Assessing Cognitive Learning of Analytical Problem Solving

by

Elodie V Billionniere

A Dissertation Presented in Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy

Approved September 2011 by the
Graduate Supervisory Committee:

James Collofello, Co-Chair
Tirupalavanam Ganesh, Co-Chair
Kurt VanLehn
Winslow Burleson

ARIZONA STATE UNIVERSITY

December 2011

ABSTRACT

Introductory programming courses, also known as CS1, have a specific set of expected outcomes related to the learning of the most basic and essential computational concepts in computer science (CS). However, two of the most often heard complaints in such courses are that (1) they are divorced from the reality of application and (2) they make the learning of the basic concepts tedious. The concepts introduced in CS1 courses are highly abstract and not easily comprehensible. In general, the difficulty is intrinsic to the field of computing, often described as "too mathematical or too abstract."

This dissertation presents a small-scale mixed method study conducted during the fall 2009 semester of CS1 courses at Arizona State University. This study explored and assessed students' comprehension of three core computational concepts - abstraction, arrays of objects, and inheritance – in both algorithm design and problem solving. Through this investigation students' profiles were categorized based on their scores and based on their mistakes categorized into instances of five computational thinking concepts: abstraction, algorithm, scalability, linguistics, and reasoning. It was shown that even though the notion of computational thinking is not explicit in the curriculum, participants possessed and/or developed this skill through the learning and application of the CS1 core concepts. Furthermore, problem-solving experiences had a direct impact on participants' knowledge skills, explanation skills, and confidence. Implications for teaching CS1 and for future research are also considered.

DEDICATION

I dedicate my dissertation work to my family and many friends. A special feeling of gratitude to my loving mother, Ms. Edmée Eugenie Billionnière whose words of encouragement and push for tenacity ring in my ears each morning. Her prayer has been answered.

I also dedicate this dissertation to my many friends and church family who have supported me throughout the process. I will always appreciate all they have done, especially my friend and mentor Debra Crusoe for being there for me throughout the entire doctorate program. She has been my best cheerleader!

Last but not least, I dedicate this work and give all the glory to God for keeping me on point and showing me the way when I thought this was not possible.

> "For we were saved in this hope, but hope that is seen is not hope; for why does one still hope for what he sees? But if we hope for what we do not see, we eagerly wait for *it* with perseverance." (Romans 8:24-25 NKJV)

ACKNOWLEDGMENTS

I wish to thank my committee members who were more than generous with their expertise and precious time. A special thanks to my committee co-chairs, Dr. James Collofello for taking me under your wing as a newcomer to the computer science education research and Dr. Tirupalavanam Ganesh for his countless hours of reflecting, reading, and proofreading my work. Thank you to Dr. Winslow Burleson and Dr. Kurt VanLehn for providing inputs.

I would like to acknowledge and thank my department for allowing me to conduct my research and providing any assistance requested. Special thanks goes to Audrey Avant, secretary administrative, and Martha Vander Berg, academic success specialist, for their continued support.

Finally I would like to thank the introductory programming teachers and teaching assistants in our department who assisted me with this project. Their flexibility and willingness to provide feedback made the completion of this research an enjoyable experience.

TABLE OF CONTENTS

APPENDIX Page

LIST OF TABLES

LIST OF FIGURES

I. INTRODUCTION

In this 21st century, the computing field has never mattered more. The explosion of new information technologies makes it possible to deliver more trusted, accurate, and timely information to the decision makers. Many applications created have changed how work is carried out and how business is organized worldwide as well as provided local and global solutions to environmental and societal matters. Yet, inappropriate use of these tools can lead to disaster for leaders and their organizations. Thus, the knowledge and skills that computer scientists acquire are critical resources for American society and the world.

Today's computer scientists are key players in problem solving as they identify, formulate, and solve complex real world problems. Therefore, there is a growing interest in better understanding how higher education institutions prepare future computer scientists, especially how students write computer programming code; which is important to success in the digital age. It is widely known that learning to program, even at a simple level, is a difficult task to achieve. A substantial number of students, at a rate as high as 50 percent, compared to 30 percent in the early 2000s, fail their introductory programming courses in every university, worldwide [1, 2]. Despite several academic interventions, the number of students failing the courses seems to increase rather than decrease over the years. The computer science education community still cannot fully understand why some students learn to program more easily and quickly while others

struggle. According to Ford and Venema [1], two potential causes that may have a direct impact on learning to program are: (1) novices' judgment of their abilities to achieve a specific task and (2) novices' internalization of real world objects and applications. The dropout and failure rates in the introductory programming courses at the university level are proof that learning to program is a difficult task. Moreover, if students drop out, fail, or struggle to pass a course in their desired major, it is unlikely that they will enroll in the subsequent computer science (CS) course. Thus, there has been ongoing investigation into the study of novice programmer errors as well as studies to examine how novice students write their programs in the introductory programming courses [3-6].

Programming is the core of CS, and therefore most national CS programs start with introductory programming courses (referred as CS1 courses). Regardless of the recognized importance of learning programming, there are two primary problems with CS1 courses: (1) the wide discrepancy in student preparation [7-11] and (2) the level of complexity of material to cover [12-15]. Many higher education institutions use their CS1 courses as general programming classes open to majors and non-majors. This results in a group of students with a wide range of previous computer experience, learning styles, backgrounds, goals, and expectations. Even when these classes are restricted to CS majors, the problem persists because the students' experience with programming is still widely varied. Furthermore, students who have no prior programming experience will most likely feel inadequately prepared, despite the fact that CS1 classes serve as introductory programming courses to teach programming.

The CS1 courses are often perceived as competitive environments where students "make it or break it." Indeed, the lack of self-confidence and the competitive environment have been identified as major contributing factors to the high drop-out rate in Science and Engineering courses, particularly among women and minorities [16, 17]. The abstract concepts of programming can be very challenging for CS1 students, particularly for those with little programming background and low confidence in their abilities. Because programming abilities are at the core of CS, skills in abstraction, conceptualization, design, and evaluation are essential for the success of students majoring in the computing field [14, 15, 18].

The Department of Computer Science and Engineering at Arizona State University (ASU) is concerned with the persistence of their freshman students and the improvement of student success. Students take either CSE100 (Principles of Programming with C++) or CSE 110 (Principles of Programming with Java) as their CS1 course. Both courses teach first year college students fundamental programming skills such as data representation in programs, running and compiling programs, simple input and output operations, control statements such as selection and repetition, and functions and parameter passing.

Many students struggle in CS1 courses and eventually are unsuccessful in their attempt to complete their first year programming courses. Table 1 summarizes the letter grade distribution for the academic years 2007-08 and 2008-09 at Arizona State University for CS1 courses. According to Arizona State University's grades and grading policies, the letter grades A and B for

undergraduate studies are equivalent to excellent and good standing whereas C, D, and E are equivalent to average, passing, and failure. Thus, it is fair to conclude that successful students in CS1 courses are the students who passed the course with an A or a B whereas students who received a C, D, or an E barely passed the course or failed the course. The letter grade W was assigned to students who dropped or withdrew from the course for unknown reasons.

Table 1 - Letter Grade Distribution: Academic Years 2007-08 and 2008-09

	CSE 100				**CSE 110**			
	Fall 2007	*Spring 2008*	*Fall 2008*	*Spring 2009*	*Fall 2007*	*Spring 2008*	*Fall 2008*	*Spring 2009*
Student Enrolled	*323*	*237*	*409*	*246*	*243*	*162*	*244*	*163*
A	104	92	116	120	84	65	102	63
B	80	44	87	42	48	41	43	34
C	42	28	69	15	33	19	26	21
D	18	14	36	7	10	7	12	7
E	21	17	31	13	23	10	29	12
W	58	42	70	49	45	20	32	26

Based on Table 1, Figures 1 and 2 depict the percentage of students who received a particular letter grade for CS1 courses. During the academic year 2007-2008, approximately 18 percent of the students enrolled in CSE 100 and 15 percent of the students enrolled in CSE 110 dropped or withdrew from the course, and an additional 25 percent or more failed in both courses by received a C, D or an E. These results show that approximately 40 percent of the students drop-out or fail the CS1 courses. Similarly for the academic year 2008-2009, the results show that approximately 35 percent of the students drop-out or fail the CS1 courses.

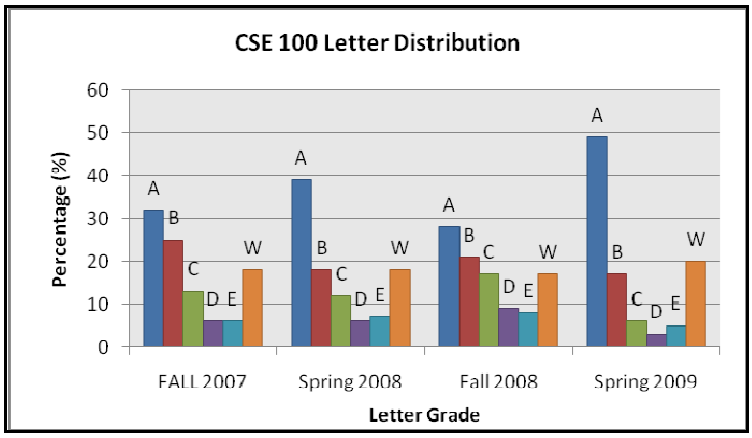


Figure 1: Letter Grade Distribution for CSE 100

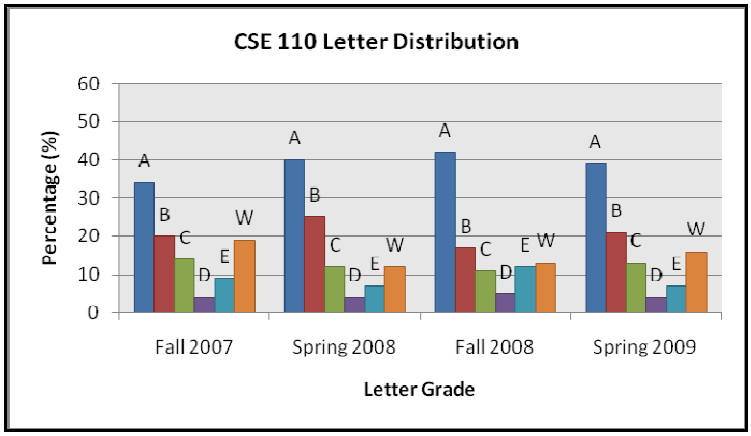


Figure 2: Letter Grade Distribution for CSE 110

As Ford and Venema [1] stated, the current drop-out/failure rate at other observed institutions ranges from 30 to 50 percent, and the ASU drop-out/failure rate is within this range. This high rate requires a closer look at the various factors affecting CS novices' success in CS1 courses.

A. Problem Statement

The increase of the drop-out/failure rate in CS1 courses over the last decade is a wakeup call for the computer science education community. It is critical that higher education institutions retain their students in their engineering/computer

science programs (quantity) and better prepare engineers/computer scientists for the workforce (quality). Over the last few years, we have heard from many hiring industries that the majority of the interviewed candidates are not able to write a basic program that prints the numbers from 1 to 100. To address this concern, one needs to examine what may be at the root of this problem: the initial programming courses that students experience in their undergraduate program. Basic programming is taught during the freshman year. As mentioned earlier, students are introduced to the fundamentals and core concepts of programming in the CS1 courses. Most likely if students have not mastered those basic concepts, then the more advanced concepts built upon the CS1 concepts cannot be learned appropriately. Therefore, students either (1) drop out from the program or (2) navigate through the program with skills gaps that may not be detected by instructors during the course experience.

This dissertation study attempts to inform this problem by investigating how students write code in CS1 courses and if any skills gaps can be identified at such an early stage in the computing field.

This problem of student success in first programming courses is not new, and many studies have been conducted to increase student learning in CS1 and CS2 (refers to more advanced programming courses emphasizing data structures, linked lists, binary trees and recursion) courses. Some of the studies have used active teaching-learning methods such as pair programming [19] and tracing program execution [20]. Additionally, some studies have used web-based interactive learning such as intelligent tutoring systems and video games to adapt

instruction to the learning style of each student [20-24] and *Just-in-time Teaching* to adapt lectures based on the learning progress of students [25]. Those studies have focused on improving learning through motivation and engagement.

In order to improve student success in CS1 courses, I propose to analyze one important factor: students' algorithm design based on CS1 core concepts. Novices spend a lot of time learning core concepts before thinking about the programming language they are coding in. Core concepts allow students to go through a change that enables them to begin to "think more like a computer scientist" [26]. Some of the core concepts in CS are abstraction, dependency, decomposition, encapsulation, iteration, and recursion [27]. As students progress in the course, their algorithm design evolves and reflects the core concepts they have learned in class, if understood. Furthermore, in order to understand their algorithm design, it is important to use different methods, which vary in their information abstraction level.

I am not proposing that it is necessary to collect data in regards to algorithm design for students to succeed in CS1. However, analyzing such data may enable CS1 instructors to better assess the comprehension and internalization of the materials presented in class and in the textbook.

The intent of this dissertation study is 1) to identify the core concepts that students have the most difficulty with and 2) to assess their ability in designing algorithms and solving problems based on the core concepts. This study aims to classify instances of student's mistakes based on the computational concepts described in the literature. Evaluating students on the core concepts will enable

documentation of the roots of the problem, if any, and help make recommendations to overcome these barriers in terms of teaching methods. Furthermore, assessing students on their logical reasoning and programming skills is essential to determine students' skill gaps in order to make the necessary interventions to close those gaps and instill confidence in their learning and abilities. Students who are confident about their information processing skills perform better [28] in their CS1 courses than those who are not as confident. Students who succeed in their classes tend to be more motivated to continue in their chosen major and more engaged in the classroom than students who are struggling with core concepts in CS.

B. Purpose

This dissertation research proposed to study how CS1 students design algorithms by using paper-and-pencil exercises, think aloud protocols, and interviews with a focus on three predefined core concepts based on a survey given to the CS1 instructors and teaching assistants. These three core concepts were: abstraction in object-oriented analysis, arrays of objects, and inheritance. Understanding how CS1 students think and solve problems is essential to identify students' skill gaps, improve teaching practices, and make recommendations to improve the learning of algorithm design.

The outcomes of this research study are to identify any problematic concepts, logical reasoning difficulties, and problem-solving difficulties that CS1 students may encounter. Additionally, this study will draw pertinent profiles of "good",

"average", and "poor" students based on the outcomes from the research methods used in this study to potentially make recommendations to improve teaching in the CS1 courses.

II. BACKGROUND LITERATURE

Algorithm design is a tedious task to achieve. Over the years, studies have shown that students experience substantial difficulties with CS1 courses. This literature review focuses on the various initiatives taken to ease the task of learning algorithm design using supplemental activities, studies that have analyzed novice programmers' problem-solution, and emergent research related to the notion of computational thinking as a mean to learn/teach the core principles in any discipline.

A. Student Success

Many students across the Science, Technology, Engineering, and Mathematics (STEM) disciplines take the introductory programming courses to learn the fundamentals about problem-solving and algorithm design. Research has shown that students taking those courses confront three main challenges that have a direct effect on student success: performance outcomes, engagement, and motivation.

1) Performance Outcomes: To improve student success, it is imperative to keep motivated students in the program and help students who are struggling to perform better. To do so, identifying students' skills gaps at an early stage is necessary to enforce intensive intervention in the course development [29]. Furthermore, it is of interest to discover patterns of successful, struggling, and repeating students enrolled in CS courses to isolate some of the causes affecting student success [30]. Some studies have shown the profile of successful students

in CS as passing both CS1 and CS2 on the first try with at least a B. On the other hand, students who have passed both CS1 and CS2 earning at most a C were unable to continue in the program or dropped out. However, students who have successfully repeated CS1 were able to acquire the skills required to succeed in CS2. Therefore we may conclude that doing well in CS1 is crucial to be successfully prepared for CS2. In addition, two feasible constructs that may have a direct impact on performance outcomes in the CS1 courses are self-efficacy and mental models [1]. The importance of student's self-efficacy can be at stake if the content is too advanced for the students to feel capable of learning the programming instructions whereas mental models have a direct impact on the student's ability to transfer conceptual ideas into concrete ideas. Thus, building good mental models strengthens self-efficacy [1].

2) Engagement: Studies have shown that the CS field is struggling with balancing theory and practice throughout its curriculum [31-33]. Academic environments in the computing field fail to reflect real-world problems that students can come across during their professional career. Additionally, CS concepts can be difficult for new students to fully grasp. Most of the concepts taught in the computing field involve an abstract knowledge base and, therefore, it is preferable to integrate meaningful projects to prepare and sustain successful students in this field. Easy-to-understand real-world applications, such as the web crawler and the spam evaluator, enable students to connect with the application and process the concepts easier [34-36].

Students find themselves spending many hours in front of the computer coding and debugging; often times they feel overwhelmed, discouraged, or disengaged by their programming projects [12, 18]. The absence of some type of engagement results in students' disinterest. It is essential to find ways to engage students in a fun and challenging environment without losing their confidence in 'doing' and by diversifying the programming projects [17, 37].

Engagement in activities such as paper and pencil exercises (i.e. tracing the logic of programs) and kinesthetic learning activities (i.e. matching types exercise for visual understanding of data type and how parameters must match when passed to a function) have proven to offer an increase in students' engagement in learning programming and have provided important information in terms of students' skills gap [6, 38].

3) Motivation: The literature suggests that high levels of academic and social integration will in many cases result in higher levels of the retention of students [39]. Social integration such as peers' collaboration and group activities are seen as having an impact on students' sense of belonging to a group or community [40]. Through these exercises, students are able to practice active, interactive, and/or constructive learning [41-43]. These different ways of learning keep all types of learners engaged, and therefore students' social integration in the computing field increased as well as students' motivation in staying part of this community.

Teaching techniques such as visualization activities and web-based applications enable students to assess their own knowledge and learn materials in

a way that fits their learning style. These techniques have offered promise in helping classrooms move toward an equitable learning environment, encouraging students to have positive beliefs about CS, and integrating CS with other disciplines [44, 45].

B. Program Design

Some of the above described initiatives used to improve student success in CS1 courses were based on research related to problem solving. This type of research provides more details on the common mistakes made by programmers as well as insights into the programmers' problem-solving methods, and thus enables researchers and instructors to have a better understanding of how students think and/or program. For the purpose of the scope of this study, the research investigation focused on how people, specifically novice programmers (since CS1 courses are primarily composed of freshman students who have no or little experience in programming), write their programming code. Even though much research has been done on how experts and novices write/solve their program, the root of the problem for the inability of students to solve a problem is still ongoing. It seems that the problem resides in the prerequisites to problem solving. More investigations in this area are needed.

