AN ABSTRACT OF THE DISSERTATION OF

Naeem Zaman for the degree of Doctor of Philosophy in Science Education presented on March 10, 2003. Title: Strategies Utilized in Computer Problem Solving and Object-Oriented Programming.

Abstract approved Redacted for Privacy

Margaret L. Niess

The purpose of this qualitative study was to describe how novice students solved computer programming problems in a beginning college level computer science (CS) course with an introduction to object-oriented programming (OOP) and what knowledge they obtained about OOP and computer problem solving (CPS) as a result of their experiences. Additionally, this descriptive study attempted to characterize the instruction provided to students in a beginning CS course as well as students' CPS strategies.

An introduction to computer science class at the college level was selected for the sample. One experienced instructor and four students participated in this study. Data were collected through classroom observations, interviews with the instructor and students, classroom documents and researcher's journals.

The analysis of the results revealed a teacher-centered instruction focused on syntactical details with an emphasis on the imperative paradigm and an introduction to object-oriented aspects of the C++ language. Results revealed that to develop the solution code for the given problems, students consistently approached them without a comprehensive written plan/design. The process students typically used in developing a solution for the given computer problem involved: (1) problem understanding, (2) preliminary problem analysis, (3) reliance on examples and (4) trial-and-error. Students typically approached debugging syntax and logic errors by (1) following the compiler generated messages, (2) using trial-and-error, (3) performing a desk-check strategy and (4) using the VISUAL C++ debugger. This study identified the features of CPS and OOP learning that can be studied for identifying how students approach CPS and OOP processes in other object-oriented languages (such as JAVA) and how their CPS and OOP processes develop as compared to C++. Differences in programming performances were found among males and females. Males in this study were more comfortable with the mechanical-orientation of programming as compared with their female counterparts. Future research is needed in CPS and OOP to explore gender issues in learning OOP languages. This study identified potential student CPS and OOP learning processes and factors using a qualitative approach. Future research should investigate the factors effecting introductory CS problem solving using a quantitative methodology or perhaps a combination of qualitative and quantitative approaches.

STRATEGIES UTILIZED IN COMPUTER PROBLEM SOLVING AND OBJECT-ORIENTED PROGRAMMING

By

Naeem Zaman

A DISSERTATION

Submitted to

Oregon State University

in partial fulfillment of

the requirements for the

degree of

Doctor of Philosophy

Completed March 10,2003

Commencement June, 2003

Doctor of philosophy dissertation of Naeem Zaman presented March 10, 2003.

APPROVED

Redacted for Privacy

Major Professor, representing Science Education

Redacted for Privacy

Chair of Department of Science and Mathematics Education

Redacted for Privacy

Dean of Graduate School

I understand that my dissertation will become part of the permanent collection of Oregon State University libraries. My signature below authorizes release of my dissertation to any reader upon request.

Redacted for Privacy

Naeem Zaman, Author

ACKNOWLEDGEMENTS

My thanks and appreciations go to all committee members: Dr. Maggie Niess, Dr. Dianne Erickson, Dr. Larry Enochs, Dr. Tadepali Prasad and Dr. Thomas Wolpret.

My special thanks to Dr. Maggie Niess for her advice, guidance and patience. She has been a great source of encouragement throughout my education at Oregon State University. I thank her for believing in me and making a difference in my life and those who depend on me.

Many thanks to my parents, Miwako Kimura, children, sisters, brother and many other important individuals in my life who have encouraged me.

TABLE OF CONTENTS

TABLE OF CONTENTS (Continued)

LIST OF APPENDICES

STRATEGIES UTILIZED IN COMPUTER PROBLEM SOLVING AND OBJECT-ORIENTED PROGRAMMING

CHAPTER I

THE PROBLEM

Introduction

Teaching and learning in the introductory course of the undergraduate computer science curriculum often sparks debate about what and how to teach novices (Baldwin & Macredie, 1999). Because of this, prominent professional associations of computer science such as Academic Computing Machinery (ACM), the ACM's Special Interest Group on Computer Science Education (SIGCSE), and the Institute of Electrical and Electronic Engineers, Computer Society (IEEE/CS) have extended considerable effort both discussing and researching the teaching and learning of novices in the introductory course in computer science (CS). This is true in part because studies (Greer, 1986; Taylor & Mounfield, 1991) found that students who are successful in an introductory level CS course remain successful in the subsequent CS discipline courses.

The introductory course in computer science warrants a continued assessment and understanding of teaching and learning strategies, the curriculum, and its content. This assessment and understanding is needed due to the changing forces in the field including technological advancements in hardware and software tools, societal needs, and shifts in computer science paradigms (ACM & IEEE/CS, 1965, 1968, 1978, 1991, 2001).

The role of programming in the introductory course also generates an interminable debate within the computer science education community. Some have argued that the focus of introductory courses in the computer science discipline should only be on teaching and learning programming. However, others suggest that the introductory course should not only introduce students to programming, but should also offer a broad overview of the computer science discipline by including sub-disciplines such as social sciences within the context of computer science (Tucker & Wagner, 1994).

Varying views also exist on the implementation strategies for the introductory computer science course. One such view holds that CS departments should teach problem-solving strategies in addition to programming, while others argue that only the syntax and semantics of a programming language should be highlighted (Shackleford & Badre, 1993).

With these diverse concerns and competing debates regarding teaching and learning in the introductory course of CS, there has been a little agreement as to which objectives and/or goals CS educators should adopt in their introductory course. However, during the past three decades, premier CS associations such as ACM and IEEE/CS provided invaluable guidelines and recommendations for the typical goals/objectives for the introductory course in CS at the undergraduate level (Baldwin & Macredie, 1999).

The latest complete recommendation by ACM and IEEE/CS entitled Computing Curricula 2001, was developed in the United States in consultation with educational institutions and was designed to satisfy the requirements of the Computer Science Accreditation Commission (CSAC) and the Computer Science Accreditation Board (CSAB). Many CS departments in the United States and abroad have already "endorsed" the Curricula 2001 and have implemented it as a model both in their undergraduate CS curriculum and in their introductory courses (Baldwin & Macredie, 1999; ACM & IEEE/CS, 2001). In addition, the American Association of Colleges ([AAC], 2002) also concurred with ACM and IEEE/CS recommendations (ACM, 2001).

In ACM's curriculum recommendations (ACM & IEEE/CS, 1991; ACM & IEEE/CS, 2001) three goals/objectives were identified to address the major concern with respect to teaching and learning in the introductory course in CS. First, they stated that students have to learn the algorithmic problem solving and programming skills central to the CS discipline. Second, they emphasized the importance of teaching students how to transfer and apply programming and problem-solving skills to solve real world problems. Third, they reinforced the importance of showing students how to develop cognitive (thinking) models (ACM & IEEE/CS, 1991; ACM & IEEE/CS, 2001).

McCauley and Manaris (2000) found that across the United States, nearly all CS departments have adopted a programming-first model, recommended by the ACM, to teach introductory computer science courses. Furthermore, their study found that

CS departments used two major programming paradigms, imperative and object-oriented programming (OOP), to teach the introductory course. Dann (1990) stated that a programming language "encompasses a set of assumptions about how the programmer will think about what can be done. These assumptions are intricately linked to the paradigm -- a distinctive conceptual organizing principle -- on which the language is based." (Dann, 1990, P.70). Some languages allow either a single paradigm approach or a combined paradigm approach. For example, Stroustrup (2001) states that C++ supports a multi-paradigm framework, i.e. imperative and object-oriented, whereas the C programming language only supports the imperative paradigm. The imperative paradigm demands learners to think of a problem solution in terms of "sequential and ordered steps," whereas, the object-oriented paradigm demands to think problem solutions in terms of objects (Dann, 1990; Ross, 1997).

In the past, computer science departments viewed the imperative paradigm as a better way to meet the goals of the introductory course in CS. The reasoning for this belief was to help students learn computer programming and computer problem solving (CPS), thus laying a foundation for subsequent CS courses.

The popularity of using the imperative paradigm is waning. McCauley and Manaris (2000) reported recent trends among accredited colleges and universities showing that the majority of CS programs in the United States are now using OOP, to teach programming in their introductory CS courses. For example, in their study of 151 CS departments accredited by the CSAC/CSAB, McCauley and Manaris (2000) found that during the 1999-00 academic year, 83% of CS departments were using OOP, whereas, in 1995-96 academic year, 36% were using OOP.

The rationale for this switch from the imperative paradigm to the object-oriented paradigm among CS departments includes: (1) new innovations and technological changes in computer hardware and software (ACM & IEEE/CS, 2001); (2) a general paradigm shift in the CS field from imperative to OOP (McCauley & Manaris, 2000); (3) curriculum recommendations made by the ACM and IEEE/CS, CSAB, and CSAB to include OOP in the introductory course in computer science (ACM & IEEE/CS, 2001; CSAB, 2003); and (4) adoption of object-oriented languages in the Advanced Placement (AP) tests in computer science by the Educational Testing Services (ETS, 2003).

As a result of the shift from the imperative paradigm to OOP in CS departments, a large number of beginning students are now learning OOP in their introductory CS courses. Proponents maintain that OOP "lends itself to the natural attributes of the thinking process," making it easier for beginning students to learn to solve problem (Dann, 1990; Goldenson, 1996; Ross, 1997; Willis, 1999). However, OOP languages may be difficult to learn for novice students (Corritore & Wiedenbeck, 1999; Rist, 1995). Two main issues were identified by Dann (1990). "First, the student may not be adequately prepared or have the cognitive skills required in the programming process. Second, the student may possess a cognitive style which is unsuited for the imposed language and methodology" (Dann, 1990, p. 100).

In spite of OOP's popularity and its adoption among CS departments, and the impact this paradigm change has on beginning computer science students, relatively little scientific evidence exists about how students develop abilities and strategies to apply CPS skills in an OOP environment. Thus far, the research has focused on investigating the novice versus expert programmers in computer programming (Corritoree & Wiedenbeck, 1999; Lee, Pennington & Rehder, 1995). Other studies (Allwood & Bjorhag, 1990; Corritore & Wiedenbeck, 1999; Ebrahimi, 1994; Rist, 1995) have investigated only few aspects of OOP, rather than studying the program development process to create a problem solution in its entirety. Moreover, the existing research has focused on the student learning process of computer programming and CPS without considering the important element of instruction in the student learning process (Allwood & Bjorhag, 1990; Corritore & Weidenbeck, 1999; Ebrahimi, 1994; Pennington, Lee & Rehder, 1995; Rist, 1995).

Other studies (Choi, 1991; Lee & Thompson, 1997; Mains, 1997; Knox-Quinn, 1995; Willis, 1999) focused on the effects of OOP and imperative programming languages on students' problem solving skills "without having fundamental knowledge about students' learning processes" involved in OOP (Ahmed, 1992). This knowledge is needed since the evidence on the effects of programming on the problem-solving skills is inconclusive (Ahmed, 1992; Palumbo, 1990; Singh & Zwringer, 1996).

A study is needed to identify students' strategies and skills used in solving computer problems in a beginning CS course with an introduction to OOP. Clarifying students' learning processes may describe how students learn to program a computer by solving computer problems and how they connect learning to the instruction they receive.

Statement of the Problem

The purpose of this study was to describe how novice students learned computer problem solving in a beginning CS course with an introduction to OOP and what knowledge they obtained about OOP and CPS as a result of their experiences. Additionally, this exploratory study attempted to connect the instruction provided to the students to the development of their computer problem solving and programming skills.

The study focused on the following questions:

(1) What instructional strategies characterized a beginning computer science course with an introduction to object-oriented programming at the college level to engage students in computer problem solving?

(2) How did novice students solve computer problems as a result of instruction in a beginning computer science course with an introduction to object-oriented programming at the college level?

Significance of the Study

One of the most significant goals in CS education is to teach students programming and problem-solving skills in their beginning course in CS (ACM & IEEE/CS, 1968, 1978, 1991, 2001). By using detailed descriptions of students' strategies and collecting other descriptive data about student learning, this study aimed to identify how students approached computer problems in a beginning CS course with an introduction to OOP. This study also guides future directions for CS programs using OOP as the environment for developing students' ability to solve computer programming problems.

In addition, this study provides a more detailed description of instructional practices, and investigates how novice students apply CPS strategies in a beginning CS course with an introduction to OOP. These insights are important because they help instructors better identify what situations provide positive and/or negative experiences while students are engaged in programming and problem solving. Finally, describing student experiences helps CS departments design more effective instructional experiences for beginning programming students and can provide a foundation for further research in answering questions related to the effectiveness of the introductory CS courses.

CHAPTER II

REVIEW OF THE LITERATURE

Introduction

The explicit goal of the introductory course in undergraduate computer science (CS) education is to teach computer problem solving and programming (ACM & IEEE/CS, 1991; ACM & IEEE/CS, 2001). However, research indicates that students have difficulty learning the required computer problem solving (CPS) and programming skills (Corritore & Weidenbeck, 1999). This difficulty in learning CPS and programming skills may be due to the "educator's lack of understanding" of how students are applying CPS strategies and programming processes. Recent emphasis on object-orientated programming (OOP) and its impact on beginning students warrant more investigation about students who take an introductory CS course. Until now, relatively little research has been conducted on how students learn to solve computer problems in a programming environment using OOP and what connection students make with the instruction they receive.

The primary purpose of this study was to obtain descriptive information of how beginning students' learned and applied CPS and programming strategies as a result of their experiences in the first course in CS with an introduction to OOP.

Additionally, this study attempted to analyze the instruction provided to students in a beginning CS course.

This chapter reviews the previous research literature in the following sections: (1) curriculum and goals for the introductory course in computer science (2) students' learning of computer science at the college level (3) students' learning of computer programming and problem-solving.

Curriculum and Goals for the Introductory Course in Computer Science

This section looks at the introductory college-level course in computer science through its past and present by summarizing efforts of Academic Computing Machinery (ACM) to guide the introductory CS course curriculum through its model curriculum. Since its inception, the academic discipline of computer science has adapted a programming-first model. The programming-first model is aimed to develop the fundamental skills of computer programming and computer problem solving among beginning students of computer science (ACM & IEEE/CS, 1965, 1968, 1978, 1991, 2001).

The primacy of the programming-first model can be seen as far back to 1968 when the ACM first recommended a course entitled "Introduction to Computing." Since then, the progression towards including programming and programming-related topics into the introductory course curriculum has remained dominant over forty years. Furthermore, in 2001 the members of the ACM Curriculum Task Force predicted that the "programming-first model is likely to remain the dominant part of the introductory course curriculum for the foreseeable future" (ACM, 2001, p. 21). The members of the task force provided two reasons for the importance of maintaining a programming-first approach: "(1) programming is an essential skill that must be mastered by anyone studying CS; (2) placing it early in the curriculum ensures that students have the necessary skills when they enroll in intermediate and advanced courses" (ACM & IEEE/CS, 2001, p.17).

Despite the ACM's support for a programming-first model and its dominance in introductory CS courses; it has instigated arguments among CS educators. The following arguments represent the most significant concerns about the programming-first model (adapted from ACM, 2001: 3).

1. The recommendations to use a programming-first model in the introductory course by the leading curriculum developers such as ACM and its widespread adoption by CS departments is viewed by critics as "computer science equals programming" where computer science theory and its relationship to the broader cultural and societal issues has been ignored. Furthermore, critics of the model believe that limiting the scope of computer science to just programming may lead beginning students to believe that "theory is irrelevant to their educational and professional needs" (ACM, 2001).

2. In many implementations of the programming-first model, the focus remains on the syntactical details of a programming language in use. However, this emphasis on syntax comes at the "expense" of not teaching beginning students' proper

problem-solving strategies. As a result, students use an "ad hoc process of trial and error" rather than understanding the underlying "essential algorithmic model that transcends from a particular programming language" (ACM, 2001).

3. The programming-first model can be detrimental to both students with no prior computer background and to students with significant computer operating experience. Students with no prior computer experience frequently feel "overwhelmed" with a cognitively challenging task such as programming. Whereas those with the prior computer background might feel they have the necessary skills to deal with programming a computer. As a result, those students may feel overconfident with their computer operating background and "simply continue the bad habits" referred to as computer hacking (ACM, 2001).

4. The programming-first model does not appeal to non-majors because it reinforces the image that problem solving can only be approached through programming. However, the latest advancements in software tools have proven the contrary. New application programs have won the image of being comprehensive and dynamic among many non-majors using computer as a problem-solving tool (ACM, 2001).

In spite of all these concerns, McCauley and Manaris (2000) in their study found that almost every CS department in the United States had implemented the programming-first model. The curriculum task force established by ACM also stated that the programming-first model "has proven to be extraordinarily durable" (ACM, 2001, p.16). Certain factors have contributed to the adoption of programming-first model among computer science departments: "(1) programming is a prerequisite for many advanced courses in computer science. Curricular strategies that delay mastery of fundamental programming skills make it harder for students and the CS departments to ensure student success in advanced CS courses; (2) students often like programming more than other aspects of the field. Programming-based courses therefore tend to attract more students to computer science; and (3) programming courses offer skills and training that meets many of the needs expressed by students, their near-term employers, and non-CS faculty" (ACM, 2001, p. 4).

ACM (2001) has also recognized certain curriculum implementation strategies for the programming-first model. These implementation strategies serve as models for the introductory course in computer science. A brief description of each of the model implementations is as follows: (adapted from ACM, 2001)

Imperative-first

The imperative-first strategy utilizes structured programming concepts. The programming languages often used for the implementation of the imperative-first strategy are C, C++ and Pascal. This implementation strategy "focuses on the imperative aspects of the language in use: expressions, control structures, procedures and functions" (ACM, 2001, p.10). The primary disadvantage of adopting an imperative-first strategy is that because it is not the most commonly practiced paradigm among CS departments, it leaves students to "face difficulties later adopting an object-oriented approach. However, others counter that students who have grown used to working in an object-oriented language will chafe at the idea of learning to do without those features that makes object-oriented programming so powerful" (ACM, 2001, p. 10).

Objects-first

Object-first implementation strategy suggests that introductory courses in CS initiate students immediately with object-oriented programming concepts. At a later stage, control structures such as selection and repetition are introduced within the context of OOP to students. Proponents of object-first implementation strategy see it as fulfilling the needs of their students in subsequent CS courses. However, opponents of the object-first strategy raise similar objections to the object-first implementation as the programming-first model (ACM, 2001).

Breadth-first

Introduced in the Curriculum 91 (ACM, IEEE/CS, 1991) the breadth-first strategy envisioned that "the first courses in computer science would not only introduce programming, algorithms, and data structures, but introduce material from all the other sub-disciplines as well, making sure that mathematics and other theory would be well integrated into the lectures at appropriate points" (Denning, 89, p.107). The breadth-first strategy offered CS educators a response to the concerns regarding the programming-first model. For example, certain CS educators and researchers viewed the focus of programming only in the introductory course as introducing students to the discipline with a "limited view" rather than a "holistic and/or broader view." However, a successful implementation of the breadth-first implementation strategy by a significant number of educational institutions has not been materialized (ACM, 2001). A sample course description of a "Breadth-First" introductory computer science course is as follows:

> This course offers a broad overview of computer science designed to provide students with an appreciation for and an understanding of the many different aspects of computer science. Topics include discrete mathematics, an introduction to programming languages, algorithmic problem solving, analysis of algorithmic complexity, basic concepts in hardware, operating systems, networks, graphics, and an overview of the social context of computing. No background in computer science is assumed or expected. The course is intended for both students who expect to major or minor in computer science as well as for those not planning on taking additional course work (Tucker, 1991, p.35).

Algorithms-first

The algorithm-first implementation strategy introduces students to the computer problem solving and/or algorithmic process to learn the fundamentals of the computer science discipline. In this approach, no executable programming language is used to teach programming. However, the major emphasis is on non-executable, language-independent algorithm development techniques such as writing pseudocodes or developing program flowcharts. The ACM (2001) reports two major advantages to the algorithm-first approach: "(1) for non-majors, it permits some access to the science of computer science; and (2) for computer science majors, it permits them to encounter appropriate aspects of theory of problem-solving from the very beginning of their course of study. However, the algorithm-first implementation strategy demands an extraordinarily time consuming effort from the faculty to grade" (ACM, 2001, p.16)

Functional-first

The Functional-first approach was developed at the Massachusetts Institute of Technology (MIT) during the 1980s. In it, Scheme, a functional language, is used to teach the functional-first implementation strategy of the programming-first model. This approach places less emphasis on the syntax of the programming language and more on the problem solving. The disadvantage is that the Functional-first approach is viewed by students as "outside of the mainstream" computer science since Scheme is not a popular language (ACM, 2001, p. 17).

Hardware-first

In the hardware-first approach, students are first introduced to the hardware aspects of computer such as switching circuits and registers. This instruction is then followed by computer programming using a higher-level language such as Pascal or C. Among the advantages of the hardware-first approach is that students learn the theory and processes of computation with minimal details of the syntax. However, students are placed at a disadvantage since "the hardware-first approach is also somewhat at odds with the growing centrality of software and the tendency of increasingly sophisticated virtual machines to separate the programming process from the underlying hardware."(ACM, 2001, p.18)

Students Learning of Computer Science at the College Level

This section reviews the literature on students' learning processes in introductory computer science courses (focusing on computer programming and computer problem solving). It wasn't until the mid 1980's that the profession began seriously studying students' learning processes in CS courses. As part of this initial movement Anderson and colleagues (1983, 1987) identified the gradual learning model (cited in Nelson, Irwin & Monarchi, 1997). According to the gradual learning model, the beginning student achieves programming knowledge in three stages. In the first stage, the student gains declarative knowledge where he/she attempts to learn the "basic concept definitions, methods, and skill performance needed in programming." During the second stage, the student achieves procedural knowledge by utilizing examples extensively, which guides him/her to apply declarative knowledge in the problem-solving process. During the third stage, as a result of practice and experience, the learner attains the needed procedural knowledge and moves towards handling more challenging computer problem solving (Nelson, Irwin & Monarchi, 1997).

Corritore and Wiedenbeck (1999) examined how subjects comprehended programs when making modifications to procedural and object-oriented programs. The sample for the study included 30 participants. Fifteen participants modified a program in C++, and 15 modified a program in C.

During a two 2-hour session, participants studied and modified programs. In another session, they completed a second and third modification. The statistical analysis of variance (ANOVA) revealed that significant main effects of knowledge category, interaction of knowledge category and paradigm were found. Follow-up analysis on the interaction revealed that between paradigms, procedural participants had significantly more knowledge of operation than object-oriented participants. However, object-oriented participants had more knowledge of structure (selection, repetition) than procedural participants.

Allwood and Bjorhag (1990) attempted to identify how students debugged Pascal programs. The sample for the study included eight undergraduate students from the computer science department at the University of Gotenborg, Sweden. Seven of the eight students were males and one was female.

A computer program was provided to the students. Students were asked to think-aloud during the experiment. Researchers used a coding sheet to develop an analysis of the "verbal descriptions" of student responses to detect errors.

Results indicated that students made a variety of errors. The authors described the results of the study using different episodes. Students spent 67% of their time (on average) in evaluative episodes. The most common evaluation episodes were "triggered as a reaction to a test value," 23% (range 0% - 47%), followed by spontaneous episodes, 6% (range 0% - 16%), and hint episodes, 4% (range 0% - 16%). Categories were used by the authors to code the subjects' response to triggering event. Actions taken in response to triggering events by students included: "(1) interpreting meaning of the error message, (2) following flow of information in the program, (3) describing symptoms, (4) hypothesizing errors, (5) testing, (6) planning changes to the program, (7) experiencing general dissatisfaction, and (8) making changes to the program."

Ebrahimi (1994) investigated, novice programming errors, error types, and the causes of errors based on language constructs and plan composition in different programming languages. The sample for the study included 80 undergraduate students enrolled in computer programming courses at State University of New York, College at Old Westbury. Students were divided into four groups, each containing 20 students. One group attended a programming course in Pascal, one in C, one in FORTRAN and one in LISP.

