + ' ');
        document.write(this.string4[i] + ' ');
        document.write(this.string3[i] + ' ');
        document.write(this.string2[i] + ' ');
        document.write(this.string1[i] + ' ');
        document.write('<br>');
    }
    document.write('<br>');
}

// ============================================================================

// Step-3. Create an instance of the object and use it in a program.

var guitar1 = new guitar();
var guitar2 = new guitar();
var fret_num = prompt('Enter a fret number [1-12]: ','0')*1;

guitar1.displayAllNotes(fret_num);
guitar2.changeToFlats();
guitar2.displayAllNotes(fret_num);
```

Try Example

The above example requires some explanation:

- Methods are listed in the constructor function along with the properties. Whenever yo **this.methodName**, methodName is what you'll use to run the method. You can use an calling method name. To the right of the '=' sign is the actual function name used to def actual function names must match.

```
this.changeToFlats = changeToFlats;
this.displayAllNotes = displayAllNotes;
```

- In the changeToFlats() methods, notice how only the properties that need to be chan the natural notes (with no sharps) are left alone. The word **this** references the origina constructor function - so you're changing the actual property value even though the cod

```
this.string1[2] = 'Gb';
```

• **function displayAllNotes(fret_start) {** Notice that a parameter is used in this me
is that the variable and value of fret_num are defined outside of the method definition.
an argument to the method in order to preserve the method's independence from the ex
words, if the entire object was copied and pasted into someone else's program, all they
the parameter list and what values the parameters are expecting. For example, if the pa
then the method would require the new programmer (who copied the object) to use the
the calling program. It is better to preserve re-useability by making objects independent

```
function displayAllNotes() {
    document.write('Starting from fret-' + fret_num +
    ...
```

• Notice that two instances of the guitar object were created and used in this program -
the other for flat notes. You can create and use as many instances of an object as your

This last example illustrates the benefits of using a visual interface with your programs.
to your program, HTML forms can be used to provide you (the user) with greater control
runs. The use of interfaces will be covered in Chapter-5 so, for now, just run the exampl
interface, a program runs from top to bottom - line by line. With a visual interface, you
when the program runs.

Example 4

```
<html>
<head>
<title>Guitar Object Interface</title>

<script>

// This example illustrates the use of an interface to the guitar object.

// ============================================================================
// The entire collection of functions between these two lines defines the o

// Step-1. Define the object and it's properties by using a constructor fur

function guitar() {
     this.string1 = new Array('E','F','F#','G','G#','A',
          'A#','B','C','C#','D','D#','E');
     this.string2 = new Array('B','C','C#','D','D#','E',
          'F','F#','G','G#','A','A#','B');
     this.string3 = new Array('G','G#','A','A#','B','C',
          'C#','D','D#','E','F','F#','G');
     this.string4 = new Array('D','D#','E','F','F#','G',
          'G#','A','A#','B','C','C#','D');
     this.string5 = new Array('A','A#','B','C','C#','D',
          'D#','E','F','F#','G','G#','A');
     this.string6 = new Array('E','F','F#','G','G#','A',
          'A#','B','C','C#','D','D#','E');
     this.fret = 0;
     this.changeToFlats = changeToFlats;
     this.displayAllNotes = displayAllNotes;
}

// Step-2. Define the methods for the object using separate functions
```

```
// The 1st method changes the property values to contain flatted notes
// The 2nd method display all notes staring with a user prompted fret numbe

function changeToFlats() {
     this.string1[2] = 'Gb'; this.string1[4] = 'Ab'; this.string1[6] = 'Bb'
          this.string1[9] = 'Db'; this.string1[11] = 'Eb';
     this.string2[2] = 'Db'; this.string2[4] = 'Eb'; this.string2[6] = 'Gb'
          this.string2[9] = 'Ab'; this.string2[11] = 'Bb';
     this.string3[1] = 'Ab'; this.string3[3] = 'Bb'; this.string3[6] = 'Db'
          this.string3[8] = 'Eb'; this.string3[11] = 'Gb';
     this.string4[1] = 'Eb'; this.string4[4] = 'Gb'; this.string4[6] = 'Ab'
          this.string4[8] = 'Bb'; this.string4[11] = 'Db';
     this.string5[1] = 'Bb'; this.string5[4] = 'Db'; this.string5[6] = 'Eb'
          this.string5[8] = 'Gb'; this.string5[11] = 'Ab';
     this.string6[2] = 'Gb'; this.string6[4] = 'Ab'; this.string6[6] = 'Bb'
          this.string6[9] = 'Db'; this.string6[11] = 'Eb';
}

function displayAllNotes(fret_start) {
     document.write('Starting from fret-' + fret_start +
          ', all notes on strings 6 5 4 3 2 1 are: <br><br>');
     for (var i = fret_start; i <= 12; i++) {
          document.write('Fret-' + i + ': ' + this.string6[i] + ' ');
          document.write(this.string5[i] + ' ');
          document.write(this.string4[i] + ' ');
          document.write(this.string3[i] + ' ');
          document.write(this.string2[i] + ' ');
          document.write(this.string1[i] + ' ');
          document.write('<br>');
     }
     document.write('<br>');
}

// ===================================================================

// Step-3. Create an instance of the object and use it in a program.

function runProgram() {

     var guitar1 = new guitar();
     var fret_num = document.guitarForm.txtFret.value*1;
     var note_type = document.guitarForm.radNotes[1].checked;

     if (note_type) {
          guitar1.changeToFlats();
     }
     guitar1.displayAllNotes(fret_num);

}
</script>

<meta http-equiv='Content-Type' content='text/html; charset=iso-8859-1'>
</head>

<form name='guitarForm' method='post' action=''>
  <table width='32%' border='1'>
    <tr bgcolor='#8899CC'>
      <td colspan='3'>
        <div align='center'><font face='Verdana, Arial, Helvetica, sans-ser
          <font color='#FFFFFF'>Guitar
```

```
          Object with Interface<br>
          </font></b><font color='#FFFFFF'>,enter a fret number 0-12, click
          type of notes, then click Run!</font><b><font color='#FFFFFF'>
           </font></b></font></div>
      </td>
    </tr>
    <tr>
      <td width='35%'>
        <div align='right'><font face='Verdana, Arial, Helvetica, sans-seri
          </font>
          <input type='text' name='txtFret' value='0' size='5' maxlength='5
        </div>
      </td>
      <td width='29%'>
        <input type='radio' name='radNotes' value='sharp' checked>
        <font face='Verdana, Arial, Helvetica, sans-serif' size='2'>Sharp <
        <font face='Verdana, Arial, Helvetica, sans-serif' size='2'><br>
        <input type='radio' name='radNotes' value='flat'>
        <font face='Verdana, Arial, Helvetica, sans-serif' size='2'>Flat</f
        </font></td>
      <td width='36%'>
        <input type='button' name='runExample' value='Run' onClick='runProg
      </td>
    </tr>
  </table>
  <p> </p>
  <p><font face='Verdana, Arial, Helvetica, sans-serif' size='2'> </font> <
</form>
```

Try Example

Exercises

APPENDIX B

THREE HINT LEVELS AND THE EXERCISE SOLUTION EXAMPLE

```
// Solution
// Predict which rodeo rider will have the best time
//

function avg_5(n1,n2,n3,n4,n5) {
     // The n's are the numbers to be averaged
     return (n1 + n2 + n3 + n4 + n5)/5;
}

function predict(X1,X2,X3,X4,X5,X,Y1,Y2,Y3,Y4,Y5) {

     var sum_XY = 0;
     var sum_X2 = 0;
     var avg_X = 0;
     var avg_Y = 0;
     var a = 0;
     var b = 0;
     var n = 5;
     var Tx = 0;

     sum_XY = (X1*Y1) + (X2*Y2) + (X3*Y3) + (X4*Y4) + (X5*Y5);
     sum_X2 = Math.pow(X1,2) + Math.pow(X2,2) + Math.pow(X3,2)
          Math.pow(X4,2) + Math.pow(X5,2);
     avg_X = avg_5(X1,X2,X3,X4,X5);
     avg_Y = avg_5(Y1,Y2,Y3,Y4,Y5);
     b = (sum_XY - n*avg_X*avg_Y)/(sum_X2 - n*Math.pow(avg_X,2));
     a = avg_Y - b*avg_X;

     Tx = a + b*X;
     return Tx;

}

var dave_time = predict(1,2,3,4,5,6,24.8,29.2,31.4,27.6,35.1);
var don_time = predict(1,2,3,4,5,6,27.8,31.5,26.3,30.2,29.9);
var randy_time = predict(1,2,3,4,5,6,30.4,24.6,27.2,24.8,30.6);

document.write('It is predicted that Dave\'s time will be <b>' +
dave_time + '</b> seconds<br><br>');
document.write('It is predicted that Don\'s time will be <b>' +
don_time + '</b> seconds<br><br>');
document.write('It is predicted that Randy\'s time will be <b>' +
randy_time + '</b> seconds<br><br>');
```

```
// Hint-1
// Predict which rodeo rider will have the best time
//
// Pseudocode description of the solution
```

```
//
// (1) Copy and paste the avg_5() function
// (2) Copy and paste the predict() function
// (3) Define variables for dave, don, an randy's times.
// (4) Assign the results of function calls to predict() to the three
variables.
//     Be sure to pass appropriate argument values to the expected
function parameters.
// (5) Output the three scores using document.write()




// Hint-2
// Predict which rodeo rider will have the best time
//

function avg_5(n1,n2,n3,n4,n5) {
     // The n's are the numbers to be averaged
     return (n1 + n2 + n3 + n4 + n5)/5;
}

function predict(X1,X2,X3,X4,X5,X,Y1,Y2,Y3,Y4,Y5) {

     var sum_XY = 0;
     var sum_X2 = 0;
     var avg_X = 0;
     var avg_Y = 0;
     var a = 0;
     var b = 0;
     var n = 5;
     var Tx = 0;

     sum_XY = (X1*Y1) + (X2*Y2) + (X3*Y3) + (X4*Y4) + (X5*Y5);
     sum_X2 = Math.pow(X1,2) + Math.pow(X2,2) + Math.pow(X3,2)
          Math.pow(X4,2) + Math.pow(X5,2);
     avg_X = avg_5(X1,X2,X3,X4,X5);
     avg_Y = avg_5(Y1,Y2,Y3,Y4,Y5);
     b = (sum_XY - n*avg_X*avg_Y)/(sum_X2 - n*Math.pow(avg_X,2));
     a = avg_Y - b*avg_X;

     Tx = a + b*X;
     return Tx;

}

var dave_time = predict(?);
var don_time = ?;
?

document.write('It is predicted that Dave\'s time will be <b>' +  ? +
'</b> seconds<br><br>');
...
```