1) The Nineteen Eighties Period: Soloway led the way in the area of studying the novice programmers in the 1980s. Soloway *et al.* [3] found that only 38 percent of novice programmers could write a program that successfully calculates the average of a set of numbers. In one of the case studies, Bonar and Soloway

provided a clear case of novice's programming bugs due to inappropriate use of natural language specification strategy [46]. The student, a novice programmer in Pascal, was writing pseudo-code for the problem: "Write a program which reads in ten integers and prints the average of those integers." She wrote the following (see Figure 3):

```
Repeat
(1) Read a number (Num)
        (l a) Count := Count + 1
(2) Add the number to Sum
        (2a) Sum := Sum + Num
(3) until Count :=I0
(4) Average := Sum div Num
(5) written ('average = ', Average)
```

Figure 3: Pseudo-Code for the Problem 3 in [46]

Despite some inconsistencies in the pseudo-code notation, her write-up is correct. However, when the interviewer asked whether (la) was the "same kind of statement" as (2a), it seems "that she thinks the Pascal translator knows far more about these roles than it does." Below is an extract from the interview after the student completed her pseudo-code [46, p. 12].

> **Subject:** How's that, are they the same *kind*. Ahhh, ummm, not exactly, because with this [la] you are adding-you initialize it as zero and you're adding one to it [points to the right side of la], which is just a constant kind of thing.
>
> **Interviewer:** Yes

Subject: [points to the right side of 2a] Sum, initialized to, uhh, Sum to Sum plus Num, ahh-that's [points to left side of 2a] storing two values in one, two variables [points to Sum and Num on the right side of 2a]. That's [now points to 1a] a counter, that's what keeps the whole loop under control. Whereas this thing [points to 2a], this was probably the most interesting thing. . .about Pascal when I hit it. That you could have the same, you sorta have the same thing here [points to la], it was interesting that you could have-you could save space by having the Sum re-storing information on the left with two different things there [points to right side of 2a], so I didn't need to have two. No, they're different to me.

Interviewer: So – in summary, how do you think of 1a?

Subject: I think of this [points to la] as just a constant, something that keeps the loop under control. And this [points to 2a] has something to do with something that you are gonna, that stores more kinds of information that you are going to take out of the loop with you.

Here, we see the novice programmer believing that the programming language knows more about her intentions than it possibly can. Hence, their results opined that the natural language seems to have a key effect on early conceptions and misconceptions of programming [46]. Furthermore, Soloway *et al.* [47] used a methodology named Goals and Plans (GAP) Trees, which specifies the relationship between goals and plans, to analyze the different type of errors that novice programmers make (see Figure 4). This descriptive methodology of buggy programs is based on the cognitively plausible, deep structure knowledge (i.e.

plan and goal) that describes a programming plan as a strategy for implementing a goal. "The relationship between programming goals and plans is that a goal can be achieved by any one of a number of different plans and a plan may give rise to several subgoals" [47]. Therefore the structure of a plan may be cut into pieces of knowledge that build the complete plan. These pieces of knowledge are programming plan schemas, which are stereotyped ways of solving a common programming problem [48].

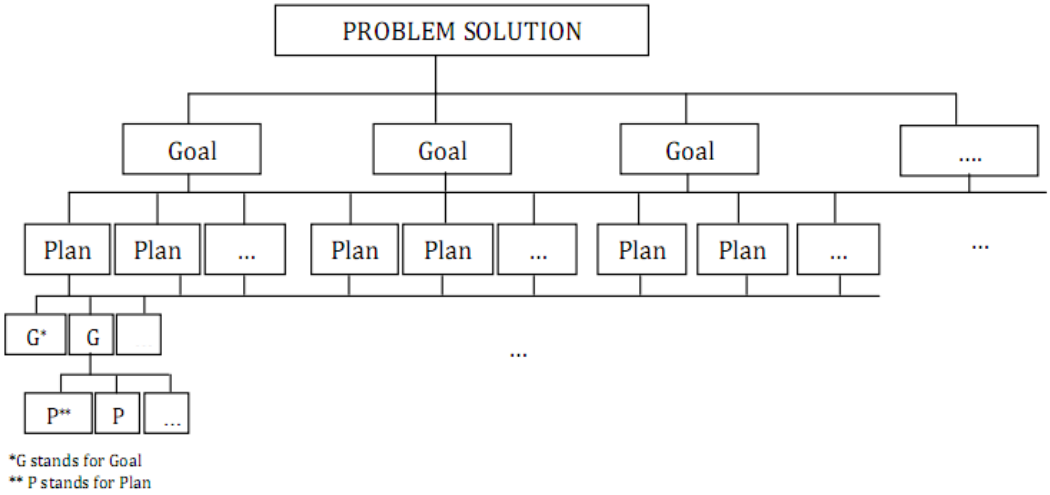


Figure 4: Simplified GAP Tree

From the GAP Tree, a schema is defined as a "remembered framework" [49]; the schema captures knowledge about the structure of the situation, derived from past experience. A plan schema is knowledge about the global structure of a problem. It is a series of ordered actions needed to execute the plan: first do this, then do this, and so on [49].

There are two types of GAP trees: Inferred GAP Tree (several plans per goal) and Solution Subtree of a GAP Tree (one plan per goal). An Inferred GAP Tree refers to all of the plans that can be used to achieve the goal of the problem whereas a

Solution Subtree of a GAP Tree refers to one particular plan that can be used to achieve each goal of the problem. A simplified version of the GAP Tree is shown in Figure 5. Using this methodology, Bonar and Soloway [46] identified seven components that compose a plan (see Table 2).

Table 2 - A Plan is Composed of Seven Components

Components	Description (using Pascal syntax)
Input	*READ* and *READLN* statements
Output	*WRITE* and *WRITELN* statements, for writing out either messages or variable values
Initialization	Initialization type assignment statements that give variables their initial value
Update	Assignment statements that change variables values
Guard	Conditionals, such as *IF* statements and the termination test of *WHILE*, *REPEAT*, and *FOR* statements
Syntax	Syntactic connectives which delimit the scope of blocks of code, such as *BEGIN*, *END*, *THEN*, *ELSE*, and *DO*
Plan	An entire plan, possibly composed of many of the foregoing microplan components

Soloway *et al.* were able to identify four ways that a program error can occur in a plan: Missing plan, Malformed guard, Misplaced syntax, and Spurious input [3]. Missing plan occurs when the *Plan* component is not present in the program. Malformed guard occurs when the *Plan* component is present, but it is not properly implemented. Misplaced syntax occurs when the *Plan* component is present, but it is in the wrong place in the program. Spurious input occurs when the *Plan* component is present, but it should not be. Furthermore, an extended version of the GAP Tree, see Figure 5, including the seven components of plan as well as the four program errors was developed by Segelman [50].

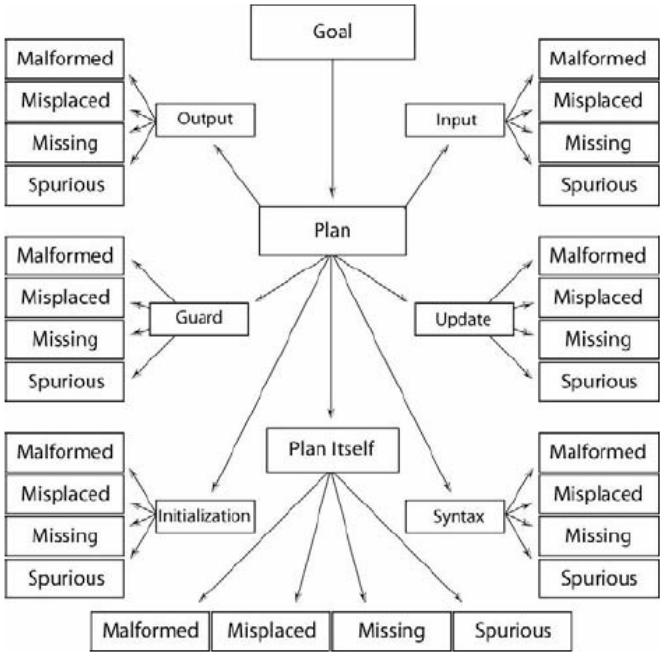


Figure 5: Extended Version of the GAP Tree [50]

The characteristics of this knowledge used in program design are forward, backward, top-down, and bottom-up design [49]. The initial approach for problem solving defines the starting point for design, but the solution path from start to end can often be long and complex. For example, consider a program that calculates the average daily rainfall for a month [48]. *If the program design is generated from the input (i.e. forward)*, then first the programmer must design the code to read in the rainfall per day. Once the input routine has been implemented, the mathematical calculations in the program must be specified, and the programmer will search for some plan that uses the rainfall data that has just been read into the program. The program design based on the input may reach an impasse (or not), and the output must be used to search for a solution. *If the program design is generated from the output (i.e. backward)*, then first the programmer must design

the plan that directly achieves the problem goal, by calculating the average rainfall (i.e. total/days). For this plan to work, the total rainfall must be found by adding the daily rainfall to a running total inside a loop. Hence, to calculate this total, the rain must be read, and the total must be initialized to zero before the start of the loop. The complete solution is executed. For the novice programmer, whose program design knowledge is limited, selection may be determined by whatever catches the attention; the novice simply captures some attributes of the problem, or even the solution, and starts from there. If the solution design begins with a search for the input, the novice looks for "read in" or "input is" in the problem statement, and thus identifies the information needed by keyword search. Whereas the experienced programmer searches the problem specification for the goals, retrieves plans to achieve these goals, and expands the plans until they match with the input data. And thus, as knowledge about how to design a solution develops, the decisions taken during design are more thought through [49]. Furthermore, Anderson *et al.* explained *decomposition design (i.e. top-down)* [51]. The knowledge of a novice consists of a set of schemas in the form of global structures of the program, such as a LISP operator, general function, or recursive function. In their study, the novices retrieved a schema from a text or from memory and implemented it to provide a solution to the problem. At the most abstract level, i.e., the program, they selected the first slot in the schema and retrieved a new structure to fill the slot, then repeated this process until the level of program code was achieved. The next abstract slot in the program was then selected and expanded, and once again the process was repeated until the program

was completely defined. Deviations from this approach are explained as *synthesis design (i.e. bottom-up)* and were first recorded by Jeffries *et al.* [52]. In their study, novices created programs by decomposing the problem into subproblems. However, novices were unable to decompose the problem at many levels of detail. Using abstract plan knowledge, they did an initial decomposition to design a solution for the problem at high levels of abstraction, but made unsuccessful attempts to retrieve more detailed schemas to continue the process. Therefore, often novices jumped straight to the level of program code, i.e. a very detailed level of planning. Their behavior changed to bottom-up design due to the lack of intermediate level schemas. Overall, studies [47-49, 51] showed that program design pattern depends on both the level of expertise of the programmer and the difficulty of the problem. If a programmer knows all the required abstract and detailed schemas, the design shows a pattern of top-down and forward solution approach, whereas a novice programmer has to create all the required plans and design, and so his/her design shows a bottom-up and backward solution approach.

Within this context of studying novice programmers, Perkins and others described novice learner's problem-solving strategies. Two types of learning styles were identified: "stoppers" and "movers" [4]. Stoppers appear to give up on the programming task at the first sign of difficulty, whereas movers use natural language knowledge to get a partial solution.

> "Stoppers and extreme movers can be viewed as being at endpoints of a continuum based on the ratio of time spent thinking (or time spent sitting in front of a terminal and not typing) to time spent entering and testing

code. But this image of a continuum is in a way misleading. It suggests a distribution with most students in the middle while extreme stoppers and movers occupy the statistically rare tails. On the contrary, the descriptions of stoppers and movers are not caricatures of the norm. Extreme stoppers or movers are common" [4, p.266].

Perkins found that stoppers can become movers if instructors encouraged them to decompose the problem and concentrate on a simpler subproblem only. Furthermore, Perkins and Martin [53] reported students have "fragile knowledge" of basic programming concepts and a "shortfall in elementary problem-solving strategies." This fragile knowledge is manifested through missing knowledge, inert knowledge, and misused knowledge [53]. *Missing knowledge* can be observed when a novice is asked to apply that knowledge in a program and the student "sort of knows, has some fragments, can make some moves, has a notion, without being able to marshal enough knowledge with sufficient precision to carry a problem through to a clean solution" [53, p.214]. This knowledge is commonly seen when students did not retain the knowledge taught. *Inert knowledge* can be observed when simple nonspecific prompts lead the students to recover the relevant knowledge and proceed correctly. In other words, they did not initially "retrieve command knowledge but in fact possessed it" [53, p.215]. Studies of programming instruction have reported that a considerable fraction of novice programmers' knowledge of commands in a programming language is inert. Also, this type of knowledge was also shown in the context of active programming, where there is almost no gap to transfer across [52, 56]. *Misused*

knowledge can be observed when students mix up several disparate elements in an attempt to fix the situation when they are uncertain [53-55]. This knowledge is commonly seen when students newly acquired knowledge.

2) *The 21st Century Period:* At the turn of the millennium, the research group of McCracken assessed the programming competency of 216 first-year CS students, Java and C++ programmers, from four universities across two countries [57]. Each student was required to write a program from a set of problems. Most students performed poorly and many students did not even complete the software development task from design to coding. The average grade was only 21 percent. Based on these results, McCracken *et al.* [57] suggested that students in the computing field are not taught programming adequately. However, the McCracken Group could not identify conclusive reasons for why the students struggled, but they speculated that it may due to inability of students to problem-solve. The group defined problem-solving as an iterative five step process:

(1) Abstract the problem from its description,

(2) Generate sub-problems,

(3) Transform sub-problems into sub-solutions,

(4) Re-compose the sub-solutions into a working program, and

(5) Evaluate and iterate.

While the work of the McCracken Group pointed out the current state of novice programmers, subsequent work is required to analyze the root of the problem, specifically if it is a language problem (i.e. object-orientation) or if it is a design problem (i.e. thinking process). To clearly make a distinction between the two,

one way is to ask students to demonstrate their understanding of existing code. This task does not involve problem solving. Building upon the McCracken research, the Leeds Group studied performance of students from seven countries on programming-related tasks. The novice programmers were required to answer multiple choice questions based on two types: "fixed code" questions and "skeleton code" questions [58]. *Fixed code* questions, also known as single value tracing, required students to predict the outcome value in a variable after execution of a given code. This type of questions required students to understand the constructs in the given code as well as to be able to trace by hand through code. In contrast, *skeleton code* questions required students to identify the correct missing lines of code from a set of four options. The results from this study showed that many students performed weakly at these tasks, specifically the skeleton questions which suggest that these students are "lacking knowledge and skills that are precursor to problem-solving" [58, p.139]. Therefore, this relates to the inability of students to read code rather than to write code. To further investigate these results, the BRACElet project currently focuses on the relationship between tracing iterative code, explaining code, and writing code [59]. So far, their findings have indicated that students who do not trace code cannot explain the code in plain English, and students who usually perform well at code writing are usually capable of tracing code and explaining code well [6].

C. Computational Thinking

In the midst of the struggle to resolve the underlying misconception that equates CS with programming, a new movement has emerged called "computational thinking" [60, 61]. Computational Thinking (CT) is one of the key practices of CS; a combination of logic skills with core CS concepts as an approach to problem solving.

The idea of CT is to integrate problem solving techniques and approaches into all disciplines, from the sciences to humanities. Just as the current three fundamental skills - reading, writing, and arithmetic - CT is a fundamental analytical skill needed for every citizen to function in today's global society [60, 61, 62]. These fundamental skills are to "describe and explain complex problems to others" [71]. Wing [60] goes even further by prophesizing that CT will be a fundamental skill used by everyone in the world by the middle of the 21st century. Recent recognition by the National Science Foundation (NSF) seems to support the idea that CT is an important component for science, technology, and society; and thus deserves our immediate attention. The NSF's Computer and Information Science and Engineering (CISE) directorate has requested that most proposals include a discussion of how their projects advance computational thinking. In particular, the NSF CISE Pathways to Revitalized Undergraduate Computing Education initiative has asked educators to present projects that introduce computational thinking into some aspect of education, research, and outreach. Furthermore, from the website of Carnegie-Mellon University's Center for Computational Thinking, one can read "it is nearly impossible to […] research

in any scientific or engineering discipline without an ability to think computationally. […] [We] advocate for the widespread use of computational thinking to improve people's lives" [63].

1) What is Computational Thinking? CT is a way of reasoning in such a manner that one defines problems, processes and relationships to solve those problems. Seymour Papert first introduced this term in 1996 as a way to solve problems more efficiently using novel approaches to problem-solving [64]. Nowadays, the concept of CT is being spearheaded by Jeannette Wing, President's Professor of computer science and department head at Carnegie Mellon University who also works at the NSF as Assistant Director for its CISE Directorate. Wing [60] defined CT as follows:

> "Computational thinking involves solving problems, designing systems, and understanding human behavior, by drawing on the concepts fundamental to computer science. Computational thinking includes a range of mental tools that reflect the breadth of the field of computer science."

Wing defined CT as the use of CS concepts to solve a problem in any domain. Some "everyday examples" [60] of computational thinking that she outlines include:

> "When your daughter goes to school in the morning, she puts in her backpack the things she needs for the day; that's prefetching and caching. When your son loses his mittens, you suggest he retrace his steps; that's backtracking. At what point do you stop renting skis and buy yourself a

pair?; that's online algorithms. Which line do you stand in at the supermarket?; that's performance modeling for multi-server systems. Why does your telephone still work during a power outrage?; that's independence of failure and redundancy in design."

Furthermore, to help clarify the notion of computational thinking, Wing [60] listed six characteristics:

1. CT is conceptualizing via multiple levels of abstraction
2. CT is a fundamental skill needed for everyone to function in modern society
3. CT is not about solving problems like computers, but rather it develops all critical skills of humans to solve problems
4. CT complements and combines mathematics and engineering thinking
5. CT is principally concerned with ideas as opposed to artifacts
6. CT should be an integral part of everyone's education

Despite the great efforts from the computer science educators, the definition of CT at the present remains abstract, and thus this method of instruction is difficult to apply without knowing exactly what we expect students to learn [65, 66, 67].

2) Computational Thinking and Computer Science: CT has a long history within CS. Known in the 1950s and 1960s as "algorithmic thinking", it meant a mental practice to formulating problems in terms of step-by-step procedures involving the conversions of some input to an output to solve the problems [68]. Today, the term CT has been expanded to include (1) thinking with many levels of abstractions to understand and solve problems more effectively, (2) use of

mathematics to develop more efficient, fair, and secure algorithms, and (3) examining how well a solution scales across different sizes of problems for efficiency, economic and social reasons [63, 69, 70].