Two experiments, one in language constructs and one in plan composition, were conducted on each group. The purpose of students' evaluations was to examine their understanding of the language constructs. For both experiments, the students were asked to write a program named "rainfall" which read the amount of rainfall for each day. Students verbalized their thoughts while developing program solutions during the interviews.

Results of the study for the [language constructs] revealed that in Pascal and C programming languages students made most errors in the use of IF statements. For the FORTRAN programming language, the most common errors were made in assignment statements. In LISP, students made errors in the use of logical operators. The most common errors in plan composition for all languages were: "(1) Guard IF:

when using IF statements, the need to check for special situations, such as division by zero; (2) Update: students had problems with both improper and unnecessary updating of variables; (3) Loops: students had difficulty with what type of loop to use, how to terminate the loop, and the structure of the loop." The results of the plan composition experiment showed that students had difficulty composing plans together.

Lee, Pennington and Rehder (1995) studied how expert procedural and object-oriented designers developed program design activities compared to how novice OOP designers performed the same task. The sample for the study included 10 subjects (8 males and 2 females). All subjects received a "swim meet competition" problem. The problem involved designing a scoring system, which could record scores and then report results for individual competitors and teams in the swim meet competition. Subjects were asked to read the swim meet competition problem and then complete a design.

The data analysis included transcribing all verbal protocols and "annotating each subject's diagramming activity." The results of novice object-oriented designers revealed that they spent more time in describing *objects* and significantly less time in designing the problem than did the experts. Novices also tended to create input and output procedures as did the experts.

Rist (1995) attempted to discover the common strategies used for object-oriented program design among university students. The sample for the study included nine students at three universities. Each student designed and coded solutions for four problems. Students were given a problem description and asked to design and code a solution on paper, in either Eiffel or C++. Subjects were asked to verbalize their thoughts while they worked, and all interview sessions were video-taped.

The results showed that all nine subjects used the global code generation strategy, and designs were typically goal-oriented (one goal at a time), and top-down within a goal. The main design strategy was forward, procedural design. Subjects did not perform prior comprehensive planning while designing the solution. However, they did identify the *classes*. The author concluded that the research provided an accurate and reliable picture of design among students. For future research, the author recommended a study of detailed cognitive models used by students in OOP.

Student Learning of Programming and Problem-solving Skills

In general finding a solution to a problem, using computer programming or general problem-solving strategies include similar cognitive actions and typical problem-solving heuristics used by a programmer (Ahmed, 1992; Dann, 1990; Kurland, Pea, Mawby, & Pea, 1986; Ross, 1997; Shnidermann, 1976). This section reviews studies on the relationship between programming and problem solving and the effects of programming on problem-solving skills.

Willis (1999) investigated the effects of learning object-oriented programming (OOP) on students' problem-solving skills. Willis hypothesized that OOP learning would improve students' problem-solving skills.

The sample for the study included, 87 students (46 females and 41 males) who were enrolled in a course titled "Computer Science I" and were considered as the treatment group. Forty-six (20 females and 16 males) were placed into a control group enrolled in a course titled "Business Computer Applications" at a local high school. The subjects' ages ranged from 15 to 18 years, and their grade level ranged from 10 to 12. The treatment group received instruction in an OOP language (C++), and the control group received instruction in Microsoft Office. For the study, a pretest-posttest non-equivalent control group quasi-experimental design method was used. In addition, the Watson-Glaser Critical Thinking Appraisal (CTA) was used to determine the existence of problem-solving skills. The test reliability was determined with coefficients for internal consistency ranging from .69 to .85.

Statistical analysis of variance (ANOVA) revealed no significant relationship between improved problem-solving skills in the two groups. The lack of improvement in CPS skills in the OOP group led the author to conclude that educators should help students become more "independent thinkers and problem solvers and not merely users of technology."

Choi (1991) studied whether programming in Pascal or FORTRAN improved the problem-solving skills of college students. The sample for the study consisted of 58 students enrolled in fall semester courses at Texas Tech University. Two experimental groups included 18 students enrolled in Pascal and 19 enrolled in FORTRAN. Students enrolled in the beginning keyboarding, course were selected as control group (n = 21) and had no prior programming experience.

To study the effects of Pascal and FORTRAN on student problem-solving ability, the study employed the Ross Test of Higher Cognitive Processes as the measuring instrument. The statistical analysis of variance (ANOVA) revealed that a significant correlation between increased problem-solving skills and programming in Pascal or FORTRAN was found. The authors concluded that problem-solving skills increased "through systematic exposure and interaction by programming in Pascal or FORTRAN." For further research, the authors suggested obtaining a more "accurate picture using descriptive data on the effects of computer programming on problem-solving."

Mains (1997) investigated the effects of computer programming language on logical thinking skills. The sample for the study consisted of students from two classes (Introduction to Programming QBasic and Graphics/ Desktop publishing) at a community college in Las Vegas, Nevada. Students enrolled in the Introduction to Programming in QBasic class served as the experimental group, and the students enrolled in the graphics/desktop publishing class served as the control group. Twenty-seven students took the pretest measuring logical reasoning skills. Statistical analysis (ANCOVA) of the results revealed no significant difference between the pretest scores for the computer programming and the graphic group. However, only 15 students took the posttest measuring the logical reasoning skills. No further information on the instrument and the type was provided. Statistical analysis (ANCOVA) of the results revealed no significant difference between the two groups on posttest. The author concluded that students with good mathematics background showed higher levels of comfort with computer programming.

Knox-Quinn (1995) designed a study to investigate how student construction of expert systems in LISP programming language would impact the problem-solving skills. The sample for the study included seven business students. Six of the students had previous computer knowledge, and two of them had prior computer programming experience.

The research design consisted of five stages:

(1) Students read four articles about expert systems, and a week later, they attended a lecture/demonstration about expert systems.

(2) Students were taught how to develop knowledge bases using examples.

(3) Students solved passive activity limitations (PAL) tax problems.

(4) Student reports and anecdotes were recorded while they were developing knowledge bases.

(5) Problems were given to students, and their verbal protocols were recorded.

The results of the study showed that programming expert systems increased problem solving and/or higher order thinking, such as being able to classify information and being able to break down content knowledge to find the relationship between pieces of information. The authors concluded that developing an expert system improved students' problem-solving strategies and that effective computer problem solving can be achieved by allowing students to spend time solving problems.

Discussion and Conclusion

This chapter has reviewed the relevant literature on three major areas. The first area which focused attention on the curriculum and goals for the introductory course in computer science, revealed an ongoing debate about what and how to teach students in the introductory course. However, teaching of OOP and computer problem solving is the evident choice among CS departments and is recommended by the major professional computer societies (ACM & IEEE/CS, 1991, 2001).

The second area of the literature review, which focused attention on student learning of computer science at the college level, revealed information on selected CPS and programming activities/strategies employed by novice students. Students displayed "expanded mental representations when they gained more programming experience" (Corritore & Wiedenbeck, 1999). *Objects* were salient in OOP understanding. Object-oriented student designers identified *objects* and *methods* (Pennigton, Lee & Rehder, 1995). Moreover, the studies (Allwood & Bjorhag, 1990; Pennington, Lee & Rehder, 1995; Rist, 1995) suggested that different strategies were used by novices to cope with problems. In addition, the studies found that students' were able to write better programs after being exposed to programming for a longer period of time. Program understanding played a vital role in finding the correct solution and during the debugging process (Allwood & Bjorhag, 1990; Ebrahimi, 1994). One of the studies (Rist, 1995) revealed a mixed conclusion that OOP is difficult to learn due to its complex nature, but that it is easy to learn due to its modeling of the real world entities.

The third area of research, which focused on the computer programming and problem-solving, revealed "mixed results" about the effects of programming on problem-solving skills (Ahmed, 1992). In some studies, there seemed to be a positive correlation between programming and increased problem-solving ability, for example, studies found that writing expert systems in LISP and computer programming in FORTRAN and Pascal resulted in improvement of problem-solving skills (Choi, 1991; Knox-Quinn, 1995). On the other hand, other studies found that problem-solving skills did not improve after receiving instruction in QBASIC or OOP (Mains, 1997; Willis, 1999).

The studies reviewed in the literature reveal that there are issues that warrant attention. First, in each study, only one or two aspects of the programming process were examined. None of them chose to examine the programming process in its entirety. Second, a wide range of CPS and OOP learning processes and strategies were studied by the researchers. However, each study stopped short of examining how the instruction students received influenced their learning processes. Thus, current studies provide an incomplete picture of the CPS and OOP learning processes in a programming class. Third, the question of what effect computer programming has on problem-solving skills has produced mixed and often confusing results. Several concerns have surfaced from the previous research and the methodology those studies used. Some studies (Choi, 1991; Willis, 1999), for example, used only a paper and pencil instrument for data collection purposes. Use of the paper-and-pencil instrument may raise some concerns such as "whether the respondent is interpreting the items on the test according to the researcher/developer's framework."

Fourth, some studies (Allwood & Bjorhag, 1990; Corritore & Wiedenbeck, 1999; Pennington, Lee & Rehder, 1995) suffered methodological shortcomings. Some studies, for example, used questionnaires (only one data collection source). Since no other data source was applied to support the collected data, it is doubtful if the information collected actually reflected the intended responses by the subjects. Furthermore, some of the researchers failed to establish the validity or reliability of the instruments used in their research (Corritorre & Weidenbeck, 1999; Willis, 1999). Finally, some studies (Allwood & Bjohrag, 1990; Corritoree &Wiedenbeck, 1999; Ebrahimi, 1994; Lee, Pennington & Rehder, 1995; Willis, 1999) failed to provide background information/selection criteria about the sample. Hence, these shortcomings make it difficult to generalize the results to subjects other than those sampled.

In addition to the various problems associated with the studies mentioned in the literature review, most of the studies were conducted on the imperative paradigm rather than studying the OOP learning process. Because computer science curricula, has already switched from imperative paradigm to OOP (ACM, 2001; CSAB, 2003; McCauley & Manaris, 2000), new studies are needed. Currently, the learning processes used by the students in an introductory OOP class are relatively unknown. Moreover, in responding to the weaknesses and shortcomings, which surfaced during the literature review, additional research is needed to determine computer problem solving strategies used by students while engaged in programming with an introduction to OOP.

Recommendations

Despite the shortcomings and weaknesses, the studies reviewed did provide directions for study and indicated a need for further research in investigating the strategies used by students in an OOP environment. First, among the studies reviewed, none accounted for instruction and its connection to OOP learning. Second, none of the studies examined the OOP learning process in its entirety. Third, a number of studies on the relationship between programming and CPS have moved to examine the effects of computer programming on problem solving without collecting fundamental knowledge (Ahmed, 1992) about CPS and programming strategies.

To avoid the problems of the research reviewed, some methodological recommendations are made. First, a study is needed that accommodates the Reed and Palumbo (1992) recommendation of gathering the basic student information on the development of problem-solving and programming skills (Ahmed, 1992). Rather than investigating the relationship between computer programming and problem solving, a study is needed which investigates students' thinking strategies and the characteristics of the instructors' instruction in CPS and OOP. Such a study will provide a more comprehensive view on students' OOP learning. Furthermore, information about the transfer effects of programming cannot necessarily be revealed by only a pencil-and-paper test. Instead, the investigation on students' OOP learning needs to employ qualitative methods of research and use multiple sources of data collection including classroom observations, student and instructor semi-structured, open-ended interviews, and a comprehensive review of the classroom documents. This approach will help avoid the methodological shortcoming in the literature reviewed.

Second, in studies (Choi, 1991; Corritore & Weidenbeck, 1999; Lee, Pennington & Rehder, 1995) students were asked to provide information using closed-ended interviews and/or questionnaires. There is a need to conduct research on students' OOP learning strategies by using a more open-ended methodology that is sensitive to students' and instructors' personal understanding of the content; such an approach may produce significantly different results (Lederman & Chang, 1997). Therefore, it is recommended that open-ended interviews be conducted to gather the information on OOP teaching and learning.

Finally, none of the studies (Allwood & Bjorhag, 1990; Corrotorie & Weidenbeck, 1999; Ebrahimi, 1994; Rist, 1995) investigated the connection of the teacher's instruction on student learning. The student learning of programming has been the focus of much research attention. But without a comprehensive look at the instruction with student learning, an incomplete picture of the student learning of CPS and OOP exists.

CHAPTER III

DESIGN AND METHOD

Purpose

The purpose of this study was to explore students' computer problem solving and computer programming learning experiences to understand the dynamics of students' approaches in learning to solve problems in a beginning computer science (CS) course with an introduction to object-oriented programming (OOP). The following questions directed the study:

(1) What instructional strategies characterized a beginning computer science course with an introduction to object-oriented programming at the college level to engage students in computer problem solving?

(2) How did novice students solve computer problems as a result of instruction in a beginning computer science course with an introduction to object-oriented programming at the college level?

Setting

Student participants were diverse in terms of gender, age, ethnicity, math placement scores, and academic majors. Two female and two male students with ages ranging from 19–30, represented a racial mix consisting of two European Americans

(1 male and 1 female), one Native American (female) and one Mexican American (male).

All 24 classroom sessions of the introductory computer science (CS 101) course were observed for the duration of five weeks during the summer 2001 academic term. The course was taught five days a week, Monday through Friday from 11:20 a.m. - 12:50 p.m. A total of 10 students were registered for the course. The class was diverse in terms of gender, age, ethnicity, and academic majors. The class contained five female and five male students, with ages ranging from 18 – 30. The racial mix of the class consisted of five European Americans (two males and three females), one Native American (female), one Mexican American (male), one Vietnamese American (male), one Chinese American (male) and one international student (female).

The classroom setting included a state-of-the-art classroom and equipment. Each student had access to a computer on his or her individual desk during the classroom sessions. Each computer was equipped with Intel® Pentium® 4 processor and Microsoft ® Visual C++ Compiler 6.0, and was connected to the Microsoft ® Visual Studio development environment providing maximum optimization for the Intel processor architecture and access to laser printers. Students used a VISUAL C++ programming environment to generate Win32 console applications. The instructor used the electronic white board and overhead projector. The instructor had Internet access to refer to his web site specifically designed for the course.

The selected course (CS 101) was designed as an overview for computer science majors at the undergraduate level. CS 101 was described in the instructor's course syllabus as follows:

> This course is designed to introduce you to the fundamental principles of computer programming. While most of us are familiar with the use of computer applications to assist in well-defined tasks such as writing a report or playing a game, you will often come across unique problems for which no application is available. Learning to program a computer allows you to create new applications to solve such unique problems by giving the computer new instructions in a general-purpose language. We will use the programming language C++ to introduce *object* design and object-oriented problem solving techniques. *Object* -oriented programming allows us to develop programs in a natural way, by organizing information and instructions as *objects* that correspond to the way in which we think about problem solving. Prerequisite: Students must have completed the basic math skills requirement.

The CS department offered CS 101 for CS majors and minors at the undergraduate level. CS 101 served as the preparatory course for advanced CS courses by providing the foundation in CS coursework. The computer facilities for students majoring in the CS Department included 24-hour access to departmental computer labs, with Pentium III & IV running Windows 2000 and Sun Ultra Sparc stations.

Method

This study was exploratory in nature, specifically designed to identify new directions for teaching computer science, computer programming and computer problem solving. In addition, it aimed to search for potential factors affecting students' success in computer problem solving, in computer programming, and developing novice students' ability to solve computer problems. A variety of qualitative research techniques were employed to collect and analyze the data.

This study focused on a college level introductory computer science class that incorporated OOP. The curriculum of the class was focused on teaching students introductory CS concepts and introductory OOP. After receiving permission from the instructor, four students were selected who were willing to participate in the study. Observations of each class session were made to thoroughly describe the curriculum and instruction. Student interviews were conducted to gather information on how students implemented instruction in their responses. Two formal interviews gathered students' computer problem solving (CPS) and computer programming approaches.

Two practice sessions (an hour each) were conducted during the second and fourth weeks. The purpose of these sessions was to allow students to orient themselves with the protocols that were used during the formal interviews, to practice verbalizing their thinking process while engaged in CPS, and to practice similar problems provided during the formal interviews. Two different computer problems were designed by the participating instructor for each practice session. However, the practice session problems were designed to demand less student time to solve as compared to the formal interview problems. Two problems were selected for each session based on the students' programming skills at a particular point in the course and were representative of problems typically found in introductory CS textbooks.

The course instructor designed and prepared the problems by embedding common student errors into a program. The instructor also designed two problems without a solution.

Subjects

This study proposed the identification of a beginning CS course with an introduction to OOP at the college level to teach CPS and computer programming where the instructor and four students volunteered to participate in the study. The instructor was willing to be observed in all classes and reflect on the teaching of the OOP concepts and CPS through multiple interviews. The four students selected for participation in the study had no previous experience in programming and were selected to assure diversity in the following categories: (1) mathematics skill levels (high, medium, low) since previous studies have linked mathematics skill levels to programming abilities (Campbell & McCabe, 1984; Dey & Mand, 1986); (2) enrollment in the course for the purpose of fulfilling (a) CS major and/or minor requirements, (b) an elective for their major, or (c) a general elective. All students were willing to participate in the practice sessions as well as fulfill research expectations. They allowed complete access to their graded assignments, quizzes and exams, agreed to informal twice a week interviews, and participated in two computer problem solving and programming interviews.

During the first day of the class, all students in the class were asked to complete a questionnaire (Appendix C) which asked for background information. Permission from the student volunteers and the volunteer instructor was also obtained before the research began (Appendices A and B). Mathematics skills test information was gathered for the volunteers, and the selection of the students was completed by the end of the first day of the term.

Data Sources

To investigate the instructional strategies emphasized by the instructor and the strategies used by students to solve computer-programming problems in a beginning CS course, five sources of data collection were used. These data collection sources helped to collect data about the class, the instructor, and the students. A description of each type of data collection source is provided below.

Classroom Observations

Classroom observations were conducted in each class session for a complete academic term. The purpose of the observations was to document the curriculum that was taught, observe the instructional strategies (activities, settings and classroom engagements), gather data on the instructor and student interactions (instructor-student and student-student interactions), and document student behavior during instruction.

The observations included the instructor's presentation of the material and the collection and incorporation of the relevant classroom material, such as exams, handouts, and worksheets. The purpose for these observations was to provide a detailed description of how a beginning CS course with an introduction to OOP was taught and to identify salient characteristics that supported student learning in the class. The observations were used in answering the first research question.

All classroom observations were audio-taped, and field notes were taken. A special microphone was attached to the instructor to record instructor-student interactions. The purpose of the field notes was to minimize the researcher's classroom influence. Field notes and audio-taped transcripts were transcribed and organized at the end of every class session. In order to minimize the researcher's bias, the researcher separated description of activities in the class from personal reactions to events, questions, and interpretations by logging personal reactions in a journal.

In addition to classroom observations, outside classroom interactions (instructor-student) were documented. These interactions were gathered by documenting students' visits to the instructor during his office hours. The instructor reported the contents of the office hour visits. Informal discussions and/or interviews were conducted with the instructor at the end of each day to clarify questions that surfaced from the classroom observation, and to review any questions students asked during his office hours.

Instructor Interviews

A series of audio-taped interviews were conducted with the instructor (Appendix D). The instructor was reminded that the data collected during the interviews were confidential and would not be used in any way for evaluation.

An initial semi-structured, open-ended interview was conducted prior to observing the classroom instruction. The initial interview was designed to establish an overview of (1) the course's curriculum, (2) the instructor's approach to teaching the class, (3) the OOP concepts and CPS strategies the instructor planned to stress, and (4) the instructor's method and purpose for teaching the OOP concepts.

Direct observations of all classes were used as a springboard for the interviews (except for the initial interview) with the instructor. Arrangements were made with the instructor prior to each observation. At the end of each day, the researcher informally interviewed the instructor to (1) clarify the observations, (2) gain the instructor's perspective on the progress of the students, (3) identify the students' progress on the assignments, and (4) gather the instructor's plan for the following day of instruction.

A final semi-structured, open-ended interview was conducted close to the end of the academic term. The purpose of the final interview was to identify the instructor's perception of the progress of the students with CPS and OOP programming concepts. The final interview allowed the instructor time to talk about the programming concepts and CPS strategies he stressed and the reflection on students' abilities to use those concepts in their programming activities. Data from the interviews were transcribed immediately after each interview. When discrepancies between interview data, classroom observations, and other data occurred, informal interviews with the instructor were arranged for clarification.

Classroom Documents

All classroom documents pertaining to the teaching the introductory CS class were collected and examined. Classroom documents included syllabi, initial course information sheets, lesson plans, lecture notes, Power Point presentations, textbook activities, laboratory sheets, assignment sheets, homework assignments, hands-on activities, tests, and programming projects. The classroom documents were then analyzed in terms of the information on (1) CPS and the programming concepts taught, the classroom activities and processes the instructor utilized, and the areas to pursue through observations and interviews. The assignments (written and programming homework) and tests (quizzes or exams) were included with other important data to develop a response to the first research question.

Student Interviews

A variety of activities were used to monitor the progress of the four students taking part in the study. The researcher conducted two informal interviews per week

(Appendix E) with the students to gauge their understanding of the CPS and OOP concepts they were learning. During the first interview, the researcher reminded students that the information would not affect their grade and would be kept confidential.

During the informal interviews students were asked to explain their understanding of the programming concepts that had been the focus of instruction that particular week. These informal interviews (twice a week) also incorporated questions about the students' perceptions and learning of CPS and OOP, their study practices, and how well they were able to determine efficient and correct computer problem solutions.

Twice during the term (at the middle and close to the end of the term) the researcher conducted problem-solving interviews (Appendix E) of the four students. The purpose of these student interviews was to allow students an opportunity to demonstrate the following: (1) their abilities with CPS strategies and the OOP concepts learned as a result of their instruction; (2) their perception of what the program was doing at different stages; and (3) any confusion they had about their programs or the concepts they were taught to use for particular programs.

In the interviews, students offered oral and/or written responses to problems and then explained the CPS and computer programming strategies they employed. All written materials were collected at the conclusion of the interview. Interviews were audio-taped and transcribed to capture all the information in an accurate manner. The written materials provided additional documentation of the audio-taped transcription and were used to clarify the students' work on the problems.

Ultimately, the focus of all the problem-solving interviews was to capture the entire program development process by observing how students approached problems and worked out solutions. Student interviews provided extensive performance examples and detailed accounts of student learning of OOP concepts and CPS strategies throughout the course. The student interviews were in-depth and conducted in a relaxed environment.

Two different computer programming problems were designed by the participating instructor for each of the formal interview sessions. The two problems for each session were problems typically found in introductory CS textbooks and were based upon the level of programming skills the students could be expected to demonstrate at that point in the course. For each session, one of the problems was designed without a solution and asked students to develop a complete solution. The other problem presented a program in which the instructor had embedded common student errors for the participants to identify and correct.

In order to establish the content and face validity of the problems given by the instructor, the problems were reviewed by five computer science instructors with recent teaching experience in an introductory course in OOP at the college level. These instructors were asked to critically review the problems and solutions with regard to problem appropriateness based on the course objectives and the material the students had covered in class. Review and modifications continued until 80% agreement was reached among the instructors.

During the formal interview, students were given two problems. Problem 1 offered a proposed solution but contained errors. Problem 2 required students to develop a complete programmed solution. For Problem 1, students were told that errors were embedded in the computer program and were asked to review the provided solution and provide the expected output. Furthermore, students were asked to identify why the output was erroneous. For Problem 2, students were asked to develop an entire computer solution to the given problem. The time period for the completion of the two tasks was a maximum of two hours. The students were told that the most valuable part of the interview was explaining their thoughts and their solution rather than just obtaining correct answers. Students were allowed to work on the computer problems using pencil-paper and/or a computer.

During the formal interview, students were asked to (1) describe their corrected solution to Problem 1, and (2) describe their solution for Problem 2. Additionally, students were asked to identify, describe, and interpret, particular strategies they used debugging the solution to Problem 1 or in creating and debugging the solution to Problem 2.