```
// Hint-3
// Predict which rodeo rider will have the best time
//

function avg_5(n1,n2,n3,n4,n5) {
     // The n's are the numbers to be averaged
     return (n1 + n2 + n3 + n4 + n5)/5;
}

function predict(X1,X2,X3,X4,X5,X,Y1,Y2,Y3,Y4,Y5) {

     var sum_XY = 0;
     var sum_X2 = 0;
     var avg_X = 0;
     var avg_Y = 0;
     var a = 0;
     var b = 0;
     var n = 5;
     var Tx = 0;

     sum_XY = (X1*Y1) + (X2*Y2) + (X3*Y3) + (X4*Y4) + (X5*Y5);
     sum_X2 = Math.pow(X1,2) + Math.pow(X2,2) + Math.pow(X3,2)
          Math.pow(X4,2) + Math.pow(X5,2);
     avg_X = avg_5(X1,X2,X3,X4,X5);
     avg_Y = avg_5(Y1,Y2,Y3,Y4,Y5);
     b = (sum_XY - n*avg_X*avg_Y)/(sum_X2 - n*Math.pow(avg_X,2));
     a = avg_Y - b*avg_X;

     Tx = a + b*X;
     return Tx;

}

var dave_time = predict(1,2,3,?,?,?,24.8,29.2,31.4,?,?);
var don_time = ?;
var randy_time = ?;

document.write('It is predicted that Dave\'s time will be <b>' +
dave_time + '</b> seconds<br><br>');
...
```

APPENDIX C

QUESTIONNAIRES USED IN THE PRESENT STUDY

Directions: A number of statements which people have used to describe their computer programming experience and ability are given below. Read each statement and indicate how you generally think or feel by clicking the appropriate button (online questionnaire). There are no right or wrong answers. Do not spend too much time on any one statement.

Computer Experience and Ability Questionnaire[1]

1. I know how to type without looking at my hands. (Yes, No)
2. I understand the basics about how to use the computer including how to power-up, shut-down, the mouse, and the keyboard. (Yes, No)
3. I have taken and completed a Computer Expectations or Computer Literacy class in the past. (Yes, No)
4. If you completed a Computer Expectations or Computer Literacy class in the past, what was your grade? (A, B, C, D or below)
5. I am comfortable using the computer to do school work. (Yes, No)
6. I have used HTML. (Yes, No)
7. I can use HTML to create a web page. (Yes, No)
8. HTML is easy for me. (Yes, No)
9. I have used Visual Basic. (Yes, No)
10. I have used Visual Basic to create a computer program. (Yes, No)
11. Visual Basic is easy for me. (Yes, No)
12. I have used JavaScript. (Yes, No)
13. I have used JavaScript to create a computer program. (Yes, No)
14. JavaScript is easy for me. (Yes, No)

[1] (Hong & Halopoff, 2003)

Directions: A number of statements which people have used to describe their math experience and ability are given below. Read each statement and indicate how you generally think or feel by clicking the appropriate button (online questionnaire). There are no right or wrong answers. Do not spend too much time on any one statement.

Math Experience and Ability Questionnaire[2]

1. I have taken and completed Math Applications in the past. (Yes, No)
2. If you completed Math Applications in the past, what was your grade? (A, B, C, D or below)
3. I have taken and completed Pre-Algebra in the past. (Yes, No)
4. If you completed Pre-Algebra in the past, what was your grade? (A, B, C, D or below)
5. I have taken and completed Algebra I in the past. (Yes, No)
6. If you completed Algebra I in the past, what was your grade? (A, B, C, D or below)
7. I have taken and completed Algebra II in the past. (Yes, No)
8. If you answered 'Yes' to the previous question, what was your grade? (A, B, C, D or below)
9. I have taken and completed Geometry in the past. (Yes, No)
10. If you completed Geometry in the past, what was your grade? (A, B, C, D or below)
11. I have taken and completed Algebra II in the past. (Yes, No)
12. If you completed Algebra II in the past, what was your grade? (A, B, C, D or below)
13. I have taken and completed Pre-Calculus in the past. (Yes, No)
14. If you completed Pre-Calculus in the past, what was your grade? (A, B, C, D or below)
15. I have taken and completed Advanced Placement Calculus in the past. (Yes, No)
16. If you completed Advanced Placement Calculus in the past, what was your grade? (A, B, C, D or below)
17. I have taken and completed Trigonometry in the past. (Yes, No)
18. If you completed Trigonometry in the past, what was your grade? (A, B, C, D or below)
19. I have taken and completed Advanced Placement Statistics in the past. (Yes, No)
20. If you completed Advanced Placement Statistics in the past, what was your grade? (A, B, C, D or below)

[2] (Hong, 2003)

Self-Assessment Questionnaire[3]

Directions: A number of statements which people have used to describe themselves are given below. Read each statement and indicate how you thought or felt by circling 1, 2, 3, or 4 that best describes your mind. There are no right or wrong answers. Do not spend too much time on any one statement. (1 = Not at all, 2 = Somewhat, 3 = Moderately so, 4 = Very much so)

		Not at all	Some-what	Modera-tely so	Very much so
1	I determined how to solve the problem before I began.	1	2	3	4
2	I checked my work while I was doing it.	1	2	3	4
3	I worked as hard as possible on all exercise items.	1	2	3	4
4	Considering the difficulty of the items, I think I did well on the exercise.	1	2	3	4
5	Thinking about my grade in the course interfered with my work on the exercise items.	1	2	3	4
6	Compared to other subjects, this exercise was difficult.	1	2	3	4
7	It is important for me to do well on the exercise item.	1	2	3	4
8	I tried to understand the goal of the exercise questions before I attempted to answer.	1	2	3	4
9	I judged the correctness of my work.	1	2	3	4
10	I concentrated fully when I was doing the exercise items.	1	2	3	4
11	I think I did a good job on the exercise items.	1	2	3	4
12	Thoughts of doing poorly interfered with my concentration on the exercise items.	1	2	3	4
13	This exercise was easy for me.	1	2	3	4
14	I think the exercise items are useful for me to learn.	1	2	3	4
15	I carefully planned my course of action before I solved problems.	1	2	3	4
16	I checked how well I was doing when I was solving the exercise items.	1	2	3	4
17	I put forth my best effort on all exercise items.	1	2	3	4
18	I think I will receive a good score on the exercise.	1	2	3	4
19	During the exercise, I though about the consequences of failing.	1	2	3	4
20	This exercise was a difficult one for me.	1	2	3	4
21	Understanding the content of the exercise is important to me.	1	2	3	4
22	I thought through the steps in my mind before I attempted to solve the exercise items.	1	2	3	4
23	I asked myself questions to stay on track as I did the exercise items.	1	2	3	4

24	I kept working even on difficult exercise items.	1	2	3	4
25	I understood the content of the exercise items quite well.	1	2	3	4
26	During the exercise, I got so nervous I forgot the information that I really knew.	1	2	3	4
27	Since I understood the material well, this exercise was easy for me.	1	2	3	4
28	Getting a good grade in this exercise is important for me.	1	2	3	4
29	I asked myself questions about what the problem required me to do before I did it.	1	2	3	4
30	As I proceeded through the exercise items, I asked myself how well I was doing.	1	2	3	4
31	I didn't give up even if the problems were hard.	1	2	3	4
32	I think I did well on the exercise items.	1	2	3	4
33	I tried to determine what the exercise items required.	1	2	3	4
34	I checked the accuracy as I progressed through the exercise items.	1	2	3	4
35	I worked hard to do well on all exercise items.	1	2	3	4
36	Even when the questions were difficult, I knew I could succeed.	1	2	3	4

[3] (Hong, 2001b)

APPENDIX D

SOURCE CODE COMPARISON SAMPLE

First Run

```
var sentence = 'The skies in Montana are big.';
var sentence_length = 0;
var reversed_sentence = 'big very are Montana in skies The';
var sentence_array = new Array();

sentence = prompt('Enter any sentence: ',
'One ring to rule them all, one ring to find them...');
sentence_length = sentence.length;

sentence_array = sentence.split(' ');
sentence_array.reverse();
for (i = 0; i < sentence_array.length; i++) {
    reversed_sentence += sentence_array[i] + ' ';
}

document.write(sentence + '<br><br>');
document.write(reversed_sentence);
```

Second Run

```
var sentence = 'The skies in Montana are big.';
var sentence_length = 0;
var reversed_sentence = 'big very are Montana in skies The';
var sentence_array = new Array();

sentence = prompt('Enter any sentence: ',
'The skies in Montana are very big.');
                    [ default prompt value was changed – Build upon ]
sentence_length = sentence.length;

sentence_array = sentence.split(' ');
sentence_array.reverse();
for (i = 0; i < sentence_array.length; i++) {
    reversed_sentence += sentence_array[i] + ' ';
}

document.write(sentence + '<br><br>');
document.write(reversed_sentence);
```

Third Run

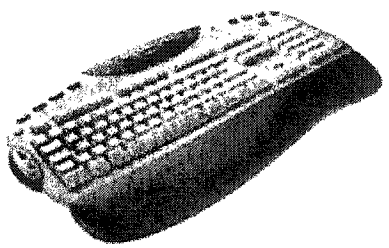
```
var sentence = 'The skies in Montana are big.';
var sentence_length = 0;
var reversed_sentence = 'big very are Montana in skies The';
var sentence_array = new Array();

sentence = prompt('Enter any sentence: ',
'The skies in Montana are very big.');
sentence_length = sentence.length;

sentence_array = sentence.split('T h e s k i e s i n M o n t a n a a r e v e r y b i g ');
                          [ tried to split a sentence other than the sentence in the variable
                          – Logic Error ]
sentence_array.reverse();
for (i = 0; i < sentence_array.length; i++) {
    reversed_sentence += sentence_array[i] + ' ';
}

document.write(sentence + '<br><br>');
document.write(reversed_sentence);
```

APPENDIX E

COMPARISON REPORT FOR JAVASCRIPT ERROR DOMAINS

CS ONLINE, SPRING 2003

The purpose of this analysis is to compare consecutive attempts by students to solve exercise 3-5-1. By comparing attempts the hope is to understand how the students learned to code Javascript, as well how to improve upon the questions used within the online program. Below is an example of how 3-5-1 appeared to the student.



Here's a helpful hint - the faster you're able to touch type, the quicker you'll be able to write programs. If you don't know how to touch type, get a program that you can install on your computer, and learn! In touch typing, words are counted in chunks of five (5) characters. In other words, the sentence 'Touch typing will help you become a better programmer since you will be able to type faster. Typing faster will give you more time to concentrate on the programming - not the typing.' has how many words? Write a program that inputs a sentence, like this one, then tells you how many words there are in the sentence.

The student also had access to three hints, which they could use by choice. Each exercise was worth 10 points, each hint level cost ½ point. The hints appeared to the students as the examples on the following pages demonstrate.

Hint 1

```
// Hint-1
// Count the number of touch typing words in a
sentence.