Furthermore, CT is seen by the computer science community as a revolutionary movement to define what the core of the field is about, to provide a way to reverse the decline of enrollments in the CS field by making the field more attractive for students to major in and for other disciplines to collaborate with, and to recognize CS as a legitimate field of science. Many computer scientists view CT as comparable to other basic cognitive abilities such as mathematical, linguistic, and logical reasoning that the average individual in modern society should possess [71]. Thus, the CS1 and CS2 courses are changing to meet the needs of students in other disciplines who are using computation and programming; thus programming is presented as a tool used to investigate areas from all disciplines (i.e. computer science, other sciences, and humanities) and an essential part of CT [69, 70, 71]. The primary objective is to give a solid foundation of basic programming and establish an understanding of the algorithmic thought process [69]. Programming is a language for expressing ideas, and therefore, you have to first learn how to read and write that language to be able to think in that language [71]. The teaching of CT should concentrate on creating vocabularies and symbols to describe computation and abstraction, recommend information and execution, and provide notation around which mental models (i.e. abstractions and methods) of processes can be build [60].

CT was defined in a number of ways such as 1) notions of procedural thinking, 2) study of mechanisms of intelligence, 3) processes, 4) formulation of precise method of doing things, and 5) open-ended and growing list of concepts that reflects the "dynamic nature of technology and human learning" [71]. These definitions are ideas extracted from the discussions among computer scientists at a workshop on CT.

According to Wing [70], computing is defined as the "automation of our abstractions" whereas CT focuses on the process of creating and managing abstractions, and defining relationships between layers of abstraction. Wing argued that CS has developed a set of CT skills that have direct impact beyond the computing field. She stated that such ideas as abstraction, layering of abstractions, and automation are fundamental CS concepts that have already yielded new insights. To assess CT in the CS field, one can look at the following five CT concepts described in Table 3: abstraction, algorithm, scalability, reasoning, and linguistics.

Abstraction can be defined as the process of eliminating the non-significant details of a problem to concentrate on the relevant details and their relationships. Abstraction is an essential core concept in CT. Wing mentioned that CT is "conceptualizing" and "thinking at multiple levels of abstraction" [60]. However, the concept of abstraction has been difficult to translate into CS1 courses. By categorizing abstraction as a "soft idea," Hazzan [72] indicated that teaching this concept by lecture is not enough to increase students' awareness about the concept of abstraction. Students must be able to identify the level of abstraction,

recognize the existence of different abstraction levels, and use abstraction in the learning process. If students are not able to apply abstraction then it may be of interest to "train" them for such task [73]. This remark was made by Kramer through the observation that in the CS curriculum offered at Imperial College, no one course explicitly focuses on teaching the concept of abstraction.

Algorithm is another fundamental CT concept that is often introduced in CS1 courses as a set of rules that describes how to solve a problem [74]. This concept may be described as a program, pseudo-code or step-by-step explanation (in plain English) of how to do something. This CT concept shows the ability of students to specify a problem precisely and construct a correct algorithm to a given problem using basic action steps.

Scalability is the ability of an algorithm and design to handle future growth plan in a graceful manner or its ability to be enlarged to accommodate that growth. It must be suitably efficient to plan ahead for scalable algorithm and design based on potential future growth of the problem. This CT concept is sometimes introduced in the CS1 curriculum towards the end of the semester. If not, it is definitely covered in CS2 courses. Usually, scalability is referred in the curriculum when covering sorting and searching algorithm techniques to help students to understand how to improve a problem solution; and thus the importance to design and construct scalable problem solutions [75].

Reasoning constitutes rules that underlie logical and mathematical structures in the algorithm and design. The formulation of reasoning is seen through logic constructs such as automation, loops, and recursion. This involves the repeating of

a procedure until a desired goal is reached such as *if conditions then conclusion*. This CT concept underlines the ability to apply mathematical constructs to the algorithm [63].

Linguistics includes primarily semantics and syntax. Semantics in problem design and solution is the meaning that is used to express the abstraction of information whereas syntax is mainly bounded to the programming language used and/or modeling language annotations. This CT concept provides clear and meaningful descriptive annotations and follows the principles and rules governing the behavior of the chosen programming/modeling language used to design and solve the problem.

Through those CT concepts, it is expected that undergraduate courses taught during the freshmen year would enable students to "adopt the thinking habits and reasoning methods of computer scientists", i.e. students would learn about the core computational concepts [71]. However, as of today, the computer science education community is still focusing on exploring the scope and nature of what CT is/is not and its cognitive and educational implications [70, 71].

Table 3 – CT Concepts

CT Concepts	Description
Abstraction	- Deciding what details need to be highlighted and what details need to be ignored - Defining the layers of interest such as classes, data members, methods, and the relationships between the layers
Algorithm	Correctness of the program should answer the following questions: - Does it do anything? - Does it do the right thing? - Does it compute the right answer?
Scalability	Ability of the program to be enlarged to accommodate growth in a graceful manner
Reasoning	Correctness of the controls such as recursion, iteration, and conditional statements
Linguistics	Correctness of the syntax and semantics

D. Summary

Today, with universities attempting to improve student success in the CS1 courses, many computer science programs are trying different strategies. The visual web-based and real-world applications may be ways for some programs to check whether their efforts are successful or whether further adjustments need to be made. Certainly, the research investigation is more insightful when students' problem-solving and program design is tracked through those applications rather than a focus on the number of correct answers. Finally, it is important to further

investigate program design in our current era of computer science. The McCraken and Lister working groups came to the conclusion that many first-year programming students cannot program at the end of their CS1 courses mainly due to difficulty with problem-solving. From their observations, they deducted that knowledge and skills are the precursors to problem solving. Thus, the next logical research step is to assess students' algorithm design and problem solving skills, students' knowledge of the CS1 computational concepts, and classify their mistakes in their work in terms of computational concepts; which is the intent of my proposed study.

III. METHODOLOGY

This study was designed to understand the skills that freshmen develop in their introductory computer programming courses. This study aimed to inform CS1 instructors with a better understanding of how their students design algorithms (Unified Modeling Language, also known as UML) and how their students solve a given problem through programming (coding). UML works as an architecture tool providing a high level view of the problems by extracting key information such as classes, data members, methods, and connections showing relationships. Coding creates a program that exhibits a certain desired behavior that requires basic instructions such as input, output, arithmetic, conditional execution, and repetition. To investigate skills such as design and problem solving, I collected quantitative and qualitative data described in Figure 6. At the beginning of the Fall 2009 semester, instructors and teaching assistants (TAs) were first surveyed on the concepts that students struggle the most with based on the instructors and TAs' teaching experience within the past two years. Based on their answers and the curriculum, with the assistance of the instructors, I developed paper-and-pencil exercises which focus on these particular troublesome concepts. Each paper-and-pencil exercise was divided into three sections: algorithm design, problem-solving, and bonus points pertinent to "fresh" knowledge (i.e. material covered the day prior or the day of the exercise). The paper-and-pencil exercises were given to the CS1 students enrolled during the Fall 2009 semester. The CS1 students representing the student body who took the

written exercises are referred as “Group 1”. From these paper-and-pencil exercises, a letter grade was assigned based on specific criteria to assess the identified concepts. The exercises were scored to ensure scores represent quantitative data. Furthermore, students from “Group 1” were invited to participate in a think aloud experiment as well as an interview to explain their algorithm design and problem-solving method. This small set of students is referred as “Group 2”. Data obtained through these methods represent qualitative data.

The primary aim was to identify any problematic concepts, logical reasoning difficulties and problem-solving difficulties that CS1 students may encounter when attempting to do the paper-and-pencil exercises. Secondary issues to be examined included the comparison of students’ level of expertise by drawing pertinent profiles of “good”, “average”, and “poor” students based on the outcomes from the research methods used in this study.

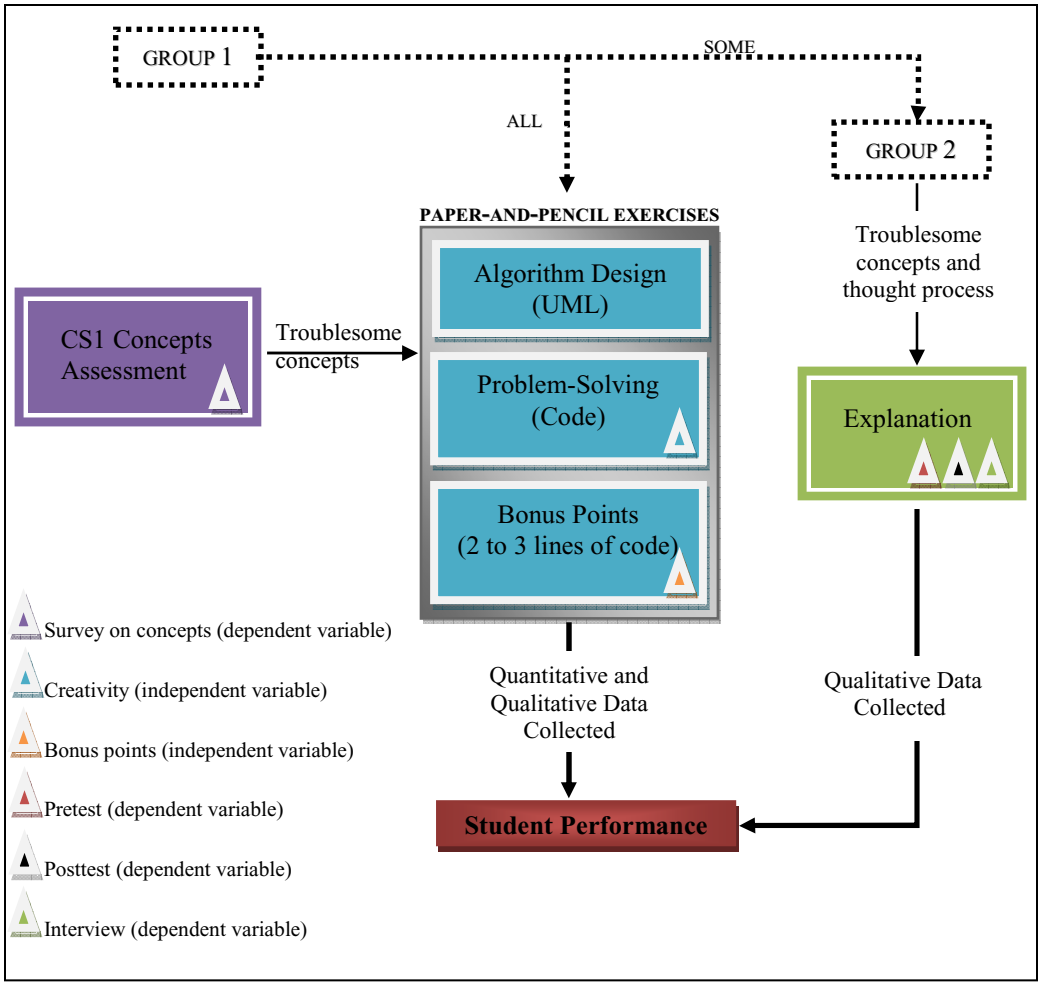


Figure 6: Assessment of Student Performance with CS1 Concepts

A. Data Collection Design

The data collection design was developed during the Summer 2009 semester to ensure that permission to conduct research was obtained from Institutional Review Board (IRB) so I could begin the study during the fall semester.

1) Participants and Site: The "Group 1" was composed of all students enrolled in CS1 courses whereas the "Group 2" was a subset of "Group 1"

composed of students of age at least 18. This age limitation for Group 2 was chosen to avoid the challenge of seeking parental permission for study participants who are younger than 18 years of age. The instructor for CS1 courses used the paper-and-pencil exercise as quizzes in his classes. The instructor provided me with a copy of students' responses to these exercises without revealing students' identity. The study was conducted at the ASU Tempe campus. The paper-and-pencil exercises were in-class tests, and thus were conducted in the classrooms where the lectures were held on a Thursday whereas the think aloud and interview exercises took place the following Monday in a non-classroom setting.

2) Sampling Strategy: Based on Patton [76, p. 243], the sampling strategy was a maximum variation sampling to ensure that the selected volunteers for the think aloud and interview exercises were diverse in terms of programming language, age, gender, grade, and major; and thus, well-representation of the CS1 student body in the sampling.

3) Sample Size and Groups: Enrollment for CSE 100 (Class# 72301) and CSE 110 (Class# 72321) was at 81 and 64 students respectively. Out of these 145 students, six to ten students volunteered to do the think aloud and interview exercises. This study included two groups. Group 1 is the group of CS1 students who did the tests in-class whereas Group 2 is a smaller group of Group 1 who volunteered to do the think aloud and interview exercises.

B. Data Collection Procedures & Protocols

The data collection procedures and protocols were designed while completing the IRB application for the approval of this study (Appendix A). All the processes described in this section were required as part of the IRB application before any study may take place.

1) Survey on CS1 Concepts: Based on the procedure used in [77, 78] to identify the most difficult topics in CS1, I similarly surveyed by email, instructors and teaching assistants (TAs) in CS1 courses to identify the most troublesome concepts in the courses (see Appendix B) and then I rank-ordered the troublesome concepts based on the frequency of their occurrences. From this ranking, there were three top troublesome concepts: abstraction in object-oriented analysis, arrays of objects, and inheritance.

2) Designing Tests: Based on the outcomes of the survey given to the instructors and TAs, paper-and-pencil tests were developed to address the basic elements that encompass each troublesome concept. The instructor who taught CSE 100 and CSE 110 courses then reviewed these tests. The instructor gave some feedback and/or made necessary changes based on the progress made in class. The first two tests (see Appendices C and D) were the same for both courses and the last one (see Appendices E and F) was different because the class CSE 100 was behind in the curriculum; thus, adjustment was needed to fairly assess the participants based on what they have learned in class.

3) Recruiting Participants: To recruit participants from the CS1 courses, I asked permission from the CS1 instructor to come into his class to make an

announcement and have the TAs to email the recruitment forms to the roster. Once I received all the forms back, I tried to select students for participation in the study to reflect the overall CS1 student body in terms of diversity in age, major and gender. However, ultimately, participants were chosen based on their availabilities in order to maximize the number of student participants in the study.

4) Collecting Tests: To collect the in-class tests, I met with the instructor after the classes ended and he handed me the copies to make photocopies of them. Then I returned the copies to him within 24 hours. A random numerical number was assigned to each participant. These numbers were used throughout the study to maintain the confidentiality of all information concerning research participants. This information included, but was not limited to, all identifying information and research data of participants and all information accruing from any direct or indirect contact I had with the participants.

5) Think Aloud Protocol: This protocol was used for the selected volunteers who decided to partake in the survey and interview phases (i.e. "Group 2"). I explained to the participants about the verbalization that I expected throughout the exercise (see Appendix G) and a warm-up exercise was conducted to ensure that the participants fully understood the think aloud protocol. Then the participants were prompted to complete the test as they stated aloud their thinking while the participants were audiotaped. The participants all had the same amount of time to complete the test, which was 30 minutes. This protocol may assist with the assessment of subjects' communication skills and detection of their misconceptions and confusions about the concepts.

6) *Survey Questionnaires:* After completing a paper-and-pencil exercise on a Thursday, participants were asked to come back the following Monday to respond to individualized pretest and posttest surveys (see Appendices H and I) as well as an interview. The surveys were developed based on a similar study that investigated the struggles encountered with CS1 concepts [78]. The participants' answers to the pretest and posttest surveys helped assess pertinent (1) knowledge skills and (2) explanation skills such as:

a) Comprehension of the core concept (1, 2)
b) Rephrasing of the core concept with no technical words (1)
c) Prior knowledge needed to gain a good understanding of the core concept, if any (1)
d) Real-world examples in regards to the core concept (2)
e) Context of utility of the core concept (1)
f) Thoughts and reactions, before, during and after the process of solving the paper-and- pencil exercise based on the core concept (2)
g) Concepts and/or elements where the participants were stuck at first but then became clearer, if applicable (1, 2)
h) Concepts and/or elements where the participants were stuck and how they dealt with this situation (1, 2)
i) Concepts and/or elements where the participants were stuck and suggestions/advice to help other students who might be struggling with the same concepts and/or elements (2)

j) Impact that the understanding of the core concept has on other things, if any (1, 2)

7) Interview Questionnaire: The interview was added in case that the think aloud protocol was not very conclusive. Also, it provided a temporal dimension to the thought processes that arose within the context of solving the problem (see Appendix J). Participants had a chance to reflect on the given problem over couple days, precisely four days, and come back to debrief on their answers as well as to reiterate their reactions and thoughts when solving the problem.

C. Data Analysis Procedures

The data analysis was divided into two parts: quantitative data analysis and qualitative data analysis.

1) Quantitative Data Analysis: The tests were assessed using grading criteria for each section. For example, the grading of the first exercise was based on the assessment criteria indicated in Table 4.

Table 4 – Scoring Grading Criteria

Tests	Criteria
Problem design (Part I – 4 points)	✓ Successfully indicated classes ✓ Successfully identified data members ✓ Successfully identified methods ✓ Assigned proper data types to data members ✓ Assigned proper parameters and return types to methods
Problem solving (Part II – 6 points)	✓ Properly formed method signatures ✓ Properly formed variable declarations ✓ Properly formed method invocations ✓ Included correct methods ✓ Properly formed variable assignment ✓ Properly formed method declarations ✓ Proper reasoning/logics ✓ Proper syntax
Bonus points criteria (Part III – 5 points)	✓ Problem design solution to bonus points exercise ✓ Creativity

The instructor of the courses determined the assigned points for each section. The base score was 10 points, which does not include the two independent variables - bonus points and creativity.

The sections of the tests (i.e. quizzes) were open-ended questions. The open-ended questions used in tests often require a more in-depth thinking from the students and can disclose more about how students understand and reason with the course concepts than do multiple choice questions. Traditional tests (see example in Appendix K) often do not disclose much about how students think about the course concepts. Since students do not have to use their conceptual understanding, their solutions often accentuate a single value response either a

numeric answer when tracing code or an alphabet answer when choosing a response from a multiple-choice question; and thus such type of tests demand a single correct response. The instructor and/or TA can only opt by assigning full credit or no credit. Whereas in tests with open-ended questions, students must give a solution that accentuates how they came up with an answer which can be more informative than traditional tests; showing students' understanding, ability to reason, and ability to apply knowledge in less traditional contexts. Such tests can communicate the levels of student achievement more clearly than multiple-choice items, and thus, give better guidance for instructions.