The role of the researcher as an interviewer during the problem-solving interviews was to: (1) prompt silent students, (2) clarify the students' ideas, and (3) probe more deeply when students made interesting comments or responses to the problems. The researcher prompted students with questions as needed to obtain as much as information about the CPS and programming strategies they used. The researcher also reiterated that the interview was not a test as a way to reassure students who were anxious about spending too much time on a problem or generating incorrect answers.

The protocol for the two problem-solving interviews followed a set of actions and questions:

(1) The audio recorder was tested, turned-on and time was noted.

(2) Each student was asked to state his/her name, and the recorders were checked again for audio transmission.

(3) Each student was provided with a computer, a desk, a pencil, a calculator, blank pieces of paper, a word-processed hard copy of the two problems, and a soft copy of the solution code for the problem with the embedded errors. Both problems were used to investigate the CPS and programming strategies that students used to develop (1) an accurate solution to a proposed solution, where errors were embedded and (2) an original solution to a given problem.

The directions for Problem 1 (solution with embedded errors) asked the students to:

1. Explain their understanding of the expectations in the problem.
2. Correct the embedded errors.
3. Generate the correct output.

The directions for Problem 2 (developing an original solution code) asked the students to:

1. Explain their understanding of the expectations in the problem.
2. Develop the solution.
3. Correct errors (if any) in their solution.
4. Generate the correct output.

(4) Students were asked to verbalize their thoughts.

(5) The researcher noted student behavior and actions as the student worked on the problems. These notes were compiled with the transcriptions of the audio-tapes.

(6) The researcher kept track of the total time taken by the student to solve each of the problems during the interview session.

(7) The researcher asked students to refrain from discussing any of the problems with others in the class; this request was made to prevent contamination of the subsequent interviews.

When a student completed the interview, all materials including the audio-tapes, handwritten notes by the student or researcher, scrap papers, the computer disk copy of the solutions, and the hard copy of the source and object code were placed in a secure envelope. Copies were also made of the material collected.

Researcher's Journal

The researcher maintained a daily journal on classroom observations and interviews with the instructor and students. In it, the researcher recorded reflections on the classroom observation and research activities. It also included thoughts, questions, reactions, interpretations, and insights during the observations. The researcher's journals helped to identify potential sources of biases and misinterpretations by the researcher. By doing so, the researcher attempted to minimize threats to the reliability of the data analysis since the major source of data collection and analysis was the researcher. The journal also served a guide in interviews and assisted the researcher in clarifying observations.

Data Analysis

Qualitative data analysis techniques were used to analyze the collected data used in this study. Data were stored in a file coded under a pseudo-name for the instructor and students, date, and type (observation, interview, etc) in the researcher's office along with a backup copy. The data analysis process for the research involved ongoing data review of instructor and student interviews, classroom observations, classroom documents, as well as the researcher's journals and field notes. The process involved preparation of a descriptive analysis and a summary of basic trends or relationships evident in the data. The data analysis and report of conclusions were structured around the research questions.

A narrative description of the experiences of the instructor and the students was developed during the data analysis. This narrative provided an in-depth description or a picture of an introductory CS class at the college level that included CPS and OOP concepts and strategies. The narrative included the transcriptions of the selected parts of the lessons with verbatim quotes from observations and interviews.

Analysis for the data collected from the instructor's classroom observations, interviews, classroom documents, and the researcher's journals was initiated with transcription of all the audio-tapes and the field notes. Data were searched, organized, examined, and classified to find ideas, similarities, constructs, themes, regularities, and patterns of similarities and differences; all of the data sources throughout the academic term were used. Any key words or phrases, representing any patterns and recurring regularities, were used to code categories and to search for patterns and comparisons in the data. The end product of the data analysis of the instructor's interviews, classroom observation, and classroom documents were summarized in a narrative summary. The narrative described the instructor's characteristics, actions/reflections of his/her teaching, a narrative description of lessons, and a list of the CPS strategies and OOP concepts emphasized.

Data analysis of student interviews began with a transcription of all students' audio-recorded interviews, researcher's journal and field notes. In order to develop a detailed and thorough description of each individual students' knowledge, skills, and understanding of solving computer problems, their interviews were reviewed and synthesized several times. A profile of each student contained background information, such as demographics and information of how each student solved the given computer programming problems; these data were related to the instruction the student received in the introductory CS class. This analysis was designed to identify: (1) patterns of similarities and differences among the students' CPS strategies and programming performances, (2) any words or phrases representative of these similarities, patterns and regularities, (3) any categories or sub-categories, if needed, to describe their problem-solving solutions, programming, and computer problem solving strategies.

Researcher

The researcher was the primary investigator and data collector for this study. Obviously the researcher had personal biases, experiences, viewpoints, training, and influences that would impact the interpretations, therefore threatening the credibility of this study. In order to minimize or eliminate personal biases, the researcher's roles were documented. The researcher kept a journal of personal questions, reactions, decisions, preconceptions, values, experiences etc. This journal was intended to help distinguish personal biases and therefore allowed a less biased understanding of the teaching and learning in the class.

The researcher received his Bachelor's degree in CS. The researcher also completed two Master of Science degrees, one in CS and the second in Science Education. The researcher had been involved in teaching introductory and advanced undergraduate classes in CS for 12 years. His teaching and industry experience included a variety of programming languages. The researcher had previously established a CS department/lab, developed and implemented a complete curriculum for an Associate of Science degree program in CS and Computer Information Systems. The researcher had also mentored new instructors, performed supervisory observations, evaluations, as well as peer observations for a variety of CS classes including introductory CS classes/labs, which included instruction in OOP. Currently, the researcher teaches a wide range of computer science classes at a community college. The researcher also advises a computer club for students.

As an undergraduate student and a CS educator, the researcher realized the high dropout rates in introductory programming classes. After reviewing the literature on the subject of teaching and learning introductory courses in undergraduate CS, the researcher recognized that most students lack the necessary skills to solve computer problems and are unable to transfer their programming and CPS skills to other areas. After attending several computer professional conferences, workshops and completing an extensive review of the literature on CPS and OOP, the researcher questioned students' understanding of fundamental CPS and programming concepts. From the researcher's perspective, the research questions presented in this study are of vital importance for CS programs.

CHAPTER IV

ANALYSIS OF DATA

Introduction

This chapter provides the results of the data analysis in three sections. The first section provides Tim's profile (the instructor for the class), including an in-depth description of his academic and professional background, his teaching characteristics and actions and his reflections about the course. It also includes his instructional strategies and the object-oriented programming (OOP) concepts and computer problem solving (CPS) strategies he emphasized. The second section includes student profiles describing how each student involved in this study approached solution to the given problem along with their demographic information, class attendance, and class work information. The third section provides a synthesis of the results directed at answering the research questions posed for this study.

Instructor Tim

The Computer Science (CS) Department Chair of the university recommended that Tim join the study. When Tim was contacted, he showed enthusiasm to participate. Tim was a cordial, friendly, and helpful individual. He had a strong educational background in the area of computer science with a Ph.D. and Master's degree in computer science with a specialization in databases. Tim's Bachelor's degree was in Electrical Engineering with a specialization in control theory.

Tim valued conducting research. Tim's impressive research work in the area of databases was published in a variety of CS and Engineering professional journals. He recently completed research in the areas of multi-database-state services and in the design of a visual object-oriented programming language.

During the past 10 years, Tim has taught a variety CS courses such as introduction to programming, computer organization, programming languages, databases, and networking. Tim stated that he enjoyed the teaching profession because it allowed him flexibility in managing his time, saying that "Teaching allows me to enjoy my life during summer and do research." Tim particularly enjoyed interacting with his students. He believed that classroom interactions and assignment grading helped him to understand how students' mental models worked, and how they thought about the object-oriented concepts.

Tim's Plans for Instruction and Assessment

Tim was a dedicated teacher who was concerned with and cared about student learning and improvement. Tim stated, "I help them to learn and make them better in what they do." Tim characterized his teaching of the introductory computer science course as "student-centered." In describing his teaching philosophy, Tim explained that "I believe in presenting opportunities for students to learn rather than telling them how to learn; I prefer self-discovery of knowledge and the light bulb theory." Tim believed his approach to teaching allowed students to have fun. He told his students "Do not really worry about what it all means; let's just solve the problem and play with the computer and make it playable rather than a chore." Tim said that he used feedback from his students in the introductory CS courses to help him decide "what works well and what does not." With regards to his teaching style, Tim explained that he wanted to provide his students with the opportunity to be "on their own" so he could build their confidence. Tim offered one critique at his own teaching, stating that "One thing which may or may not be a good thing of my style as a teacher is to have a tendency to not really jump on people who are lagging sometimes, when I probably ought to, and this may lead me to ignore some of the quieter students."

Prior to the beginning of the term, Tim perceived that students were generally "unprepared" and lacked, well developed "mental models" to create object-oriented computer programs. Tim's also noted that students were unprepared to manage low levels of details of the program implementation and that they lacked sophistication in developing their own computer programs.

Tim had strong opinions about the title of the course "Introduction to Computer Science" and called it "misleading in its implications." According to Tim, the name of the course was "wrong," because the course was not an introduction to computer science but rather an introduction to computer programming. Tim also had reservations with regards to the textbook assigned for the class. He described it as "confusing for students" and as offering "very little new facts."

Tim also disagreed with the use of C++ language in the introductory CS course. He called C++ an "awful language," explaining that, "there are too many complexities in C++ that cause frustration without giving any more of an inside inspiration among students." Furthermore, Tim described the VISUAL C++ compiler as "notoriously bad" with obscure error messages.

In the initial interview, Tim explained his planning and implementation of the instructional strategies as "textbook bound," stating that "Mainly I develop my teaching around the book and will have 50% time for lecture only." Tim also planned to make the course more "like a lab" where the majority of teaching involved interactive lab exercises, and students were able to explore how the programs actually worked. Tim planned to teach the introductory course by using examples and placing less emphasis on syntax and more on CPS. According to Tim, " I plan to augment my lectures with examples and not to emphasize too much on the syntax. I also have decided to develop 50% of the homework assignments based on examples I will be discussing in the course and 50% where students have to develop an entire solution from scratch." However, Tim confided that his personal approach to teaching with regards to OOP concepts was still "open" and in the "experimental stages."

Tim was still unsure of how he would approach teaching the introductory computer science class. "Should I introduce the object-oriented concepts right from the first day or should I wait and introduce these concepts at a later stage in the course?" Tim thought that OOP concepts should be taught in a non-confrontational way. "According to my past experience, students seem to be pretty good about the object-oriented concepts such as the concept of *class* as long as they are not confronted directly with it."

Tim planned to emphasize the following OOP concepts in his introductory CS course: *object*, *methods, class*, *constructor*, *destructor*, *encapsulation*, *inheritance*, *polymorphism* and *information hiding*. He thought that the most difficult OOP concepts to teach would be *abstraction* and *encapsulation* and the easiest would be the concept of *methods*.

When asked how the concepts he planned to emphasize would be helpful to beginning students, Tim replied:

> The concept of *abstraction* will assist students in organizing material. It will help them to manage problem complexity and will help students to place mental walls and put shells around to hide details and to move thinking to other problems and situations.

Tim elaborated on his teaching goals and objectives for the introductory CS course. He wanted students to master fundamental object-oriented problem-solving techniques by using the object-oriented design and to solve problems using the object-oriented programming. Tim also wanted students to learn other programming concepts such as selection, repetition, and functions.

With regards to assessment goals and objectives, Tim's goal was to make sure that students were able to develop basic OOP and CPS skills. He explained that the purpose of quizzes and exams was to help assess students' understanding of the conceptual problems and CPS. However, Tim felt that "weighing too much on computer problem solving is not fair because you are asking students to do creative work in a stressful situation." The programming assignments were designed to assess students' ability to create, execute, and test simple programs.

As part of the initial interview, Tim was asked to explain his understanding of the CPS process. Tim viewed CPS as a discovery process.

> I see problem-solving as a discover and refine process. Layout your basic thoughts, your *objects*, *methods*, and *classes* and see how they fit together and keep refining it. It is like a black box approach i.e. to see what information you give and what information you get back. One should keep distance from the computer language or the details of the syntax.

Tim planned to help students develop CPS skills by asking them to work out computer problems in written English first rather than C++ code. However, Tim was not in favor of any particular problem-solving strategy and/or tool.

When asked about his understanding of some fundamental OOP concepts that he planned to teach, he responded:

> *Encapsulation* is basically just structuring of everything into a central entity. It is like putting all the pieces together in the same box, and so it's all there together. *Information hiding*, that is once you get the stuff in the box you only let people see what you want them to see in the box, and rather than putting it all for them to see. It is like you have a cardboard box then you make a hole and you see what you need to see. *Class* is simply a data type. It's basically the group of things that potentially can be an instance of a *class*. It is the mechanism by which you define the potential instances. In a more abstract way, it is a set of all potential values. OOP is any programming in which your first principle is a definition of an *object* by defining the *classes*. This definition has nothing to do with languages. It is a way of thinking and developing structure.

Tim's Instruction and Assessment

Based on classroom observations/documents, field notes, and interviews, Tim's instructional and assessment strategies were identified. The course and weekly lesson contents (Appendix F) typically included teacher-directed, text-bound lectures using Power Point presentations. The presentations were followed by demonstration of examples on the computer and a time specified to answer student questions. Students were assigned homework assignments from the textbook that involved both written exercises and programming projects. Course assessment included quizzes, written (midterm and final) exams, and homework. At the conclusion of the course, students were expected to design, code, and test programs.

During the first part of the first week (Appendix F), Tim focused on discussing primary (computer programs, input/output, *objects*) and general (*object* types, *object* attributes, *object* actions/operations) concepts using a Power Point presentation. He discussed the C++ programming environment (source code, compiling, pre-processing and linking using VISUAL C++ compiler).

During the second half of the first week, Tim highlighted the imperative aspects of the C++ programming language with a continuing introduction to *objects* (such as cin, cout). Tim lectured on topics such as C++ constructs and basic C++ types and programs. He also introduced expressions and control flow concepts using C++ code examples with an emphasis on details of C++ syntax. At the end of each lecture, Tim allowed brief lab sessions where students worked at their computers individually with the sample program code he provided. Typically, students copied the sample program code into the computer and then compiled the programs. Instructor-student interactions observed during the lab sessions focused on questions regarding the working of the VISUAL C++ compiler and its error messages. During the first week, the written homework assignments focused on memorization, familiarization, and understanding of basic object-oriented programming concepts and C++ language constructs. The programming assignments focused on entering provided sample code and practice with the VISUAL C++ compiler commands. Few students visited Tim's office during the first week. The students who did mostly ask questions about VISUAL C++ program compilation and sought clarification on compiler error messages. At the end of the first week, Tim gave the students the first quiz. The quiz focused on testing students' knowledge of programming concepts, its applications, and C++ syntax. Tim was satisfied with the quiz results.

The second week of instruction focused on the object-oriented aspects of C++ language. During the first part of the second week (Appendix F), Tim described the process of creating an object-oriented program by stating, "To create a program, we must define *object* types, create specific *objects* (*instances*) of those types, give instructions for manipulating the *objects* that we have created, and finally C++ *class* declarations."

Tim further explained how to think about *classes* by describing the concept of *encapsulation*. At this point, a typical interaction between Tim and a student occurred about the concept of classes.

Student: Why do we need *classes*?

> Tim: A general purpose language like C++ is designed for the most generic applications. Primitive types hold the most common kinds of values for particular applications. As a result, we want more meaningful types of *objects*: For example, bank accounts, student's records in the registrar's office, airline tickets etc. and that's where *classes* become practical in OOP.

Later Tim indicated that *encapsulation* was useful to put all similar or related information in the same *object*. Tim described *information hiding* as another way to think about *classes*. According to Tim, "*Information hiding* is where a programmer doesn't want to worry about how an *object* works. He just wants to be able to use it. And finally we think of *classes* in terms of *object* types."

Tim supplemented his comments by showing Power Point slides on *information hiding*. Meanwhile a student asked him a question:

> Student: What role as programmers do we have here?
> Tim: When programming with *classes* and *objects*, you need to shift between two roles: *class* designer, the mechanic who understands how the internal details of the *objects* work and *object* user, the driver who just wants to use the *objects* to get some task accomplished. Whereas, *abstraction* helps to keep these roles separate. This allows us to concentrate on just what is important at a particular time.
>
> Tim: For example, bank account. What information needs to be stored in a Bank Account *object*?
>
> Tim: (showing a slide on *information hiding*) The account number, the owner's name, balance of the account, list of transactions, restrictions and penalties, the owner's phone number and address, the bank's name and phone number etc. name of the *attributes* in this case are:
>
>> AcctNumber: the account number
>> Owners Name: the owner's name
>> Balance: the balance of the account
>> Determine the type of each *attribute*

AcctNumber: *string*
Owners Name: *string*
Balance: double

After discussing the concept of *information hiding*, Tim explained the concept of *operations*, by writing the following on the board: "*Operations* are functions that have access to an *object's attributes*. An *operation* may have the side effect of changing the value of some *attributes*." Then he showed the slide to describe the concept of *operations* followed by the syntactical details on how to create *classes*.

During the second half of the second week (Appendix F), Tim's emphasis was on teaching CPS within the context of OOP. Tim started the lesson by writing the following problem on the board: "Create a program to compute bills and coins needed to give exact amount of change." Then Tim asked students questions. Tim's strategy was to engage students in the CPS process through dialogue. A sample of these interactions is:

Tim: What is the input for this program?

Student: Amount of purchase.

Tim: What will be the output for this program?

Student: Change.

Tim: What *objects* do we need here?

Student: Amount.

Tim: Let's identify *methods*. What *methods* do we need?

Student: *Public* and *Private*.

> Tim: [writing the following on the board] The object-oriented problem-solving process is as follows:
>
> Analyze and understand the problem
> List all input values
> List all output values
> Define *objects*
> Define *methods*
> Define *classes*
> Do computation
> Write code
> Test code

During the second week's lab sessions, Tim provided instructions on how to work through example code and on how to perform a desk-check to detect and correct syntax and logic errors. Tim also taught VISUAL C++ compiler instructions, dealing with error messages and writing and testing C++ programs. Students worked individually on their programming assignments while Tim circulated and helped students.

Students frequently visited Tim's office during the second week. Their questions focused on the example code, library functions (provided by Tim), and their use. Students were confused about library files (provided by Tim) and how these libraries converted information into a graphic representation. Students also sought help in understanding the programming assignment ACCUMULATOR-CLASS problem (Appendix H) and had difficulties in understanding concepts such as *constructor*, declaring *classes* and using *class objects.*

Tim felt that he was successful this second week in helping students understand the OOP concepts. However, Tim mentioned that "students understood the concepts, but I did not get enough feedback on that yet. I think that the students' understanding about OOP concepts is mature." Overall, Tim described on his instructional strategies during the second week as less textbook bound and more analytical in nature. He stated, "During the second week there was far too much information. This week was to get not only that here are the facts, but what you do with the facts."

Tim talked about his rationale for the second week's homework assignments. According to Tim, the written exercises were meant to ascertain that students had read the assigned chapters, while the programming exercises he assigned tested the concept of code reusability in solving problems with an emphasis on graphics, math and string *classes*. The assigned work involved students in understanding the mechanics of how the new *class* or the *operation* works. In evaluating the homework for the unit, Tim thought that students faced difficulties dealing with C++ syntax. Tim described the quiz given during the second week as a way to teach and assess students on the syntax and semantics of the C++ language and to evaluate students' understanding of the basic C++ concepts and their applications.

The third week (Appendix F) of instruction was dedicated to imperative aspects with a touch of *objects* (cin and cout) of the C++ language. Tim focused on basic C++ control structures (selection and repetition) and the concept of functions. According to Tim, "control structures will allow us to write programs that are better organized and understandable. We'll learn the logic required to make decisions and the instructions that allow us to tell a computer to select from a number of options and to perform repeated tasks." Tim lectured throughout the week using Power Point presentations on control structures. Lectures during this week were text bound and focused on the syntactical details of the C++. Tim provided example codes on control structures and functions with complete C++ program code. He advised students to follow the example code while working on their programming assignments. Tim also provided instructions (during lectures) on how to use the VISUAL C++ debugger and how to insert print statements at various locations in the program code by using the pause feature to examine the values of the variables.

During the third week, lab sessions did not follow the lecture. Students were advised to work on their homework and programming assignments on their own time. Written homework assignments focused on C++ syntax by evaluating simple and complex relational expressions, if-else statements and repetition (*while*, *for* and *do-while*) statements. The programming assignments included problems in which students had to modify the solution (provided by Tim). In his directions, Tim specifically advised students to modify the provided solution code rather than start with their own solution. During the third week's office hours, Tim answered some student questions and felt that students had difficulty understanding the concepts of control structures and transcribing the problem into C++ code. Tim felt that some students were not fully exploring the problem before they were attempting to complete the problem's solution.

One of the major events of the third week of instruction was the midterm exam. Tim viewed the midterm exam as a tool to identify whether students were "becoming programmers or not." Tim explained that his intent for the exam questions was to ascertain students' progress by getting a "snapshot" of their understanding of the course topics covered. He asserted that the primary focus of the midterm exam questions was to evaluate "students' thinking about programming concepts and their applications." Tim did not emphasize CPS, on the midterm exam. Instead, the midterm examination asked students to solve a multi-level set of questions. According to Tim,

> The midterm exam had three levels of questions. The first was about basic vocabulary and definitions; second, specific concepts, higher level than first level, i.e. more like mechanical. What was the output of a program? The second level was more detailed to get more specific concepts, for instance *reference parameters*, which pass back the information out of the function. I always tested this [*functions* concept] because without it you cannot solve the problem. *Functions* focus on a number of other concepts. *Functions* are pinnacle. The third level, or the application questions, was more general problem-solving; they begin to regurgitate the information or can they apply that information? One point I like to make here is that I used to weigh too much on problem-solving. It was not fair because you are asking students to do creative work in a stressful situation. I pulled it back to 30%.

Tim had mixed feelings about the overall student performance on the midterm exam. He explained that students performed poorly on their syntax knowledge. Tim also commented on his grading of the midterm exam, especially the questions involving students in CPS. "Grades for the midterm exam especially for the CPS part, were extremely soft; they [students] had lots of credit for things that were very wrong." Tim also said that he gave partial credit for incomplete work.

For the fourth week (Appendix F), Tim explained his plans by stating, "I want to complete what I started during the previous week and continue teaching additional

C++ control structures." Tim's instruction during the fourth week also focused on the imperative aspects of C++. Among the topics introduced and discussed was additional information on C++ control structures (selection, repetition) and an introduction to text files. The instruction during week four was textbook bound and was delivered using Power Point presentations. The additional information on C++ control structures during this week included nested control structures, control structures for special case selection and repetition, break, and continue statements for modifying the usual flow of control within a control structure. Due to time constraints, Tim did not allow lab sessions after lectures during week four. The instructor-student interactions were limited to clarifications of syntactical details of the control structures in C++. The written assignment problems covered syntactical understanding of control structures (nested IF, switch, for, nested loops statements, reference arguments and text files). The programming assignments included adding C++ syntax to existing partial codes provided by Tim. Students' questions during Tim's office hours in week four focused on problems such as "dangling-else," pass-by-value and reference arguments, and properly opening and closing file streams.

Tim was asked about student progress with regards to CPS and OOP concepts during the fourth week. Tim felt that class interactions, assignment grading and office hours gave him insight into students' understanding of OOP concepts and evolving student mental models. Tim felt that students were beginning to understand the OOP concepts, especially the concept of an *object*. According to Tim, "Some students' mental models are expanding out enough that they are seeing where next things will fit, but they do not know what the next things are yet."

Tim also felt that some students should be able to develop solutions to the programming problems from scratch. However, he felt that students were still having difficulty transcribing code into C++ language during the fourth week.

Tim described his teaching experience as "100% positive" at the end of the fourth week. He also felt that it was "tough" to evaluate how well students were able to understand and apply CPS and OOP concepts at this point. According to Tim, "I am not sure of what students actually learned with regards to CPS during the fourth week."