// Pseudocode description of the solution
// (1) Use prompt to input the sentence
// (2) The number of words is the length of the
sentence divided by 5
// (3) Output the length using alert() or
document.write()
```

Hint 2

```
// Hint-2
// Count the number of touch typing words in a
sentence.

var sentence = prompt(?);
var num_of_words = ?;
alert(?);
```

Hint 3

```
// Hint-3
// Count the number of touch typing words in a
sentence.

var sentence = prompt('Enter the sentence.','Touch
typing will help you become a better programmer
since you will be able to type faster.');
var num_of_words = ? / 5;
alert('The number of words in the sentence is ' ?);
```

In addition to the hints, there is an in-exercise example available for the student that resembles the exercise. There is no point cost for using the example. The example for 3-5-1 is rather lengthy, and is demonstrated on the following pages:

Example

```
// Examples of string properties and methods using
a broccoli soup recipe.

var broccoli_soup = 'Broccoli soup is a blend of
Broccoli, chicken broth, and onions.'
var soup_slogan = ''; // an empty string

// Show how some string methods work on this
string:
document.write('The string length is: ' +
broccoli_soup.length + '

');
document.write(broccoli_soup.toLowerCase() + '

');
document.write(broccoli_soup.toUpperCase() + '

');

// Display each character in the string followed by
its index value:
// The  HTML pair display the character in bold
for (var i = 0; i <= broccoli_soup.length; i++) {
     document.write('' + broccoli_soup.charAt(i)+''
+ i + ' ');
```

```
}
document.write('

');

// Show how more methods work on the string:
document.write('The first occurrence of the word
broccoli starts at character ' +
broccoli_soup.indexOf('Broccoli') + '

');
document.write('The last occurrence of the word
broccoli starts at character ' +
broccoli_soup.lastIndexOf('Broccoli') + '

');
document.write('Characters 9-12 of the string are
[' + broccoli_soup.substring(9,12) +
']

');
document.write('The first 10 characters of the
string are [' + broccoli_soup.substr(0,10) +
']

');
document.write('The character at index 3 is [' +
broccoli_soup.charAt(3) + ']

');

soup_slogan = broccoli_soup.concat(' Soup is good
food!');
document.write(soup_slogan);
```

This particular analysis of 3-5-1 was comprised of 25 students and 198 attempts and was conducted by one of the teaching assistants assigned to review student work. Students submitted an average of seven to eight attempts for this exercise. Unless otherwise stated, it is assumed that the final attempt was successful. The next section demonstrates the actual ethnographic record and domain analysis as worded by the teaching assistant for exercise 3-5-1:

Analysis:
Student # 1

General impressions of first attempt:

Three code lines, appears to have answer correct on first attempt.

2 compared to 1:

Added zero to prompt box. CLEAN UP

3 compared to 2:

Eliminated unneeded words in prompt. CLEAN UP

4 compared to 3:

Spaced sentence in prompt differently. CLEAN UP

5 compared to 4:

Same NO CHANGE (other)

6 compared to 5:

Removed zero from prompt. CLEAN UP

7 compared to 6:

Same NO CHANGE (other)

8 compared to 7:

Same NO CHANGE (other)

Overall comments:

Student has basic understanding of question. Six compared to five creates undefined area in prompt box. The written area in prompt box above where user inputs sentence could be better worded or at least presented better. I do not know if the student understands how the prompt function

works or if the aesthetics of the box is even a concern for this student.

Overall the student met the criteria of the question.

Analysis:
Student # 2

General impressions of first attempt:

Two line first attempt. Hard math values used rather than one that is dependant upon an input.

2 compared to 1:

Change in variable names and prompt added. Math dependant on input in a variable. Addition of alert for an output.

SUDDEN CHANGE (HINT)

3 compared to 2:

Spelling corrected in variable name. Prompt gets parentheses.
Grammatical improvements in alert. GRAMMAR(misspell & variable name)

SYNTAX (parentheses)

4 compared to 3:

Elimination of single quote prior to sentence.length/5 . Deleted spaces in alert.

CLEAN UP
SYNTAX (mis-quote)

5 compared to 4:

Same. NO CHANGE (other)

6 compared to 5:

Addition of "+' is' " in alert. CLEAN UP

7 compared to 6:

Elimination of spaces around word "is" in alert

CLEAN UP

8 compared to 7:

Addition of space after "is" in alert. CLEAN UP

9 compared to 8:

Same NO CHANGE (other)

Overall comments:

Student seems to work well after what seemed to be the use of a hint. Most of the steps showed clean up and some normal syntax and grammatical errors.

Analysis:
Student # 3

General impressions of first attempt:

Proper logic, but spelling errors present. Appears similar to the hint provided.

2 compared to 1:

Correction of spelling in prompt. GRAMMAR (misspell)

3 compared to 2:

Added words to prompt display. CLEAN UP

4 compared to 3:

Changed"word" to " words" in alert. CLEAN UP

5 compared to 4:

Added spaces to alert wording. CLEAN UP

6 compared to 5:

Changed "the" to "a" in prompt. Returns moved as a result of space change in prompt.

CLEAN UP

7 compared to 6:

Same NO CHANGE (other)

8 compared to 7:

Same NO CHANGE (other)

Overall comments:

Student had typical grammar and clean up issues. Note misspelling in variable name, variable still worked as it was consistently misspelled.

Analysis:
Student # 4

General impressions of first attempt:

Appears to be a copy of a hint level of some kind.

2 compared to 1:

Elimination of a line space, and an addition of a semi colon closing a line. CLEAN UP

3 compared to 2:

Fills in question mark with a variable. BUILD UPON PROGRAM

4 compared to 3:

Incorrectly eliminates part of math formula SYNTAX (general confusion)

Corrects capitalization in variable name. GRAMMAR (case sensitivity)

5 compared to 4:

Eliminates value of variable completely OTHER

Adds ".length" to alert statement BUILD UPON PROGRAM

6 compared to 5:

Changes variable name, and uses new name to be an equivalent in another variable. Not a drastic change.

OTHER

7 compared to 6:

Adds "+ / 5" to variable formula. BUILD UPON PROGRAM

8 compared to 7:

Eliminates "+/ 5" in variable and adds" /5" to a variable in an alert function.

CLEAN UP
BUILD UPON PROGRAM

9 compared to 8:

Introduces two strings, and changes alert function to use new string rather than previous formula. BUILD UPON PROGRAM

EX: var strng1=num_of_words.length/5
var strng2=Math.round(strng1)

10 compared to 9:

Eliminates two string previously added, but creates a sophisticated alert function to do the work of the two strings. BUILD UPON PROGRAM

EX: alert('The number of words in the sentence is '
+Math.round(num_of_words.length/5));

11 compared to 10:

Same NO CHANGE (other)

Overall comments:

It's always exciting when a student surpasses the expectations of the problem. Rounding is not a part of the expected solution, but this student accounted for it very well. This demonstrates a certain sophistication in java script coding.

Analysis:

Student # 5

General impressions of first attempt:

Starts with variable, no output.

2 compared to 1:

Variable name change, addition of a prompt with parentheses added. Elimination of part of a sentence. Created new variable with partial formula. Created alert.

CLEAN UP

SUDDEN CHANGE (HINT)
SYNTAX (parentheses)

3 compared to 2:

Change in math formula using hard numbers instead of dependant variables.

SYNTAX (general confusion)

4 compared to 3:

Put previously deleted sentence back into prompt. Corrected math formula to be dependant on variable length.

CLEAN UP

LOGIC CHANGE

Overall comments:

Student arrived at answer more quickly than the average. I suspect he had some form of outside help, perhaps the teacher or another student provided some direction.

Analysis:
Student # 6

General impressions of first attempt:

Four lines with text book correct answer. Identical to hints provided. Only one attempt made.

Overall comments:

Perhaps the server had gone down losing students previous attempts. The student may have remembered solution from previous work.

Analysis:
Student # 7

General impressions of first attempt:

One line, prompt with "your Mom looks like a dinosaur.", preloaded in window. Able to generate an output.

2 compared to 1:

Adds variable with proper math that is dependant on another variable.
Note use of variable names different then hints.
Adds alert complete with sentence, "the number or you suck my peepee is"

BUILD UPON PROGRAM

3 compared to 2:

Changes alert to more appropriate sentence. CLEAN UP

4 compared to 3:

Same NO CHANGE (other)

Overall comments:

The student seems to be entertaining himself during programming. Demonstrates a constant build with a consistent attitude. I would be curios to verify hint levels on this one.

Analysis:
Student # 8

General impressions of first attempt:

Text book correct answer, same variables as the hints.

2 compared to 1:

Same NO CHANGE (other)

3 compared to 2:

Addition of space in alert between variable and written sentence.
CLEAN UP

Overall comments:

If a student starts with a correct answer, there is little change to track. This is the second student, possibly the third up to this point who arrive at the answer curiously quick.

Analysis:

Student # 9

General impressions of first attempt:

No attempt was made. This is the only student out of the received data that had not made it as far as 3-5-1. I still included this student in the data because this student was a part of the test group. When figuring out mean, median, and mode of attempts of the test group, removing the two highest and two lowest numbers made little difference on the answers.

Overall comments:

See Appendix E1 for additional information

Analysis:

Student # 10

General impressions of first attempt:

Variable established with content of "Your mom is so fat"
A string is established with no content.
No output would be generated from this attempt.

2 compared to 1:

Document.write added. BUILD UPON PROGRAM

3 compared to 2:

Addition of prompt to initial variable as well as more appropriate sentence.
BUILD UPON PROGRAM

4 compared to 3:

Deletion of string variable. Deletion of document.write . Added alert.
CLEAN UP
BUILD UPON PROGRAM

5 compared to 4:

Changes wording of prompt for what would show above entry window.
CLEAN UP

6 compared to 5:

Adds "+" to alert to fix math syntax in alert.

SYNTAX (general confusion)

7 compared to 6:
Changes what would appear in "fill in" window on prompt.
CLEAN UP

8 compared to 7:
Same NO CHANGE (other)

9 compared to 8:
Same NO CHANGE (other)

10 compared to 9:
Same NO CHANGE (other)

Overall comments:

I would imagine a hint level was used in 4th attempt. Otherwise a standard build up.

Analysis:
Student # 11
General impressions of first attempt:
Prompt is aesthetically well done. Variable has hard math value rather than one dependant on variable. Appears to be based of hint with similar variable names.

2 compared to 1:
Change in formula to be dependant on variable.
LOGIC CHANGE

Overall comments:
Another quickly solved problem, I believe changing from a hard math formula to a formula dependant on a variable is a change in logic. With this student starting out with a close copy of a hint, its hard to believe there was a sudden change in reasoning. More likely is a better job of following a given hint.

Analysis:
Student # 12

General impressions of first attempt:

Started with a variable prompt and an empty string with a comment identifying it as an empty string.

2 compared to 1:

Shortened prompt sentence. CLEAN UP

Variable Change that is equivalent to another variables length (that does not exist yet). divided by five. SUDDEN CHANGE (other)

Addition of an alert statement.

3 compared to 2:

Change of variable name again with corresponding change in formula that uses the variable. No difference in output.

CLEAN UP

4 compared to 3:

A change made to variable name in prompt now makes it correlate to formula in another variable. Note that name matches hint name.

GRAMMAR (variable name)

5 compared to 4:

Shortened Prompt sentence again. CLEAN UP

Overall comments:

I would imagine a use of a hint in the sudden change, but because of misnamed variables from the hint, it could be some other change.

Analysis:
Student # 13

General impressions of first attempt:

Unlike any other students first attempt so far. It appears to be a baseball related program that changes the case of the letters. Obviously copied from somewhere. Student is trying to find a similar program to what is being asked, and plans to cut it up and modify it as needed to achieve the desired results. This is rather like reducing a block of wood to a sculpture. This is an example of another strategy that students utilize

2 compared to 1:

Eliminates a zero in second to last line, as well as a pair of single quotes.

CLEAN UP

3 compared to 2:

Eliminates Upper and Lower case part of code as well as the sentence that would be a repeat if left in. CLEAN UP

4 compared to 3:

Eliminates two large blocks of program. SUDDEN CHANGE (other)

5 compared to 4:

Eliminates one break from a double break. CLEAN UP

6 compared to 5:

Adds a sentence to a variable. Sentence added will be generated on output. OTHER

7 compared to 6:

Adds two variables, both related to length. BUILD UPON PROGRAM

8 compared to 7:

Changes name of first variable, and adds a sentence to the variable that resembles the question at hand. Partially eliminates another variable. Changes a variable in a doc.write formula to match new name of first variable. SUDDEN CHANGE (other)

9 compared to 8:

Creates a new undefined variable and eliminates two variables related to length. SUDDEN CHANGE (other)

10 compared to 9:

Restores previously deleted variables related to length. SUDDEN CHANGE (Return previous code)

11 compared to 10:

Capitalized a variable's first letter in a formula, although variable referenced does not yet exist. GRAMMAR (case sensitivity)

12 compared to 11:

Eliminates two variables that reference length, and adds a space in the after an undefined variable's name. CLEAN UP

13 compared to 12:

Added a document.write that references another variable correctly and determines that variables length correctly. BUILD UPON PROGRAM

14 compared to 13:

Removed space after single quotation last line. CLEAN UP

15 compared to 14:

Changed sentence in document write output. CLEAN UP

16 compared to 15:

Same. NO CHANGE (other)

17 compared to 16:

Addition of parentheses with correct length divide by five formula in document.write statement. BUILD UPON PROGRAM

18 compared to 17:

Same. NO CHANGE (other)

Overall comments:

This seemed a particularly painful journey to a solution. It is difficult to say where and if hints were used. I did find myself wondering how clear the question was that started this journey. This particular answer does not use a prompt. Perhaps the question should be specific about using a prompt. Perhaps this just demonstrates that the reduction method of programming is not very efficient.

Analysis:

Student # 14

General impressions of first attempt:

Another long copied program probably from the text of chapter. Appears to be another attempt of reducing a large program to met the needs of the question.

2 compared to 1:

Changes sentence in prompt so that prompt window will default to display the sentence "Stephanie wines to much about everything…" .

CLEAN UP

3 compared to 2:
Cuts notes, functions, for loop, and document.write reverse.
SUDDEN CHANGE (other)

4 compared to 3:
Added word "length" to variable named in formula, variable does exist.
Added word "array" to variable named sentence, variable does exist.
BUILD UPON PROGRAM

5 compared to 4:
Reversed order of two lines.
CLEAN UP

6 compared to 5:
New final sentence in document.write, uses quotes.
CLEAN UP

7 compared to 6:
Added document.write command referring to a proper variable.
BUILD UPON PROGRAM

8 compared to 7:
Changes sentence in prompt window to test program.
OTHER

9 compared to 8:
Same NO CHANGE (other)

Overall comments:
Student built a program that seems to generate the proper output in a sophisticated manner different from the given solution. I would have to admit that I don't get how this program works, but I believe it is counting spaces between words. It does not count every fifth character as it should. Even if this is wrong, I can appreciate the journey in problem solving.

EX: Final Program