An application of Amabile's consensual assessment technique for rating the tests was applied. Based on [79], the three requirements for the task itself must be satisfied [79, p.1001]:

1) The programming task did not depend on specialized skills. It was solely based on what was taught in the classroom/lab setting. There was no pre-requisite for CS1 courses and, therefore, no prior knowledge in programming was expected.
2) The programming task was an open-ended question, which enabled flexibility in responses. For example, a typical exercise was "Based on your UML diagram above, please develop the [*ClassName*] class".
3) The programming task was a paper-and-pencil type exercise, i.e. a written response, which was easily accessible.

In addition, this study used the following assessment procedure [79, p. 1002]:

1) The judges were two graduate students (one female and one male) in the computing field who have taken the CS1 courses as part of their undergraduate curriculum. If the two judges have a different point of view, a third judge, another graduate student in the program, would evaluate the specific test(s) to break the tie. However, in this study, a third judge was not needed as the two judges were able to come to an agreement for all the participants.
2) The judges assessed tests based on their "own subjective definition" of each criteria such as logic, clarity, identification of attributes and methods, type parameters, syntax, and more without consulting each other.
3) The judges assessed each programming exercise by comparing one programming exercise to another one.
4) The judges were given a stack of completed tests, which included copies of each programming exercise. The order of the copies in each stack was random for each judge.

Furthermore, using Amabile's assessment technique, creativity was assessed for section 2 of each test, which was the problem solving (code). The creativity in section 1, the algorithm design (UML diagram), would have been a bit difficult to assess. This section was a straightforward exercise and therefore this section was not assessed for creativity to stay away from negative creativity. For example, a few students, who were unable to retrieve their knowledge about the UML diagram, came up with 'strange' answers such as a spiral and a one-paragraph write-up. One may have thought that their answers were creative but the display

of their knowledge about algorithm design was very poor. Based on the notion of "creativity," some of the initial impressions of the data were found in students who did the following:

1) Checked for positive deposit. Only a few of them thought about error handling for deposit. Who would think about making a deposit of a negative amount?
2) Used an array in test 1 when it was not the concept tested on. It was supposed to be in test 2 since arrays were learned after the test 1.
3) Underlined methods and attributes in the problem description
4) Used Boolean methods instead of void methods
5) Named the variables

The assessment of quantitative data was conducted using the software Statistical Package for the Social Sciences (SPSS). The statistical analysis explored the overall student performances over the three tests and attempted to find any correlations existing among exercises. Furthermore, data on students' background were collected, and thus, comparison within-students was applied to differentiate any students' design and problem-solving performance based on factors such as course, major, gender, ethnicity, and prior programming experience.

2) Qualitative Data Analysis: Even in quizzes with open ended questions it may not be that simple to see how a student was thinking on a problem or why they answered a particular way. In such a case, the computer science education researcher may decide to conduct interviews with some of the students. The

written data (tests) and the verbal data (think aloud/interviews) were assessed by first identifying episodes demonstrating some skills and then the judges came together to compare their respective episodes and agreed on the skills to use for assessment. The goal of the written data is to present samples of student performance to showcase common mistakes made by the student body participating in this study. Whereas the goal of the verbal data is to present the skills that were identified in the think aloud and interview protocols and showcase some interesting excerpts from the interviews that are related to the CS core concepts described earlier and the skills identified.

In this study, participants were students enrolled in at least one of the CS1 courses and the problems were open-ended design problems. Three datasets – Problem design, Problem solving, and Think aloud/Interviews - can be distinguished in Figure 7 below. Three different analyses were conducted which included students' profiles, computational thinking skills, and core concepts abilities.

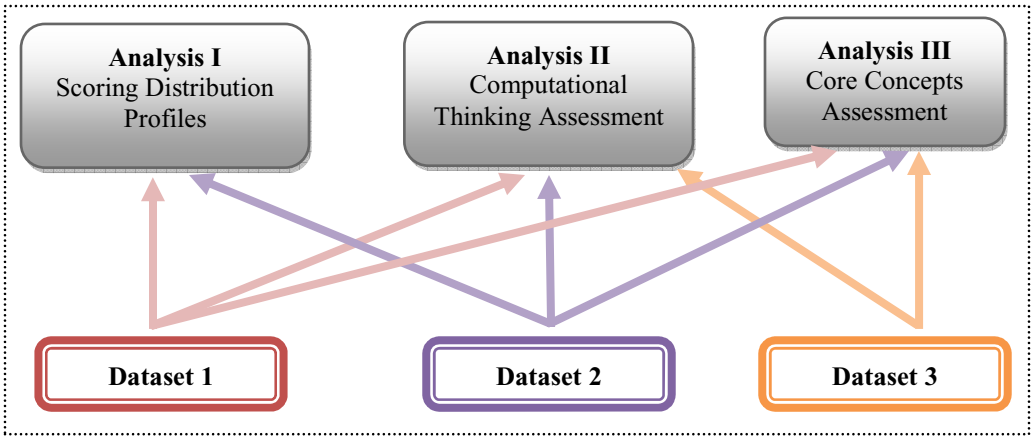


Figure 7 – Overview of Qualitative Data Analysis Procedure

D. Verifying Data Accuracy

The data accuracy was verified during the data collection. As the study progressed, changes were made to avoid clerical errors, subjective errors, and methodological errors.

1) Avoiding Clerical Errors: The data collection worksheets were checked against the original source of documents (i.e. copies) to ensure consistency with the assigned identification numbers for both the verbal and written data. Only one individual was in charge of recording the data in question to limit inconsistency and inaccuracy.

2) Avoiding Subjective Errors: When dealing with data involving subjective ratings such as those provided by teaching assistants, an effort was made to determine the accuracy of the rating system. This was accomplished by examining the rating scale to determine how clear and comprehensive the descriptions were of the various rating categories. To ensure accuracy, a designated individual double-checked the grades entered based on the assessment criteria.

3) Avoiding Methodological Errors: Of the data collection techniques existing, surveys are most prone to methodological error. The survey instrument (i.e. questionnaires) was reviewed for possible bias by a dissertation committee member and the Office of Research Integrity and Assurance at ASU.

4) Avoiding Assessment Errors: The assessment of both qualitative and quantitative analyses was conducted by two graduate students who scored the data independently using the assessment criteria described earlier in this chapter. Then the two judges got together and compared their assigned scores. If the scores were the same then this was the final score. Otherwise, the judges had to reassess the data until they came into an agreement.

E. Limitations

This study was limited by the number of college students willing to participate in the interview process. Although the study began with 145 students, only 93 participants from CS1 courses took part in all tests (49 from CSE 100 and 44 from CSE 110). Thus, the study only looked at this subset of 93 participants to determine their overall performance across the three tests. Furthermore, only 6 students were interested in participating in the corresponding think aloud and interview phases. There was a delay in curricular implementation during the semester, closer to the time when the third test was to be implemented. As a result, the third test had to be adjusted to reflect the latest material covered in class and the initially planned assessment could not be fully executed. Because this exercise was finally implemented closer to the end of the semester, the

number of participants for the corresponding interview and think aloud protocol was low. Although the data collected was useful, the delay in curricular implementation bounded the data collection for think aloud and interviews to primarily the first two tests. Nonetheless, this study has useful results that will inform instructors of CS1 courses.

A secondary limitation to the study was the time needed to conduct assessment on additional troublesome concepts. Due to the full curriculum already in place for CS1 courses, the study focused only on the top three concepts identified by the survey given to the instructors and TAs.

Last but not least, the duration of each test was only 30 minutes, which is a short time to complete the exercises. Therefore students had to make decisions rapidly, and the scope for reflection was limited, especially for the bonus exercise. This time limitation for test was due to the allocation of time for lecture and review of materials prior to taking the test.

IV. DATA ANALYSES AND RESULTS

In this chapter the results of the data are presented. The data were collected and then processed in response to the problems posed in chapter 1 of this dissertation. Two fundamental goals drove the collection of the data and the subsequent data analysis. Those goals were to develop a base of knowledge skills about what CS1 students know or do not know about the core concepts in terms of the computational thinking's characteristics: abstraction, algorithm, scalability, reasoning, and linguistics; and to compare their overall design and problem-solving solutions. These objectives were accomplished. The findings presented in this chapter demonstrate the potential for making significant recommendations to the CS1 instructors.

A. Response Rate

One hundred and forty five CS1 students were initially identified to take the tests, including 81 in CSE 100 and 64 in CSE 110. However, only 93 participants completed all tests. With 93 participants out of 145, the response rate was 64 percent. Furthermore, the last test for CSE 100 did not include any design and coding due to a delay in the curricular implementation which bounded the data collection for CSE100 to the first tests. Therefore, 93 participants were considered and only the first two tests for CSE 100 were considered to be legitimate for this research. Two hundred and seventy-nine tests were obtained but only 230 usable responses (98 in CSE 100 and 132 in CSE 110) were analyzed. With 230 usable responses out of 279, the utility rate was 82 percent.

I compared the means from the two samples in each course to ensure that the actual sample that I am using represents the students who took the tests. The first sample (iUML1, iUML2, iUML3, iCoding1, iCoding2, and iCoding3) represents all the students who did not take each specific piece of each test whereas the second sample (UML1, UML2, UML3, Coding1, Coding2, and Coding3) represents the students who took all three tests (see Table 5).

Table 5 – Comparing Means among Specific Pieces of Each Test

Pieces of each Test	**CSE 100 mean**	**CSE 110 mean**
iUML1	.790	.653
UML1	.801	.622
iUML2	.864	.629
UML2	.847	.660
iUML3	n/a	.748
UML3	n/a	.773
iCoding1	.613	.520
Coding1	.626	.583
iCoding2	.739	.790
Coding2	.724	.789
iCoding3	n/a	.750
Coding3	n/a	.778

As it can be noticed, the difference in means is somewhat minimal; and thus the proposed sample that I used (93 participants) in this study is accurate and also very close to the source sample (145 participants).

B. Participant Background

Of the 93 participants, 53 percent were enrolled in CSE 100 (C++) and 47 percent were enrolled in CSE 110 (Java). Table 6 represents the overall background information of the participants based on major, gender, ethnicity, and prior programming experience. Even though the study attempted to achieve balance, this was not possible due to the student population in CS1 at that time. Nonetheless, the participant background distribution is a good representation of the CS1 students at Arizona State University.

Table 6 – Participation Background Distribution

	Major		**Gender**		**Ethnicity**		**Prior Programming Experience**	
	CS	*Non-CS*	*Female*	*Male*	*White*	*Non-White*	*Yes*	*No*
CSE 100	82%	18%	31%	69%	59%	41%	10%	90%
CSE 110	93%	7%	25%	75%	64%	36%	43%	57%
CSE 100 ∩ CSE 110	87%	13%	28%	72%	61%	39%	26%	74%
Total	*100%*		*100%*		*100%*		*100%*	

C. Intercoder Reliability

As mentioned in the previous chapter, two judges independently evaluated the tests and reached an agreement. As Neuendorf indicated "[w]ithout the establishment of reliability, content analysis measures are useless" [80]. Furthermore, Kolbe and Burnett [81] note that "interjudge reliability is often perceived as the standard measure of research quality. High levels of

disagreement among judges suggest weakness in research methods, including the possibility of poor operational definitions, categories, and judge training." Thus, intercode reliability is necessary because its proper assessment makes coding more efficient and all the work involved - data gathering, analysis, and interpretation - is unlikely to be dismissed by skeptical reviewers [82].

There are many different measures of intercoder reliability and despite all the efforts devoted to develop and test measures, there is no consensus on one universally accepted measure [82]. However, the Cohen's kappa measure seems to be the norm in research that involves behavior and learning [83]. In addition, Cohen's kappa can be calculated using SPSS. To do so, in the data setup format, each row represented a single case (i.e. a single participant) and each column represented the coding judgments of a particular coder for a particular variable (i.e. UML1, Coding1, and etc). It is rare that a perfect agreement is reached. Different people have different interpretations. As a rule of thumb values of Kappa from 0.40 to 0.59 are considered moderate, 0.60 to 0.79 substantial, and 0.80 outstanding [84]. Most statisticians prefer for Kappa values to be at least 0.6 and most often higher than 0.7 before claiming a good level of agreement. From the SPSS program outputs, the level of reliability for the kappa index for UML1, Coding1, UML2, Coding2, UML3, and Coding3 is summarized in Table 7. In any case the level of reliability is always acceptable as all the Kappa coefficients are greater than 0.90.

Table 7 – Quantitative Symmetric Measures

	Measure of Agreement Kappa Value	Approx. Sig.	N of Valid Cases
UML1_c1 * UML1_c2	.919	.000	93
UML2_c1 * UML2_c2	.919	.000	93
UML3_c1 * UML3_c2	.911	.000	44
CODING1_c1 * CODING1_c2	.914	.000	93
CODING2_c1 * CODING2_c2	.933	.000	93
CODING3_c1 * CODING3_c2	.940	.000	44

Disagreements in the reliability coding were resolved by the two judges as an agreement was reached after a second round of evaluation and thus the two judges came to the same conclusion for all the participants.

D. Quantitative Analysis

As presented in the previous chapter, the quantitative analysis was conducted on the tests which are primarily divided into three specific pieces: UML (design), Coding (problem-solving), and bonus points. For the scope of this study, we limit the analysis of each test on the first two pieces, UML and Coding.

The statistical data analysis of the exercises focuses on the following research question hypotheses:

1) How did the group perform on the three tests overall?
2) Are there any differences between how students scored on specific parts of test1 compared to test 2 (and test 2 compared to test 3)?
3) Are there any relationships between any of the four factors (major, gender, ethnicity, and prior programming experience) and student performance scores for each specific pieces of each test?

4) Are there any relationships between how students scored on specific parts of test 1 compared to test 2 (and test 2 compared to test 3)?

To answer the above research hypotheses, I used the analytical software SPSS to test the data sets for normality; and to conduct dependent t-test, multiple analysis of variance test (MONAVA), and the correlation test. Furthermore, since CSE 100 participants were bounded to test 1 and test 2, I have three datasets. The first dataset analyzed the three tests for CSE 110 (Appendix K), the second dataset analyzed the first two tests for CSE 100 (Appendix L), and the third dataset is a combination of the two first datasets for an analysis of the overall CS1 student performance (Appendix M). The data sets included the percentage scores for each UML exercise (UML1, UML2, and UML3) and each Coding exercise (CODING1, CODING2, and CODING3).

1) Testing for Normality: A test for normality is a prerequisite for many statistical tests where normal distribution of data is an underlying assumption in parametric testing. There are two main methods to assess normality, graphically and numerically. Numerical tests have the advantage of making an objective judgment of normality but are disadvantaged by sometimes not being sensitive enough at low sample sizes or overly sensitive to large sample sizes. Graphical interpretation has the advantage of allowing good judgment to assess normality in situations when numerical tests might be over or under sensitive. As such, since my data sets are of small size samples (< 100 samples), I used the *normal Q-Q Plot* as a graphical representation of normality. Based on the plots, the data points

were close enough to the diagonal line to conclude that the three data sets can be considered as normal distributions.

2) Dependent t-Test: The dependent t-test compares the means between two related groups on the same continuous variable. In this study, a group of freshman students enrolled in one of the introductory courses were selected from the student population to investigate whether design (UML) and problem-solving (Coding) improve their performance in the course. In order to test whether these types of exercises are useful measures that can show an improvement in performance, the sample groups were first tested for their performance in test 1, and then measured again (test 2 and test 3) before the end of the semester.

Using SPSS paired-samples t-test procedure, from the two tables - *Paired Sample Statistics* and *Paired Samples Test* - the first data set (i.e. CSE 110 participants for all three tests) showed the following score improvement:

> Due to the significance level value of UML1-UML2 and CODING2-CODING3 ($p > 0.05$), there was no statistically significant score improvement between UML1 and UML2 and CODING2 and CODING3.
>
> $t(43) = -3.513$, $p < 0.05$. There was a statistically significant paired difference for UML2 (0.66 ± 0.24 pt) - UML3 (0.77 ± 0.20 pt);
>
> $t(43) = -3.218$, $p < 0.05$. There was a statistically significant paired difference for UML1 (0.62 ± 0.32 pt) - UML3 (0.77 ± 0.20 pt);
>
> $t(43) = -3.707$, $p < 0.05$. There was a statistically significant paired difference for CODING1 ($0.58 \pm .34$ pt) - CODING2 (0.79 ± 0.26 pt);
>
> $t(43) = -3.605$, $p < 0.05$. There was a statistically significant paired difference for CODING1 ($0.58 \pm .34$ pt) - CODING3 (0.78 ± 0.23 pt);

Based on the first dataset results shown above (means and direction of the t-value), a summary of the overall score performance between each specific piece of each test is presented in Table 8 below.

Table 8 – Dataset 1 Overall Score Performance between Exercises

		Significant Score Performance Difference	If Yes then Positive or Negative
Pair 1	**UML1 - UML2**	No	
Pair 2	**UML2 - UML3**	Yes	Positive
Pair 3	**UML1 - UML3**	Yes	Positive
Pair 4	**CODING1 - CODING2**	Yes	Positive
Pair 5	**CODING2 - CODING3**	No	
Pair 6	**CODING1 - CODING3**	Yes	Positive

From this table, I concluded that overall CSE 110 students' problem design scores improved significantly. Even though, CSE 110 students' problem solving scores improved significantly, the scores between test2 and test3 were similar with a mean difference of 0.01.

The second dataset (i.e. CSE 100 participants for the first two tests) showed the following score improvement:

> Due to the significance level value of UML1-UML2 ($p > 0.05$), there was no statistically significant score improvement between UML1 and UML2.
>
> $t(48) = -3.020$, $p < 0.05$. There was a statistically paired difference for CODING1 (0.63 ± 0.27 pt) - CODING2 (0.72 ± 0.27 pt);

Based on the second data set results shown above, I concluded that overall CSE 100 students' problem design scores improved (but not significantly) with a mean difference of 0.05, and thus this shows that students scored were about the same.

However, CSE 100 students' problem solving scores improved significantly, and thus this shows that students scored better.

The third dataset (i.e. CS1 participants for the first two tests) showed the following score improvement:

> $t(92) = -4.707$, $P < 0.05$. There was a statistically significant paired difference for CODING1 (0.61 ± .30 pt) - CODING2 (0.76 ± 0.27 pt);

Based on the third data set results shown above, similarly to dataset2, I concluded that overall CS1 students' problem design scores improved (but not significantly) with a mean difference of 0.04, and thus this shows that students scored were about the same. Whereas, CSE 100 students' problem solving scores improved significantly, and thus this shows that students scored better as they progressed through the semester. However, it is important to keep in mind that in this dataset there are more CSE 100 students than CSE 110 so this difference of 6 students may have played a role in the overall CS1 student performance.