Tim started the fifth week (Appendix F) focused on both the imperative and the object-oriented aspects of C++ language. He continued discussing the concept of *classes* during the first part of the fifth week and then discussed arrays. However, Tim was unsure how well students were learning OOP concepts from his instruction. "How they are developing their understanding of the OOP concepts? That I really don't know. They go out and figure it out by themselves and I make some suggestions as to how they should think about it." Tim began the fifth week instruction by providing directions on how to take a pre-existing *class* and augment it with another *operation* and write and test that *operation.* To further explain the concept, Tim made two columns on the board, one for *attributes* and the other for operations. Then Tim asked students to "describe a baseball player" and waited for student responses:

Tim: What *attributes* can we list here?

Student: Height

Tim: Ok. What else?

Student: Team.

Tim: Very good. Give me another one.

Student: Position

Tim. Great. How about the *operations*?

Student: How about create a new player?

Tim: Yea.

Student: Compute batting average?

Tim: Excellent. What else?

Student: Update statistics?

Tim: Very good.

During his Power Point (textbook bound) lectures, Tim explained the concepts of the implementing *class* member functions. He also introduced program design concepts such as designing and implementing new *classes* for specific problems as well as concepts such as member function implementations, scope resolution operator (::), and arrays (single and multi-dimensional). There were no lab sessions after the lecture during the fifth week. Tim explained his rationale for week five programming assignments as "discovering the concepts of *object* and *class*." Typically, in week five, students were provided with a partial solution code for the problem and were asked to complete the solution.

Students' questions during the office hours in week five focused on understanding of certain OOP concepts. Tim thought that students were having difficulties understanding concepts, such as *constructor* names, *constructor attribute* names, and *constructor's* argument names for the *attributes*. However, he argued students were gaining an understanding of the concept of *object.*

During the last day of the week, Tim provided the final exam. The final exam was comprehensive in nature and was similar to the midterm exam. Tim explained his rationale of the final exam questions as a way to evaluate students' understanding of the programming concepts and their applications. The final exam questions tested students' knowledge of basic vocabulary and definitions, specific concepts with syntactical details, program output, and application questions without an emphasis on CPS.

Tim felt students again performed poorly on syntax knowledge. He found that students misunderstood the concept of an *assignment expression* being an *expression* and they did not make progress in understanding the concept of *object* and *class*. The most common mistake was to neglect naming the *object* properly. Students also faced difficulties and even sometimes failed to develop a *class* from scratch.

During Tim's final interview, he was asked about his perception of the progress of the students had made and their ability to understand and apply CPS and OOP concepts. Initially, Tim thought students would lack well-developed mental models to create object-oriented computer programs. He remained concerned with this idea until the end of the term.

Tim described the nature of the CPS process that he taught to students as "anecdotal." He noted that the most difficult OOP concepts to teach and for students to learn were "visualization of the *objects* and understanding *class* as a type." He said that students at the end of the term were still "immature" in their understanding of the CPS process and OOP concepts. According to Tim, students had difficulties in finding appropriate ways of organizing their solutions in part because OOP added a level of complexity by obscuring the flow of control. Students had difficulties with the idea of an *object* and how information flows or communicates in and out of a program.

At the conclusion of the term, Tim believed he had been successful in developing an "aggressive attitude" among students towards computer programming and that he had developed a relationship with his students by interacting with them. He stated, "I think I developed pretty good relationships with the ones who interacted with me. But there is not much time in this short period to really interact with each and with every one of them."

Comparison of Plans vs. Actual Implementation

This section provides analysis of the observations and field transcripts on how Tim's plans and beliefs differed from his actual instruction and assessment. Tim planned his instruction for the introductory CS course to be "student-centered," "like a lab" and that only 50% of the class time would be spent lecturing. However, the analysis of data revealed that Tim's class was teacher-centered, lecture-oriented rather than “like a lab.” He spent more than 50% time on lecture throughout the term, and less on lab session (first two weeks of the term only) with a primary emphasis on the C++ language syntax only.

Tim planned to place less emphasis on the syntactical details of the C++ language and also teach CPS to help students develop CPS skills. His plan was to accomplish this by asking them to use visualization and solve problems in “English-first” (i.e. problem planning). However, Tim did not provide any specific detailed instructions on how to develop comprehensive CPS skills. During his lectures, he periodically recommended that students use flowcharts to design a problem solution, but he claimed that, “It is not important that they [students] draw a flowchart. What is important is that they [students] visualize.” However, in actual implementation of his instructional plans, Tim did not provide specific instructions on how to work out problems in “English-first” or how to draw a flowchart. He also did not explain to students what visualization actually meant or how students could achieve/use visualization to solve problems. On the contrary, Tim commented that teaching problem-solving seemed “time consuming.” The one area Tim did teach problem-solving a technique was through preliminary problem analysis.

For the assessment, Tim’s goal and/or plan was to make sure that students were able to develop CPS skills. As part of this plan, Tim wanted to design 50% of assignments (written homework and programming) so that students would develop solutions from scratch. However, the analysis of assignments revealed that more than 50% of the homework problems had partial solutions. In them students were asked to enter the missing code to get a final solution rather than developing the solution from scratch. Moreover, the analysis revealed that this planned strategy of providing partial solutions to develop the complete solution did not engage students in developing comprehensive program analysis and design practices to develop and/or enhance their CPS skills. The written exams focused on assessing students' basic vocabulary, C++ syntax knowledge, specific concepts and the application of learned concepts, rather than the development of CPS skills. In fact, Tim asserted that, "asking students to do creative work in a stressful situation" such as exams would be unfair.

Tim planned to introduce the OOP concepts in a "non-confrontational way." The analysis revealed that Tim was successful in this goal because he was approachable to his students. As documented by observation and through student interviews, students' comments were favorable towards his "non-confrontational" approach to instruction. For example, Ann (student) said, "Prof. Tim cares, and he pays attention."

Student Profiles

This section includes profiles of the four students who participated in the study and a comparison of their CPS strategies and OOP performances. Each profile includes a brief description of the students' background and their approaches to the problems provided in the formal interviews. Their class work (selected problems from written home work assignments, programming assignments, quizzes and midterm and final exams and excerpts from the student informal interviews) was also considered in the development of each profile. Pseudonyms were used to assure confidentiality and anonymity.

Students were given two problem sets labeled Set I and II to solve during the two formal interviews. Problem Set I (AVERAGE-PRODUCT and CASH-REGISTER) was given in middle of third week of the term, and Problem Set II (PLAYER-STATUS and TRIP-TRACKER problems) was given towards the end of the fifth week. Appendix G contains the problems.

For the AVERAGE-PRODUCT (Set I) and PLAYER-STATUS (Set II) problems, proposed solutions with errors embedded were provided to the students. Students were asked to review the solutions, debug the errors and generate the expected output. Whereas, for the CASH-REGISTER (Set I) and TRIP-TRACKER (Set II) problems, students were provided with a problem statement without a solution and were asked to develop an entire solution.

Adam

Adam was a 19-year-old male student in his sophomore year. He had a score of 590 in the quantitative portion of the Scholastic Aptitude Test (SAT) and a 3.12 GPA prior to taking this course. Adam registered for the course to fulfill the technical elective requirements for his Engineering Physics major. He had no prior programming background. Adam had taken Calculus classes prior to registering for the programming class. In the course, Adam had a perfect class attendance record.

Around middle of the term (middle of the third week), Adam approached the AVERAGE-PRODUCT (Set I) problem by compiling the provided solution code and found syntax errors. The compiler identified syntax errors including (1) "missing ; [semi colon] after identifier in the *cin* statement" and (2) "undeclared identifier." To correct these syntax errors he printed the solution code and the compiler generated syntax error messages. Then he began to desk-check each line of the solution code in sequential order. He circled the lines of code he thought were generating errors.

He shifted to the computer to read each error message generated by the compiler and began by double-clicking at each error message. This process allowed him to examine the line of code and/or areas of the code generating the syntax error(s). He followed the messages and directions provided by the compiler and corrected such errors as "missing ; [semi colon] after identifier *cin* in the statement" by placing the semi colon after the identifier in the *cin* statement. When Adam read the syntax error message "undeclared identifier" his strategy was to change program statements without specifically thinking about the results of such a change. This process did not help him correct the error "undeclared identifier." His next strategy was to continue to guess by listing different choices for *objects* (on paper) and eliminating choices in a sequential order. This trial-and-error strategy led him to reduce the syntax errors but not eliminate them. Adam often injected more syntax errors while correcting the existing errors. He spent most of his time in correcting syntax errors without even an opportunity to correct logic errors and ran out of time. His final solution contained syntax as well as logic errors.

For the second problem (CASH-REGISTER, Set I) in the middle of the term where no solution code was provided, Adam read the problem and then identified on paper the real number *objects* such as purchase price, amount of purchase, and amount of change. Next he referenced the textbook and class notes searching for a similar problem solution. Later without any written plan to approach the problem solution, he began entering C++ code directly into the computer, meanwhile referring to the problem statement and his references. First, he entered the pre-processor directive command #include <iostream.h>. He declared the *objects* (price, payment, change) followed by the code to prompt the user to enter the purchase price within the *main function*. After spending a few minutes to enter the entire solution code into the computer Adam executed the code. However, the execution resulted in a syntax error "not an l-value." Adam realized his mistake, double-clicked at the compiler generated error message to reach the area generating the error and corrected the statement "payment – price = change;" to "change = payment – price;" Later he encountered problems with the output formatting especially with precision; the number of decimal places in his floating-point variables were inaccurate. For example for his variable "dollars" the output statement he coded was "cout << setprecision(3) << dollars;" instead he needed a coded statement "cout << setiosflags(ios::fixed) << setprecision(3) << dollars;" by first including the fixed-point format setiosflags(ios::fixed).

Basically, his strategies correcting these errors was to search for and review similar example codes from his references for output formatting. Adam spent a significant amount of his time in formatting the output without realizing that his solution code was generating an incorrect output due to a logical error since he was incorrectly determining the coin change. His solution code line to determine the coin change was “cchange = change – dollars;” instead he needed his statement similar to “cchange = int((change- dollars)*100);” to accurately determine the number of cents required. To correct the coin change problem, Adam referred to his references to search for a similar example solution code to the given problem and found one. After reviewing the textbook example solution code, Adam incorrectly guessed ways to include a solution code statement “cchange = ((change- dollars)*100);” Adam was frustrated and randomly generated code to obtain the accurate coin change, but ultimately he ran out of time.

During the fifth week, Adam approached the PLAYER-STATUS (Set II) problem with the solution code provided by compiling it. After compilation he realized that the output was inaccurate. His next strategy was to trace the solution code by inserting print statements (learned during class instruction) at certain key locations in the solution code. The insertion of print statements did help Adam solve the problem. He traced through the solution code and detected the point at which the value of the health *attribute* became illegal. Tracing also helped him to keep track of the *state* of the *object* values at different locations in the code and to successfully identify the effect of *operations* on the “health” *attribute* in the code to generate an output free of errors.

Next Adam worked on the TRIP-TRACKER problem (no solution code provided). He identified the *objects*, *methods* and *class* on a piece of paper. He referenced his textbook and class notes to search for examples that were similar to the given problem and used them as a guide for solving the problem. Adam then entered code into the computer by simultaneously referring to the problem statement and his references to create a *class* (TripTracker), *public methods* (*constructor*, reset, purchase gas, miles_per_gallon and cost_per_mile) and *private methods* (total_gallons_of_gas, total_cost_of_gas and trip_miles). Adam incorrectly used a *for* loop to calculate and print the gallons purchased, cost per gallon and miles driven. After completing and executing the code, the compiler reported syntax errors. Some of the syntax errors were "loop has no body," "including a return type with constructor's prototype" and "missing : : scope operator." Adam successfully corrected the errors by following the error message instructions provided by the compiler and then by adding a needed semicolon to close the loop, a return statement for the *constructor*, and the *scope operator* : : in the header line of the member function defined in the *class* implementation section. However, these corrections of the syntax errors did not obtain a correct result. Adam had a logic error because he had not defined the *object* "cost." He attempted to use the *operations* calculating cost that read and/or manipulated data for "cost," but the *object* "cost" never existed. To correct the logic error he read his solution code on the screen sequentially and guessed at a correction by defining different objects without considering the effects of the creation of new *objects*. This guessing strategy generated no solution for the problem. His next strategy was to invoke the VISUAL C++ debugger (learned during class instruction) to divide the solution code into smaller sections and execute the solution code one line at a time. This divide-and-solve strategy helped Adam examine his program solution code in a step-by-step mode and to keep track of all *objects* declared. By examining the program solution code in a step-by-step mode, he pointed to his error and defined the *object* cost. For his final solution Adam generated a correct output for the problem.

Adam's approach to solve problems without prior comprehensive planning/designing was noticeable in the formal interview problems (Appendix G) throughout the term. However, changes were noticed in Adam's debugging approaches. Around the middle of the term, Tim introduced inserting print statements at several locations in the code to examine values of the variables in pause and the VISUAL C++ debugger to help students deal with the errors (syntax and logic). Adam's abilities to detect and correct both types of errors improved towards the end of the term, as a result of class instruction on debugging. Adam successfully used print statements in the PLAYER-STATUS problem to correct the logic error. He also successfully used the VISUAL C++ debugger to divide the TRIP-TRACKER problem into smaller chunks dealing with them more efficiently as compared to middle of the term problems.

An analysis of Adam's class work problems (Appendix H) further described his CPS strategies and OOP performances while solving the given problems. Adam typically did not pre-plan his approach to problems, such as conducting problem analysis, and/or by developing an algorithm. He approached them by directly inputting the solution code into the computer. He typically designed his solution code in terms of C++ language code by converting the directions given in the problem specifications line-by-line into C++ language. However, in some instances he performed some incomplete problem analysis such as identifying *object* and *methods* prior to generating the solution code.

Adam's work (throughout the term) also revealed his misunderstanding of the concept of *object.* He initially named/declared *objects* that he never used in his solution code. Adam's class work also revealed misunderstandings towards the beginning of the term with control structures such as selection and loops. For example, Adam did not use compound statements (when needed). As a result, the compiler defaulted to unpaired "*if's* and *elses*." In some instances he used the assignment operator =, instead of using the relational operator = =, generating an infinite loop situation since the expression with assignment =, was placed in a statement prior to the loop. However, by the end of the term Adam improved his understanding of control structures such as loops. For example, for the RE-WRITE A LOOP problem (Appendix H), Adam rewrote a *for* loop into a *while* loop demonstrating his basic level of understanding of both types of loops. Adam did not deal with *function arguments* appropriately. He said, "I am always confused between the *actual* and the *formal arguments*."

Adam approached the debugging process using multiple strategies throughout the term (1) following the compiler generated messages/directions and (2) guessing. However, from the middle towards the end of the term his approach to the debugging process also included: (1) performing a desk-check to mimic and execute each C++ statement as the compiler would perform (writing each *object* encountered in the solution code and then listing each value that should be stored in the *object* as each input and assignment statement was encountered), (2) program tracing (using the print statements into key location in the program to track the *object* values), and/or (3) using the VISUAL C++ system interactive debugger. In many instances, program tracing and the use of VISUAL C++ system debugger helped Adam to divide-and-solve his problems.

During the informal interviews, Adam explained the computer problem solving process as:

> When you are given a problem you need to read the problem, what the problem is asking you to do. If the problem is asking you to write a *class* so you write a *class.* If the program is asking you to write a *while*-loop, then you write a loop. A programmer has to first create a blue print before he can solve the problem.

Adam viewed OOP as "writing programs based on objects, where *objects* interact among each other and objects hold values." He expressed his feelings towards the class by complaining about the lack of actual programming and examples done in the class. According to Adam, "I am used to Calculus class where half of the assignments are done in class, so students know how to solve the homework problems since they have seen many examples." When asked about his reaction to the quizzes with respect to CPS involved, Adam replied, "Quizzes are mostly about syntax and vocabulary of the C++ but not about problem-solving and programming." He further explained his feelings towards class instruction. "During the first three weeks we just copied programs; we need to write programs from scratch." Adam's study practices included reading each chapter in the textbook prior to and after attending the classes and working on homework problems.

Throughout the term, Adam completed and submitted all his written home work and programming assignments on time and earned 100% grade on average. His average score for all quizzes was 80%. For the midterm exam he earned 85% and 82% in his final exam. His final grade for the course was "B."

Ann

Ann was a 30-year-old female student in her freshman year. She had a GPA of 2.67 prior to taking this course. Ann's SAT scores were not available. However, in the institutional entrance exam in Mathematics, Ann obtained 75% (grade C). She registered for the course to fulfill her major course requirements in computer science. Ann had no prior programming background but she had taken courses in mathematics, including Elementary Mathematics and Pre-Calculus, prior to registering for the class. Ann had perfect class attendance record.

Ann explained that she had grown up in one of the southern states where she had to face racism on regular basis. According to Ann, "My way to deal with the racism was sports and I love sports." Ann wanted to earn a college degree in computer science to pursue a career as a computer programmer.

For the AVERAGE-PRODUCT (Set I) problem in the middle of the term Ann compiled the solution code and encountered syntax errors. Her strategy in fixing syntax errors was to read the error messages generated by the compiler and then double click at the error messages to correct them on the computer. Ann responded to the exact code line(s) and/or the area of the code(s) generating the syntax errors. She corrected the syntax error "newline in constant" by adding the missing double quote i.e. changing the given code line "cout << "Please enter the values << endl;" to "cout << "Please enter the values" << endl;" For other syntax errors, such as "undeclared identifiers," Ann read each line of the code in a sequential order, searching for code lines that were generating errors and correcting the errors by following the messages and directions provided by the compiler until she eliminated all the syntax errors. She compiled the code again without any syntax errors but was stumped to find out that the output was incorrect because the code was not generating the value of the *object* product. Ann reviewed the code again in a sequential order line-by-line and from top-to-bottom and was able to figure out that statement "product = num0*num1*num2*num3*num4;" needed to be placed prior to the statement "cout << "The product is:" << product << endl;" She was successful in generating the correct output.

Ann read and re-read the second problem, CASH-REGISTER (Set I) problem. On a piece of paper she identified cash register, clerk and change as *objects*, amount of purchase, payments and the amount as *attributes* of the *objects* and change (number of dollars, quarters, dimes, nickels and pennies) as *operations* from the given problem specification. Next she searched for a similar example program code from the class references (textbook and class notes). Once she found a similar example, she copied the code from the example program code by altering the code sequence hoping that it would work for the CASH-REGISTER problem also. She entered the pre-processor directive "#include <iostream.h>" followed by another pre-processor command "include <math.h>." Her main program code included statements declaring the *objects*, followed by the statements that allowed the user to enter values, such as purchase amount and amount paid.

After completing her coding, Ann compiled her solution code and encountered a syntax error, "undeclared variable," because of the missing declaration of "purtotal" identifier. To correct this syntax error, her strategy was to double click at the error message generated by the compiler to identify the location in the code generating the error. Ann declared the identifier "purtotal" in her solution code. She compiled the code again but was unable to produce the correct output since her solution code had logic errors because of her calculations for change (dollars, quarters, dimes, nickels and pennies). In order to generate the correct code, Ann wrote on a piece of paper different possible codes and eliminated the ones she deemed incorrect to calculate the correct amount of change. However, this strategy did not help her generate the correct solution code. To calculate dollars in change, Ann had the statement "dollars = change;" whereas she needed a statement similar to "dollars = int(change);" followed by “coinChange = int ((change – dollars) * 100);” To calculate quarters, dimes, nickels and pennies she generated the following code respectively “quarters = 100% 6.92; quarters = %25; dimes = /10; dimes %; nickels/5; nickels%; pennies = nickels;” However, she needed statements similar to “quarters = coinchange / 25; coinChange = coinChange % 25; dimes = coinchange / 10; coinChange = coinChange % 10; nickels = coinchange / 5; coinChange = coinChange % 5; and pennies = coinChange;” respectively. Later Ann guessed by changing statements without specifically thinking about the effects of such a change on her solution. However, she was unable to find the correct solution in the allotted time for the CASH-REGISTER problem since her program contained logical errors.

In the fifth week, Ann began her work with the PLAYER-STATUS (Set II) problem by compiling the given solution code. Next she used the VISUAL C++ debugger to trace the solution code. The VISUAL C++ debugger allowed her to examine the code in smaller segments and one line at a time. She studied the changing values of variables at the different stages of the program execution but was unable to figure out the operation that was affecting the health *attribute*. Ann changed different *operations* without realizing the effect of changes made. She spent a significant amount of her time in changing the given code and operations and then changing them back. Meanwhile she ran out of time in the process. Her final solution contained logic errors and Ann did not consider working on them.

For the TRIP-TRACKER problem (Set II) Ann began by reading the problem and searching for examples similar to the given problem in her class notes, textbook and the Internet. Ann was familiar with the use of search engines over the Internet (class instruction was not provided on using Internet to find example solution code) and found a similar example. Later she copied and pasted the solution onto the editing area of the Visual C++ project and then adjusted the code. Her solution code included the declaration of the Trip-Tracker *class* with *private* and *public methods* in the *class* declaration section. For the *private method* Ann made a declaration of the following *objects* as data members: "Trip Tracker, Gas, Cost-Gas, Miles-driven" and for the *public method* she declared the following member functions: "Trip-Tracker ( ), reset( ), get_cost_gas( ), get_miles_driven( ) and cost_trip ( )." Ann used a *while* loop to allow the user to enter the gallons purchased, cost per gallon and miles driven. After compilation of the solution code, Ann encountered syntax errors because she had used the assignment operator = instead of the relational equality operator = = in the tested expression of the *while* loop and also placed a semicolon at the end of the *while* loop parentheses. To correct these errors Ann double clicked at the compiler provided error messages to reach the area(s) of the code generating the errors. She corrected the errors by replacing the assignment with the equality operator and by removing the semicolon from the end of the *while* loop. Later Ann compiled her solution code but found her output to be incorrect since her solution code included logic errors because (1) she misunderstood the operator precedence and (2) there was a division by zero attempted. In case of the first logic error, her solution code included "milesPerGallon = endMileage – startMileage / gallonsUsed;" However, the code should have been similar to "milesPerGallon = (endMileage – startMileage) / gallonsUsed;" placing division at a higher precedence than subtraction. In case of the second logic error Ann computed and returned the average milesPerGallon. To avoid a division by zero error, she did not return zero, meaning she did not know how and when the Total-Gallons-Of-Gas variable was zero.

Her strategy to correct the logic errors was to review her solution code line-by-line in a sequential order on the computer screen but she was unable to find the problem. Next she printed her solution code and desk-checked the code. During the desk-checking process, Ann wrote different possible code combinations eliminating the ones that in her opinion did not work to attain the correct solution. After spending a few minutes on desk-checking without producing a result, Ann invoked the Visual C++ debugger to execute her solution code and then examined one line at a time and different variable values with each pause. She was unable to detect her logic errors in the time allotted.

Throughout the term, for the formal interview problems (Appendix G) changes were noticed in Ann's debugging approaches but her approach to solve problems remained unchanged; no comprehensive plan and/or design was used prior to finding the solution code. She successfully used VISUAL C++ debugger to detect and correct the logic error in the PLAYER-STATUS problem. She also desk-checked and used the VISUAL C++ debugger for the TRIP-TRACKER problem. Ann's abilities to detect and correct both types of errors improved towards the end of the term, as a result of Tim's instruction on debugging techniques such as desk-checking, VISUAL C++ debugger.

The analysis of Ann's class work problems (Appendix H) revealed that throughout the term she typically approached problems without planning, beginning her solutions by entering the codes directly into the computer. However, in some instances Ann did approach the problem solution by providing a preliminary problem analysis and design such as, identifying the *objects, attributes, operations* and *methods* prior to entering her solution code into the computer.