```
// Declare variables to be used by this program
var sentence = '';
var sentence_length = 0;
var sentence_array = new Array();
```

```
sentence = prompt('Enter any sentence: ',
'Jack and Jill went up the hill to fetch a pail of water...');

sentence_array = sentence.split(' ');

sentence_length = sentence_array.length;
document.write(sentence + '<br><br>');
document.write("The number of words in this sentence is " + sentence_length +
'<br><br>');
```

Analysis:
Student # 15

General impressions of first attempt:

Student has copied example and will try to reduce this program to meet the question at hand.

2 compared to 1:

Same — NO CHANGE (other)

3 compared to 2:

Cuts five document.writes and a variable. — SUDDEN CHANGE (other)

4 compared to 3:

Cuts rest of program, and creates document.write related to the matter at hand.

SUDDEN CHANGE (other)

5 compared to 4:

Changes variable name in line one and adds prompt with appropriate sentence in same line. Makes change in document.write by adding proper variable to formula. Makes additional change in final document.write with proper length/5 element.

BUILD UPON PROGRAM

6 compared to 5:

Same — NO CHANGE (other)

7 compared to 6:

Same — NO CHANGE (other)

Overall comments:

Student was efficient in using reduction method. I would be curious to see hint levels used. I suspect hints may not have been used.

Analysis:
Student # 16

General impressions of first attempt:

Four line code using variables called string. One line splits string. Document write produces string length.

2 compared to 1:

Added zero in brackets to string variable. BUILD UPON PROGRAM

3 compared to 2:

Changed all variable names, added a a variable equivalent of sentence.split (a properly defined variable in program). Added an output alert with length function.

SUDDEN CHANGE (other)

4 compared to 3:

Same NO CHANGE (other)

5 compared to 4:

Added double Quotes. SYNTAX (mis-quote)
Removed parentheses from alert. SYNTAX (parentheses)

6 compared to 5:

Added prompt and changed sentence in parentheses.
Added sentence to alert. BUILD UPON PROGRAM

7 compared to 6:

Same NO CHANGE (other)

Overall comments:

Although student would generate an output, it would not be correct. It would display the number of characters in a string. Very close, needs the divide by five after the length function.

Analysis:
Student # 17

General impressions of first attempt:

Two lines with out put.

2 compared to 1:

Adds a prompt, drops an alert. Adds a document.write with proper formula for length with in a sensible sentence. BUILD UPON PROGRAM

3 compared to 2:

Deletes an apostrophe due to single quote. SYNTAX (mis-quote)

4 compared to 3:

Adds an array with a split. Adds new document.write using the added array. BUILD UPON PROGRAM

5 compared to 4:

Cuts original document.write that used the .length function. CLEAN UP

6 compared to 5:

Changes sentence in final line of document.write. CLEAN UP

7 compared to 6:

Same NO CHANGE (other)

Overall comments:

The split method shows that the student is actively trying to pull information from the chapter. Unfortunately it is not the answer to this problem. Could the question be written in such a way that the student puts a predetermined sentence in the prompt to arrive at a specific output. Many other questions are like this. Hints were also not used by this individual (assumed) , What if the first hint had no points taken off? Would this facilitate a hint in right direction being used?

Analysis:

Student # 18

General impressions of first attempt:

6 lines of code. Initial code utilizes split function.

2 compared to 1:

Adds break to last line of document.write. CLEAN UP

3 compared to 2:
Adds an "s" to the word sentence in final doc.write
CLEAN UP

4 compared to 3:
Changes
 to <p> CLEAN UP

5 compared to 4:
Added Parentheses to Array SYNTAX (parentheses)

Overall comments:
Another student uses split due to no way to check answer. A particular sentence with some short words and many spaces that has to be entered into the prompt to determine if it matches some specific number given in the question would be helpful to this situation.

Analysis:
Student # 18
General impressions of first attempt:
Copy of a program dealing with track. Another student tries to whittle down a program so as to match the question at hand.

2 compared to 1:
Deletes lines that deal with upper and lower case.
CLEANS UP

3 compared to 2:
Deletes 5 unneeded document write lines. CLEANS UP

4 compared to 3:
Deletes last six lines of program. SUDDEN CHANGE (other)

5 compared to 4:
Deletes document.write CLEAN UP
Adds HTML code to Justify word track in current last line.
SYNTAX (language confusion)

6 compared to 5:
Changes initial variable and installs a sentence related to problem at hand.

Changes document write to create output of above variable installed.
First time code resembles problem being attempted.
OTHER

7 compared to 6:
Shortened initial variable sentence.
Added variable called Programming
BUILD UPON PROGRAM

8 compared to 7:
Copied broccoli example from problem, all previous program gone.
SUDDEN CHANGE (other)

9 compared to 8:
All new program, broccoli is gone. Seems correct except missing divide by five component.
SUDDEN CHANGE (other)

10 compared to 9:
Adds additional break to document.write.
Adds a document .write to final line. BUILD UPON PROGRAM

11 compared to 10:
Adds parentheses to final document.write that contain the .length/5 function. BUILD UPON PROGRAM

12 compared to 11:
Cuts a variable, and a single quote followed by a plus sign in final document.write. This creates a misquote and general confusion. I will call this other for now. OTHER

13 compared to 12:
Added word "Example" to initial variable string.
Added quote and plus back in. SYNTAX (misquote)
SYNTAX (general confusion)
Added alert with .length/5 function BUILD UPON PROGRAM

14 compared to 13:
Added new variable with prompt.
Added new alert line. BUILD UPON PROGRAM

15 compared to 14:

Code drops to two lines, new variable is used. Alert with proper .length/5 with addition sentence added. SUDDEN CHANGE (other)

16 compared to 15:

Added a document.write that has same out put as alert.
BUILD UPON PROGRAM

Overall comments:
A long trip to a solution with many changes along the way.

Analysis:
Student # 20

General impressions of first attempt:

First attempt is a copy of a modified broccoli example.

2 compared to 1:

Broccoli goes away and is replaced three lines of code. Code has hard math rather than variable dependant math formula.
SUDDEN CHANGE (other)

3 compared to 2:

Moves single quote mark in final line. CLEAN UP

4 compared to 3:

Adds a period in final quote. CLEAN UP

5 compared to 4:

Shortens initial prompt.
Changes alert sentence (still not correct) CLEAN UP

6 compared to 5:

Changes variable equivalent to proper formula sentence.length/5.
SYNTAX (general confusion)

7 compared to 6:

Adds single quotes to separate prompt components.
SYNTAX (misquote)
Changed alert sentence, still not correct. BUILD UPON PROGRAM

8 compared to 7:

Adds a period in prompt default sentence, Added single quote to middle of alert. BUILD UPON PROGRAM

9 compared to 8:

Adds a plus after previously added single quote.

SYNTAX (general confusion)

10 compared to 9:

Adds additionional single quote to alert SYNTAX (mis-quote)

11 compared to 10:

Removes quotes from none sentence, variable reference in final line.

SYNTAX (mis-quote)

Overall comments:

Another long journey to a solution, with much learning along the way.

Analysis:

Student # 21

General impressions of first attempt:

Appears to be hint one code with incorrect characters filling in the question mark.

2 compared to 1:

Changes alert code to reference a variable. BUILD UPON PROGRAM

3 compared to 2:

Changes wording of prompt window. CLEAN UP

Changes variable in alert code to hard number.

SYNTAX (general confusion)

4 compared to 3:

Changes alert to document.write. CLEAN UP

5 compared to 4:

Declares a variable with no content, and moves prompt line but maintains variable used with prompt that is now established at beginning of program.

Creates new variable. OTHER

Changes content of prompt to "'Enter a sentence that contains the word hi:'

' I said hi to the bum instead of giving him a handout.'", This test for an out put, but more importantly he puts a formula in that figures out the length of characters an a variable.

BUILD UPON PROGRAM

6 compared to 5:

Changes variable value. Creates new variable with hard math formula rather than one dependant on variable value.

BUILD UPON PROGRAM

7 compared to 6:

Adds alert with a hard number. BUILD UPON PROGRAM

8 compared to 7:

Same NO CHANGE (other)

9 compared to 8:

Cut out variable that determined sentence length.

CLEAN UP

10 compared to 9:

Major change. Eliminates a variable that determines sentence length, as well as a variable used to figure out number of words with hard numbers. Installs a new sentence in prompt (now in first line as in attempt 1) that is identical to the highlighted words in the question. Two new out puts replace the alert. One of the out put s uses the .length code.

SUDDEN CHANGE (other)

11 compared to 10:

Deletes one of the document.write installed in previous attempt.

CLEAN UP

12 compared to 11:

Same NO CHANGE (other)

13 compared to 12:

Same NO CHANGE (other)

14 compared to 13:

Reestablishes document.write previously deleted.

CLEAN UP