3) Multiple Analysis of Variance (MANOVA): MANOVA is used to answer the research question: "Are there any relationships between the factors and all (or each of) the dependent variables?" In this section, I only presented partial multivariate tests tables with relevant information (i.e. $p < 0.05$).

In the first dataset, there was a significant relationship between major and Coding3 and ethnicity and Coding3. Also, the combined factors major and ethnicity were found significant with Coding3 (See Table 9).

Table 9 – Dataset 1 Tests of Between-Subjects Effects

Source	Dependent Variable	Type III Sum of Squares	Df	Mean Square	F	Sig.
MAJOR	UML1	.040	1	.040	.392	.535
	UML2	.009	1	.009	.136	.715
	UML3	.133	1	.133	3.387	.074
	CODING1	.005	1	.005	.049	.826
	CODING2	.094	1	.094	1.808	.188
	CODING3	.422	1	.422	11.210	.002
ETHNICITY	UML1	.073	1	.073	.704	.407
	UML2	.013	1	.013	.186	.669
	UML3	.162	1	.162	4.123	.050
	CODING1	.071	1	.071	.657	.423
	CODING2	.087	1	.087	1.666	.205
	CODING3	.177	1	.177	4.707	.037
MAJOR * ETHNICITY	UML1	.134	1	.134	1.300	.262
	UML2	.005	1	.005	.067	.798
	UML3	.222	1	.222	5.654	.023
	CODING1	.005	1	.005	.045	.833
	CODING2	.011	1	.011	.210	.650
	CODING3	.260	1	.260	6.902	.013

In the second dataset, no significant relationships between the factors and the specific pieces of the tests were found.

In the third dataset, there was a significant relationship between course and both UML1 and UML2. Also, the combined factors major and gender were found significant with all three Coding tests (See Table 10).

Table 10 – Dataset 3 Tests of Between-Subjects Effects

Source	Dependent Variable	Type III Sum of Squares	df	Mean Square	F	Sig.
COURSE	UML1	.118	1	.118	1.657	.202
	UML2	.309	1	.309	6.411	.014
	CODING1	.006	1	.006	.065	.799
	CODING2	.018	1	.018	.277	.600
MAJOR * GENDER	UML1	.062	1	.062	.877	.352
	UML2	.027	1	.027	.565	.455
	CODING1	.526	1	.526	6.020	.017
	CODING2	.274	1	.274	4.165	.045

CSE 110 dataset shows that computer science male students performed higher in the third coding exercise than the rest of the students. Furthermore CS1 dataset shows that CSE 100 students performed higher in the second UML exercise than CSE 110 students. Also, computer science male students performed higher in both the second and third coding exercises than the rest of the students.

4) Pearson's Product-Moment Correlation: This correlation test aims at comparing the scores obtained in UML tests and CODING tests to determine if there is a relationship. The research question is: "Does a student who performed well in UML1 also performed well in Coding1?" The Pearson product-moment correlation was run to determine the relationship between UML performance test scores and Coding performance test scores. Since I am performing several correlations, I must consider a corrected significance level to minimize the chances of making a Type I error. I used the Bonferroni approach, which required dividing .05 by the number of computed correlations. I used .0056 (0.05/9) for the

first dataset and .0125 (0.05/4) for the second and third datasets. From the first dataset, students scored similarly on both UML1 and Coding2, on UML2 and Coding2 and Coding3, and on UML3 and Coding3 ($p < 0.0056$) (See Table 10). From the second dataset, students scored similarly on both UML1 and Coding1, and on UML2 and Coding1 and Coding2 ($p < 0.0125$) (See Table 12). Whereas from the third dataset, students scored similarly on both UML2 and Coding2, and on UML1 and Coding1 and Coding2 ($p < 0.0125$) (See Table 13)

Table 11 – Dataset 1 Correlation between UML and Coding

		UML1	UML2	UML3	**CODING1**	**CODING2**	**CODING3**
UML1	Pearson Correlation				.091	.423**	.395**
	Sig. (2-tailed)				.557	.004	.008
UML2	Pearson Correlation				.093	.510**	.485**
	Sig. (2-tailed)				.549	.000	.001
UML3	Pearson Correlation				.151	.377*	.788**
	Sig. (2-tailed)				.329	.012	.000

**. Correlation is significant at the 0.01 level (2-tailed).

*. Correlation is significant at the 0.05 level (2-tailed).

Table 12 – Dataset 2 Correlation between UML and Coding

		UML1	UML2	**CODING1**	**CODING2**
UML1	Pearson Correlation			.548**	.274
	Sig. (2-tailed)			.000	.057
UML2	Pearson Correlation			.456**	.481**
	Sig. (2-tailed)			.001	.000

**. Correlation is significant at the 0.01 level (2-tailed).

*. Correlation is significant at the 0.05 level (2-tailed).

Table 13 – Dataset 3 Correlation between UML and Coding

		UML1	UML2	CODING1	CODING2
UML1	Pearson Correlation			.277**	.289**
	Sig. (2-tailed)			.007	.005
UML2	Pearson Correlation			.241*	.389**
	Sig. (2-tailed)			.020	.000

**. Correlation is significant at the 0.01 level (2-tailed).

*. Correlation is significant at the 0.05 level (2-tailed).

E. Qualitative Analysis

In this section I provide a sample of student performance profiles and identify the computational thinking errors that students made, if any. To do so, first a scoring guide was developed to assess the quality of student performance in relation to design, coding, and troublesome concepts. The scoring guides for the exercises indicate specific criteria to describe a range of possible student responses and a consistent set of guidelines to grade student work.

I describe below the scoring guides for both problem design (UML exercises) and problem solving (Coding exercises). Since the UML class diagram was assigned 4 points by the instructor, its scoring guide is divided into five categories - excellent, good, average, marginal, and unsatisfactory. The Coding exercises was assigned 6 points by the instructor and thus its scoring guide is divided into seven categories - excellent, very good, good, average, poor, very poor, and unsatisfactory. In addition, I have included samples of student responses for common mistakes found. The three tests given to the students can be found in Appendices C to F.

1) Analysis of Problem Design: The problem-design scoring guide was used as an assessment tool to judge the quality of student performance in relation to UML content standards. The scoring criteria were primarily generated based on the following concepts: classes, data members, methods, connections, and syntax (Table 14).

Table 14 – Problem Design (UML) Score & Description

SCORING	DESCRIPTION	
EXCELLENT *S = 4*	+	Classes were named with descriptive names
	+	All data members are well-described and include their data types
	+	All methods including constructors are well-described and include their parameters' data type and return types
	+	All connections are indicated correctly
	+	UML class diagram format is correct
GOOD *4 < S ≤ 3*	+	Classes were named with descriptive names
	+/-	Most data members are well-described and include their data types
	+/-	Most methods including constructors are well-described and include their parameters' data type and return types
	+	All connections are indicated correctly
	+	UML class diagram format is correct
AVERAGE *3 < S ≤ 2*	+	Classes were named with descriptive names
	-	Few or no data members are well-described and include their data types
	+/-	Most methods are well-described and include their parameters' data type and return types. Constructors may or may not be included
	+	All connections are indicated correctly
	+	UML class diagram format is correct
MARGINAL *2 <S ≤ 1*	+/-	Classes may or may not be named (with descriptive names)
	-	Few or no data members are included with their data types
	-	Few or no methods are included with their parameters' data type and return types. Constructors may or may not be included
	+/-	Connections may not be indicated correctly
	+	UML class diagram format is correct
UNSATISFACTORY *1 < S ≤ 0*	+/-	Classes may or may not be named (with descriptive names)
	-	Few or no data members are included with their data types
	-	Few or no methods are included with their parameters' data type or return types. Constructors are not included
	-	Connections are not indicated
	+/-	UML class diagram format may not be correct

The student performance in problem design (UML) was measured three times over the semester. The first UML exercise was given seven weeks after school began. Students were knowledgeable about classes, data members, and objects. The second UML exercise was given five weeks after the first UML exercise was given. Students were knowledgeable about arrays of objects, conditional statements, and repetition. The last UML exercise was given three weeks after the second UML. Students were knowledgeable about abstraction, inheritance, and polymorphism.

Based on the student performance in UML throughout the semester, I have included samples of student performance below indicated the common mistakes found frequently. Figure 8 shows an example of an excellent response for a problem design. The student listed the relevant data members with their respective data types. In addition, the student listed all the relevant methods, including the constructor, with their respective parameters' data type and return types. This example shows that the student was able to abstract the relevant information from the given problem as well as organized the information in such a manner that he/she understood the concepts of class, data members, and methods. Furthermore, he/she specified the return types of each method in such a manner that it is clear that he/she understood how the outputs will be accessed, particularly the method getBalance().

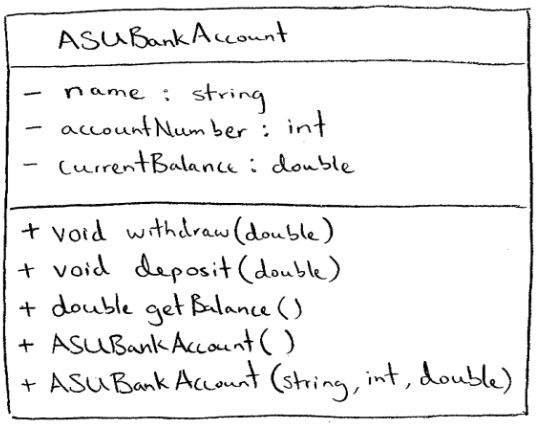


Figure 8 – Sample for UML Score Excellent

The next example, Figure 9, shows an overall good response from a student. The student listed the relevant data members with their respective data types. However, the data type for the array is not consistent with the data type for the final average. Furthermore, the relevant methods are listed including the constructors. However, the student did not include the parameters' data type, and thus failed to abstract all the relevant details from the given problem.

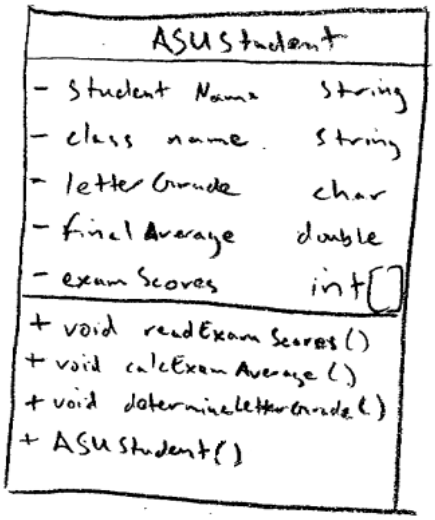


Figure 9 – Sample for UML Score Good

Figure 10 shows another example where the student's problem design failed to abstract the relevant information form the given problem. First, the majority of the data members are not indicated except for one, balance. Secondly, most of the methods are included except for the constructor. Last but not least, this response included the incorrect return type for the method viewBalance(). As a result, this student's response was categorized as average.

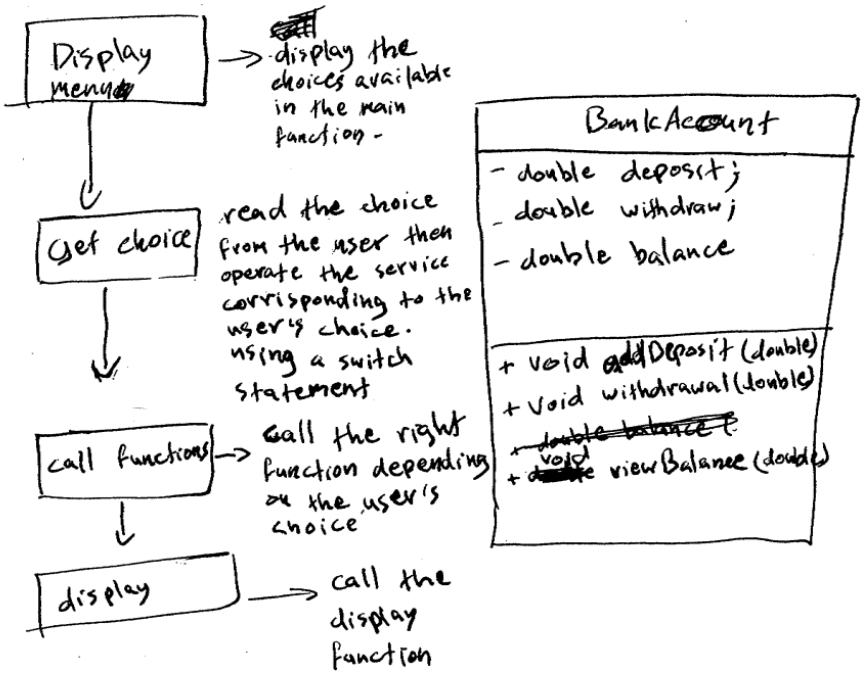


Figure 10 – Sample for UML Score Average

Figure 11 is similar to the mistakes made in Figure 10 in terms of the abstraction of the relevant data members, but also the student failed to correctly define the data. For example, the syntax to define an array is incorrect and some of the data types for the data members are incorrect such as letter grade. Furthermore, the methods do not include their parameters' data type and their return types. Last but not least, the constructor was omitted. Thus, this response was scored as marginal.

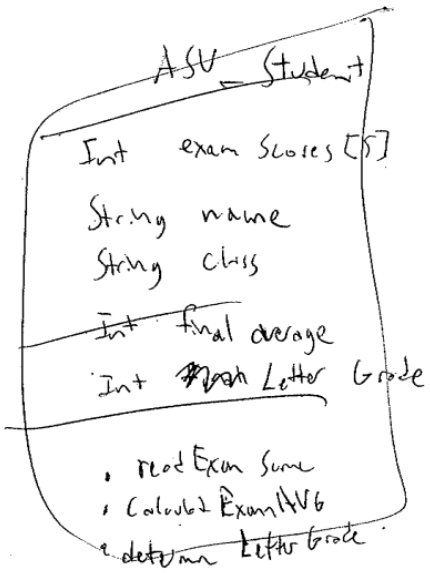


Figure 11 – Sample for UML Score Marginal

Figure 12 is an example of a student who was unsuccessful at the abstraction of the information as well as the correctness of his/her problem design. This response is omitting the constructors as well as the return types of the toString() method. The data members are not properly defined. It seems that they are defined as methods. In addition, the relationship for inheritance between the two classes is incorrect. The arrow should be pointing in the other direction.

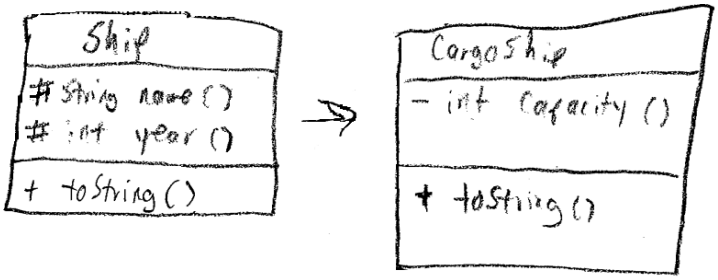


Figure 12 – Sample for UML Score Unsatisfactory

2) Problem Design Score Distribution: The scoring distribution for the three UML exercises is indicated with respect to their score range (Table 13). Note that the 'good' score percentage is the only score that consistently increased throughout the study. Also, the 'unsatisfactory' score is the only score that consistently decreased throughout the study and at the end of the study no student response was rated 'unsatisfactory.' The two primary reasons for the 'excellent' score decreasing for UML3 are: (1) this exercise was only assessed for CSE 110 and (2) many students identified one constructor instead of two constructors. Furthermore, when combining the top two scores for each UML exercise, about two-third of the CS1 students received an "excellent" or "good" grade, equivalent to the letter grade A or B, in UML1 and UML2. Furthermore, eight students out of nine students performed above 'average' in UML3.

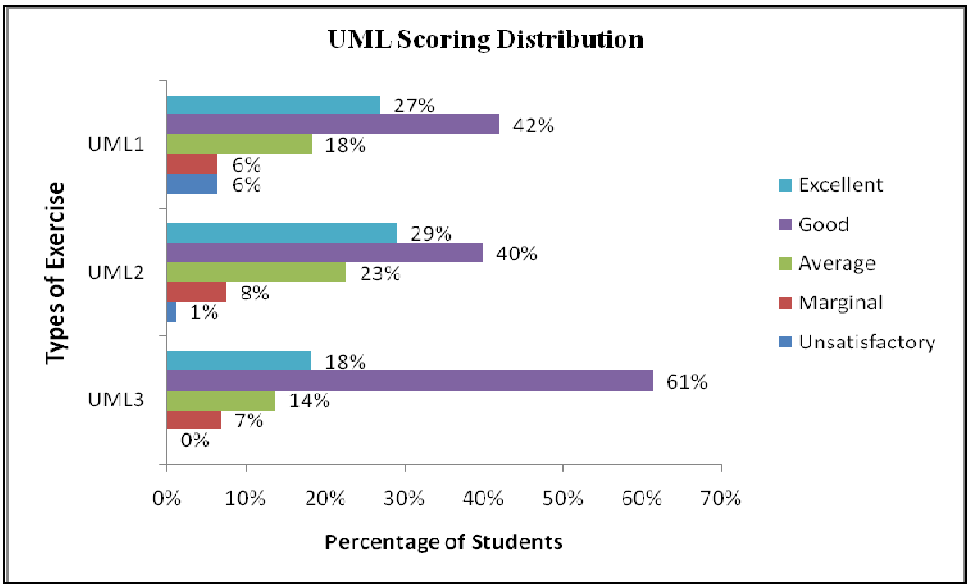


Figure 13: Scoring Distribution for UML for CS1 Courses

In addition, the UML1 and UML2 percentage scores with respect to the course variable shows that the majority of the students performing below average are enrolled in CSE 110 (Table 15). However, the study included 44 CSE 110 participants compared to 49 CSE 100 participants so this difference in 6 students may have played some role in this difference between the two courses. Nevertheless, the difference is high so a closer look into the types of mistakes found across the two courses may be useful.