Ann's work and conversations during the informal interviews revealed that she had difficulties with the concept of *object* throughout the term. For instance, she did not name, or she forgot to name, the *objects* properly in many of the given problems. Ann had difficulties in generating the correct values of the *objects*, i.e. variables. For instance, in homework problems towards the middle of the term and the end of the term, the problem asked the students to obtain the values of the *objects* from the provided solution code. However, Ann provided the resultant values in the wrong *objects*. Her work also revealed the difficulties she faced with the concept of *class* and writing the problem solution from scratch. For example, for the SALARY-CLASS (Appendix H) she was unable to correctly develop the salary *class* and calculations involved for the problem. For the *operation* retirement benefits she returned 5% of the salary. However, she needed a more complex solution by developing a *function* in the implementation section of the Salary *class* than returning the percentage of the salary in the main program. However, her *operations* and *attributes* for the SALARY-CLASS problem were correctly performed.

Ann also had difficulties throughout the term identifying the correct sequencing of the objects and their values in complex control structures such as loops. For instance for the RE-WRITE A LOOP problem (Appendix H) where Ann had to rewrite a *for* loop into a *while* loop, she needed to rotate an angle from zero to pi and print out the values. The initialization of the code for the loop was inside the body of the loop causing it to be recalculated each time. In her solution code the loop was being controlled by an *object* called "angle." She initialized "angle" to zero from inside the loop, printed the sine of the "angle" and then incremented the "angle." However, these instructions led to an infinite loop situation where the solution code would print the value of the sine of zero indefinitely.

Ann approached the debugging process throughout the term by first reading error messages generated by the compiler and then double clicking at the error messages. By double clicking Ann was able to reach the exact code line(s) and/or the area of the code(s) that generated the syntax errors; from this point she corrected the error messages by following the instructions given by the complier. However, in many instances she was unable to understand the error messages and directions provided by the compiler. According to Ann, "It is difficult to decipher these [compiler] messages." By the end of the term, she also used the desk-checking technique to hand-trace each line of code in a sequential order and the Visual C++ debugger to execute the code one line at a time.

During the informal interviews (towards the end of the term) Ann was asked about her own understanding of problem-solving and OOP. Ann responded without providing any supporting examples:

> (1) Understand the problem (2) analyze the problem (3) research and build parts of the problem (4) put everything together i.e. design (5) test the design and fix any errors and (6) implement working program.

Ann described her OOP programming process without supporting examples as consisting of two parts: (1) descriptive information about the *object* type and (2) the specifics of the *objects*. Ann assumed from her class instruction that in object-oriented programming most of the code already exists. According to Ann, "the teacher gave us most of the *classes* which were already written and we just had to fix some parts of the program and not much was given from scratch." When asked about her understanding of OOP concepts Ann replied, " I see the *objects* as *classes* or items. *Objects* can be reused. The *values*, *arguments*, *attributes* that are hidden are *private*, i.e. information that nobody really needs to know. You do not want this to be altered."

When asked about her feelings and experiences in the introductory OOP class, Ann replied, "This class is very difficult but I learned a lot." Ann described her study practices as "sticking close to the textbook, and class notes, memorizing definitions." When asked about her reaction to the graded class work, Ann replied, "I am satisfied with my grades."

Throughout the term, Ann did not complete and submit all her written homework and programming assignments on time and earned 82% grade on average.

Her average score for all quizzes was 70%. For the midterm exam she earned 75% and 72% in her final exam. Her final grade for the course was "C."

Mel

Mel was a 22-year-old female student in her senior year. She had a GPA of 2.8 prior to taking this course. Mel's SAT score in the quantitative portion was 500. She registered for the course to fulfill her required elective requirements from the CS department. Mel had no prior programming background. She had taken courses in mathematics including College Algebra and Business Mathematics at the college level prior to registering for the class. Mel did not attend all classes.

Mel's reason for taking this course was her father. Mel explained, "My father works as a computer consultant and I might want to minor in Management Information Systems and this class might help me. Besides it is required that I take one computer class for my major."

At the middle of the term, to solve the AVERAGE-PRODUCT problem (Set I), Mel compiled the code and identified syntax errors. Her strategy to fix the syntax errors was to read the error messages provided by the compiler and correct the given solution code by guessing in order to alter the sequence of the given solution code. For instance, when Mel encountered the "sum as undeclared identifier" she declared the sum identifier at various other locations in the solution code. This strategy of introducing code led to the introduction of more syntax errors, in particular the "undeclared identifier" error. Mel repeatedly made similar mistakes and in her final solution she had several syntax errors including "undeclared identifier, left operand must be l-value" she ran out of time prior to removing syntax errors and never considered potential logic errors.

After Mel read the second problem, the CASH-REGISTER (Set I) problem, she referred to the textbook and the class notes to search for similar problems. After finding a similar problem, Mel entered code directly into the computer from the references assuming that the code she copied would work as a problem solution. Her solution code included the preprocessor command "#include <iostream.h>" followed by object declarations such as price, payment and change. Next she added C++ code to ask the user to input values followed by calculations for change, number of dollars needed in change, quarters, dimes, nickels and pennies. Finally she inserted print statements to print out the desired results. After entering the code into the computer, Mel compiled the code and received the syntax error: "error LNK1120:1 unresolved external error executing link.exe.." because she created a Win32 Application project, rather than a Win32 Console Application project. To correct this error, Mel created a new project in VISUAL C++ environment and then successfully ran the code without the "LNK 1120" error. However, she encountered several other syntax errors such as undeclared identifier "purchase space total" since she attempted to use the variable "purchase space total" but had not declared this variable prior to its use. To correct this error Mel introduced new variables into the solution code without planning for their use. This strategy led to more syntax errors such as "undeclared identifiers." Later Mel ignored the error message "undeclared identifier" and read other error messages generated by the complier in the hope that if she could correct other syntax errors the error "undeclared identifier" would also be corrected. Her strategy to correct the syntax errors was to guess code to be entered into the existing solution code without recognition of the error messages/directions generated by the complier. In her final solution to the CASH-REGISTER problem, Mel had several syntax errors as well as logic errors. Some of her logic errors included incorrect calculations for determining quarters, dimes, nickels and pennies such as her calculation for determining quarters was "quarters = change – 25;" However, she needed a statement similar to "quarters = coinChange / 25; and coinChange = coinChange % 25;"

Close to the end of the term, Mel approached the PLAYER-STATUS (Set II) problem, by compiling the given solution code. She used the Visual C++ debugger to identify errors. The debugger helped Mel to examine the values of the variables by pausing the given code. Next Mel added and transposed code. Mel added a new member function in the *class implementation* section and than altered the sequence of the given solution code. However, guessing at code led to syntax errors. One such syntax error was generated since she forgot to include the *class* name and *scope resolution operator* :: in the header line of member *functions* defined in the *class implementation* section. Mel was unsuccessful in figuring out the syntax errors and did not recognize the logic errors in the allotted time.

For the second problem, TRIP-TRACKER (Set II) Mel identified a list of "things" on a piece of paper: "buy-gas, amount-of-gas, cost-of-gas, miles-driven, Ave-mpg and cost-per-miles." She then referred to her textbook and class notes and searched for example solution codes similar to the given problem. Next she moved directly to the computer and entered her solution code for the given problem. Mel spent a significant amount of the allotted time thinking and entering the solution code for the problem. Her solution code started by including the pre-processor commands "#include <iostream.h>, #include "TripTracker.h" followed by *object* variable declarations with initialization of some of the *object* variables. Later she included statements allowing the user to enter values and output the values. At the completion of her solution code, Mel compiled her solution code and received a syntax error "fatal error C1083: Cannot open include file: 'triptracker.h': No such file or directory Error executing cl.exe." This error was identified because the "triptracker.h" file never existed. Mel was unable to solve the problem in the allotted time. Several other syntax errors were present in her final solution code. As a result, Mel was unable to generate any solution for the given problem.

Changes in problem-solving strategies were not noticeable in Mel's formal interview problems (Appendix G) throughout the term. She approached the problem without comprehensive planning. Throughout the term, she struggled to understand and interpret with the compiler generated messages. However, she did use the VISUAL C++ debugger towards the end of the term but was unsuccessful in detecting and correcting errors (syntax and logic) using this approach. Mel did not improve her debugging abilities throughout the term.

The analysis of Mel's class work problems revealed that she approached the solution of problems without developing a plan and went directly to the computer to enter code. In a few cases she did identify *objects* prior to approaching the solutions. Her approach to identifying problems when solutions were provided involved transposing or altering the sequence and guessing at new code to be entered. For example, for the MONEY-CLASS (Appendix H) problem Mel tried to transpose the *constructor* from the provided *class* declarations for the "Money" *class*. This strategy of transposition and guessing caused the introduction of a variety of syntax errors such as using the same name for a data member of a member *function* and defining more than one default *constructor* for a *class*.

Mel's class work also revealed that in several instances her solution code contained information that was not needed and she misunderstood what was required to solve the problem correctly. For example, the COST-OF- FENCE (Appendix H) problem asked for a prototype for the *constructor*, but she provided a declaration of an *object*. When the problem asked for a prototype for a *member function* she gave what appeared to be a call to the *member function* and when the problem asked to declare a *class* she provided what appeared to be an *object* creation not a *class* declaration. However, for the COST-OF-FENCE problem she provided the *private* and *public* members correctly since she could copy similar code from the textbook. Mel also had difficulties in understanding loop sequencing. For instance, for RE-WRITE A LOOP (Appendix H) a loop problem where she had to rewrite a *for* loop into a *while* loop, Mel did not sequence the body of the loop correctly.

From the interviews and the analysis of Mel's work, her solution codes typically included the use of sample C++ syntax. Following and copying the sample code did not help Mel in generating correct solution. She was overwhelmed close to the end of the term and was not comprehending the course material. According to Mel, "Too much is going on, too much information for me at this time and I am reaching a point where I do not even know what questions to ask [the instructor] any more and I do not even know what to do."

Mel approached the debugging process by reading the error messages provided by the complier. However, mostly she did not understand the messages or the compiler provided directions to correct the errors. She also used the VISUAL C++ debugger but was unable to use it effectively throughout the term.

Mel explained her understanding of the OOP concepts and the CPS process:

> CPS is to know what *objects* need to be declared. Replicate previous programming assignments. However, to try to figure out what they want from you is the hardest. OOP is like when you use the *object* you put information to get the output. *Object* is like an alarm clock or the oven like we discussed in the class. I am not sure what is a *class*?

Mel's study practices included memorization of definitions, reading the textbook and class notes. According to Mel, "I just repeat what I learn in the class. Mostly I depend on memorization. If I see some example in the book or somewhere then I try to solve the given problem based on the example given."

Mel felt that having a strong mathematical background would have helped her in becoming more comfortable in this class. Mel explained her learning experience in the class as "difficult." According to Mel, "I think that the class is hard since it builds upon what you already know and the class speed was too fast."

Mel, throughout the term, did not attend the class regularly. She did not understand several concepts in the class and as a result was unable to complete and submit all her written homework and programming assignments on time. Her grade for the assignments was 65% on average. Her average score for all quizzes was 85%. She earned 80% on the midterm and 65% in the final exam. Her final grade for the course was "C."

Jose

Jose was a 27-year-old male student in his sophomore year. He had a GPA of 3.5 prior to taking this course. Jose's SAT scores were not available. He registered for the course to fulfill his major requirements in computer science. Jose had no prior programming background. However, he had a strong mathematical background. He had successfully completed Calculus I, Calculus II, Calculus III and Differential Equations. Jose did not attend all classes.

At the middle of the term, when Jose received the AVERAGE-PRODUCT problem (Set I), he compiled the problem and found syntax errors. His strategy to correct syntax errors was to read each error message provided by the compiler and checked each error by double clicking at the error message. By double clicking at the error messages in a sequential order, Jose reached the area(s) or the exact line(s) of the code generating the error(s). For the error message "new line in constant" he added the missing double quote at the end of the message string and corrected the error. Jose used the same error correcting strategy to correct other syntax errors until he corrected all of them. While Jose was sequentially reading the provided solution code to correct the syntax errors, he also detected and corrected logic errors such as an incorrect sequencing of the statements. For example, Jose correctly placed the variable "product" prior to the print statement for the "product." Jose generated a correct solution in the allotted time.

For the second problem, the CASH-REGISTER problem (Set I) Jose read the problem. Next he searched for a similar problem solution by referencing his textbook and class notes. His next strategy was to identify "Purchase Total, Payment, Change, Dollars, Quarters, Dimes, Nickels and Pennies" as *objects* on a piece of paper. Later Jose wrote the analysis of the given problem (as instructed in the class) as follows:

> At the cash register we will need the purchase price and the payment. The change = payment – purchase price and will be a real number. The whole part will contain the change in dollars and the fractional part will contain the change in cents i.e. quarters, dimes, nickels and pennies. So if we have $10 as a payment to buy something for $3.08, the needed change will be $6.02. The $6.92 change will contain $6 bills, 3 quarters, 1 dime, 1 nickel, and 2 pennies.

Jose continued with the program design by listing (on a piece of paper) the input, processing steps and the outputs (IPO):

```
Input: user will be asked to input price and payment.
Processing:  Calculate the change.  Change = 10.00 – 3.08
             Calculate whole part of the change i.e. dollar amount
             Calculate fractional part of the change
Output:                 Purchase Total        3.08
                        Payment               10.00
                        Change                6.92
                        Dollars               6
                        Quarters              3
                        Dimes                 1
                        Nickels               1
```

Next Jose started to enter the C++ code into the computer with the help from his problem analysis, inputs, outputs, textbook and the class notes. His solution code included the preprocessor command "#include <iostream.h>" and main function followed by the variable declarations such as "Purchase Total, Payment, Change" as doubles (real numbers) and "dollars, quarters" as integer values. Then he inserted the statements allowing the user to enter values for the purchase total, to calculate the change followed by output statements printing the purchase total, price and change. After entering the solution code, Jose compiled the code. However, he had a syntax error missing a semicolon before an identifier. Jose double clicked at the error message so he could access the area(s) and/or lines generating the errors and then corrected each error by following the directions provided by the compiler and by placing the missing semicolon where required. Jose complied the solution code again and had no syntax errors. He continued by writing the code to calculate the dollars and cents needed in the change followed by the statements to calculate quarters, dimes, nickels and pennies and then coded the statements to print the change in dollars and cents. After entering his solution code, Jose complied the code and identified syntax errors since he had forgotten to enter " ; " at the end of an assignment statement and had misspelled the identifier "change." His strategy to correct syntax errors was reading the code line-by-line, sequentially from top-to-bottom and then correcting the errors using the compiler messages and directions.

After correcting the syntax errors, Jose compiled the program and found the solution code was free of syntax errors. However, his output was incorrect due to logic errors. His statements calculating quarters, dimes, nickels and pennies were not generating the fractional parts correctly. Jose obtained the hard copy of his solution code and started to desk-check his solution code line-by-line in a sequential order until he found the logic errors. He tried various combinations of calculations until he eliminated incorrect calculations and generated correct calculations and the correct solution.

Close to the end of the term, for the PLAYER-STATUS problem (Set II), Jose began by compiling the given code. He instantly realized that the program was generating the wrong output. Next he checked the program on paper in a sequential order and pointed out that one of the *operation* was affecting the health *attribute*. However, he did not mention the *operation*. Jose invoked the VISUAL C++ debugger, which helped him to execute his solution code one line at a time and to look at the values of variables using the pause feature. Jose successfully changed the *operations* that affected the *attributes* of the *object*. He ran the program and generated the correct results.

For the second problem, the TRIP-TRACKER problem (Set II), Jose spent time reading the problem to understand it. Next he identified on paper a *class* and named it "TripTracker," followed by the identification of "total cost, total miles, total gas" as *objects*, "TripTracker" as *constructor*, "total cost, total miles, total gas" as *private methods* and "add-gas, cost-of-gas, miles-driven, average-mpg, avg-cost-per-mile" as *public* methods as part of his problem analysis. Then Jose entered the solution code into the computer. His solution code included a *class* called "TripTracker" with a *class* declaration and implementation sections followed by the main *function*. After entering the code entry for the declaration section of the "TripTracker" *class* into the computer, Jose compiled his partial solution code and encountered syntax errors because he forgot to terminate the *class* declaration section with a semicolon. His strategy to correct the error was similar to his method for the CASH-REGISTER problem at the middle of the term, double clicking at the error messages and following the directions provided by the complier. He corrected the needed semicolon in the declaration section of the "TripTracker" *class*. Jose continued to enter his remaining solution code into the computer and then complied his solution code. He encountered syntax errors since he included (1) a return type with *constructor's* prototype, (2) used the same name for a data member as for a *member function*, and (3) forgot to include the *class* name and "*scope resolution operator ::*" in the header of a member *function*. Jose repeated his syntax error correction strategy to successfully correct all the syntax errors. However, he had a logic error because the loop in the solution code had one less iteration than needed. To correct this logic error Jose invoked the VISUAL C++ debugger allowing him to examine the code one line at a time and the variable values used for his loop at each pause. Jose corrected the logic error, compiled the solution code, and generated the correct output for the TRIP-TRACKER problem in the allotted time.

Jose did not use a similar approach to plan the solution from the middle of the term towards the end of the term. For example, in the middle of the term for the CASH-REGISTER problem he listed the input, processing and the output for the problem. However, for the TRIP-TRACKER problem at the end of the term he did not generate a similar listing. Throughout the term, Jose did not approach the problem by developing a comprehensive solution plan and design. Typically, he went directly to the computer to code the solution. Jose improved in his debugging strategies from the middle to the end of the term. As soon as he received instruction on desk-checking and VISUAL C++ debugger, Jose successfully adopted these debugging tools and solved his problems.

The analysis of Jose's class work revealed that he generally declared the *objects*, named the *objects* and constructed the *class constructor* correctly. His work also revealed that he understood what and how *function* arguments were needed in developing the solution. Jose at times misunderstood control structures (loops being infinite etc.) and simple data structures such as arrays.

In one situation while solving his written homework problems, his work revealed misunderstandings such as "Array index bound errors." For instance, the following code showed the array's indices ranged from 0 through 6. However, the array's indices inside the *for* loop were incorrect and ranged from 1 to 6 since the initializing list started with variable i = 1.

```
const int index = 6;
int grades[index];
int i;
```

```
for (i =1; i <= index; i++)
{
        cout << Please enter a grade: ";
        cin >> grades [i];
}
```

Jose typically approached the debugging process throughout the term by reading each error message provided by the compiler, double clicking at each error message. He performed desk-checks and also utilized the VISUAL C++ debugger from middle to the end of the term as soon as he learned about it. By using the VISUAL C++ debugger he examined the code one line at a time and different variable values with each pause.

During the informal interviews, Jose's explained his understanding of the CPS process. According to Jose, "CPS resembles mathematics problem-solving. However, finding a computer solution is a complex process." Jose listed the steps he used while solving computer problems as "(1) look at the input/output, processing and (2) search for similar examples to the given problem."

Jose explained some OOP concept understanding as follows:

> An *object* is anything that holds a value. An *agent* can be somebody who initiates the action. *Operations* are individuals that take somebody to get involved. However, I am confused about the difference between the *attributes* and *operations.*

Jose's study practices included mainly memorization. His overall reaction to the class and grades was favorable. According to Jose, " I am really enjoying this class."

Throughout the term, Jose completed and submitted all his written home- work and programming assignments on time and earned 100% grade on average. His average score for all quizzes was 98%. He earned 93% on the midterm and 95% in his final exam. His final grade for the course was "A."

Instructional Strategies and their Impact on Student Learning

This section contains a synthesis of the results directed at answering the research questions posed in this study. The first research question addressed the characterization of Tim's instructional strategies to engage students in CPS. The analysis of the results revealed a teacher-centered, text-bound lecture/lab instruction that was focused on syntactical details rather than the underlying programming logic. Tim focused on the imperative paradigm and/or procedural aspects with an introduction to the object-oriented aspects of the C++ language. Exams and assignments were geared towards the memorization of basic definitions and facts, knowledge of specific programming concepts and their applications, and syntactical details of the C++ programming language.

Typically, Tim's instruction used a Power Point presentation and taught students to find a computer solution to a given problem without developing a comprehensive plan/design. He frequently used "example codes" from the textbook and/or other sources to teach programming. He also provided students with partial code, asking them to develop a complete solution code. During his instruction, Tim avoided a discussion of problem planning due to lack of time. Tim felt that teaching CPS in the introductory CS class was "time consuming" and that was unfair to demand students to complete "creative work i.e. problem-solving in stressful situations" such as exams and quizzes. Tim did not consistently teach any formal methodology of CPS such as problem-solving heuristics and/or strategies to engage students in CPS. Instead, he introduced CPS concepts occasionally throughout the term. For example, in one particular instance (towards the end of second week), Tim showed and instructed students to observe the following CPS process (in general) to solve given problems:

1. Analyze and understand the problem.
2. List inputs/outputs and processes.
3. Identify/define *objects*, *methods* and *class*.
4. Code and test.

However, one area Tim was consistent in teaching CPS was in identifying/defining *objects*, *attributes*, *methods* and *class* for the given problems in order to engage students in preliminary problem analysis, a CPS strategy. In addition, Tim recommended the use of abstraction (hide details and focus on a general view of the problem), visualization and thinking of related problems (from textbook and class notes) while solving the given problems. Tim also recommended that students think about the problem solutions in terms of the English language (work on the given problem in "English first" and then later think in terms of computer logic and/or C++ language codes). However, Tim did not provide specific instructions or problems for students to practice abstraction, visualization and thinking in "English first." In other words, he seemed to believe that students needed to learn programming C++ syntactical details first to solve computer programming problems rather than applying a comprehensive problem-solving approach (a comprehensive plan and design and then code).

Tim planned his instruction to place less emphasis on syntactical details of the C++ programming language. He also planned to design assignments in a manner where at least 50% of assignments would allow students to develop an entire solution from scratch. However, his implementation of the instructional plans did not achieve the stated goals. Students understood the effect of Tim's instruction on their learning of CPS and OOP. As Ann noticed, "the teacher gave us most of the *Classes* which were already written, and we just had to fix some parts of the program and not much was given from the scratch," Adam agreed, saying, "In this class we just copied programs. There was not much programming."

Most of the student effort in Tim's class concentrated on learning details of the syntax of C++ language. The instructional emphasis on syntax and/or the imperative aspects of C++ language also demanded students to think in an analytical and procedural i.e. sequential and/or mechanical manner.

The second question dealt with identifying how novice students solve computer problems in an introductory computer science course with an introduction to object-oriented programming. The analysis of the students' results revealed that in developing an original solution code for the given problems, nearly all students approached the problems without a comprehensive written plan/design throughout the term. This approach appeared to be the impact of the instruction they received in class since Tim did not instruct students to prepare a comprehensive plans/design prior to solving computer problems. Without a written plan/design, the students' approach to CPS was focused on directly entering solution code into the computer and guided by the C++ commands and instructions. Their knowledge of programming concepts such as *objects*, *operations*, control structures (selection, repetition) etc. was inadequate, fragmented, and inaccurate. In most computer problem situations, students lacked creativity and selectivity to effectively find the computer solution using OOP concepts.

The analysis also revealed that students typically used four strategies to develop the computer solutions. In order to understand the given computer problem, students (except Mel) read the problem underlining key words and/or sentences. After becoming familiar with the problem, students then did a preliminary problem analysis although that analysis was often incomplete. This analysis typically included listing of input, process and output, an identification of *objects*, *methods* (*private* and *public*), *class*, and *attributes*. Throughout the term, Tim encouraged and demonstrated the identification of *object* and *methods*. After a preliminary problem analysis, they typically searched for sample code and/or examples to solve the given problem. Students referred to their textbook and class notes to search for possible solution codes from similar problems. They also sometimes drew analogies from their personal life experiences. Then they generated C++ code by translating the given problem specification line-by-line and/or word-by-word by using the programming knowledge constructs learned in the class with a focus on C++ syntax. Their reliance on these references (textbook and class notes), personal life experiences, and application of problem solving often confused them, but at the same time it helped them generate the C++ text of the solution code. Their reliance on references decreased and the reliance on analogies increased as the term progressed.

The strategy of using model code appeared to be a result of the class instruction. Tim often used "example codes" and encouraged students to think of related problems from the textbook and class notes while solving the given problems. Later, they used a trial-and-error strategy and tried various combinations of C++ code to solve the given problems. This process often confused them or they were unable to explain how they reached the problem solution.