15 compared to 14:

Same NO CHANGE (other)

16 compared to 15:

Adds a plus sign document.write. SYNTAX (general confusion)

17 compared to 16:

Same NO CHANGE (other)

18 compared to 17:

Adds divide by five to .length component in final document write

BUILD UPON PROGRAM

19 compared to 18:

Shortens prompt wording CLEAN UP

20 compared to 19

Same. NO CHANGE (other)

Overall comments:

This student had the longest journey to a working solution, but finally made it. Final answer is different from hint solution, but contains similar components.

Analysis:

Student # 22

General impressions of first attempt:

Two lines, first one establish a variable with sentence highlighted in the problem. Second line determines variable length and out puts through document.write.

2 compared to 1:

Student begins to play with split function. A variable called split is declared and is equivalent to a newArray. Split is then made equivalent to sentence.split . Document.write is changed to split.length.

BUILD UPON PROGRAM

3 compared to 2:

Adds a new variable called length that is equal to split.length. Changes document.write output to new variable length.

BUILD UPON PROGRAM

4 compared to 3:

Adds two document.write statements, one which writes the initial variable sentence. The other is installs a space in out put.

CLEAN UP

5 compared to 4:

Adds single quotes and a semi colon to document.write that creates a space. Drops "var" from line with a variable that is already declared.

SYNTAX (general confusion)

6 compared to 5:

Deleted a space in what should be the prompt sentence but is the initial variable.

CLEAN UP

7 compared to 6:

No change NO CHANGE (other)

Overall comments:

Another split method solution. Wording in question may need attention.

The students are asked to count the words in a sentence. Then the question asks them to write a program that tells them how many words are in the sentence. It would be easy for students to physically count the words as they appear rather than every five characters, then write a program that does the same. Perhaps the "chunks of five" reference needs to be repeated where it asks the student to write the program to draw more attention to it.

Analysis:

Student # 23

General impressions of first attempt:

Copy of example regarding broccoli soup.

2 compared to 1:

Deletes document.write, adds returns to layout program better per code line. When copied some of the lines blended together.

Basically reduces program. CLEAN UP

3 compared to 2:

Deletes large chunk of program SUDDEN CHANGE (other)

4 compared to 3:

Changes variable broccoli soup to beef soup.

Deletes document.write CLEAN UP

5 compared to 4:

Same NO CHANGE (other)

6 compared to 5:

Deletes 'words' from document.write sentence out put.

CLEAN UP

7 compared to 6:

Corrects a variable name in document.write to match an existing variable established in first line of code. (beef soup) SYNTAX (function parameters)

8 compared to 7:

Adds divide by five function to document write.

BUILD UPON PROGRAM

9 compared to 8:

Replaces 'words' in out put of document.write

CLEAN UP

10 compared to 9:

Deletes 'words' again.

CLEAN UP

11 compared to 10:

reversed placement of two lines CLEAN UP

12 compared to 11:

Same NO CHANGE (other)

Overall comments:

This is a strange answer to the question at hand as it deals with beef soup.

The logic behind the program works, and ultimately it will generate a

correct answer based on the sentence used. Perhaps the question should

require a prompt?

Analysis:

Student # 24

General impressions of first attempt:

Five line code plus comments. Establishes two variables, creates a prompt and determines length of characters in prompt. Has out put of number of characters in prompt.

2 compared to 1:

Changes output of document.write to read sentence in prompt followed by "has" followed by number of characters based upon.length variable, followed by " in it".
CLEAN UP

3 compared to 2:
Adds strange use of double and single quotes in final document.write.
SYNTAX (mis-quotes)

4 compared to 3:
Replaces double quotes with parentheses. SYNTAX (parentheses)

5 compared to 4:
Adds "word" to out put. CLEAN UP

6 compared to 5:
Declares a new variable, and tries to manipulate it to split sentence length.
BUILD UPON PROGRAM

7 compared to 6:
Same NO CHANGE OTHER

Overall comments:
Another split attempt, unsuccessful. Does not work. Student had submitted this as a final and needed to be reset.

Analysis:
Student # 25
General impressions of first attempt:
Modified broccoli beginning. Demonstrates some work to answer question at hand. Interesting strategy of pasting example and modifying it to meet the question. Several students have used this as a way to arrive at answers.

2 compared to 1:
Deletes document.write. CLEAN UP

3 compared to 2:
Same NO CHANGE (other)

4 compared to 3:
Deletes a slash and return in variable sentence.
CLEAN UP

5 compared to 4:

Removes two document.write codes regarding upper and lower case letters not needed in this question. CLEAN UP

6 compared to 5:

Same NO CHANGE (other)

7 compared to 6:

Same NO CHANGE (other)

Overall comments:

Program will not give proper out put. If student was aware of proper output value, they may have continued working to solve issues.

Analysis Summary

I would say that programming is much like sculpture. You can take material and build it up into a construct, or you can take material and reduce it down to a construct. Each attempt is a journey to the end construct. Mistakes are a part of that journey and certainly a part of learning. In going through each of these students' journeys, I found I kept wondering how the journey was launched? I kept reconsidering the original question. Why were some students dividing by five and others using a split method? Why were students submitting an answer that was not reaching the goal of the question? I also found myself wondering about the aesthetics of the program solutions. Why were hints not used, and is this necessarily good or bad? I will now attempt to address each of these issues.

To begin I will address the fact that a lot of information is dumped on the student before attempting this question. I would consider it an overwhelming amount of

information to be able to narrow down the needed components of this particular question of 3-5-1. The example that students pull from is also long and has many sophisticated components that are not needed to answer this particular question. My immediate thought is that this chapter needs to be broken into smaller components, perhaps introducing a few string objects to work with at a time. The example needs to be simple utilizing one string object instead of several.

The way the question is worded is cute, but also vague. This is good as it allows the student to interpret it differently and immediately be creative in generating an answer. Although the question describes touch typing as counting words in chunks of five characters, when the question directs students to write a program is does not define to use the touch method. Another issue is that the example sentence is long. Upon reading the sentence, the student is asked to count the words in the sentence. Most students are going to count per word. Visually the long example sentence separates them far from the chunks of five character reference, which is easily forgotten after reading the sentence. After a student counts the words in a sentence, they are asked to write a program that tells them how many words are in the example sentence. Changing the write sentence to read, "Write a program that inputs a sentence and counts words in terms of five characters per word." may help the situation.

In referring to why students were submitting solutions that did not have correct answers, a possible solution would be to have several test sentences that could be put into the program that have a particular value the students had to match. Requiring the use of a prompt takes away from a creative factor, but is a logical solution to the question at hand. The input of the sentence was handled many ways by the students in this study. There

seemed no aesthetic concern on how the answer was arrived at, or how a sentence was put in, or what sentenced was put in. I would like for students to be aware of the aesthetic presentation of their codes, and demonstrate that they know how to control how a prompt appears. Requiring a prompt with specific information displayed would help this issue.

Another issue that may help students is to use hints. What if the first hint was free, and the remaining hints were ¾'s of a point? The pseudo code would certainly launch the journey in the right direction, but many students do not use the code as they lose points. Changing the value of the points may make the first code hint more welcoming to the students, and launch their journey better.

Other considerations would include that the managers window currently shows the solution when the hint levels are looked into, making it hard for the TA's to know how the use of a hint effected a students program. All the hints from the management screen show the solution, not the hint.

According to domains, students tended to clean up their work more than any other domain. I feel that is normal. Other domain counts do not seem unreasonable for students learning a computer language. This is part of the journey. Perhaps a good journey after this question would to be to adjust this program to round as one student had successfully done.