Table 15 – Breakdown of UML Scoring Distribution by Course

	UML1		*UML2*	
	CSE 110	CSE 100	CSE 110	CSE 100
Excellent	36%	64%	22%	78%
Good	36%	64%	41%	59%
Average	76%	24%	71%	29%
Marginal	50%	50%	100%	0%
Unsatisfactory	83%	17%	100%	0%

UML is a tool used to model/design a problem at the abstraction level in terms of the relevant information from a given problem. As a result, the assessment of the types of mistakes made by the students during the semester is based on the following two computational thinking criteria: abstraction and linguistics (Table 3). Table 16 illustrates a more descriptive disparity among the two classes. In the first test, both CSE 100 and CSE 110 students scored similarly for the overall mistakes made in abstraction and linguistics when modeling the problem. However CSE 100 students' mistakes were found more than twice as high as their CS 110 peers' mistakes for data members and methods. In contrast, CSE 110 students' mistakes were found more than twice as high as their CSE 110 peers'

mistakes for returned types and more than one third as high as their CSE 110 peers' mistakes for parameter's data types. As the semester progressed, overall the students' mistakes in abstraction reduced by one third when they took the second test. While CSE 100 students' mistakes in abstraction drastically dropped, CSE 110 students still struggled with some of the concepts such as data members' data types. This is primarily due to the introduction of arrays in their learning. Most students did not define an array for the exam scores but rather a single variable to characterize all the exam scores. Students either did not know how to define an array in their problem design or they defined the array improperly; thus, the mistakes in relation to linguistics skill went up. Some students who recognized they had to define an array, but did not know how to, defined five data members instead, given that the array was limited to five entries. Their alternative design of the array was correct even though they did not use the concept of the array in their problem design. Toward the end of the semester, the last test included two classes. The CSE 110 students failed to identify the second constructor, and therefore the count for the mistakes related to the omission of the constructors almost went back to the count from the beginning of the semester. Otherwise, the count would have been close to zero percent. This shows that the students have not fully grasped the concept of a 'constructor.' If students included the constructor then they did not include the parameters' data type for it.

Table 16 – Assessment of UML Computational Thinking Skills

	UML1			*UML2*			*UML3*
Computational Thinking Mistakes	CSE 110	CSE 100	All	CSE 110	CSE 100	All	CSE 110
ABSTRACTION	82%	78%	80%	77%	41%	58%	66%
Relevant classes are omitted/incorrect	0%	0%	0%	0%	0%	0%	0%
Relationships between the classes are incorrect	0%	0%	0%	0%	0%	0%	7%
Relevant data members are omitted/ incorrect	7%	12%	10%	0%	2%	1%	0%
Data members' data types are omitted/ incorrect	14%	14%	14%	34%	14%	24%	0%
Relevant methods, excluding constructors, are omitted	5%	12%	9%	5%	0%	4%	0%
Constructors are omitted	48%	31%	39%	34%	8%	21%	41%
Parameters' data types are omitted/incorrect	68%	45%	56%	27%	8%	17%	34%
Return types are omitted/ incorrect	61%	29%	44%	41%	6%	24%	27%
LINGUISTICS	*25%*	*20%*	*23%*	*27%*	*29%*	*37%*	*14%*
Improper semantics	0%	0%	0%	0%	0%	0%	0%
Improper syntax	25%	20%	23%	27%	29%	37%	14%

Overall, the CS1 students demonstrated computational thinking skills as they progressed through the semester. Students have a better grasp of abstraction in terms of data members, methods, data types, and return types. Even if the number of mistakes made by the students in linguistics did not reduce as the students progressed in the course, their ability in linguistics is consistent.

3) Analysis of Problem Solving: The problem-solving scoring guide was used to judge the quality of student performance in relation to program content standards. The scoring criteria were generated based on program sequence, inclusion of the classes, methods with constructors, reasoning within methods, and syntax. Table 17 describes the scoring for each student performance profile.

Table 17 – Program Solving (Code) Score & Description

SCORING	DESCRIPTION	
EXCELLENT *S = 6*	+ + + + + +	Sequence of the program is correct (class, data members, and methods) Classes include data members Constructors are included and initialized Methods are included with their parameters passing, data types, and return types Logic is performed correctly (arithmetic, conditional statements and repetition statements) No syntax error
VERY GOOD *6 < S ≤ 5*	+ + + + +/- +/-	Sequence of the program is correct Classes include data members Constructors are included and initialized Methods are included with their parameters passing, data types, and return types. Most logic is performed correctly Few (minor) syntax errors
GOOD *5 < S ≤ 4*	+ + +/- + +/- +/-	Sequence of the program is correct Classes include data members and methods Constructors may not be included or initialized Methods are included with their parameters passing, data types, and return types. Most logic is performed correctly Few (minor) syntax errors
AVERAGE *4 <S ≤ 3*	+ + - + +/- -	Sequence of the program may is correct Classes include data members Constructors are not be initialized properly Methods are included (excluding constructors) Most logic is performed correctly Some syntax errors
POOR *3 <S ≤ 2*	+ +/- - +/- - -	Sequence of the program is correct Classes include some data members Constructors are not be initialized properly Most methods are included (excluding constructors) Few logic is performed correctly Some syntax errors
VERY POOR *2 <S ≤ 1*	+/- +/- - - - -	Sequence of the program may not be correct Few classes with data members are included Constructors are not included/initialized Few methods are implemented (excluding constructors) Logic is performed incorrectly Many syntax errors
UNSATISFACTORY *1 < S ≤ 0*	+/- - - - - -	Sequence of the program may not be correct No classes with data members are included Constructors are not included/initialized No methods are implemented Logic is omitted Syntax is incorrect

Similar to student performance in problem design, the student performance in problem-solving (Coding) was measured three times over the semester. The first Coding exercise was given seven weeks after school began. Students were knowledgeable about classes, data members, and objects. The second Coding exercise was given five weeks after the first Coding exercise was given. Students were knowledgeable about arrays of objects, conditional statements, and repetition. The last Coding exercise was given three weeks after the second Coding exercise. Students were knowledgeable about abstraction, inheritance, and polymorphism.

Based on the student performance in coding throughout the semester, I have included samples of student performance below indicated the common mistakes found frequently. But first, Figure 14 shows an example of an excellent response for solving a problem. The student properly defined the relevant data members with their respective data types. In addition, the student defined all the relevant methods, including the constructor, with their respective parameters' data type and return types. This example shows that the student was able to abstract the relevant information from the given problem as well as organized the information in such a manner that he/she understood the concept of abstraction. Furthermore, he/she used the appropriate controls within the methods when necessary such as the conditional statement in the method withdraw().

```
class ASUBankAccount
{
    private:
        String name;
        int accountNumber;
        double currentBalance;
    public:
        ASUBankAccount();
        ASUBankAccount(string, int, double);
        void withdraw(double);
        void deposit(double);
        double getBalance();
};
ASUBankAccount::ASUBankAccount()        // Default Constructor
{
    name = "";
    accountNumber = 0;
    currentBalance = 0;
}
ASUBankAccount::ASUBankAccount(string acctName, int acctNumber, double balance)
{
    name = acctName;
    accountNumber = acctNumber;
    currentBalance = balance;
}
void ASUBankAccount::withdraw(double withdrawal)
{
    if (withdrawal <= currentBalance)
        currentBalance = currentBalance - withdrawal;
    else
        cout << "You do not have sufficient funds!" << endl;
}
void ASUBankAccount::deposit(double dep)
{
    currentBalance = currentBalance + dep;
}
double ASUBankAccount::getBalance()
{
    return currentBalance;
}
```

Figure 14 – Sample for Coding Score Excellent

The next example, Figure 15, shows an overall very good response from a student. The student defined the relevant data members with their respective data types. However, the data type for the array is not consistent with the data type for the final average. As a result, a minor syntax error will result from it in the method calcExamAverage(). However, the student failed to use a for-loop when initializing the exam scores. This solution is correct, but in terms of scalability,

this approach will not be suitable when revising the size of the array for the exam scores. In the method determineLetterGrade(), the conditional statement is improperly used. After the first "if", the students should have used "else if" rather then "if" for the next two conditional statement. Plus, the variable 'finalAverage' is misused.

```
public class ASUStudent
{
   private String studentName;
   private String className;
   private char letterGrade;
   private int[] examScores = new int[5];
   private double finalAverage;

   public ASUStudent(String name1, String class1)
   {
      studentName = name1;
      className = class1;
      letterGrade = 'F';
      examScores[0] = 0, examScores[1]=0, examScores[2]=0, examScores[3]=0, examScores[4] = 0;
      finalAverage = 0.0;
   }

   public void readExamScores()
   {
      for(int i = 0; i < examScores.length; i++)
         System.out.println("Please enter your exam scores");
         examScores[i] = scan.nextInt();
   }
   public void calcExamAverage()
   {
      finalAverage = (examScores[0] + examScores[1] + examScores[2] + examScores[3] + examScores[4]) / 5;
   }
   public      determineLetterGrade()
   {
      if(finalAverage >= 90)
         letterGrade = 'A';
      if(finalAverage >= 80 && < 90)
         letterGrade= 'B';
      if(finalAverage >= 70 && < 80)
         letterGrade = 'C';
      else
         letterGrade = 'F';
   }
}
```

Figure 15 – Sample for Coding Score Very Good

Figure 16 shows another example where the student's problem design failed to initialize the constructor Ship(). In addition, in the last method, the student should have used the 'super' for the variable 'name' and 'year' sine these variables are defined in the super class Ship(). As a result, this student's response was categorized as good.

```
Public class Ship()
{
    protected string name;
    Protected int year;
    Public Ship(-)
    {
    }

    Public string toString()
    {
       return name + "built in " + year;
    }
}

Public class CargoShip extends Ship( )
{
    private int maxCapacity;
    Public CargoShip(String name, int newCap, int newYear)
    {
        Super.name = name;
        super.year = newYear;
        maxCapacity = newCap;
    }
    Public String toString()
    {
       return name + "built in" + year + "has a" +
            "max capacity of " + maxCapacity;
    }
```

Figure 16 – Sample for Coding Score Good

Figure 17 is an example of an average score. The constructor doe not have its parameters passing and the two methods for toString() do not have their return types and a space should be included between the two variables to be displayed. In the last toString() method, there should be a dot after 'super'. Finally, the maxCapacity variable was not initialized in the constructor CargoShip().

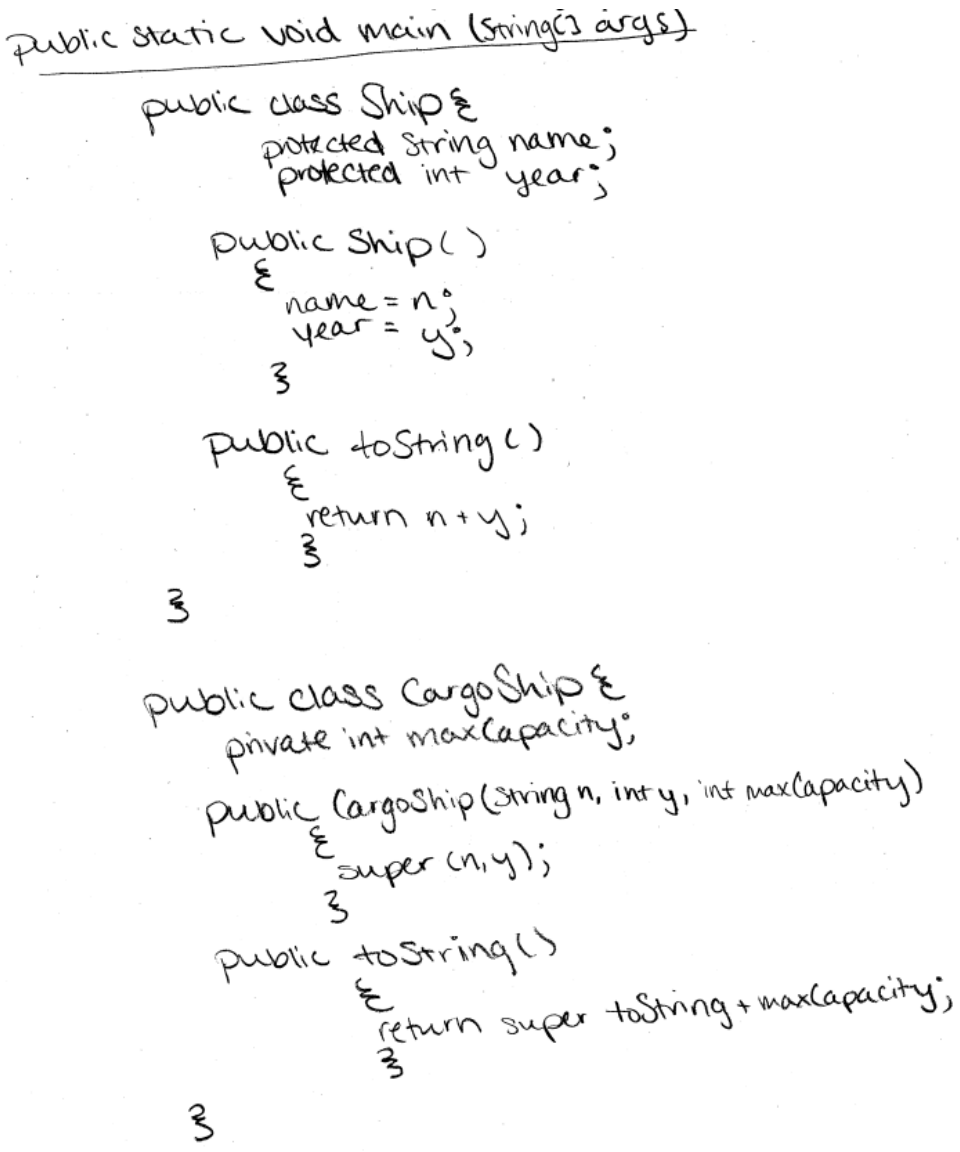


Figure 17 – Sample for Coding Score Average

Figure 18 is an example of a poor score. Some data members are not included and the constructor is not properly implemented. Also, the data types for the parameters passing are omitted and the conditional statement in the method withdraw() is not included. The last method has an incorrect return type.

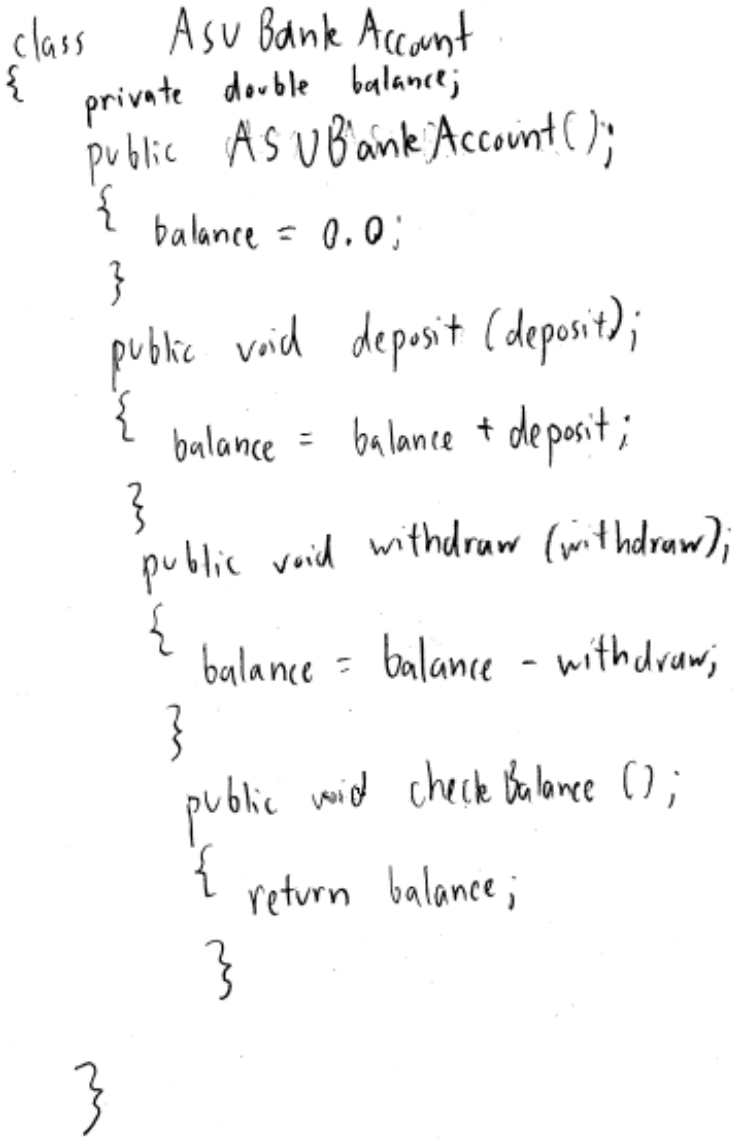


Figure 18 – Sample for Coding Score Poor

The student response below (Figure 19) has more syntax errors than the previous examples. The data members seemed to be defined as methods and the methods

do not have their return types and their parameters passing. All the methods are not implemented; and thus this is a very poor student response.