After compiling the solution code, students typically encountered syntax and logic errors. Syntax errors occurred primarily due to students' misunderstanding and/or lack of knowledge of programming syntax and constructs. Logic errors often occurred due to their inability to understand the given problem and to develop the correct solution code from the given problem specification.

While correcting errors, students (except Jose) attempted to debug first without even understanding how the solution code actually worked. In other words, they worked on the debugging without clearly understanding the computer program.

Throughout the term, when they encountered errors, students typically worked on syntax errors first. The frequent set of syntax errors encountered by students involved (1) “symbol referencing errors” (such as misspelling) leading to undeclared identifiers, undefined functions, and *class* name errors, (2) “output formatting errors” such as improper field width and precision controls, (3) “improper keyboard input” such as inserting non-numeric values for *objects* where a numeric value was expected. This last kind of error often led to the termination of the program or infinite loop situations. Throughout the term, Tim provided specific instructions on dealing with syntax errors by using the compiler provided message and directions. To correct syntax errors, students typically accessed the location of the syntax error by double clicking on the compiler-provided error messages. Then they were able to examine the area of code that generated the error message and corrected the errors sequentially. For the most part, students were able to follow the directions provided by the compiler and thus correct the syntax errors. However, at times students were confused by the compiler-generated messages, its interpretation, and parse syntax errors. Parse errors occurred, for example, if students placed a semicolon at the end of the *class* declaration. In these situations, the complier directions reflected errors several lines earlier than the actual error location. To deal with parse errors, students reviewed their solution code and then referenced their textbook and class notes, searching for a similar model code; then they followed the model code to correct the syntax of their solution code. However, the strategy to follow the model code did not always help to correct the syntax errors and often led to the introduction of new (often erroneous) code in their solution.

During second week, Tim explained how to use the desk-check strategy to correct syntax and logic errors. This strategy was meant to help the students deal with syntax errors caused by misplacing code (for example, computation of average inside a loop, unnecessary code and improper sequencing of C++ statements). Adam, Ann and, Jose used this desk-check strategy by printing their solution code and then correcting the code line-by-line in a sequential order. This strategy helped them to detect and correct syntax errors.

At times students were unsuccessful in correcting their errors by using the desk-check strategy. They then used a random trail-and-error approach to respond to the syntax error problems. However, this trial-and-error strategy often led to the introduction of unnecessary code, leading to more syntax errors. While this trial-and-error strategy was occasionally helpful to students, in many instances, they were unable to explain how they reached the correct solution.

After correcting syntax errors, students considered logic errors. The most frequent logic error sets included: (1) improper sequencing of variable *objects* placed in a complex control structure (i.e. sequencing mistakes), (2) language construct errors which created problems such as infinite loops, (3) misunderstanding operator precedence, (4) using multiple object roles such as assigning two values to the same object, (5) improperly naming *objects*, (6) using non-existent *objects* and *operations* on the *object* such as misunderstanding the scope (global and local) of *objects*, and (7) misinterpreting the *class* declaration and implementation sections. To deal with the logic errors, Adam, Ann, and Jose typically followed the desk-check strategy provided in Tim's instruction. This strategy frequently helped students detect and then correct logic errors.

As the term progressed and close to the end of the term, class instruction provided exposure to additional automated debugging tools such as the use of VISUAL C++ debugger and the insertion of tracing statements (the print statements at certain key locations in the solution code). With the use of the debugger and the tracing statements, students were able to execute their solution code one line at a time and examine the status of the values of the variables at key points. This allowed them to divide the code in small pieces and work on each piece individually. In other words, it offered students the opportunity to divide-and-solve their problems. However, students also faced difficulties in understanding the VISUAL C++ compiler-provided error messages/directions. These messages and directions were oriented towards professional programmers and did not address the needs of beginners. As a result, students were often confused and misunderstood the meanings of the message and/or direction. These misunderstandings resulted in parse errors and/or the introduction of new errors.

Another noticeable feature was the improvement in students' debugging techniques. At the beginning of the term, students received instruction on following and identifying compiler-provided error messages. During the second week, they received instruction on the desk-checking technique where they could sequentially examine their code for syntax and logic errors. And then towards the middle of the term, they received instruction on how to use the VISUAL C++ debugger and the insertion of print statements in the code. Students used these tools as soon as they received instruction. These tools improved their error detection and correction abilities.

Students did not improve in their planning and designing techniques to develop the programming solutions since Tim did not provide specific instructions on comprehensive program analysis and design. However, he did instruct students on preliminary problem analysis such as identification of *objects*, *methods* and listing of input, output and processing steps. As a result, students also limited themselves to performing the preliminary problem analysis to complete the solution.

Analysis of students' work also revealed difficulties among students when learning some OOP and control structure concepts (selection and repetition). Students had difficulties in understanding the exact definition of an object, the meaning of an *object*, and how to efficiently access and name an *object*. Students also had difficulties in understanding the OOP concept of *constructor*. They were unable to distinguish between a C++ function and a *constructor*. They also listed attribute names as the *constructor* arguments, and when the problem asked for a constructor, students' commonly provided a declaration of an *object*.

A separation of students became visible towards the end of the third week of the term. Mel and Ann became overwhelmed during this time. They did not perform as well as their male counterparts (Adam and Jose) in the class. Both females seemed to have relatively more difficulties in sequencing program statements (in general) and in complex control structure (in particular) than their male counterparts. For example, one problem provided a *for* loop to rotate an angle from zero to Π (pi) and print out the values. Both Ann and Mel initialized code for the loop inside the body of the loop. The males seemed more comfortable with the mechanical, analytical, and sequential aspects of computer programming.

CHAPTER V

DISCUSSIONS AND CONCLUSIONS

Introduction

The purpose of this exploratory study was to describe novice students' learning of computer problem solving (CPS) in a beginning computer science (CS) course with an introduction to object-oriented programming (OOP). Additionally, this study attempted to connect the instruction with the knowledge students obtained about OOP concepts and CPS strategies to support their learning of computer programming.

The institution where the study was conducted enrolled about 5000 students from an ethnically diverse population in the western United States. One experienced instructor and four students participated in the study. Pseudonyms were used to assure confidentiality and anonymity.

This study began by selecting a college level introductory CS class. A volunteer instructor Tim and four volunteer student participants (Adam, Ann, Mel and Jose) participated in the study. All classroom documents used to teach the introductory CS course were collected and analyzed. Classroom observations of each session documented the curriculum (in particular the CPS strategies and OOP concepts), instructional strategies (activities, settings and classroom engagements), and instructor-student interactions during instruction and his office hours. The goal of the classroom observations was to provide a detailed description of how a beginning

CS course was taught. An initial, semi-structured (open-ended) interview was conducted to establish Tim's overall philosophy and approach for incorporating OOP and CPS strategies in the introductory CS course. Informal discussions and semi-structured interviews were conducted with him at the end of each class day to clarify any questions from the classroom observation and to document the questions students asked during his office hours. A final semi-structured, open-ended interview was conducted to identify Tim's perception of his students' progress with CPS and OOP programming concepts.

To gather student information, twice during the week informal student interviews were conducted to encourage the students to explain their understanding of the OOP concepts and CPS, their perceptions, their study practices, and their ability to determine efficient and correct computer problem solutions. Two formal interviews (one at the middle of the term and a second close to the end) were used to gather specific data about how students approached computer problems in terms of programming and computer problem solving in an introductory computer science course. During each interview, students were given one problem solution to debug and one problem which asked them to develop an original computer solution.

The study was designed to answer the following questions:

(1) What instructional strategies characterized a beginning computer science course with an introduction to object-oriented programming at the college level to engage students in computer problem solving?

(2) How did novice students solve computer problems as a result of instruction in a beginning computer science course with an introduction to object-oriented programming at the college level?

This chapter includes discussion and interpretations of the findings presented in Chapter IV in light of previous literature on teaching and learning in computer science. It also includes the limitations of the study, the implications of the study and recommendations for future research.

Interpretation and Discussion of the Results

To address the first research question, this study investigated the instructional strategies utilized in an introductory CS course. An analysis of the results found that despite Tim's initial intensions, the class was teacher-centered and emphasized the syntactical details and with a focus on imperative aspects with an introduction to object-oriented aspects of the C++ programming language. Tim's lectures, labs, homework, exams, and quizzes did not address the underlying programming logic, such as the comprehensive problem analysis/design involved in computer programming and OOP conceptual approaches. Tim did succeed in teaching some introductory OOP and preliminary CPS strategies; however, for example, he did provide instruction about conducting a limited preliminary problem analysis with a focus on identifying and defining *objects*, *methods* and *classes*. He also recommended that students use abstraction and think about the problem solutions first in English language i.e. the program logic before attempting the solutions in C++ code. However, Tim did not provide specific instruction to support students' abstraction or on how to think in "English first."

The Computer Science Department at the university in this study adopted a programming-first model as described in Academic Computing Machinery (ACM) curriculum reports (ACM 1991, 2001). The programming-first model was aimed at helping beginning computer science students develop the fundamental skills of computer programming and computer problem solving (ACM, 1965, 1968, 1978, 1991, 2001).

The ACM (2001) curricula report recognized a variety of implementations strategies for the programming-first model (adapted from ACM, 2001: 3).

(1) Imperative-first strategy: This strategy focuses first "on the imperative aspects of the language: expressions, control structures, procedures and functions" (ACM, 2001, p.10).

(2) Objects-first strategy: This strategy initiates the introductory course in computer science with object-oriented programming concepts. Control structures such as selection and repetition are introduced within the context of OOP at a later stage.

(3) Breadth-first strategy: This strategy introduces students to programming along with sub-disciplines such as mathematics as well as computer programming. It teaches programming language with the purpose of providing a "holistic view" of the computer science field.

(4) Algorithms-first strategy: This strategy introduces students to computer problem solving and/or algorithmic processes without using an executable programming language; this strategy is used to teach programming with a major emphasis on non-executable, language-independent algorithm development techniques, such as writing pseudocode or developing program flowcharts.

(5) Functional-first strategy: This strategy introduces students to a functional language, such as Scheme. Students are introduced to topics such as procedural abstraction, data abstraction, algorithms, and problem-solving.

(6) Hardware-first strategy: This strategy initially introduces students to the computer hardware concepts such as switching circuits, simple registers and then programming in a high-level programming language such as Pascal, C.

Tim implemented a programming-first model using an "imperative-first" implementation strategy. The results in this study concurred with the previous research of McCauley and Manaris (2000) found that a majority of computer science departments across the United States have adopted the programming-first model for the introductory CS class.

Tim helped students to learn some of the programming skills needed in C++ language. However, his emphasis on C++ syntax details may have lead students involved in this study to approach computer programming without a comprehensive plan and to develop solution in an "ad hoc process of trial and error" method (ACM, 2001, p.10). Some students in Tim's class lost interest and motivation for computer programming as a result of his instructional approach, which emphasized the syntactical details of the C++ language. For example, Mel (Business major) felt "overwhelmed" towards the end of the class and mentioned that taking Tim's class was "irrelevant" to her major. Students without prior computer programming in general and females in particular were also placed at a disadvantage because of the emphasis on the mechanistic details of programming constructs in Tim's class. Tim also agreed that some of the students were put at a disadvantage because of this focus stating, "students especially Ann and Mel seemed uncomfortable with the mechanical details involved in computer programming in this class. They wanted to understand programming as a process. However, programming was more of a mechanical activity in this class. Programming is actually more for the person who is mechanically-oriented."

The premier computing professional organizations such as the Academic Computing Machinery (ACM), the Special Interest Group on Computer Science Education (SIGCSE), the Institute of Electrical and Electronic Engineers, Computer Society (IEEE/CS), the Computer Science Accreditation Commission (CSAC) and the Computer Science Accreditation Board (CSAB) recommend that students be taught problem-solving/programming skills, the development of cognitive models and effective analysis/design of computer problem solutions in the introductory computer science course. In short, these associations assert that the curriculum and instruction in an introductory computer science course should not only be focused on sets of syntax rules but should also help students develop the necessary cognitive thinking skills to deal with complex tasks such as CPS and computer programming. Research

(Carter, 1992) supports that introductory CS classes taught with an emphasis on teaching CPS improves programming performance among students.

In addition, Tim's instruction did not provide a comprehensive exposure and practice with object-oriented programming. The analysis of the results revealed that during the five weeks of instruction, Tim spent almost four weeks focusing on the imperative aspects with just a touch of object-oriented aspects (such as cin and cout) of the C++ language. Only the second week and part of the fourth week was used to emphasize the object-oriented aspects of the C++ language. ACM (2001) strongly recommends a comprehensive exposure to object-orientation in an introductory course indicating object-orientation as "central" to the introductory computer science curriculum. Furthermore, both CSAB and CSAC notified the computer science education community that object-oriented topics would have significant emphasis in the Advanced Placement curriculum (AP 2000).

Close to the end of the term, students in Tim's class showed a constant struggle in their learning due to shifts in the way they were taught to approach programming problems, i.e. from imperative to object-oriented and vise versa. While students were exposed to the imperative aspect during the first week, the second week of the class emphasized the object-oriented aspect of C++. The third, fourth and the fifth week focused again on the imperative aspect while part of the fourth week focused on the object-oriented aspect of C++ language. According to Ross (1997), little or a late introduction to object-orientation can be counterproductive since students exposed earlier to imperative programming practices may have to "unlearn" their procedural thinking in order to learn object-oriented programming.

> One problem that learners of OOP run into is that of false understanding. This results in programmers thinking an approach makes a lot of sense, liking it, adopting it, and then having to back away from it. It takes time to abandon what seemed like a good idea. (Ross, 1997, p.48)

Some of Tim's planned views about teaching the introductory course were not noticeable in his instructional practices. For example, Tim stated that he would apply a student-centered approach to his instruction and said that he believed in presenting opportunities for students to learn rather than telling them how to learn. He preferred "self discovery of knowledge and the light bulb theory." He stated that he would encourage student participation and believed that classroom interactions, assignments etc. would help him understand how students were learning. However, Tim's instruction was primarily teacher-centered, lecture-oriented, and contained little student participation and discussion during the lectures.

Before the course began, Tim stated that he would not emphasize C++ syntax during his instruction. However, his instructional approach was mainly syntax-based. The syntax-based approach did not appear to provide a facilitation of the cognitive models and/or skills required to be a successful programmer since programming activity requires high-order thinking while solving problems (ACM, 2001).

Tim viewed teaching problem solving as "time consuming" and "unfair" because students would have been asked to "do creative work in a stressful situation." As a result, Tim's instructional strategies appeared to exclude core higher order thinking skills, which lead to well-rounded computer programming experiences. Because of this, students depended mostly on reproduction (copying existing code) rather than a combination of reproduction and a production (creating new from an existing) thinking pattern. Overall, Tim's instruction seemed to have a profound effect on student's CPS and OOP abilities.

The second research question of this study addressed the strategies used by students to solve computer problems in a beginning course in computer science with an introduction to object-oriented programming. Students' problem-solving approaches lacked the use of a comprehensive written plan/design throughout the term. Students viewed the solution to the given problems as a collection of the parts of C++ statements rather than with a comprehensive, integrated view of the problem. They seemed somewhat familiar with C++ syntax, but at the same time they were uncomfortable and at several instances unsuccessful in developing complete solutions in C++ syntax from scratch. Their strategies typically followed a specific process. First, students attempted to understand the problem by reading and underlining keywords or sentences. Second, they performed a preliminary problem analysis by identifying *objects*, *methods* (*private* and *public*), *class*, *attributes*, as well as listing of input, processing steps and output. Throughout the term, Tim encouraged and demonstrated the identification of *objects*, and *methods*. Third, they used examples and model code from the textbook and class notes, analogies from their real-life experiences, prior learning experiences from other educational domains such as mathematics/algebra problem-solving and the programming knowledge/concepts attained through class instruction. During this third step, the students typically generated a solution code. Their strategy to code- first without comprehensive planning often failed to lead them to an efficient solution. As a result, they resorted to a fourth strategy: trial-and-error. After compiling the solution code, students often encountered syntax and logic errors. In the debugging process, students began the process without understanding how the solution code actually worked. To correct the syntax errors, they accessed the location of the syntax error by double clicking on the compiler-provided error messages and sequentially corrected the errors. Students also used a desk-check strategy to correct syntax and logic errors by printing their solution code and then examining the code in a sequential order line-by-line. When this desk-checking strategy was not successful, they resorted to trial-and-error. Correction of syntax errors was followed by the correction of logic errors. Students used the desk-checking strategy, the automated debugging tools provided by the VISUAL C++ debugger, and inserted tracing statements at certain key locations in the solution code. With the use of the debugger and the tracing statements, students efficiently executed their solution code one line at a time and examined the status of the values of the variables at key points.

In short, students typically approached the problems without developing a comprehensive written plan/design. Instead, they accepted the given problem and began to code. This code-first strategy engaged students in thinking about their problem solutions in terms of C++ code. This strategy has also been reported by other investigators (Carter, 1991; Lee, Pennington & Redher, 1995; Rist, 1995).

Another interesting result of the study was that the students' learning processes seemed to follow the gradual learning model (cited in Nelson, Irwin & Monarchi, 1997, proposed by Anderson and colleagues, 1983, 1987) built from a computer programming perspective. According to this model, a beginner gains programming knowledge in three stages. In the first stage, a declarative knowledge is developed as the beginner attempts to learn the basic definitions, methods, and skill performance needed in programming. In the second stage, beginners gain a procedural knowledge by using examples extensively to apply the declarative knowledge in the problem-solving process. During the third stage, the beginning student gains the experience and procedural knowledge needed to handle more challenging problem solving (Nelson, Irwin & Monarchi, 97). Analysis of the students' knowledge in this study revealed similar stages of knowledge gain. In the beginning, they learned the declarative knowledge involved in the C++ programming language. Then they used model codes, examples, and analogous solutions from their textbook and class notes. Finally, they practiced their new skills by completing written homework and programming assignments thus attaining the necessary procedural knowledge.

While students' learning processes allowed them to gain some of the procedural knowledge to solve programming problems, their ability to design and develop correct program solutions remained quite limited. Evidence from this study suggests that the reason is linked to their inability to use effective problem-solving heuristics and/or strategies.

Brooks (1982) developed a model of cognitive processes in computer programming. According to his model, programmers go through three major steps to solve a given problem: They (1) work to understand the problem; (2) find a method (algorithm); and (3) convert the method to a solution code. The students involved in this study attempted to understand the problem by underlining key words and/or sentences in the problem specification. However, they did not identify an algorithm and its conversion to an actual code. Rather, they performed an incomplete preliminary problem analysis and followed examples to generate code. Eventually, they resorted to trial-and-error correction of the solution code.

Much discussion in the literature (Choi, 1991; Lee & Thompson, 1997; Mains, 1997; Knox-Quinn, 1995; Willis, 1999) has considered whether computer programming helps to develop students' general problem-solving skills. While this study did not attempt to study the question of general problem-solving abilities, the research question did focus on understanding students' computer problem solving processes. In general, the processes the students used included: (1) understanding the problem by reading and underlining and/or identifying keywords; (2) searching for the analogous or model problem solutions from their textbook and class notes or parts of an analogous solution code in the hopes of a correct solution.

Similarities of this process were considered in relationship to ideas promoted by Polya (1988). Polya described a four stage model where a problem solver:

(1) Attempts to understand the problem. He or she looks at what is known

and unknown, analyze the problem's conditions and situation, and then identifies the key words and data in the given problem.

(2) Devises a plan by finding a connection between the data and the unknown in the given problem. Here, the problem solver looks at analogous solutions and uses some portions of the analogous solution to solve the given problem.

(3) Carries out the plan and checks that each step is correct.

(4) Examines the problem solution by looking back.

The process students used in this study was somewhat analogous to the first two stages of Polya's problem-solving model. However, they did not prepare comprehensive written plans/designs nor did their processes mirror ideas in the third and fourth stages of Polya's model. With more computer science course- work and instruction, their patterns may become more aligned with Polya's model. Perhaps a more efficient and progressive problem-solving model develops with more problem-solving experience.

Another noteworthy aspect that emerged during the data analysis was the kind of thinking students exhibited in this study. Lowen (1982) mentioned two kinds of thinking modes and/or thinkers in computer programming, i.e. analytical and intuitive. Analytical thinkers are well planned, detailed-oriented, and sequential (cited in Dann, 1990 proposed by Lowen, 1982). On the other hand, intuitive thinkers are gestalt and experimental. As in Lowen's (1982) findings, students involved in this study exhibited a "dichotomy of these two modes of thinking. However, one mode was dominant over the other." Adam, Ann and Mel seemed to be more intuitive thinkers.

They did not exhibit a well-planned, methodical, and detailed- oriented thinking pattern. However, Jose exhibited more of an analytical thinking pattern. During his CPS and OOP processes, he paid attention to details and methodically solved given computer problems. Of the students involved in this study, Jose was the most successful problem solver earning an "A" in the course. During the interviews and observations, Jose displayed confidence and comfort while engaged in CPS, which perhaps was due to his analytical thinking style. In addition, as mentioned earlier, Tim primarily focused on the imperative aspects of the C++ programming language during his instruction. This could have favored Jose's thinking style. As Dann (1990) stated, "The underlying conceptual principles of imperative programming can be seen to richly accommodate the analytic, well-ordered, step-by-step, procedural cognitive style." Consequently, students (Adam, Ann and Mel) who were less analytical in their thought processes were less successful in the class.

An important question in this study thus becomes whether Tim "adequately prepared" his students to deal with the "cognitively-challenging" task of computer programming, a task that requires the use of productive and reproductive thought patterns. Tim's instruction typically involved students in a drill-and-practice regiment. He encouraged students to use example code to solve the computer problems. As a result, students depended on examples in their textbook and class notes to solve their interview, homework, quiz, etc. while developing computer solutions. In other words, Tim's instruction focused students on reproductive thinking.

In addition, the findings of this study supported Dann's (1990) identification of problems some students face while learning OOP. According to Dann, "First, the student may not be adequately prepared, i.e. the cognitive skills required in the programming process. Second, the student may possess a cognitive style which is unsuited for the imposed language and methodology" (1990, p. 100). "Inadequate preparation" in the cognitive skills such as productive and high-order thinking required for computer programming combined with incompatible thinking and learning styles with the C++ language appeared to limit students' understanding of CPS processes and computer programming (Dann, 1990). As a result, students' in this study frequently encountered syntax and logic errors while developing their computer solutions. During the debugging and the CPS process, students in this study often depended, on a trial-and-error strategy to correct their errors (i.e. syntax and logic errors). However, the use of a trial-and-error strategy became more harmful when students applied it while working on logical errors.

The instructor in this study focused on the imperative aspects of the C++ programming language. This approach had a profound effect on students' thinking and problem-solving approaches while engaged in programming using C++ language. It required them to think sequentially, mechanically, analytically, and procedurally. According to Sutter (2002), the thinking process involved in C++ programming language's imperative aspects requires a store, fetch, and execute cycle similar to the mechanical aspects of computer hardware. Students in this study had to think in an analytical and sequential manner to solve problems, making it difficult for those who think more intuitively. The analysis of student results revealed that students in general had difficulties in proper sequencing of the statements in control structures such as loop. However, the females (Ann and Mel) had more difficulties as compared to males (Adam and Jose) in understanding and correctly sequencing statements in complex control structures. For example, Mel and Ann had similar problems with nested loops and particularly with the proper sequencing of statements. However, Adam and Jose performed better with nested loops and sequencing. Instead of being able to view the computer as a machine, Mel and Ann viewed it as a thinking process similar to what is used in mathematics or algebra. On the other hand, the males seemed to be more comfortable with the mechanical aspects of computers and computer programming. In other words, females in this study seemed to be less mechanically-oriented than the males. An interesting note, however, is that the students in general, and females in particular, performed better on the declarative aspects of C++ programming language where no control was involved. Other research (Colley, 1995; McClelland, 2001) has also reported subtle gender differences in approach and understanding of computers and programming.