I believe the CS Online course is a very exciting option for students and schools to have available to them. The refinement this type of study offers in constructing CS Online can only serve to make it stronger.

SOURCE CODE KEY FOR 3-5-1

Attachment:	Date:	ICON:
researchQ5DataGroup2a 233k	may1st	☼
researchQ5DataGroup1a 725k	may1st	✠
researchQ5DataGroup1b 337k	may1st	❄
researchQ5DataGroup2b 85k	may1st	❖
researchQ5DataGroup2b 72k	may2nd	☒
researchQ5DataGroup1c 75k	may2nd	⌘
researchQ5DataGroup2c 49k	may5th	❍
researchQ5DataGroup2d 140k	may5th	◆
researchQ5DataGroup1d 56k	may5th	❁

Student	#1	ROJAF	☼	Attempts	8
Student	#2	STAMENKOVICM	☼	Attempts	9
Student	#3	RICHC	✠	Attempts	8
Student	#4	OTTD	✠	Attempts	11
Student	#5	BALINTNM	❄	Attempts	4
Student	#6	GLENNYC	❄	Attempts	1
Student	#7	HUTCHIN	❄	Attempts	4
Student	#8	JOHNSONG	❄	Attempts	3
Student	#9	MALISA	❄	Attempts	0
Student	#10	MCINTIRE	❄	Attempts	10
Student	#11	HOWARDL	❄	Attempts	2
Student	#12	IANID	❄	Attempts	5
Student	#13	JOSHPOWEL	❄	Attempts	18
Student	#14	TERRELLE	❄	Attempts	9
Student	#15	BALINTS	❖	Attempts	7
Student	#16	KELLER	❖	Attempts	7
Student	#17	KISS	❖	Attempts	7
Student	#18	LYONSA	❖	Attempts	5
Student	#19	LOF	☒	Attempts	16
Student	#20	MC	☒	Attempts	11
Student	#21	BOYDB	⌘	Attempts	20
Student	#22	WILLBEGR	◆	Attempts	7
Student	#23	SCHAFFERS	◆	Attempts	12
Student	#24	STATS	❍	Attempts	7
Student	#25	REIKO	❁	Attempts	7

From lowest to highest number of attempts would look like this

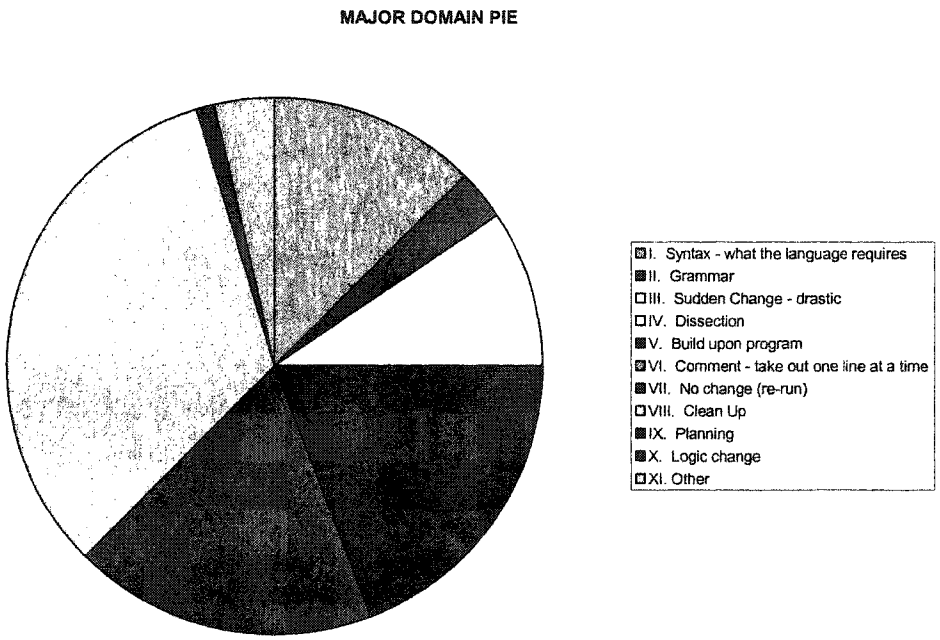
0,1,2,3,4,4,5,5,7,7,7,7,7,7,8,8,9,9,10,11,11,12,16,18,20 for a total of **198 attempts**

From this data we can determine the following;
Mean 7.92 attempts Median 7 attempts Mode 7 attempts

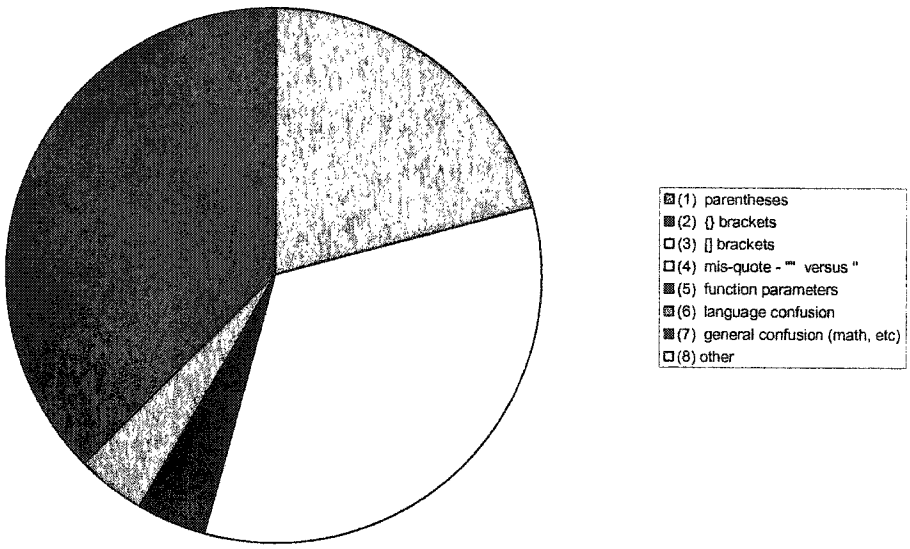
If we eliminate the two lowest and highest attempts
2,3,4,4,5,5,7,7,7,7,7,7,8,8,9,9,10,11,11,12,16, for a total of **159 attempts**
Mean 7.57 attempts Median 7 attempts Mode 7 attempts

This demonstrates little change in the outcome of the data, making the mean, median, and mode valid throughout wide variations of attempts of the students.

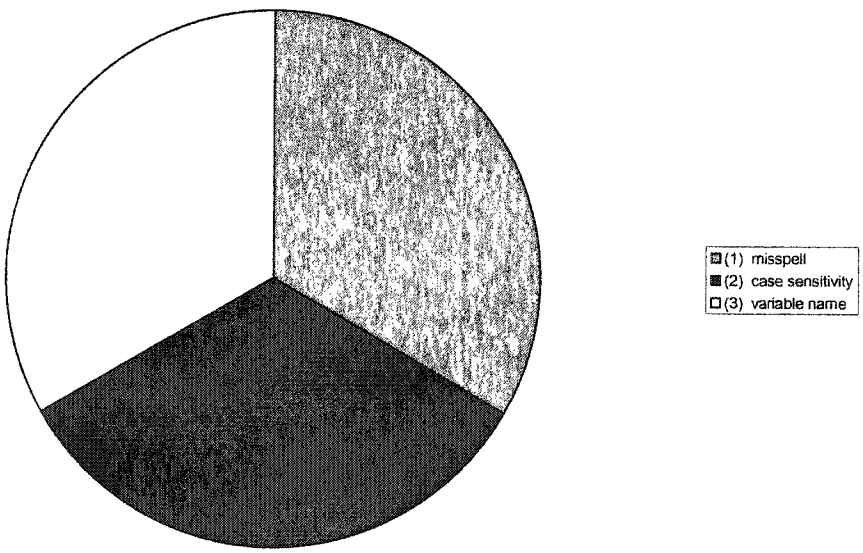
Appendix Y

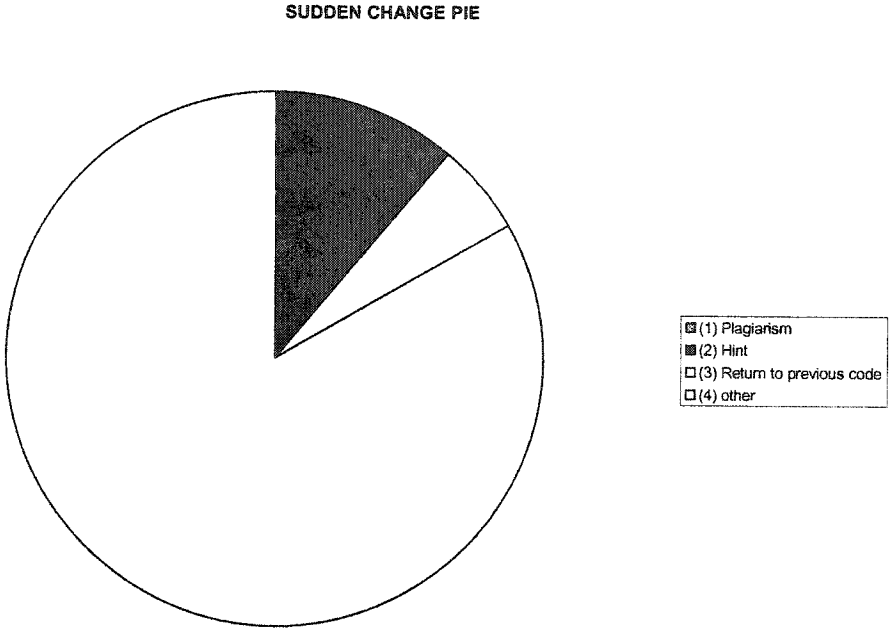


SYNTAX



GRAMAR PIE





REFERENCES

Allan, V. H. & Kolesar, M. V. (1996). Teaching computer science: A problem solving approach that works. Call of the North, NECC 1996. Proceedings of the Annual National Educational Computing Conference – Call of the North NECC, Minneapolis, MN, June 1996.

Anderson, J. R., Fincham, J. M., & Douglass, S. (1997). The role of examples and rules in the acquisition of a cognitive skill. *Journal of Experimental Psychology: Learning, Memory, and Cognition*, 23, 932-945.

Association for Computing Machinery (ACM). (1985a). Computer science for secondary schools: Course content. Communications of the ACM task force on curriculum for secondary school computer science, 28, 270-274.

Association for Computing Machinery (ACM). (1985b). Proposed curriculum for programs leading to teacher certification in computer science. Communications of the ACM task force on teacher certification in computer science, 28, 275-279.

Association for Computing Machinery (ACM). (1993). ACM task force of the pre-college committee: ACM model high school computer science curriculum. New York: ACM Press.

Association for Computing Machinery (ACM). (2003). ACM Education Task Force: Survey of the present state of computer science education in the United States. 2003.

Atkinson, R. K., Derry, S. J., Renkl, A., & Wortham, D. (2000). Learning from Examples: Instructional Principals from the Worked Examples Research. *Review of Educational Research, 70(2)*, 181-214.

Ayers, P. (1993). Why goal free problems can facilitate learning. *Contemporary Educational Psychology, 18*, 376-381.

Baylor, A. L., & Kozbe, B. (1998). A personal intelligent mentor for promoting metacognition in solving logic word puzzles. Paper presented at the Fourth World Congress on Expert Systems (Mexico City, Mexico, March 1998).

Bayman, P., & Mayer, R. E. (1988). Using conceptual models to teach BASIC computer programming. *Journal of Educational Psychology, 80(3)*, 291-98.

Becker, B.W., Graham, S. (2000). The university perspective on the new high school computer science curriculum. Available from http://www.math.uwaterloo.ca/%7Ecshsliai/Symposium. Presentation V3/sld001.htm.

Booth, S. (1990). Conceptions of programming: A study into learning to program. Goteborg Univ., Molndal (Sweden), Institute of Education.

Bruner, J. S. (1966). Toward a theory of instruction. New York: W. W. Norton.

Brusilovsky, P. (1994). Teaching programming to novices: A review of approaches and tools. Educational Multimedia and Hypermedia. Proceedings of ED-MEDIA 94 – World Conference on Educational Multimedia and Hypermedia, Vancouver, B.C., 1994.

Butler, D. L., & Winne, P. H. (1995). Feedback and self-regulated learning: A theoretical synthesis. *Review of Educational Research, 65*, 245-281.

Campbell, A. (1984). Vocational education in an information age: Society at risk? Occasional Paper No. 99. National Center for Research in Vocational Education, Ohio State University. Columbus, Ohio, 1984.

Casey, P. J. (1997). Computer programming: A medium for teaching problem solving. *Computers in the Schools, 13(1-2)*, 41-51.

Catrambone, R. (1994a). The effects of labels in example on problem solving transfer. In A. Ram, & K. Eiselt (Eds.). Proceedings of the Sixteenth Annual Conference of the Cognitive Science Society (pp. 159-164). Hillsdale, NF: Erlbaum.

Catrambone, R. (1994b). Improving examples to improve transfer to novel problems. *Memory & Cognition, 22*, 606-615.

Catrambone, R. (1995a). Aiding subgoal learning: Effects on transfer. *Journal of Educational Psychology, 87*, 5-17.

Catrambone, R. (1995b). Effects of background on subgoal learning. In J. D. Moore & J. F. Lehman (Eds.), Proceedings of the Seventeenth Annual Conference of the Cognitive Science Society (pp. 259-264). Hillsdale, NJ: Erlbaum.

Catrambone, R. (1996). Generalizing solution procedures learned from examples. Journal of Experimental Psychology: *Learning, Memory, and Cognition, 22*, 1020-1031.

Chandler, P., & Sweller, J. (1991). Cognitive load theory and the format of instruction. *Cognition and Instruction, 8*, 293-332.

Chandler, P., & Sweller, J. (1992). The split attention effect as a factor in the design of instruction. *British Journal of Education Psychology, 62*, 233-246.

Chandler, P., & Sweller, J. (1996). Cognitive load while learning to use a computer program. *Applied Cognitive Psychology, 10*, 151-170.

Chi, M. T., Bassok, M., Lewis, M. W., Reimann, P., & Glaser, R. (1989). Self-explanations: How students study and use examples in learning to solve problems. *Cognitive Science, 13*, 145-182.

Chi, M., Glaser, R., & Rees, E. (1982). Expertise in problem solving. In Sternberg, R. (ed.), Advances in the Psychology of Human Intelligence, Erlbaum, Hillsdal, NJ, pp. 7-75.

Coad, P., & Yourdon, E. (1993). Object-oriented programming (second edition). Englewood Cliffs, NJ: Prentice Hall, 1993.

College Board, (2003a). Computer Science A Overview. Available from http://apcentral.collegeboard.com/article/0,1281,151-162-0-4345,00.html.

College Board, (2003b). Advanced placement program course description: Computer Science. New York: The College Board, May 2003. Available from http://www.collegeboard.com/ap/pdf/cd_computer_science_03.pdf.

Colorado Department of Education (CDE). (1996). Colorado model content standards. Denver, CO: Department of Education.

Connolly, M.V. (1996). Starting computer science using C++ with objects: A workable approach. Proceedings from the 1996 ASCUE Conference.

Cooper, G. (1998). Research into cognitive load theory and instructional design at UNSW. Available from http://www.arts.unsw.edu.au/education/CLT_NET_Aug_97.HTML.

Dalbey, J., & Linn, M. C. (1985). The demands and requirements of computer programming: A literature review. *Journal of Educational Computing Research, 1(3)*, 253-274.

Dalton, D. W., & Goodrum, D. A. (1991). The effects of computer programming on problem-solving skills and attitudes. *Journal of Educational Computing Research, 7(4)*, 483-506.

Deek, F. P., & Kimmel, H. (Eds.). (1998). Computer science education in the secondary schools: Curriculum guidelines, content and professional development. Proceedings of the 1995, 1996 and 1997 conferences. Newark, NJ: New Jersey Institute of Technology.

Deek, F. G., & Kimmel, H. (1999). Status of computer science education in secondary schools: One state's perspective. *Computer Science Education, 9(2)*, 89-113.

Delcros, V. R., & Burns, S. (1993). Mediational elements in computer programming instruction: An exploratory study. *Journal of Computing in Childhood Education, 4(2)*, 137-52.

Dillon, A., & Gabbard, R. (1998). Hypermedia as an educational technology: A Review of the Quantitative Research Literature on Learner Comprehension, Control, and Style. *Review of Educational Research, 68*, 322-349.

DuPoint, A. P. (1998). Technology Night. *Education Leadership, 55(8)*, 74-75.

Education Week, (2002). E-Defining education. *Education Week, 21(35)*, 1-11.

Foreman, K. H. (1990). Cognitive characteristics and initial acquisition of computer programming competence. *School of Education Review, 2*, 55-61.