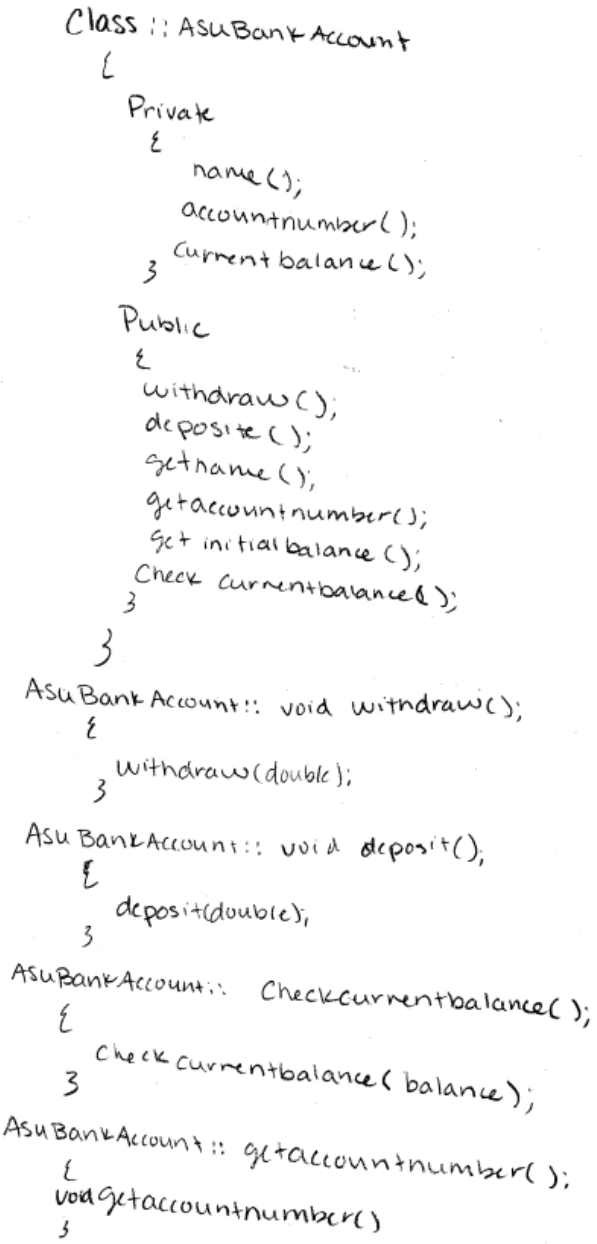
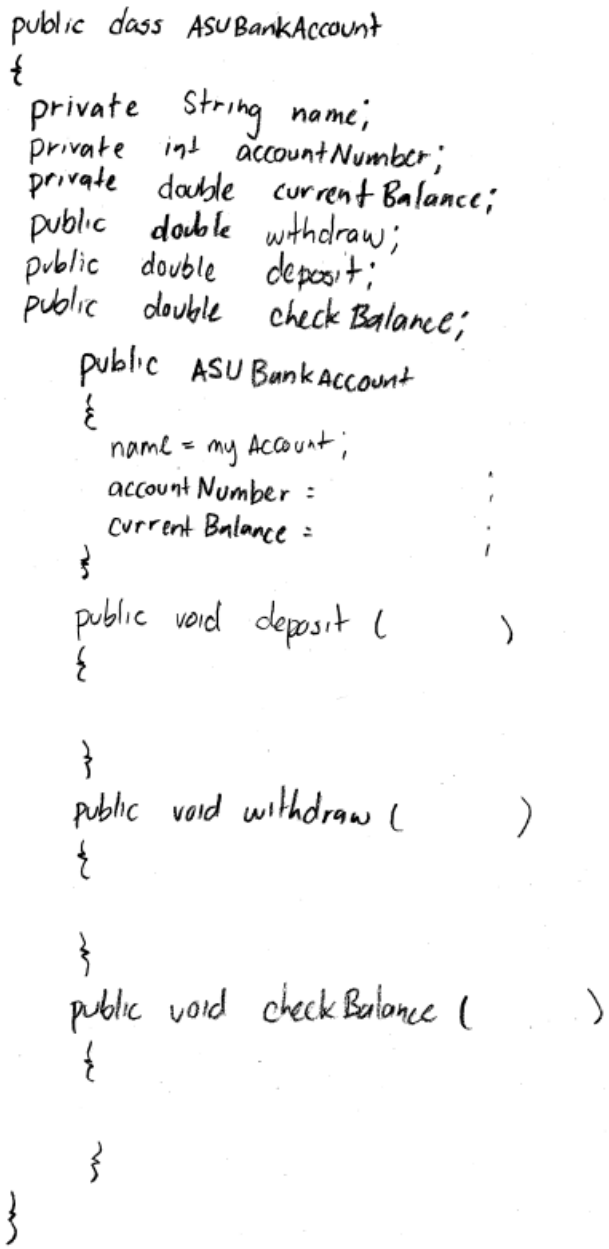


Figure 19 – Sample for Coding Score Very Poor

The last student response example clearly shows that the student defined the methods as variables at the beginning of the class and the constructor does not

have its parameters passing and is not properly initialized. The rest of the problem is not implemented. This response is unsatisfactory.



```
public dass ASUBankAccount
{
 private String name;
 private int accountNumber;
 private double currentBalance;
 public double withdraw;
 public double deposit;
 public double checkBalance;
    public ASUBankAccount
    {
      name = myAccount;
      accountNumber =          ;
      currentBalance =         ;
    }
    public void deposit (      )
    {

    }
    public void withdraw (      )
    {

    }
    public void checkBalance (      )
    {

     }
}
```

Figure 20 – Sample for Coding Score Unsatisfactory

4) *Problem Solving Score Distribution*: The scoring distribution for the three Coding exercises is indicated with respect to their score range in Table 21. Note

that the 'good' score percentage is the only score that consistently increased throughout the study. Also, the 'very poor' and 'poor' scores are the only scores that consistently decreased throughout the study and that at the end of the study the 'very poor' score indicated that no student response failed under that category. Furthermore, when combining the top two scores for each Coding exercise, about one-third of the CS1 students received an "excellent" or "good" grade, equivalent to the letter grade A or B, in Coding1 and two-third of CS1 students received an "excellent" or "good" grade, equivalent to the letter grade A or B in Coding2. Furthermore, nine students out of 10 students performed above 'average' in Coding3.

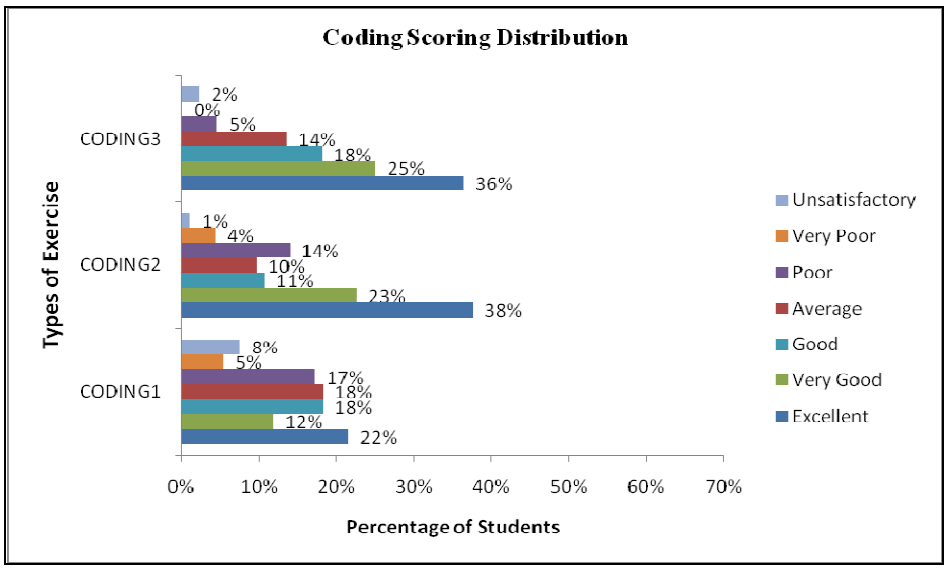


Figure 21: Scoring Distribution for Coding for CS1 courses

A closer look at Coding1 and Coding2 percentage scores with respect to the course variable shows that half of the students for both CSE 100 students and CSE 110 students performed above average for Coding1 (Table 18). However, the

CSE 110 students performed higher for Coding2. The difference is minimal but a closer look into the types of mistakes found across the two courses may be useful.

Table 18 – Breakdown of Coding Scoring Distribution by Course

	CODING1		*CODING2*	
	CSE 110	CSE 100	CSE 110	CSE 100
Excellent	18%	25%	43%	33%
Very Good	14%	10%	21%	25%
Good	20%	16%	16%	6%
Average	18%	18%	7%	12%
Poor	9%	25%	9%	18%
Very Poor	7%	4%	2%	6%
Unsatisfactory	14%	2%	2%	0%

Coding is a detailed approach used to problem solve which requires expertise in abstraction, algorithm, reasoning, linguistics and scalability. These abilities are part of the assessment of the types of mistakes made by the students during the semester based on computational thinking skills (Table 3). A more descriptive disparity among the two classes is highlighted in Table 19. In the first test CSE 100 students performed almost twice as worse as the CSE 110 students in abstraction, however the two classes performed similarly in algorithm, scalability, and linguistics. As the semester progressed, overall, the number of mistakes in abstraction reduced considerably for both courses in the second test. Despite the use of arrays in this test, students performed much better than in their problem design. Similar to problem design, in the third test, CSE 110 students failed to identify the second constructor, and therefore the count for the mistakes related to the omission of the constructors almost double their count from the beginning of the semester. Otherwise, the count would have been minimal.

Table 19 – Assessment of Coding Computational Thinking Skills

	CODING1			*CODING 2*			*CODING 3*
Computational Thinking Mistakes	CSE 110	CSE 100	Overall	CSE 110	CSE 100	Overall	CSE 110
ABSTRACTION	27%	51%	40%	16%	16%	16%	48%
Data members are omitted	9%	14%	12%	14%	10%	12%	12%
Constructors are omitted	18%	22%	20%	5%	14%	10%	30%
Methods are omitted	11%	12%	12%	5%	8%	6%	5%
Relationships between the classes are omitted	0%	0%	0%	0%	0%	0%	14%
ALGORITHM	39%	45%	42%	20%	33%	27%	20%
Program sequence order is incorrect	11%	12%	12%	5%	10%	8%	0%
Program does not do anything	16%	16%	16%	5%	10%	8%	2%
Program does not do the right thing	25%	31%	28%	2%	12%	8%	11%
Program does not compute the right answer (for at least one method)	30%	29%	29%	16%	14%	15%	14%
REASONING	61%	49%	55%	18%	29%	24%	14%
Control statements are incorrect	14%	6%	10%	18%	29%	24%	14%
Control statements are omitted	59%	43%	51%	0%	0%	0%	0%
SCALABILITY	30%	37%	33%	11%	27%	19%	23%
Program requires more lines of code than others for future expansion	30%	37%	33%	11%	27%	19%	23%
LINGUISTICS	59%	63%	62%	45%	43%	44%	34%
Improper semantics	0%	0%	0%	0%	0%	0%	0%
Improper syntax	59%	63%	62%	45%	43%	44%	34%

Overall, the CS1 students have acquired computational thinking skills as they progressed through the semester. The students have a better grasp of abstraction in terms of data members, methods, algorithm correctness, scalability, and linguistics. Even if the number of mistakes made by the students in reasoning did

not reduce as the students progressed in the course, their ability in reasoning was still satisfactory.

5) Analysis of Think-aloud/Interviews: The questionnaires and interviews were conducted to assess students' knowledge skills (i.e. CS core concept assessment), explanation skills, and skills that were related to computational thinking skills based on literature. As shown in Table 20, 16 participants for test 1 and 11 participants for test 2 were involved in this phase. Due to final semester examination schedule, no interviews were conducted for test 3. Thus, this phase was limited to test 1 and test 2.

Table 20 –Think-aloud/Interviews Participation Distribution

	Test 1		*Test 2*	
	CSE 100	CSE 110	CSE 100	CSE 110
Think-aloud/Interviews	6	10	5	6

Students' knowledge skills and explanation skills were assessed based on the criteria indicated in Table 21. From this table, overall students' knowledge skills increased from test 1 to test 2, with a very similar rate of change for both classes. Students' explanation skills have decreased between the two tests, primarily due to the concept of arrays (i.e. modeling); and thus students struggled with their explanations. Students' computational thinking skills were increasing and decreasing depending on the criteria of interest. In general, students performed well in the acquisition of knowledge over the two tests, but the introduction of new concepts such as arrays of objects showed some struggles in both their explanation skills and computational skills. Furthermore, students seemed to

memorize the materials in class rather than internalizing the information when it comes to the core computational concepts. This can be observed when one looked at the percentage difference between in-class examples versus other examples. In addition, participants who were able to successfully explain abstraction represented a little less than two-third of the participants in this study.

Table 21 – Questionnaires/Interviews Assessment

	Test 1			*Test 2*		
Identified Skills	CSE 100	CSE 110	All	CSE 100	CSE 110	All
KNOWLEDGE SKILLS						
Understanding the core concept	50%	78%	67%	67%	100%	88%
Recognizing prior knowledge needed to apply the core concept	50%	67%	60%	67%	80%	75%
Knowing the context of utility of the core concept	50%	67%	60%	67%	80%	88%
Sharing examples in relation to the core concept	100%	78%	87%	67%	80%	88%
Referring to/Remembering in-class material while solving the problem	67%	78%	73%	67%	60%	63%
EXPLANATION SKILLS						
Rephrasing the core concept with no technical words	50%	67%	60%	33%	80%	63%
Reiterating thoughts during the process of solving the problems	67%	78%	73%	33%	60%	50%
Having confidence when dealing with a problem	33%	89%	73%	33%	80%	63%
Communicating the goal or solution	67%	89%	80%	67%	80%	75%
COMPUTATIONAL SKILLS						
Logically organizing and analyzing data	67%	89%	80%	67%	100%	88%
Representing data through abstractions	50%	67%	60%	33%	100%	75%
Automating solutions through algorithmic thinking	50%	78%	67%	33%	80%	63%
Analyzing and implementing possible solution with the goal of achieving the most efficient and effective combination of steps and resources	33%	56%	47%	0%	20%	13%

I coded the interview transcripts to illustrate students' skills broadly categorized as knowledge, explanation, and computational skills (as in Table 21).

The interview text in bold print is representative of the specific skill that is noted in square brackets immediately following the text.

Excerpt:

> So, arrays **can store data, either primitive data structures or even objects themselves** [understanding the core concept and recognizing prior knowledge to apply the core concept], and they're **useful because they can refer to multiple since they have like indices** [knowing the context of utility of the core concept] and so forth. They can actually **store a lot of information** [understanding the core concept], which **prevents the programmer from having to use repetitive means to declare all the variables** [knowing the context of utility of the core concept] for a program.

This participant was able to give a basic definition of the core concept, arrays of objects, which could be characterized as a response at the level of CS1. The participant did not use any technical words. The participant explained the core concept in his own words. I interpreted this explanation to indicate that the participant has understood the core concept in its technicality and he was also able to explain the core concept to others (both majors and non-majors). Also, the participant's response included the utility of arrays of objects, which indicates that the participant understood the context and modeling of the core concept.

Excerpt:

> **Objects are just a subgroup of classes. So they're smaller** [understanding of core concept and confidence]**.** You know, the very vague generalized section is the class. And then these are types of that section like you know you

could say, **you can have a car and then the object could be the type of car, the year of the car, the make of the car and so forth** [sharing example in relation to the core concept]**.**

This participant was not able to provide a clear definition of the core concept, classes and objects, but rather was seeking for approval of his response. I interpreted that this participant was uncertain about the idea of "objects" and thus showed his lack of confidence. However, the participant was able to recall the 'car' example presented by the professor in-class from the previous lecture. The participant has an initial understanding of the core concept but his response still presents some missing information, which prevented him from providing a more precise definition of the core concept.

Excerpt:

Well, in this particular exercise, **I had to create a class that was going to store information such as the student name, the class name of the student** [reiterating thought process]**. I had to do some computation on the scores to figure out what the student's final score is** [communicating to others the goal]**.** So, the **first thing I did was I read the problem specifications** [reiterating thought process] to **underline and list all the attributes and all the operations or the methods that would be performed** [reiterating thought process, logically organizing/analyzing data, and representing data through abstractions]**.** Also, I had **to remember that I had an array as an attribute** [understanding the core concept] and **so I had to perform a slightly different series of operations on that array** [remembering in-class material]**.**

I had **to remember the syntax for initializing an array** [remembering in-class material] with a certain number of elements in the array, one-dimensional of course of this problem. And, oh, yeah. Well, **when I had to add a method to get the highest score, we had learned an algorithm we could implement for finding maximum scores** [referring to in-class material]**.** So, I implemented the algorithm **by creating a variable that would store the highest score in the element zero of the exam score array and then it would actually go through and compare it with the other scores. And if I found one that was actually higher, if it founded an element there with a value that was greater than the initial highest score, it would replace that variable with the element** [reiterating thought process and automating solutions through algorithmic thinking] from the – whichever exam score element had been higher. I was able to approach it that way and **it's very efficient algorithm I think, well, that I know of to find the maximum in arrays** [analyzing solution]**.**

This participant's response showcased knowledge, explanation and computational skills - some of them were explicit and others were implicit. The participant reiterated his thought process from reading the given problem to his optimal solution. The participant was able to determine that the modeling of his solution needed an array and then was able to recall the material learned in class to apply it to the given problem. Furthermore, the participant explained his solution in a simple manner which showed the confidence and ease of the participant to implement a solution. Last but not least, the participant even thought about

efficiency even though it could only be based on his knowledge acquired so far. This indicates that participant had algorithm efficiency in mind that is a bit advanced compared to the rest of his peers who participated in this study.

Excerpt:

> **I just underlined the important things that are probably going to be either a variable or a method or the actual name of the object itself** [logically organizing/analyzing data and representing data through abstraction]**.** Things like that. Then I went ahead and did my best to put that into a UML, which is theoretically just like a code list of programming. Hmmm **the problem asked for the average score grade and then the final score letter grade** [communicating to others the goal]**.** Even though they didn't explicitly mention an array for the exam score, **I figured that an array would save me a lot of trouble with storing the scores** [knowing the context of utility of the core concept]**.** To calculate the exam average, **I just summed them all up with a "for loop" and then divided by the total of exams** [reiterating the thought process and automating solution]**.** And then I ran out of space so I went over here. And this is just **a series of if-statements asking if it's bigger than 90. No, is it bigger than 80? No. Is it bigger than 70? No. Fine then he gets an F. And then it returns that score** [reiterating the thought process]**.** I don't know if they wanted me to return it or not, but I figure because you can always have it return, I might as well give them the option to make their life a little easier.

This participant was able to abstract the data from the given problem and modeled his solution using an array. The participant understood the advantage of using an array over multiple variables and thus reiterated his thought process for the array using logic to determine the solution for the problem. The participant explained the thought process behind the if-statements to determine the final letter grade. This showed that the participant was confident in his algorithm thinking.

Excerpt:

> **Because we kind of were taught that we should look at methods as sort of the actions and the variables are the – basically variables as the nouns in a problem statement** [referring to the material in-class]**.** I was able **to deduce** that **name, account number and current balance were attributes to variables that held a string value, an integer value, and a double value** [understanding the core concept, and logically organizing/analyzing data]**.** And that the verbs that you wanted to deposit or withdraw or check current balance were all methods. And by having that kind of – by being able **to compare then in that way, it was easier to take the problem statement, decompose it into its component parts** [confidence] and then create the UML from there. For the coding section, first we have the basic standard declaration that its enumeration is public. **It obviously is a class and we give the class name so that it encapsulates the entire class under that name. We go on just to state that the variables in this class are going to be accessible only by the class itself and so other classes will have to instantiate the object in order to be able to access those variables. So**

obviously I declared each variable in itself as private. I then go on to declare some of my methods and I start out be creating the constructor method, which is the basic method used to call an object or use that object [reiterating thought process and understanding the core concept]**.** It's simply passing values into it and then storing them into the variables, which will allow the other methods to manipulate the variables later on. **One of the tasks that the problem description wanted me to perform was depositing and withdrawing** [communicating the goal]**,** I created two methods. One that had no returns so it was a void. **All it simply did was set the current balance equal to the current balance at that time plus the additional of money or basically adding in money to the account, which is what a deposit would do. Withdraw was simply taking a double value – oh yeah and both of these methods have parameters. So in this case, the withdraw is going to be a double type and it's going to be subtracting the money and then it's going to set the current balance equal to the current balance minus money** [automating solution]**.** Hmmm **I should have added an if statement that checked for overdraft, which would have been the proper thing to do** [analyzing the solution]**.** Otherwise, **I feel quite confident that my program would perform as it should if I complied** [confidence]**.** But because I'm quite new at writing code on paper and not compiling it, I did have some misgivings about if it would throw up a compilation error or if something in my program might have a logical error. **But overall, I felt pretty confident that this particular class would assist in solving the problem** [confidence]**.**

The participant was able to share all three skills – knowledge, explanation and computational. The participant recalled the material learned in-class and thus was able to deduce the abstraction of the data from the given problem. Furthermore, the participant was able to describe the step-by-step procedure to compute the higher score with a variable and the array. However, the participant's description presented many technical words.