Limitations of the study

Limitations of the study, including sample size, the researcher, the methodology, and the programming language and analysis of data, make it difficult to widely generalize the findings of the study. After all, only one instructor, one course and four students from a limited geographical area were included in the study. As a result, the representativeness of the sample was limited. However, the sample was appropriate for the research design and method by providing an opportunity to study the participants in greater depth. While generalization of the results is not appropriate, the study did identify potential factors that affect introductory computer science problem-solving which should be studied in more detail using a quantitative methodology or perhaps a combination of using qualitative and quantitative approaches.

In addition to the limited sample size of the study, its qualitative design may have presented a bias since the researcher was the primary investigator and data collector. As a result, the researcher's background and/or unintentional biases could have led to the contamination of the data and an unintentional bias in its interpretation and/or analysis. Precautions were taken to minimize the researcher's biases. On a daily basis, for example, the researcher maintained a journal to record classroom observations, interviews with the instructor and students, and reflections on classroom and research activities. The journal also included thoughts, questions, reactions, interpretations and insights during the observations. Through this detailed reflection and analysis the researcher worked to identify potential sources of biases and misinterpretations.

Course duration was also a limitation in this study. Typically, instruction of an introductory course in computer science involves an 11 to 16 week term, whereas the course involved in this study was only five weeks. One of the significant factors that influenced Tim's instructional approaches and student's learning of the CPS and OOP in the introductory CS course was time. Tim and students were under tremendous constraints to complete the required course material in a shortened time period.

Implications for Computer Science Education

Students in this study faced challenges to constantly adjust their individual and/or personal thinking and learning styles in accordance with the imposed methodologies of the C++ programming language. Tim's teaching approach imposed an analytical thinking and learning style to accommodate the imperative aspects of the C++ programming language. As a result, those who adapted well (Jose) to the analytical aspects (i.e. step-by-step, detailed-oriented aspects of the C++ programming language) were more successful than others (Adam, Ann and Mel). Instructors of introductory computer science classes need to be aware of, and sensitive to, different student thinking and learning approaches to computer programming.

The use of the C++ programming language in this study also revealed that C++ is not a student-friendly programming language for beginning computer science students. The students in this study generally felt that the C++ language was a difficult language to learn. Adam called C++ a "cryptic" language, and the instructor called C++ an "awful" programming language for the beginning students. As mentioned earlier, C++, if taught with the imperative or "object-first" implementation strategy, could favor a particular thinking and learning approaches. For this study, the use of C++ programming language imposed the analytical approach/strategy to solve the given problems. Computer science departments need to consider this problem and select a computer programming language that is more student-friendly for beginners and also can accommodate diverse student thinking and learning approaches while at the same time helping students develop an acceptable foundation in computer problem solving.

Students involved in this study typically solved problems without using a comprehensive problem analysis/design and at times faced difficulties to develop the solution code. The ACM (2001) curriculum report recommends the teaching and learning of effective problem analysis/design skills among introductory computer science students. CS instruction at the introductory level must include instruction concerning the underlying computer logic to support skills in a comprehensive and effective analysis and design.

In this study, Tim's instruction did not provide a comprehensive exposure of the object-oriented concepts. Computer associations such as, ACM, IEEE, CSAS and CSAB, on the other hand, highly recommend an early and comprehensive exposure to object-oriented concepts for introductory computer science students. Tim's instruction also switched between the imperative and object-oriented aspects of C++ language. Previous research (Ross, 1997) has shown that exchange between the paradigms during instruction is counterproductive. A major concern noted in Ross's research was the effort students had to extend to "unlearn" the other methodology (imperative) prior to their journey in learning object-orientation. Computer science departments therefore need to consider implementing an "object-first" strategy for their introductory computer science courses. This kind of strategy may help students in their transition towards advanced computer science courses such as data structures.

Adoption of VISUAL C++, a commercial-based and professionally/expert oriented compiler, by the CS department where the study was conducted, was found to have consequences for both teaching and learning in the introductory course in computer science. The results of this study indicated that students initially had difficulties dealing with the VISUAL C++ compiler and its error messages. If the goal of a beginning computer programming course is to teach students the basics of an OOP language, a complier should be selected that takes into account the needs of beginning students and that generates information understandable by beginners.

Recommendations for Future Research

In this study, the teaching and learning involved the use of C++, a hybrid programming language. C++ allows both imperative and object-oriented approaches. Tim adopted a programming-first model with a focus on the imperative aspect with an introduction to the object-oriented aspect of C++. McCauley and Manaris (2000) reported an upward trend towards a complete adoption of object-oriented approach in the CS departments across the United States. The ACM (2001), CSAB and CSAC (2000) and SIGCSE (2002) recommended a focus on object-oriented concepts. Future research is needed to identify how students approach CPS and OOP processes in other object-oriented languages (such as JAVA) and if their CPS and OOP processes develop more naturally.

A small sample within a limited geographical area was utilized in this study. For the current study, the sample was appropriate since the study provided results that can help CS educators begin to understand how students solve computer problems in an OOP environment. For future research, a more diverse geographical area and a more diverse sample could help to identify the approaches students might use in an object-oriented environment.

Students in the current study were not interviewed collaboratively since the research questions were focused on the individual student. Real life situations, however, require computer programmers to often work collaboratively in projects thus solving problems in a collaborative manner. Häkkinen (2001) reported a substantial body of research demonstrating the benefits of collaborative learning. Therefore, it is recommended that research on collaborative CPS learning should be completed, allowing students to work in groups of two or more; interviews should be conducted on those students involved in collaborative learning of CPS in an OOP environment. This approach can provide more understanding of effective computer problem solving skills in a collaborative environment.

A variety of qualitative research techniques (e.g., classroom observations, interviews, researchers' journal and classroom documents) were employed to collect and analyze the data. The purpose of employing multiple sources was to strengthen the validity of data analyses, to sustain assertions, and to assure viability of the data collected (Lincoln & Guba, 1985; Patton, 2002). Therefore, it is recommended that future research in CPS and OOP should also involve multiple resources. In addition to the kinds of resources used in this study, researchers should consider adding the element of personal journals written by the subjects involved. These reflections will help enrich the data and provide a more in-depth perspective of the subjects involved.

The instructor and students in this study were under time constraints to teach and learn CPS and OOP in a short period. This time limitation presented a burden in the teaching and learning of CPS and OOP. The instructor thought that teaching CPS would be a time consuming process and, in many instances, students resorted to rushing to identify a solution by any means, including copying from examples without using the underlying CPS processes. Future research is needed where multiple approaches to instructional formats such as a longer time periods are used. Perhaps, an open entry and exit instructional formats would allow time for instructors and students involved in the study to focus on teaching and learning of the CPS and OOP processes.

Students in this study depended on examples to solve problems. They used examples of solution code (similar to the problem given) from their textbook and class notes. There were several instances where using examples did help guide solutions. However, at other times students tended to copy the examples without understanding them. Future research is needed to identify examples that support introductory computer science students in learning the CPS and OOP.

The analysis of the results in this study indicated that students in general, and females in particular, had difficulties using the C++ programming language's imperative approach. Females in this study were relatively less mechanically-oriented than males. It is recommended that future research in CPS and OOP explore gender issues in depth and study how gender plays a role in learning other OOP languages. Future studies could also utilize information from the gender related studies (Charlton & Birkett, 1998; Dryburgh, 2000; Kadijevich, 2000).

The instructor in this study planned to teach the CPS with a focus on OOP. However, there was an apparent disconnect in his planned views and his actual implementation of the instructional plans. Future research needs to focus on how CS departments and instructors' state and/or plan their theory of pedagogy as compared to their actual pedagogy implemented.

The computer science accreditation board CSAB (2002) reported a continual popularity and adoption of OOP languages among CS departments across the United States. However, in spite of the popularity, change, and adoption of OOP among CS departments (McCauley & Manaris, 2000), and the resulting impact this adoption of OOP has or will have on a significant number of beginning computer science students, relatively little scientific evidence exists about learning CS with OOP languages. This study identified potential student CPS and OOP learning processes and factors using a qualitative approach. Future research should continually investigate the factors effecting introductory CS problem-solving using a quantitative methodology or perhaps a combination of using qualitative and quantitative approaches.

REFERENCES

Academic Computing Machinery. (1965). An undergraduate program in computer science preliminary recommendations. *Communications of the ACM*, 8(9), 543-552.

Academic Computing Machinery. (1968). Curriculum '68: Recommendations for the undergraduate program in computer science. *Communications of the ACM*, 11(3), 151-197.

Academic Computing Machinery. (1978). Curriculum '78: Recommendations for the undergraduate program in computer science. *Communications of the ACM*, 22(3), 147-166.

Academic Computing Machinery. (1991). Curriculum '91: Recommendations for the undergraduate program in computer science. *Communications of the ACM*, 43(2), 101-134.

Academic Computing Machinery. (2001). An undergraduate program in computer science preliminary recommendations [online]. http:www.acm.org/sigcse/2001.

Advanced Placement Program. (2003). Introduction of Java in 2003-2004 [*online*]. http:www.collegeboard.org/ap/computer-science.

Ahmed, M. Aqeel. (1992). Student Thought Processes While Engaged in Computer Programing. *Dissertation*. Oregon State University.

Allwood, M.C., Bjorhag, C. (1990). Novices' debugging when programming in Pascal. *International Journal of Man-Machine Studies*, 33(6), 707-724.

American Association of Colleges. (2002). http://www.aacu-edu.org.

Anderson, J. R. (1983). *The architecture of cognition*. Cambridge, MA; Harvard University press.

Anderson, J. R. (1987). Skill acquisition: compilation of weak-method problem solutions. *Psychological Reviews*, 94, 192-210.

Baldwin, L. P., Macredie, D. R. (1999). *Education and Information Technologies*. 4(2), 167-179.

Brooks, R. (1983). Towards a theory of cognitive processes in computer programming. *International Journal of Man-Machine Studies*, 18 (2), 543-554.

Byrne, D. J., & Moore, J. L. (1997). A comparison between the recommendation of computing curriculum 1991 and the views of software development managers in Ireland. *Computers Education*, 28(3), 145-154.

Campbell, P. F., & McCabe, G. P. (1984). Predicting the success of freshman in a computer science major. *Communications of the ACM,* 27(11), 1108-1113.

Carter, J. H. (1991). Comparison of a Problem Solving approach to computer programming curriculum with a syntax-oriented approach (programming). *Dissertation.* The University of Texas at Austin.

Charlton, J. P., & Birkett. P. E. (1998). Pyshological Characteristics of Students Taking Programming-Oriented and Applications-Oriented Computing Courses. *Journal of Educational Computing Research*, 18(2), 163-182.

Choi, S.W.(1991). Effect of Pascal and Fortran programming instruction on the problem solving cognitive ability in formal operational stage students. *Dissertation.* Texas Tech University. Lubbock.

Computer Science Accreditation Commission/Computing Sciences Accreditation Board: (2003). Criteria for accrediting programs in computer science in the United States. [*online*] http://www.csab.org/.

Computer Science Accreditation Board. (2003). Criteria for accrediting programs in computer science in the United States. [*online*] http://www.csab.org/.

Computer Science Accreditation Board (2003). [*online*] http://www.csab.org/.

Colley, A. (1995). Gender Effects in the Stereotyping of Those with Different Kinds of Computing Experience. *Journal of Educational Computing Research.* 21(1), 19-27.

Corritore, L.C., Wiedenbeck, S. (1999). Mental representations of expert procedural and object-oriented programmers in a software maintenance task. *International Journal of Human-Computer Studies*, 50, 61-83.

Dann, P. W. (1990). Cognitive aspects of programming, programming paradigms, and programming Instruction. *Master Thesis.* State University of New York Institute of Technology. Utica.

Denning, J. P. (1989). Computing as a discipline. *Communications of the ACM*, 32(1), 90-132.

Dey, S. & Mand, L. R. (1986). Effects of mathematics preparation and prior language exposure on received performance in introductory computer science courses. *SIGCSE Bulletin*, 18(1), 144-148.

Dryburgh, H. (2000). Underrespresentation of Girls and Women in Computer Science:Classification of 1990s Research. *Journal of Educational Computing Research*, 23(2), 181-202.

Ebrahimi, A. (1994). Novice programmer errors: Language constructs and plan composition. *International Journal of Human-Computer Studies*, 41, 457-480.

Educational Testing Services. (2003). [*online*]. http://www.ets.org.

Goldenson, D. (1996). *Why teach computer programming? Some evidence about generalization and transfer.* Proceeding of National Educational Computing Conference '96. 271-276.

Greer, J. (1986). High school experience and university achievement in computer science. *AEDS Journal*, 19(2-3), 216-225.

Häkkinen, P. (2001). Rethinking Collaborative Learning. European Society for Developmental Psychology. [*online*] http://www.devpsy.lboro.ac.uk/eurodev/reviews /index.htm.

Kadijevich, D. (2000). Gender Differences in Computer Attitute among Ninth-Grade Students. *Journal of Educational Computing Research*, 22(2), 145-154.

Kaplan, C. A., & Simon, H. A. (1990). In search of insight. *Cognitive Psychology*, 22, 374-419.

Knox-Quinn, C. (1995). Student construction of expert systems in the classroom. *Journal of Educational Computing Research*, 12(3), 243-262.

Kurland, D. M. , Pea, R.D., Mawby, R., & Pea, D. R. (1986). A study of the development of programming ability and thinking skills in high school students. *Journal of Educational Computing Research*, 2, 429-458.

Lederman, N. G. & Chang, H. (1997). An international investigation of preservice teacher's pedagogical and subject matter knowledge structures. National Science *Council Part D: Mathematics, Science, and Technology Education.* 7(2), 110-122.

Lee, C. M.,Thompson, A. (1997). Guided instruction in logo programming and the development of cognitive monitoring strategies among colleges students. *Journal of Educational Computing Research*, 16(2), 125-144.

Lee, A., Pennington, N. (1994). The effects of paradigm on cognitive activities in design. *International Journal of Human-Computer Studies*, 40, 577-601.

Lincoln, Y. & Guba, E. (1985). *Naturalistic inquiry.* Beverly Hills: Sage Publications, Inc.

Lowen, W. (1982). *Dichotomies of the Mind.* New York: John Wiley.

Mains,G.M. (1997). The effects of learning a programming language on logical thinking skills. *Journal of Educational Research,* 4(3), 185-202.

Mayer, R. E. (1983). *Thinking, Program Solving, Cognition.* New York: W. H. Freeman and Co.

McCauley, R.A., & Manaris, B.Z. (2000). Comprehensive report on the 1995 survey of departments offering CSAC/CSAB-accredited degree programs. *ACM SIGCSE Bulletin*, 32(2), 144-148.

McClelland, M. (2001). Closing the IT Gap for Race and Gender. *Journal of Educational Computing Research.* 25(1), 5-15.

Nelson J., Irwin G., & Monarchi, E. D. (1997). Journeys up the mountain: Different paths to learning object-oriented programming, *Accting., Mgmt & Info. Tech.*, 7(2), 53-85.

Patton, M. (2002). *Qualitative Research and Evaluation Methods, (*3rd. ed.). Thousand Oaks: Sage Publications, Inc.

Pennington, N., Lee, Y. A., Rehder, B. (1995). Cognitive activities and levels of abstraction in procedural and object-oriented design. *Human-Computer Interaction,* 10, 171-226.

Palumbo, D. B. (1990). Programming language/problem solving research: A review of relevant issues. *Review of Education Research*, 60. 65-89.

Polya, G. (1988). *How to solve it* (2nd ed). Princeton, NJ: Princeton University Press.

Reed, W. M. & Palumbo, D. B. (1992). The effect of basic instruction on problem solving skills over an extended period of time, *Journal of Educational Computing Research,* 8(3), 311-325.

Ross, M. J. (1997). Instructional design paradigm: Is object-oriented design next? *Performance Improvement Quarterly*, 9(3), 23-31.

Taylor, H. G., & Mounfield, L. (1991). An analysis of success factors in college computer science: High school methodology is a key element. *Journal of Research on Computing in Education*, 24(2), 240-245.

Tucker, A. B., & Wagner, P. (1994). New Directions in the introductory computer science curriculum. *SIGCSE Bulletin*, 26(1), 36-39.

Salataci, R., & Akyel, A. (2002). Possible Effects of Strategy Instruction on L1 and L2 Reading. *Reading in a Foreign Language*, 14(1), 13-19.

Shackelford L. R., & Badre N. A. (1993). Why can't smart students solve simple programming problems? *International Journal of Man Machine Studies*, 38, 985-997.

Shneiderman, B, (1976). Exploratory experiments in programmer behavior. *Journal of Computer Information Science*. 5(2), 123-134.

Singh, J. K., & Zwirner. W. (1996). Towards a theoretical framework of problem solving within logo programming environment. *Journal of Research on Computing in Education*, 29(1), 68-95.

Stroustrup, B. (2001). *Speaking C++ As A Native*. Advanced Computing and Analysis Techniques in Physics Research: VII International Workshop. American Institute of Physics. October 2001.

Sutter, H. (2002). The new C++. C/C++ Users Journal – Advanced Solutions for C/C++ programmers. 3(1), 1-6.

Willis, M. J. (1999). Using computer programming to teach problem solving and logic skills: The impact of object-oriented languages. *Master thesis*. The University of Houston. Clear Lake.

APPENDICES

APPENDIX A

Student Informed Consent Form

Dear ______________________________________,

I am a graduate student at the Oregon State University. I am conducting research for my thesis. This research will investigate the strategies utilized by students while engaged in computer problem solving (CPS) and object-oriented programming (OOP). You are invited to participate in this study.

Confidentiality will be maintained by using pseudonyms and by not linking or showing your name to the information in this study. All data will be in a secured place. Data will only be accessed by me and the thesis advisor. At the conclusion of the study all the data will be destroyed.

Participation in the study will not effect your grade in any way. Participation is strictly voluntary. You may withdraw at any time without any penalty or loss of benefits to which you may be otherwise entitled. You may choose to selectively not to answer any particular questions or any question at all.

You will be asked to interact with the researcher twice a week during informal interviews approximately 30-45 minutes and to participate in two (around the middle and close to the end of the term) computer problem solving interviews of approximately 2 hours each. You are asked to allow the researcher to observe the class/take notes, access your graded assignments, tests and your mathematics placement scores. For questions about personal rights as participants you may contact IRB coordinator at (541)737-3437 or via e-mail at IRB@orst.edu

I have read and understand the consent form. I am at least 18 years of age or older and I agree to participate in this research project in the manner described. I understand the general intent of the study, the type of data collected, and the time commitments involved in the study. I give my informed and voluntary consent to participate in this study. I understand that I will receive a signed copy of this consent form.

______________________________ ______________________________

Student Signature/Name Date

APPENDIX B

Instructor Informed Consent Form

Dear _________________________________:

I am a graduate student at Oregon State University. I am conducting research for my thesis. This research will investigate the strategies utilized in object-oriented programming (OOP) and computer problem solving at the introductory college level class. You are invited to participate in this study.

Your name will not be linked or shown to the information in this study. Confidentiality will be maintained through coding. All data will be in a secured place. Data will only be accessed by me and the thesis advisor. Pseudonyms will be used for the educational institutions and the subjects when reporting the results of the research. No information will be used for any class evaluation purposes. At the conclusion of the study all data will be destroyed.

You will be asked to: (1) allow observation of your classes throughout the academic term; (2) participate in an initial interview approximately 1 hour prior to the beginning of the term; (3) daily informal interviews approximately 30-45 minutes and a final interview approximately an hour long; (4) provide classroom documents; and (5) assist in the design of computer problems and OOP solutions.

Participation is strictly voluntary. You may withdraw at any time without any penalty or loss of benefits to which you may be otherwise entitled. You may choose to selectively not answer certain questions or any questions at all. For questions about personal rights as participants you may contact IRB coordinator at (541)737-3437 or via e-mail at IRB@orst.edu

I have read and understand the consent form. I am at least 18 years of age or older and I agree to participate in this research project in the manner described. I understand the general intent of the study, the type of data collected, and the time commitments involved in the study. I give my informed and voluntary consent to participate in this study. I understand that I will receive a signed copy of this consent form.

_________________________________ _________________________________

Instructor's Signature Date

APPENDIX C

Student Background Information

Please complete the following information as directed. Your cooperation is greatly appreciated.

1. Name ____

2. Address ____

3. Phone (_____)________________

4. E-mail ____________________________

5. Major _________________ Minor ________________ Undecided ________

6. Gender (please circle) Female Male

7. Academic Level (please circle) Freshman Sophomore Junior Senior Graduate

Other (please write) ________________________________

8. Are you learning computer programming for the first time? (please circle one)

YES NO

9. What computer classes you have taken so far? Please list. You may include classes taken in high school, and/or at the college level.

____
____
____
____

10. Other computer training received

- ❑ workshops
- ❑ self-taught
- ❑ high school courses
- ❑ other (Please describe in the space provided)

11. Work experience in computers or computer related field. If none, please state so.

12. Please describe your understanding of computer programming and problem-solving process? If you need additional space, please feel free to attach an extra sheet.

Computer Programming Process

Problem-solving process

Thank you for your participation!

APPENDIX D

Instructor Interview Protocols

Initial Interview

1. Please tell me about yourself and your professional background in general and in teaching Computer Science in particular.

2. How long you have been teaching the introductory computer science classes?

3. How are you planning i.e. general overall strategy to teach the introductory computer science class?

4. How would you characterize your instructional strategies in the introductory OOP class?

5. What is your understanding of the complete problem-solving process in OOP?

6. What OOP concepts and CPS strategies you are planning to stress and why?

7. What instructional strategies are you planning to apply to teach computer problem solving/OOP concepts and why?

8. What will be the most difficult OOP concept(s) to teach in this course and to engage students in CPS and why? And how you are planning to present it?

9. What will be the easiest OOP concept(s) to teach in this course and to engage students in CPS and why? And how your are planning to present it?

10. Please tell me about your perception on how students will be learning CPS and applying it in the OOP environment?

11. What sort of outcomes and engagements you envision from teaching students the CPS and OOP concepts?

Informal Interviews

1. How do you feel about students progress in the class?

2. What are your plans in terms of teaching OOP concepts and CPS strategies and assigning the homework?

3. Other questions will be based on the issues/questions raised during the classroom observations and office hour contacts.

Final Interview

1. During the initial interview, I had asked you about your initial planning to teach this class. How do you feel about your overall plans for this class at the end?

2. During the initial interview you envisioned certain outcomes and engagements from teaching students the CPS and OOP concepts. Did you meet your expectations?

3. During the initial interview you gave me your initial perception of student learning. Please tell me your final overall perception of the progress of students learning CPS in the OOP environment.

4. During the initial interview you have given me your initial characterization of the class. Please give me your final characterization of the introductory CS class.

5. During the course of instruction you stressed [certain OOP concept(s)] more frequently than others, and why?

6. During the course of instruction you stressed [certain CPS strategy] more frequently than the others, and why?

7. What was the most difficult OOP concept(s) for the students and how did you present it?

8. What was the most difficult CPS strategy(s) for the students and how did you present it?

9. What was the easiest OOP concept for the students and how did you present it?

10. In your opinion, what ideas did students got well during the instruction?

11. In your opinion, what didn't work well during the instruction?

12. Were there any CPS strategies emphasized in your class, which did not come out during the observations or I might not have observed them?

13. Were there any OOP concepts emphasized in your class, which did not come out during the observations or I might not have observed them?

14. Please explain with examples your overall reaction to the introductory OOP class.

APPENDIX E

Students Interview Protocols

Informal Interview

1. Please tell me about yourself (first interview only).

2. During the past week [certain OOP concept] and [certain CPS strategy] was taught and/or emphasized. What is your understanding of it?

3. Please explain specifically what CPS strategy(s) learned in the class helped you to solve your assignments.

4. Please explain specifically what programming concepts learned in the class helped you to solve your assignments.

5. What other kind(s) of CPS strategy(s) not learned during classroom instruction you are planning to utilize and/or utilized to solve your assignment(s)?

6. Please explain specifically what CPS strategy(s) learned in the class did or did not help you to solve your assignments.