Gesler, W., & Kaplan, A. (1993). Computer programming in a spatial analysis course. *Journal of Geography, 92(3)*, 139-145.

Glaser, R. (1976). Components of a psychology of instruction: Toward a science of design. *Review of Educational Research, 46*, 1-24.

Goktepe (1985). Design and implementation of a tool for teaching programming. Computer and Education, 3rd Edition. New York: McGraw-Hill.

Goldenson, D. (1996). Why teach computer programming? Some evidence about generalization and transfer. Call of the North, NECC '96. Proceedings of the Annual National Educational Computing Conference, Minneapolis, Minnesota, June, 1996.

Google (2003). Web definitions. Support.sbcglobal.net/general/662.shtml

Greenburg, G. (1991). A creative arts approach to computer programming. *Computers and the Humanities, 25(5)*, 267-273.

Haajanen, J., Pesonius, M., Sutinen, E., Tarhio, J., Terasvirta, T. & Vannien, P. (1997). Animation of user algorithms on the web. Proceedings from the Visual Languages '97 Conference, Capri, Italy.

Haataja, A., Suhonen, J., Sutinen, E., & Torvinen, S. (2001). High school students learning computer science over the web. IMEJ Article, Wake Forest University. Imej.wfu.edu/articles/2001/2/04/index.asp.

Hadjerrouit, S. (1998). Java as first programming language: a critical evaluation. SIGSCE Bulletin, June 1998.

Hancock, C. (1988). Context and creation in the learning of computer programming. *For the Learning of Mathematics, 8(1)*, 18-24.

Hartley, K., & Bendixen, L. (2000). Learning with hypermedia: The role of epistemological beliefs and self-regulation. Society for Information Technology & Teacher Education International Conference: Proceedings of SITE 2000 (11th, San Diego, California, February 8-12, 2000). Volumes 1-3.

Hartley, K., & Bendixen, L. D. (2001). Educational research in the Internet age: Examining the role of individual characteristics. *Educational Researcher, December 2001*, 22-26.

Hong, E. (1998). Differential stability of state and trait self-regulation in academic performance. *The Journal of Educational Research, 91(3)*, 148-158.

Hong, E. (2001a). Construct validation of a trait self-regulation model. *International Journal of Educational Psychology, 36(3)*, 186-194.

Hong, E. (2001b). Self-Assessment Questionnaire. University of Nevada, Las Vegas.

Hong, E. (2003). Math experience questionnaire. University of Nevada, Las Vegas.

Hong, E., & Halopoff, G. (2003). Computer experience questionnaire. University of Nevada, Las Vegas.

ISTE Accreditation Committee (1992). Proposed NCATE curriculum guidelines for the specialty area of educational computing and technology: Proposal to NCATE, Eugene, OR: ISTE.

Jang, Y. (1992). Cognitive transfer of computer programming skills and analogous problem solving. Paper presented at the annual conference of the American Educational Research Association (San Francisco, April 20-24, 1992).

Johnson, J. A., & Johnson, G. M. (1992). Student characteristics and computer programming competency: A correlational analysis. *Journal of Studies in Technical Careers, 14(1)*, 33-46.

Jones, P. K. (1988). The effect of computer programming instruction on the development of generalized problem solving skills in high school students. Ed.D. Practicum, Nova University.

Kelley, A. (1994). Is programming dead in teacher education? *Journal of Computing in Teacher Education, 11(1)*, 19-22.

Kolling, M. (2000). BlueJ – The interactive Java environment. www.bluej.org.

Kurland, D. M. (Ed.). (1984). Developmental studies of computer programming skills. A Symposium: Annual Meeting of the American Educational Research Association, 1984.

Kushan, B. (1994). Preparing programming teachers. ACM SIGSCE Bulletin, 26, 248-252.

Kynigos, C. (1993). Children's inductive thinking during intrinsic and euclidean geometrical activities in a computer programming environment. *Educational Studies in Mathematics, 24(2)*, 177-197.

Lai, K. (1993). Lego-Logo as a learning environment. *Journal of Computing in Childhood Education, 4(3-4)*, 229-245.

Lai, S. Repman, J. L. (1996). The effects of analogies and mathematics ability on students' programming learning using computer-based learning. *International Journal of Instructional Media, 23(4)*, 355-364.

Lanza, A., & Roselli, T. (1991). Effects of the hypertextual approach versus the structured approach on active and passive learners. *Journal of Computer-Based Instruction, 18(2)*, 48-50.

Lehrer, R., Lee, M., & Jeong, A. (1999). Cognitive consequences of LOGO with two different teaching methods. Reflective teaching of Logo. *Journal of the Learning Sciences, 8(2)*, 245-89.

Levesque, K., & Hudson, L. (2003). Trends in Introductory Technology and Computer-Related Coursetaking. U. S. Department of Education Institute of Educational Science. July, 2003. NCES 2003-025.

Liao, Y. C. (1990). Effects of computer programming on students' cognitive performance: A quantitative synthesis.

Liu, M. (1998). The effect of hypermedia authoring on elementary school students' creative thinking. *Educational Computing Research, 19 (1)*, 27-51, 1998.

Logo Foundation (2003). Available from http://el.www.media.mit.edu/logo-foundation/logo/index.html.

Lynch, P., & Horton, S. (1997). Yale center for advanced instructional media style guide. Available from http://info.med.yale.edu/caim/manual/contents.html.

Madison, S. K. (1995). A study of college students' construct of parameter passing implications for instruction. Ed.D. Dissertation, University of Wisconsin-Milwaukee.

Martin, M. (1997). Homeschooling: Parents' Reactions. (Report No. 141). Washington, DC: U.S. Department of Education.

Martin, P. (1998). Java, the good, the bad and the ugly. SIGPPLAN Notices, April 1998.

Martin, B., & Hearne, J. D. (1990). Transfer of learning and computer programming. *Educational Technology, 30(1)*, 41-44.

McCoy, L. P, & Dodl, N. R. (1989). Computer programming experience and mathematical problem solving. *Journal of Research on Computing in Education, 22(1)*, 14-25.

McCoy, L. P. (1988). General variable skill, computer programming and mathematics. Paper presented at the Annual Meeting of the International Association for Computing in Education (New Orleans, LA, April 1988).

McCoy, L. P. (1990). Literature relating critical skills for problem solving in mathematics and in computer programming . *School Science and Mathematics, 90(1)*, 48-60.

Microsoft (2003). Script Debugger. Available from http://www.microsoft.com/downloads

Milbrandt, G. (1995). Using problem solving to teach a programming language. *Learning and Leading with Technology, 23(2)*, 27-31.

Miller, G. A. (1956). The magical number seven, plus or minus two: Some limits on our capacity for processing information. *Psychology Review, 63*, 81-97.

Miller, R. B. (1988). Effects of Logo computer programming experience on problem solving and spatial relations ability. *Contemporary Educational Psychology, 13(4)*, p348-357.

Mousavi, S. Y., Low, R., Sweller, J. (1995). Reducing cognitive load by mixing auditory and visual presentation modes. *Journal of Educational Psychology, 87(2)*, 319-334.

Mwangi, w., & Sweller, J. (1998). Learning to solve compare word problems: The effect of example format and generating self-explanations. *Cognition and Instruction, 16*, 173-199.

National Council for Geographic Education (NCGE). (1994). National geography standards. Washington, DC.

National Council of Teachers of Mathematics (NCTM). (1989). Curriculum and evaluation standards for school mathematics. Reston, VA.

National Research Council (NRC). (1996). National science education standards. Washington, DC: National Academy Press.

Netscape (2000). Core JavaScript Guide 1.5. Available from http://devedge.netscape.com/library/manuals/2000/javascript/1.5/guide/intro.html #1012569.

Nevada Department of Education (NDE). (2000). Nevada Computer and Technology Education Standards. Carson City, NV: Department of Education. www.nde.state.nv.us/sca/standards.

New Jersey Department of Education (NJDE). (1994). Core curriculum content standards. Tenton, NR: Department of Education.

New York Department of Education (NYDE). (1994). Curriculum, instruction, and assessment framework for mathematics, science, and technology. Albany, NY: Education Department.

Oprea, J. M. (1988). Computer programming and mathematical thinking. *Journal of Mathematical Behavior, 7(2)*, 175-90.

Ormrod, J.E. (1999). Human Learning. New Jersey: Merrill.

Owen, E., & Sweller, J. (1985). What do students learn while solving mathematics problems. *Journal of Educational Psychology, 77*, 272-284.

Paas, F. (1992). Training strategies for attaining transfer of problem-solving skill in statistics: A cognitive load approach. *Journal of Educational Psychology, 84*, 429-434.

Paas, F., & Van Merrienboer, J. (1994). Variability of worked examples and transfer of geometrical problem-solving skills: A cognitive approach. *Journal of Educational Psychology, 86*, 122-133.

Paloff, R. M., & Pratt, K. (2001). From lessons from the cyberspace classroom: The realities of online teaching, San Francisco, CA: Jossey-Bass: A Wiley Company.

Palumbo, D. L., & Palumbo, D. B. (1993). A comparison of the effects of Lego TC Logo and problem solving software on elementary students' problem solving skills. *Journal of Computing in Childhood Education, 4(3-4)*, 307-323.

Papert, S. (1980). Mindstorm, children, computers and powerful ideas. New York: Basic Books.

Prichard, M. K. (1993). Mathematical iteration through computer programming. *Mathematics Teacher, 86(2)*, 150-56.

Quilici, J. L., & Mayor, R. E. (1996). Role of examples in how students learn to categorize statistics word problems. *Journal of Educational Psychology, 88*, 144-161.

Reed, S. K., & Bolstad, C. A. (1991). Use of examples and procedures in problem solving. *Journal of Experimental Psychology: Learning, Memory, and Cognition, 17*, 753-766.

Reed, W. & Liu, M. (1992). The comparative effects of BASIC programming versus HyperCard programming on problem-solving, computer anxiety, and performance. Paper presented at the Annual Conference of the Eastern Educational Research Affiliation, Hilton Head, South Carolina, 1992.

Ridley, D. S., Schutz, P. A., Glanz, R. S., & Weinstein, C. E. (1992). Self-regulated learning: The interactive influence of metacognitive awareness and goal setting. *Journal of Experimental Education, 60*, 293-306.

Roberts, E. (2003). Resources to Support the Use of Java in Introductory Computer Science. Special session proposal to the Special Interest Group for Computer Science Education (SIGSCE) of the Association for Computing Machinery (ACM). 2003.

Savitch, W. (2003). Java: An introduction to computer science and programming (third edition). Englewood Cliffs, NJ: Prentice Hall, 2003.

Schneider, W., & Shiffrin, R. (1977). Controlled and automatic human information processing: I. Detection, search and attention. *Psychology Review, 84*, 1-66.

Schraw, G. (1998). Promoting general meta-cognitive awareness. *Instructional Science, 26*, 113-125.

Schraw, G., & Moshman, D. (1995). Metacognitive theories. *Educational Psychology Review, 7*, 351-371.

Sebesta (1996). Concepts of programming languages. Menlo Park, CA: Addison Wesley.

Seidman, R. H. (1990). Computer programming and logical reasoning: Unintended cognitive effects. *Journal of Educational Technology Systems, 18(2)*, 123-141.

Shih, Y., & Alessi, S. M. (1994). Mental models and transfer of learning in computer programming. *Journal of Research on Computing in Education, 26(2)*, 154-175.

Shin, E., Schallert, D., & Savenye, C. (1994). Effects of learner control, advisement, and prior knowledge on young students' learning in a hypertext environment. *Educational Technology Research and Development, 42*, 33-46.

Simon, H., & Gilmartin, K. (1973). A simulation of memory for chess positions. *Cognitive Psychology, 5*, 29-46.

Spradley, J.P. (1980). Participant Observation. Fort Worth: Harcourt Brace College Publishers.

Stark, R. (1999). Lernen mit Losungsbeispielen. Der EinfluB unvollstandiger Losungsschritte auf Beispielelaboration, Motivation und Lernerfolg [Learning by worked-out examples. The impact of incomplete solution steps on example elaboration, motivation, and learning outcomes]. Bern, CH: Huber.

Stephenson, C. (1997). Revitalizing high school computer science: Finding common ground? NECC '97 Proceedings. Seattle, WA: National Education Computing Conference.

Stephenson, C., and West, T. (1998). Language choice and key concepts in introductory computer science courses. *Journal of Research on Computing in Education, 31(1), 89-95.*

Stephenson, C. (2002). Java engagement for teacher training: Proposal for a pilot project to help local secondary computer science teachers develop expertise in Java programming. ACM memorandum, August 2002.

Sun Microsystems (2003). Java 2 Platform, Standard Edition (J2SE), v1.4.2 API specification. Available from http://java.sun.com/j2se/1.4.2/docs/api.

Suomala, J. (1996). Eight-year-old-pupils' problem-solving processes within a LOGO learning environment. *Scandinavian Journal of Educational Research, 40(4)*, 291-309.

Swan, K., & Black, J. B. (1993). Knowledge-based instruction: Teaching problem solving in a Logo learning environment. *Interactive Learning Environments, 3(1)*, 17-53.

Sweller, J. (1994). Cognitive technology: Some procedures for facilitating learning and problem solving in mathematics and science. *Journal of Educational Psychology, 81*, 457-466.

Sweller, J., Chandler, P., Tierner, P., & Cooper, M. (1990). Cognitive load in the structuring of technical material. *Journal of Experimental Psychology; General, 119*, 176-192.

Sweller, J., Van Merrienboer, J. J. G., Paas, F. G. W. C (1998). Cognitive architecture and instructional design. *Educational Psychology Review, 10(3)*, 251-296.

Tarmizi, R. A., & Sweller, J. (1988). Guidance during mathematical problem solving. *Journal of Educational Psychology, 80*, 424-436.

Taylor, H. G., & Norris, C. A. (1988). Retraining pre-college teachers: A survey of state computing coordinators. ACM SIGSCE Bulleting, 20, 215-218.

Trafton, J. G., & Reiser, B. J. (1993). The contributions of studying examples and solving problems to skill acquisition. In M. Polson (Ed.), Proceedings of the Fifteenth Annual Conference of the Cognitive Science Society (pp. 1017-1022). Hillsdale, NJ: Erlbaum.

Tu, J., & Falgout, B. (1995). Teaching If-Then structures--An integrated approach. *Learning and Leading with Technology, 23(3)*, 26-28.

Tucker, A. (1996). Strategic directions in computer science education. *ACM Computing Surveys, 28*, 836-845.

Tucker, A. (2003). Toward a K-12 computer science curriculum. Concurrent session presentation at the 2003 NECC Conference. Seattle, WA., 2003.

Van Merrienboer, J. J. G. (1990a). Instructional strategies for teaching computer programming: Interactions with the cognitive style reflection-impulsivity. *Journal of Research on Computing in Education, 23(1)*, 45-53.

Van Merrienboer, J. J. G. (1990b). What cognitive science may learn from instructional design: A case study in introductory computer programming. Paper presented at the Annual Meeting of the American Educational Research Association (Boston, MA, April 16-20, 1990).

Van Merrienboer, J. J. G. (1997). Training complex cognitive skills: a four-component instructional design model for technical training, Educational Technology Publications, Englewood Cliffs, NJ.

Van Merrienboer, J. J. G., & Krammer, H. P. M. (1987). Instructional strategies and tactics for the design of introductory computer programming courses in high school. *Instructional Science, 16(3)*, 251-85.

Van Merrienboer, J. J. G., & Paas, F. G. W. C. (1990). Automation and schema acquisition in learning elementary computer programming: Implications for the design of practice. *Computers in Human Behavior, 6(3)*, 273-289.

Van Merrienboer, J. J. G., Jelsma, O., & Paas, F. G. W. C., (1992). Training for reflective expertise: a four-component instructional design model for training complex cognitive skills. *Educational Technology Resource Development, 40(2)*, 23-43.

Wallace, C., Martin, P., & Lang, B. (1997). Not whether Java but how Java. Paper presented at the Java in the Computing Conference, London, January 1997.

Ward, M., & Sweller, J. (1990). Structuring effective worked examples. *Cognition and Instruction, 7*, 1-39.

Webopedia (2003). Available from http://www.webopedia.com.

Wieschenberg, A. A. (1999). Logic via programming. Paper presented at the annual International Conference on Technology in Collegiate Mathematics (12th, San Francisco, CA, November 4-7, 1999).

Wilkerson, L., & Gijselaers, W.H. (1996). Bringing problem-based learning to higher education: Theory and practice. New directions in teaching and learning, Jossey-Bass Quarterly Sourcebooks, number 68. San Francisco: Jossey-Bass Publishers, 1996.

Williams, S. M., & Hmelo, C. E. (Eds.). (1998). Special Issue: Learning through problem solving. *The Journal of Learning Sciences, 7*.

Winne, P. H. (1995). Inherent details in self-regulated learning. *Educational Psychologist, 30*, 173-187.

Wirth, N. (2002). Computing science education: The road not taken. Proceedings of the 7th annual conference on innovation and technology in computer science education, Aarhus, Denmark, June 2002.

Zeidner, M., Boekaerts, M., & Pintrich, P. R. (2000). Self-regulation: Directions and challenges for future research. In: Handbook of Self-Regulation, San Diego, CA.: Academic Press.

Zimmerman, B. J. (1995). Self-regulation involves more than meta-cognition: A social cognitive perspective. *Educational Psychologist, 30*, 217-221.

Zimmerman, B. J. (2000). Attaining self-regulation: A social cognitive perspective. In Boekaerts, M., Pintrich, P. R., & Zeidner, M. (Eds.) Handbook of Self-Regulation. San Diego: Academic Press.

Zimmerman, B. J., & Bandura, A. (1994). Impact of self-regulatory influences on writing course attainment. *American Educational Research Journal, 31*, 845-862.

Zimmerman, B. J., & Risemberg, R. (1997). Self-regulatory dimensions of academic learning and motivation. In G. D. Phye (Ed.) Handbook of Academic Learning: Construction of Knowledge. San Diego: Academic Press.

VITA

Graduate College
University of Nevada, Las Vegas

Gregory Paul Halopoff

Home Address:
2019 Pinion Springs Drive
Henderson, Nevada 89074

Degrees:
Bachelor of Science, Electrical Engineering, 1982
University of California, Los Angeles

Master of Science, Electrical Engineering, 1986
University of Southern California

Dissertation Title: Development of Computer Science Online and Preliminary Validation of its Efficacy as an Instructional Environment

Dissertation Examination Committee:
Chairperson, Dr. Neal B. Strudler, Ph. D.
Committee Member, Dr. Randall A. Boone, Ph. D.
Committee Member, Dr. Kendall W. Hartley, Ph. D.
Committee Member, Dr. Eunsook Hong, Ph. D.