7. How did you explore ideas to solve the given problems?

8. Please explain your debugging process?

9. How do you feel about the assignment(s) assigned and/or returned during the last two weeks?

Problem solving Interviews

1. How did you explore ideas to solve the given problem?

2. Please describe your approach to find the solution for the problem 1.

3. Please describe your approach to find the solution for the problem 2.

4. Please describe area(s) where you got stuck, please provide examples.

5. Why did your approach to problem-solving on any given problem worked? why it didn't work?

6. Your instructor stressed [certain CPS strategies] during the class, how did these CPS strategies help you in solving the given problems?
7. Your instructor stressed [certain OOP concepts] during the class, how did these OOP concepts help you in solving the given problems?

8. Please explain your debugging approaches and processes for the given problems.

9. Summarize what you have learned so far about CPS and programming.

APPENDIX F

Weekly Lesson Contents

Week	Monday	Tuesday	Wednesday	Thursday	Friday
I	T: Introduction, C++ constructs, VISUAL C++ IS: PPL, TB, LS, WHE, PE, I	T: C++ data Types, definitions *objects* etc. IS : PPL, TB, LS, WHE, I	T: control flow & data concepts. IS: PPL, TB, LS, I	T: C++ concepts IS: PPL, TB, LS, WHE, PE, Q, I	No Class
II	T: Describing and declaring *classes*, control flow concepts IS: PPL, TB, LS, WHE, PE, I	T: Data concepts, input/output, IS: PPL, TB, LS, WHE, PE, I	T: Functions, Class declaration & implementation IS: PPL, TB, LS, WHE, PE,I	T: Libraries, Accumulator Class. IS: PPL, TB, LS, WHE, PE, I	T: Libraries, functions revisited IS: PPL, TB, LS, WHE, PE, Q, I
III	T: Basic C++ control Structures, selection IS: PPL, TB, WHE, PE, I	T: expressions, IS: PPL, TB, WHE, PE, I	T: Selection IS: PPL, TB, WHE, PE, I	T : Repetition IS: PPL, TB, WHE, PE, I	Selection and repetition IS: PPL, TB, WHE, PE, I, Mid-term exam
IV	T: Additional Control Structures IS: PPL, TB, WHE, PE, I	T: Nested loops IS: PPL, TB, WHE, PE, I	T: control flow concepts, data concepts, Input/output concepts IS:PPL, TB, WHE, PE, I	T: data concepts, developing your own classes IS: PPL, TB, WHE, PE, I	T: C++ concepts, control flow concepts IS: PPL, TB, LS, WHE, PE, I
V	T: member functions IS: PPL, TB, WHE, PE	T: free functions, introduction to arrays. IS: PPL, TB, WHE, PE	T: one dimensional arrays IS: PPL, TB, WHE, PE	T: two dimensional arrays. IS: PPL, TB, WHE, PE	Final exam

Coding: Topics (T); IS: Instructional Strategy; Power Point Lecture (PPL); Text Bound (TB); Lab Session (LS); Written Homework Exercises (WHE); Programming Exercises (PE); Quiz (Q); Student-Instructor, office hour Interaction (I);

Weekly Content Description

Week I - Chapter 1: Introduction to Object Technology

Primary concept: A computer *program* is a specification of some *computation.* A program operates on data that it receives (input) and delivers results (output). Object-oriented programs organize the conceptual entities in a program as *objects.* Objects are described by *object types*, which encapsulate the attributes and operations supported by all objects of that type.

General concepts: Objects, object types, object attributes, object actions/operations, algorithms, input and output

C++ constructs: C++ identifiers, the main() function, statements, comments, include statements

The C++ programming environment: source code, compiling, preprocessing, linking example object types.

Chapter 2: Basic C++ Types and Programs

Primary concept: A program consists of *instructions* and *data.* Instructions are organized as *statements*, which define the order in which operations are executed. The data is organized as objects, which are defined by types. *Types* specify the *values* that an object can hold and the *operations* that can be used to manipulate those values. *Expressions* are combinations of objects and operations on the values of those objects that result in new object values. The order in which expressions are evaluated is determined by the order of the statements in which they appear.

Control flow concepts: The order in which things happen in a program is called the *flow of control.* The primary unit of program control is a statement. Statements execute in the order in which appear in the program text, unless that order is changed by some control structure. Most statements contain expressions, which are evaluated when the statement is executed. The order in which the operations in the expression are evaluated is determined by *associativity* and *precedence.* After a subexpression is evaluated, the value it produced is used in the expression in which it appears.

Data concepts: An object has a *name*, a *type* and a *value.* C++ defines primitive object types that hold single values such as integers, real numbers and characters. The value of a primitive type object can be changed by *assignment*, and the assigned value for the object remains until it is reassigned in a later statement. When an object's name appears in an expression, that object's value is used in evaluating the expression.

Operators define the operations that can be applied to primitive type objects. The basic operators implement mathematical operations such as addition and multiplication. Each object type requires a different representation of that data that it holds, and data can be translated between the representations for different types through *type conversion*. Type conversion can be *implicit* or *explicit*. The collective value of all objects in a program is called the *program state*. The program state at any particular time is the only thing that has any memory of what has occurred earlier in the program.

Input and output concepts: The data that a program receives and the data that it delivers are controlled by input and output *streams*. Streams are abstractions of data flows. The primary source of input data is the keyboard and the primary receiver of output data is the computer display. Streams control when data is moved between the program and the I/O devices. Streams also translate data between internal and external representations. All data external to the program is represented as a sequence of characters. The internal representation of data depends on the type of the object that holds that data.

C++ concepts: Primitive types: int, short, long, float, double, char
declaration statements and object value initialization
constant objects literal objects: literals have no name and are identified directly by their value arithmetic operators: + - * / % = assignment operator: =, object values are changed by the assignment of new values precedence and associativity of operators implicit and explicit type conversions I/O streams (cin and cout), stream operators (<< and >>), stream manipulators (setw, setreal)

Week II - Chapter 3: Describing and Declaring Classes

Primary concepts: Classes introduce new object types. The value of objects of a class type is defined by the *attributes* of the class. Attributes are also called *data members*. The operations that can be applied to objects of a class type are defined by the *member functions* of the class. The collective value of the attributes of an object is called the *object state*. Member functions act as mini-programs: the accept input through the function *arguments*, deliver results through the function *return value* and change the object state through assignment of new values to the object's attributes. *Free functions* are functions that are not associated with any particular object class. Free functions may affect the program state.

Control flow concepts: When a function name appears in an expression, control is turned over to the implementation of the function. When the function completes, the control returns to the expression from which the function was called and the return value of the function is used as the function's value in the expression.

Data concepts: Classes define object types by *encapsulating* state in private attributes and providing functionality through public member functions. Classes provide *information hiding* by preventing access to private members. This allows the internal representation and operation of objects to be changed without affecting programs that use the public operations. When objects are created, initial values for the attributes of the object are given by a special member function called a *constructor*.

I/O concepts: The information that a program delivers can be presented through means other than output streams. For example, that information may be displayed graphically. The EZDraw library is an example of an output library that converts information to a graphical representation.

It utilizes classes that control the graphic display by encapsulating display information in objects that correspond to graphical objects.

C++ concepts: Arguments: Formal arguments are place-holders for values to be supplied when the function is executed. Actual arguments are the values supplied at the point from which the function is called.

Function prototypes: a prototype defines a function's name and the types of its arguments and return value

Declaration statements for class objects: initialization is implemented by providing arguments to the constructor. Class declaration syntax, private and public members, data members and member functions, default values in function prototypes: default values are used if actual values are not provided when the function is called.

Free function implementations and return statements:

Libraries and examples:
Math library functions (sqrt, pow, etc.)
EZDraw library classes: RectShape, CircleShape
The Accumulator class provides a very simple example that illustrates all the important concepts for declaring classes and using class objects.

Week III - Chapter 4: Basic C++ Control Structures

Primary concept: The flow of control of a program can be determined as a program runs, by the evaluation of true/false expressions based on the current program state. Control structures for *selection* and *repetition* control the execution of other statements by evaluating some boolean expression and then determining what to do based on that value.

Control flow concepts: Selection is the choice between option statements and repetition is repeated execution of some statement. Both selection and repetition constructs are treated as single statements, which execute by controlling the sequence of execution of the statements that constitute their body.

Data concepts: The boolean data type defines objects and expression that have values from the set {true, false}. Boolean expressions are composed of relational operators that perform comparisons and logical operators that combine boolean values.

C++ concepts: Primitive type bool
Relational operators (==, !=, <=, etc.), logical operators (&&, ||, !), logical expressions
Compound statements (block statements)
Precedence of all operators
Selection: if and if/else statements
Repetition: while and do-while loops

Week IV - Chapter 5: Developing Your Own Classes

Primary concept: The implementation of class member functions provides the specification of the computations to be performed for the operations on class objects. Class member functions are similar to free functions, except that they also have direct access to the attributes (and other private members) of class objects.

Control flow concepts: As with free functions, flow of control is passed to the function implementation when the function is called. Control returns to the calling point when the function terminates. Any changes to object state or program state that occurred during the execution of the function remain once the function terminates.

Program design concepts: designing and implementing new classes for specific problems

C++ concepts: member function implementations
Scope resolution operator (::)
Constructor implementations and constructor initialization lists
Organizing program source code, defining new header files
Private member functions for class utilities

WEEK IV - Chapter 6: Additional C++ Control Structures

Primary concepts: Control structures can be *nested*, with one control structure appearing in the statement controlled by another. Any nested statement is executed in its entirety every time it is encountered in the during the nesting statement.
Text files provide persistent copies of data external to a program. The data in text files is represented as a sequence of characters and can be read or written by file streams, using the same extraction and insertion operators previously used for the keyboard and display.

Reference arguments allow for an additional means for receiving data from functions. Arguments are normally passed to a function by creating new objects in the function, which are assigned the same values as the actual arguments. The names of reference arguments within a function simply become aliases for the actual arguments. This means that any changes that a function makes to the value of reference arguments are actually made to the original objects supplied as actual arguments when the function was called.
Control flow concepts: nested control structures, special control structures for special case selection and repetition, break and continue statements for modifying the usual flow of control within a control structure.

Data concepts: reference parameters, file stream types.

I/O concepts: Using text files to hold data external to a program. Using stream objects to read and write text file data.

C++ concepts: nested if statements and the dangling else problem;
Switch statements, break statements;
nested loops for loops, break and continue statements;
pass by value and reference arguments;
text file streams: ifstream and ofstream, opening and closing file streams;

WEEK V- Chapter 7: Arrays

Primary concept: Arrays allow a collection of objects of the same type to be stored with a single name. Individual objects within an array are identified by integer indexes. Partially filled arrays are arrays in which not all objects hold useful values. Ordered arrays are arrays in which the values of objects have a particular order based on some comparison function defined on those values. Looked at to methods for keeping arrays ordered: (i) by restricting the operations on the array such that the operations that change values in the array (insertion and deletion) are guaranteed to keep the values ordered and (ii) by sorting the values after they have been placed in the array.

Data concepts: Arrays are a first example of a data structure. Data structures collect object values in an organized manner.

Program design concepts: Algorithms for modifying and accessing partially filled arrays (ordered and unordered). The selection sort algorithm.

C++ concepts: the array data type, declaring arrays, indexing arrays.
Arrays as arguments (arrays are passed by reference) not covered on final: random number generation (sections 7-6, 7-9), multidimensional arrays (section 7-13).

APPENDIX G

Formal Interview Problems

Set I

Directions: For the following problem 1 a solution is proposed. Errors are embedded in problem 1. Please review the provided solution, explain your understanding of the expectation in the problem, debug the errors and provide the expected output.

1. AVERAGE-PRODUCT –PROBLEM 1

```
#include <iostream.h>
void main ( )
        {
                int num0, num1, num2, num3, num4;
                cout << “Please enter the values << endl;
                cin >> num0;
                cin >> num1;
                cin >> num2
                cin >> num3 >> num4;
                cout << “Thank you for your input” << endl;
                sum = (num0+ num1 + num2 + Num3 + num4;
                average  = sum/4;
                cout << “The average is:” << average << endl;
                cout << “The product is:” << product << endl;
                num0 * num1 * num2 * num3 * num4 = product;
        }
```

Directions: For the following problem 2 develop an entire computer solution. Explain your understanding of the expectation in the problem. Correct errors (if any) in the solution and generate the correct output.

2. CASH-REGISTER – PROBLEM 2

A cash register uses an automated coin machine to help make change. We assume that a clerk is handed money to pay for purchases. For change, the clerk returns to the customer any paper money and directs the coin machine to distribute any change less than $1. In this problem you are to simulate the actions of the clerk and the machine. Write a program that prints the amount of purchase, the payment, and the amount that must be returned as real numbers. Your algorithm solution should use the example of paying $10.00 to cover the purchase of $3.08-the change is $6.92. Then indicate the number of dollars, quarter, dimes, nickels, and pennies that makeup the change total. Use the following output format:

```
Purchase Total    3.08
Payment           10.00
Change            6.92
Dollars           6
Quarters          3
Dimes             1
Nickels           1
Pennies           2
```

Set II

<u>Directions:</u> For the following problem 1 a solution is proposed. Errors are embedded in problem 1. Please review the provided solution, explain your understanding of the expectation in the problem, debug the errors and provide the expected output.

<u>PLAYER-STATUS- PROBLEM 1</u>

The Player-Status *class* described here and implemented in the supplied Visual C++ project has an error. The project includes a test program that indicates one possible sequence of events that result in this error. Determine the cause of the error and correct it.

```
// PlayerStatus.h
// Declaration of class PlayerStatus
// See problem statement for descriptions of attributes and Operations.

#define PLAYERSTATUS_DOT_H
#define PLAYERSTATUS_DOT_H

class PlayerStatus
{
public:
        // Constructor
        PlayerStatus();

        // accessor Operations

        short currentHealth();
        short currentArmor();
        long  currentMoney();
        short currentFood();
        short currentSkillPoints();

        // modifier Operations

        void receiveFood(short _amount);
        void receiveArmor(short _amount);
        void receiveMoney(long _amount);
        void receiveSkillPoints(short _points);
```

```
	bool loseHealth(short _amount);
	bool receiveDamage(short _amount);
	bool spendMoney(long _amount);

	booltradeSkillForMoney(short_skill_spent, long _money_received);
	bool tradeSkillForFood(short _skill_spent, short _food_received);
	bool consumeFood(short _amount);

private:

	// attributes
	short food;
	short health;
	short armor;
	short skill_points;
	long money;
};

#endif


// PlayerStatus.cpp
//
// Implementation of methods for class PlayerStatus
//

#include "PlayerStatus.h"

PlayerStatus::PlayerStatus()
: health(100), food(0), armor(0), skill_points(0), money(0)
{
}

short PlayerStatus::currentHealth()
{
	return health;
}

short PlayerStatus::currentArmor()
{
	return armor;
}
```

```
long PlayerStatus::currentMoney()
{
        return money;
}

short PlayerStatus::currentFood()
{
        return food;
}

short PlayerStatus::currentSkillPoints()
{
        return skill_points;
}

void PlayerStatus::receiveFood(short _amount)
{
        // food attribute is limited to 100
        if ((food + _amount) > 100)
                food = 100;
        else
                food += _amount;
}

void PlayerStatus::receiveArmor(short _amount)
{
        // armor attribute is limited to 100
        if ((armor + _amount) > 100)
                armor = 100;
        else
                armor += _amount;
}

void PlayerStatus::receiveMoney(long _amount)
{
        money += _amount;
}

void PlayerStatus::receiveSkillPoints(short _points)
{
        // skill points attribute is limited to 100
        if ((skill_points + _points) > 100)
```

```
                skill_points = 100;
        else
                skill_points += _points;
}

bool PlayerStatus::loseHealth(short _amount)
{
        // health cannot be less than zero
        if (_amount > health)
                health = 0;
        else
                health -= _amount;
        return true;
}

bool PlayerStatus::receiveDamage(short _damage)
{
        // armor will absorb up to 50% of damage
        short armor_damage = _damage/2;
        if (armor_damage > armor)
                armor_damage = armor;
        armor -= armor_damage;

        // damage not absorbed by armor affects health

        health -= _damage - armor_damage;

        return true;
}

bool PlayerStatus::spendMoney(long _amount)
{
        // do not spend any money if player does not have required amount
        if (_amount > money)
                return false;
        money -= _amount;
        return true;
}

  bool PlayerStatus::tradeSkillForMoney(short _skill_spent, long _money_received)
{
        // do not trade any skill points if player does not have required amount
        if (_skill_spent > skill_points)
```

```
                return false;

        // do not trade skill points of money received is insufficient
        if (_skill_spent > _money_received)
                return false;

        skill_points -= _skill_spent;
        money += _money_received;
        return true;

}

    boolPlayerStatus::tradeSkillForFood(short _skill_spent, short _food_received)
{
        // do not trade any skill points if player does not have required amount
        if (_skill_spent > skill_points)
                return false;

        // do not trade skill points of food received is insufficient
        if (_skill_spent > _food_received)
                return false;

        skill_points -= _skill_spent;
        food += _food_received;
        return true;
}

bool PlayerStatus::consumeFood(short _amount)
{
        // do not consume anything if player has no food
        if (food == 0) return false;

        // limit amount consumed to amount available
        if (_amount > food)
                _amount = food;

        // consume food by moving food points to health points
        food -= _amount;
        health += _amount;

        return true;
}
```

```
// test_player.cpp
//
// Test program that executes a sequence of events that illustrates
// possibility of out of range values for health attribute.

#include <iostream.h>
#include <iomanip.h>

#include "PlayerStatus.h"

void displayPlayerStatus(PlayerStatus status)
{
          cout << setw(10) << "health: " << setw(5) << status.currentHealth();
          cout << setw(10) << "money: " << setw(5) << status.currentMoney();
        cout << endl;
          cout << setw(10) << "armor: " << setw(5) << status.currentArmor();
        cout << setw(10) << "food: " << setw(5) << status.currentFood();
        cout << endl;
          cout << setw(10) << "skill: " << setw(5) << status.currentSkillPoints();
        cout << endl;

}


void main()
{
        cout << "Testing PlayerStatus class:" << endl << endl;

        PlayerStatus player1;

        cout << endl << "New player's status:" << endl;
        displayPlayerStatus(player1);

        player1.receiveArmor(20);
        player1.receiveDamage(40);

        cout << endl << "Player's status after battle:" << endl;
        displayPlayerStatus(player1);

        player1.receiveSkillPoints(50);
        player1.tradeSkillForFood(50,100);
        player1.consumeFood(40);
        cout << endl << "Player's status after working and eating:" << endl;
        displayPlayerStatus(player1);

        cout << endl;

}

/* PROGRAM OUTPUT:

Testing PlayerStatus class:

New player's status:

 health: 100  money: 0
  armor: 0    food: 0
  skill: 0

Player's status after battle:
 health: 80   money: 0
  armor: 0    food: 0
  skill: 0

Player's status after working and eating:
 health: 120  money: 0
  armor: 0    food: 60
  skill: 0

*/
```

Directions: For the following problem 2 develop an entire computer solution. Explain your understanding of the expectation in the problem. Correct errors (if any) in the solution and generate the correct output..

TRIP-TRACKER PROBLEM 2

A Trip-Tracker is used to monitor gas consumption and the cost of gas while traveling. Each time the user stops to buy gas, they will enter the amount of gas, the cost of the gas and the miles driven since the last stop. The Trip-Tracker accumulates this information and uses it to compute the average miles-per-gallon for the trip and the cost-per-mile of the trip. The beginning of a trip is determined by the creation of the Trip-Tracker *object*, or by a call to the reset() *operation*. Write a C++ program to implement the required member functions of the Trip-Tracker class.

APPENDIX H

Sample Class Work Problems

Sample class work problems from students' written homework assignments, programming assignments quizzes, midterm and final exams.

RE-WRITE A LOOP

1. Rewrite the following code fragment using a while loop.

```
float angle;
for (angle = 0.0; angle <=PI; angle +=0.1)
        cout << sin (angle) << endl;
```

GRADE-RECORD

2. The C++ declarations for a GradeRecord class are given below. The attributes store a student's ID and their accumulated grade units and grade points. The functionality of the operations is described by the comments in the C++ declaration.

```
class GradeRecord
{
public:
    //  The constructor initialized the values of the attributes
    GradeRecord (String ID, int gunits=0, int gpts=0) ;
    //  Function gpa ( ) computes and returns the students current
    //  grade point average, using the accumulated values of
    //  grade points and grade units.
    double gpa ( ) ;
    //  Function writeGradeInfo ( ) sends a report on the student's
    //  grade status to the printer.
    void writeGradeInfo ( ) ;
    //  Function updateGradeInfo ( ) adds additional grade units and
    //  grade points to the accumulated values.
    void updateGradeInfo (int  newunits,  int  newgpts) ;
private:
    String studentID;
    int gradepts;
    int units;
} ;
```

(a) Give a statement that will create a GradeRecord object for a student named Frank Black whose student ID is 222-33-7777 and who currently has 20 grade units and 58 grade points.

(b) Is it possible to change the student id stored in Frank's GradeRecord, after the object has been created? If it is give a statement to do it. If it is not, explain why.

(c) Frank just completed a semester of 16 units for which he earned 43 grade points. Give statements to update his grade record and then print his new grade point average to the display console.

(d) A student is on the dean's list of honor students if that student's grade point average is above 3.3. Create a boolean object called deans_list. Use Frank's grade record to set deans_list to true or false to indicate whether he should be placed on the dean's list.

COST-OF-FENCE

3. A company builds 4 foot high chain link fences, whose cost depends on the length of the fence and the number of gates. Each gate is 3 feet wide and costs $75. The chain link portion of the fence costs $12 per foot. To handle customers, we design the Fence class with integer attribute numberOfGates and real number attributes fence-Length and totalCost. The length of the fence includes the width of the gates. The constructor takes as arguments the total fence length and the number of gates and uses them to initialize the attributes. It is assumed that the fence is long enough to accommodate the required number of gates. The member function, getTotalCost(), returns the total cost of the fence.

(a) Give the prototype for the constructor.
(b) Give the prototype for the function getTotalCost().
(c) Develop a declaration for the Fence class.

SALARY-CLASS

4. Write a program that inputs the number of hours worked by the full time employee Fred Barnes and the number of hours worked by the part time employee Sandy Rose. All company employees are paid $18.00 per hour. Place the implementation of the class salary. Declare the Salary objects fred and sandy that represent these employees. Output just the salary and retirement benefit information for Fred and complete salary information for Sandy.

ACCUMULATOR-CLASS

5. Modify the Accumulator class so that it can compute and return the average of the numbers that form the total. Do this by adding a new data member, count, that is initialized to 1 by the constructor. The value of count is increased by 1 at each execution of addValue(). A new member function, average(), returns total/count.

```
class Accumulator
          {
             private:
                //  total accumulated by the object double total;
                //  number of values accumulated in total int
                    count;
             public:
                //  constructor.  initialize total and assign count = 1
                Accumulator (double value = 0) ;

                //  return total
                double getTotal ( ) ;

                //  add value to total and increment count
                void addValue (double value = 1) ;

                //  return total / count
                double average ( ) ;
          } ;
```

(a) Implement the constructor.
(b) Implement addValue().
(c) Implement average().

MONEY-CLASS

6. The Money class defines objects that hold an amount of money. The value of the money can be accessed as either dollars or cents.

```
class Money
{
public:
    //  The constructor accepts an initial value in dollars
    Money (float init dollars=0) ;

    //  these operations allow the value to be
    //  modified using argument values in dollars or cents
    void addDollars (float dollars) ;
    void addCents (long cents) ;

    //  these operations allow the value to be
    //  accessed with values in dollars or cents
    float amountInDollars ( ) ;
    long amountInCents ( ) ;

private:
    //  the dollar value of the money is stored in this attribute
    float amount_in_dollars;
} ;
```

Here is the implementation of one of the member functions:

```
void Money: :addCents (long cents)
{
    amount_in_dollars = amount_in_dollars + cents/100.0;
}
```

Give <u>implementations</u> for the remaining four member functions of class Money.