Excerpt:

> **Object is the instance of the class. So, class somehow** [confidence] **unifies the data. It has members. It has functions. So, class operates on data but the way it does, it has variables, member variables, and member functions** [understanding the core concept]**,** but when they create object, this is actually implementation of the class. **For example, if we have let's say a triangle. There are a whole bunch of different triangles, but we can create a class triangle because they all have in common, they all have three sides** [referring to in-class examples]**. They all have three angles I guess and there are certain common characteristics that all the triangles have. So, if we create a class that does a certain function or includes certain variables about this triangle then we can simplify the program., and then we can apply it to a particular – when we create an object, we apply it to a particular triangle** [rephrasing with no technical word]**.** For me, the idea that a group of data can have common characteristics that is what helps me to understand that this is a class, that this is class of triangles. Let's say mushrooms, yeah. **There's a class of mushrooms, class of animals**

[understanding core concept]**. Maybe it's the same idea. Oh I think yeah** [confidence] we would give an example of animals that would be cool.

The participant was able to express his knowledge skills in such a manner that showed that he understood the abstraction concept but his knowledge still presented some uncertainty referred as "somehow" and "maybe." However, the participant was able to reiterate the in-class example 'triangle' except for the details of the example. It seems that the participant was shying away from explaining the common and/or certain characteristics, which may be due to missing knowledge.

Below I have included excerpts that presented negative notations of the criteria indicated in Table 20.

Excerpt:

I didn't know [confidence] like how many exams they were going to enter and so I was like trying to account for that. I have them keep entering but then I was like **I don't know** [confidence] how many and how am I going to keep track of all of them so **I created five variables to keep track of five scores for a given student** [understanding core concept]**.**

In this excerpt above, the participant failed to efficiently implement his solution using an array. And thus the modeling of the solution was incorrect. Furthermore, the participant seemed to show that he was a bit confused about how to solve the problem. He used the statement "I don't know" twice in this interview excerpt.

Excerpt:

An array of objects is like having an **array; you had the set like 0,1,1,0, and then whatever. And you would set it up like the numbers in the left-hand side go down in the left hand and the numbers in the right-hand side go to the right hand. And in the middle was like what it makes like when you have combination** [understanding the core concept]**.** Oh an array of- well you could have different – it doesn't have to be numbers in the array. It could be like say like class members or a class of students or something. You have John, Joe, Matt, and you have like the test scores also and you have John 90 or something.

In this excerpt above, the participant was not able to explain the core concept. I interpreted this student's response as showing that the participant did not fully understand the concept of arrays.

Excerpt:

(laughing) **I am panicking. I don't know. I really don't know** [confidence] how to solve it, just from scratch. I mean if I had a laptop and the Internet then maybe my Java book. I probably could figure out in more than 30 minutes I'm sure. The biggest for me right now is to bring the code out of nothing. Plus, I don't have fellow students to ask for help or see what they did to see what I did differently; help them, help me, and just me and the paper.

In this excerpt above, the participant panicked and thus he lost his confidence before he even began. It seemed that he was out of his comfort zone, which included not having access to his laptop, Internet, and textbook.

Below I have indicated some of the types of struggles/challenges that participants encountered.

Excerpt:

> **It's hard** [confidence] in the beginning because [abstraction] is a new concept. You have to switch your way of thinking. Instead of having a set of instructions and focusing on instructions, you focus on how to organize the data. I used the program that does mind map where I can make connection like classes and objects and then I broke [them] up with all the concepts that connected with. For me I'm a visual person so mind map for difficult things or an abstract thing that has a lot of concepts works great. There are a bunch of programs that do that [such as] **Mind Jet, Mind Note, Mind Manager** [sharing examples]. The website for Mind Manager has a lot of templates for teachers to use for hard concepts.

The participant noted that she encountered difficulties because the concept of abstraction is hard for her mind to understand. The participant is a visual learner and thus abstract ideas need to be presented to her in a visual manner. For example, the tool Mind Map (http://www.mindmap.com/), which is a diagram (similar to UML) used to represent words and ideas linked to a central keyword. This tool helps with studying/organizing information and solving problems.

Excerpt:

> I think sometimes people don't grasp some of the basics, like with objects, and I **have a hard time grasping** [confidence]. I am thinking I've mislabeled a few things that were variables as objects when doing the exercise.

This participant showed confusion in his learning of the core concept, and thus he was unable to build up his knowledge.

Excerpt:

> I kind of looked over it and I noticed that there are a **few problems** [confidence] that I had, but it was something that I would have to be sitting at a computer and testing it to see what would work and what wouldn't work. So, like if the current balance is equal to zero, set the current balance to the initial balance. **I thought what if the person withdrew exactly to where the current balance is zero then they would get their money back** [reiterating thought process]. So there are little things like that I have to tweak and fix before it was a **perfect program** [analyzing solution].

The participant was aware that his solution was not complete in terms of efficiency. The participant needed the computer to test and 'tweak' (i.e. debug) out his solution.

Last but not least, few of the excerpts were related to self-taught concepts and utility of learning tools. These are indicated below.

Excerpt:

> I taught myself [programming]. I mean, **I'm sure** [confidence] you understand that **with programming, there's the structure and then the syntax** [understanding core concept]. **I taught myself the structure, which is very similar among, you know, most languages. And obviously, I taught myself the syntax of the language I was learning as well** [recognizing prior knowledge]**.** But because I know the structure, it's a lot easier for me to learn

different languages. It's […] like […] music. **Once you learn how to read music, it's not difficult to learn to play a different instrument** [sharing example].

The participant demonstrated his/her confidence through sharing his/her experience in learning the concept of programming by first learning the common structure in all languages and the syntax, which is bonded to the particular language.

Now let's take a look at the score distribution of the participants who took both the written and verbal protocols in this study. Sixteen participants were identified in the first test and twelve participants in the second test. First, the intercoder reliability kappa coefficient was run on the other two variables – explanation and abstraction for test 1, and explanation and modeling for test 2. The level of reliability for the kappa index is summarized in Table 22. According to Landis & Koch [84], the level of reliability is "outstanding" as all the Kappa coefficients are greater than 0.80.

Table 22 – Qualitative Symmetric Measures

	Measure of Agreement Kappa Value	Approx. Sig.	N of Valid Cases
Exp1_c1 * Exp1_c2	.811	.000	16
Abstr_c1 * Abstr _c2	.805	.000	16
Conf1_c1 * Conf1_c2	.893	.000	16
Exp2_c1 * Exp2_c2	.862	.000	12
Model_c1 * Model_c2	.862	.000	12
Conf2_c1 * Conf2_c2	.862	.000	12

The two judges resolved disagreements in the reliability coding. An agreement was reached after a second round of evaluation, and thus the two judges came to the same conclusion for all the participants. A third judge was not needed to serve as tiebreaker.

The final score distributions for the participants are indicated below (Table 23 and Table 24). Note that the scores of the variables explanation, abstraction, modeling, and confidence are based on a scale of zero to one. Also, across the two tests, four participants took both written and verbal protocols for both tests; they are highlighted in bold. From Table 23 and Table 24, participants who performed low (i.e. scores less than 0.5) in algorithm design also performed low in problem solving. Participants who performed average in problem solving (i.e. scores equals 0.5) performed higher in algorithm design (i.e. scores greater than 0.5). This indicates that even though the participants were not able to solve the problem, they were able to abstract key elements from the problem statement, and thus they knew what information were relevant. If algorithm design scores were higher than problem solving scores then participants shown some knowledge of the core concept (i.e. 0.5 out of 1). Furthermore, their difficulty with solving the problem had a direct impact on their confidence and explanation skills. Their explanations were not clear and demonstrated misconceptions about the core concept. Participants who scored less than 0.5 in problem solving also scored zero in confidence and either zero or 0.5 in explanation. Whereas participants who performed higher in problem solving (i.e. at least 0.5) also scored at least 0.5 both in confidence and explanation. Finally, participants who successfully solved the

problem (i.e. scores equal 1) also performed highly in algorithm design, explanation, core concept, and confidence (i.e. scores greater than 0.75).

Table 23 – Score Distribution for Test 1

UML1	*Coding1*	*Explanation*	*Abstraction*	*Confidence*
0.75	0.916667	1	1	1
0.625	0.75	1	0.5	0.5
0.75	1	0.5	1	1
0	0	0	0	0
0.75	**0.333333**	**0**	**0.5**	**0.5**
0.75	**0.916667**	**1**	**1**	**1**
0.5	0.5	0	0.5	0
0.666667	1	0.5	1	1
0.875	**1**	**1**	**1**	**1**
0.25	0	0.5	0	0
0.75	0.5	0.5	0.5	0
0.75	1	1	1	1
0.875	1	1	1	1
0	0.166667	0	0	0
0.5	0.666667	0	0.5	0
0.5	**0**	**0**	**0.5**	**0**

Table 24 – Score Distribution for Test 2

UML2	*Coding2*	*Explanation*	*Modeling*	*Confidence*
0.75	**0.833333**	**1**	**1**	**1**
0.875	**1**	**1**	**1**	**1**
0.75	1	1	1	1
0.75	0.5	0.5	0.5	0
0.75	0.666667	0.5	0.5	0.5
0.25	0.166667	0	0.5	0
1	1	1	1	1
0	**0**	**0**	**0**	**0**
0.75	0.916667	1	1	1
0.5	0.166667	0.5	0.5	0
0	0	0.5	0	0
0.25	**0.333333**	**0**	**0.5**	**0**

E. Summary and Discussion

This study's purpose was to explore the core computational concepts in CS1 courses and to assess students' skills in algorithm design and problem solving. Due to the limitations of the study, this chapter focused on primarily two core computational concepts – abstraction and modeling. From the participants' written and verbal responses, students' profiles were drawn based on their algorithm design (i.e UML) and problem solving (i.e. coding) and students' common mistakes were categorized based on the computational thinking criteria described in the review of literature.

First, it is important to acknowledge that students are on a path from novice to skilled programmers. That is, CS1 students first must learn to solve structured problems involving concepts, as in their introductory courses, to be able to both formulate and solve less structured and uncertain types of problems, as in the real-world applications. Developing such ability requires a continuing back-and-forth between theory and application as the students acquire more sophisticated skills through experience. In addition, computer science students are primarily eighteen to twenty-two years old, and thus students are still in the early phase of their cognitive development. Students' learning abilities at this phase can help computer science educators understand their students' cognitive development and thus improve assessment and instructions in terms of knowledge and practice. By the end of the introductory courses, students are expected to be able to use the computational concepts to solve specific and well-defined problems. It is assumed that the more they practice applying these concepts, the deeper their understanding of the concepts become.

Findings have shown an increase in higher scores in both algorithm design and problem solving. Even though, the number of participants who performed above average in algorithm design (i.e. abstraction of the class, attributes, and methods) showed no significant difference between test 1 and test 2 (69 percent), participants who performed at an average level, increased from 18 percent to 23 percent. The number of participants who performed above average in problem solving (i.e. implementation of the class, variables,

methods, and logic) increased from 52 percent in test 1 to 78 percent in test 2. In addition, the number of mistakes identified as computation thinking criteria decreased from test 1 to test 2 by one-third to two-third. When adding the verbal responses to the written responses, it was found that high scores in algorithm design were consistent with higher score in problem solving which was no surprise. Abstraction is the first step before solving a problem and thus a well-written abstraction of a given problem enables better guidelines for solving the problem. However, the problem solving scores were found to have a direct impact on the other variables, particularly on explanation skills and confidence. When solving a problem, more than two-third of the participants referred to knowledge such as definition and examples that were mostly visited in-class. This shows that students have acquired transferable knowledge, i.e. they have the ability to map problems' solutions to very similar problems given earlier.

It can be observed through algorithm design and problem solving that participants have indirectly acquired some of the skills in computational thinking. Since UML represents modeling the problem, students must identify the relevant information from the given problem. By doing so, students are using a form of abstraction, which is a key aspect of computational thinking; and thus, this is a fundamental step when attempting to solve a problem. As described in the background literature, the 'grand vision' of computational thinking is to enable everyone in any discipline of study to have a common understanding of the core computational concepts in the computing field to

solve real world problems. In this study, the assessment of abstraction was for participants to demonstrate their abilities in separating valuable and non-valuable information from the given problems. Also, abstraction included the representation of the valuable information by programming concepts such as class, object, data members, and methods. This step of abstraction in learning programming in CS1 is important because it assisted instructors/TAs in evaluating students' modeling these concepts which served as a primarily base for the next step in programming which was problem-solving. The assessment of problem solving was supported by the modeling of the abstraction step. The assessment of problem solving was for participants to demonstrate the application of abstraction to the given problems including the logic behind it. This step of problem solving in learning programming in CS1 is very relevant because it assisted instructors/TAs in evaluating students' reasoning in terms of basic operations such as arithmetic, conditional statements, and repetition statements. This step enables the discovery of mistakes in basic mathematical operations and thus incorrect logical thinking to solve a problem. Also, in this study, the problem solving step included the modeling of the solution using a programming language (Java or C++) so instructors/TAs were able to assess specifics about syntax mistakes which was resourceful to determine the level of complexity for concept specific syntax. Thus, abstraction and problem solving are essential in the learning of programming in CS1 because they represent the fundamental steps that any novice programmers would take to solve a given problem. It is crucial

that instructors teach students the importance of the step of abstraction before the step of solving a problem. The correct order of execution of these two steps will benefit students in long term when they will have to tackle more larger and complex problems.

Reinforcing the model of software design in CS1 curricula would enable this 'grand vision.' In CS1 courses, instructors teach students from all disciplines, i.e. computer science and non-computer science majors. The use of a tool, which does not require the knowledge of any programming language, would enable instructors to assess the notion of abstraction (classes, attributes, methods, and relationships) defined by computational thinking. In this study, UML was used as it is part of the curriculum. Furthermore, problem solving of simple real-world problems that can be identified by students as daily activities, such as bank account transactions and gradebook, enable students to develop their basic analytic skills such as abstraction, algorithm, reasoning and scalability. Such skills are critical to tackle larger problems using the computer. Sometimes, the programming language editors-compilers allow students to arrive at answers without thinking, if the students have mastered debugging skills. This study used paper-and-pencil and open-ended exercises to minimize 'guessing' when dealing with single value answer. Using such type of exercises, I was able to follow how students came up with an answer, which was more informative than traditional tests. However, half of the students were thrown off by the open-ended questions, and thus they encountered some difficulties in their algorithmic thinking. In

addition, participants wrestled with problems given in plain English, and they had to translate them into step-by-step problems, which involved mathematical operations. And one thing that is being stressed in the CS1 courses is that in the work environment, if an individual comes to you as a computer scientist and asks you to solve a problem stated in plain English then it is your responsibility to get this problem translated into an abstract problem and use your way of thinking to solve it.

V. RECOMMENDATIONS AND FUTURE RESEARCH

One of the recommendations to improve the understanding of the field of computer science (i.e. tackling almost all types of problem situation) is to use concrete real-world examples and problems that are not only related to daily activities but also to public service matters. Learners in the field want to make a difference in society, and thus, problems such as voting system, banking system, electronic health records, and traveling salesman address this interest and make learning to be more engaging and relevant. Through such problems, students are able to (1) combine data and ideas to solve problems, (2) create tools and information, and (3) manipulate data using abstractions and computational thinking. These real-world applications enable learning in context. The CS1 concepts can be learned in the context of a computing situation representative of the practice. Learning in context enable students the opportunity to interact with the body of knowledge in a way that connects with the practice for which they are being prepared for. This type of learning helps students relate what they are learning to how it may be used and results in a deeper understanding of the field. This means that instructors should introduce concepts in context to enable students to both internalize and transfer knowledge to other contexts. Besides making a direct connection to something real or familiar motivates students to be engaged and confident in their own learning.

Computational thinking is very similar to the field of computer science minus the domain-specific and the usage of the computer. It deals with (1) how difficult

problems are to solve, (2) how to think about and manage problems, and (3) how to create procedures for solving them. Nowadays the emergence of fields of study such as bioinformatics, computational biology, and computational mathematics has given an opportunity to apply computational concepts to a specific discipline such as biology, mathematics, and physics. Such fields have made computational science a third pillar of science, along with theory and experimentation. Thus, computational thinking is not one more thing to add to the curriculum but rather it emphasizes the application of the knowledge of the core computational concepts in various fields of study. Students develop their ability to abstract the information from a given problem and modeling the solution based on the computational strategies which can vary depending on the individual's thinking process. Looking at the CS1 curriculum, computational thinking is not explicitly stated and students may not be aware that in fact they are developing their computational thinking skills through the application of the core computational concepts in context-specific knowledge. And therefore, another recommendation is to make computational thinking concepts more visible in the curriculum. To do so, computational thinking skills can be stated in the syllabus under the section "course objectives and outcomes." In this section, the instructor has already stated that students should have an understanding of methods and variables, searching and basic sorting algorithms, and basic recursions. Also, students should be able to read, understand, and develop programs. These aptitudes are computational thinking skills and thus methods and variables represent the concept of "abstraction," searching and basic sorting algorithms represent the concept of

"algorithm," basic recursions represent the concept of "reasoning," and programs represent "systems."

In addition, the focus of assessment must be on how one thinks about a problem, not just the correct answer. To do this, instructors should challenge their students with responding to open-ended questions to determine how well they understand (explanation) and synthesize the concepts they have learned (thinking process). Multiple-choice questions may not give an accurate assessment of students' knowledge. Students can guess an answer and get it right. Students can also know the answer but their thinking process to get to the correct answer may include errors. Students may have just memorized the answer but they are not able to transfer this answer to another similar problem. As a consequence, a more detail-oriented response to a given question/ problem will allow instructors to more efficiently track down misconceptions and correct students' misconceptions at the next class period. Students focus on the approach to the problem rather than on their final answer. Moreover, the problem solving solution does not need to be programming language-specific. In this study, it was observed that novices spent quite some time on syntax during problem solving which took time away from their algorithmic thinking. So the recommendation is to use a common language, which is plain English pseudo-code. This would remove the programming language factor into the assessment equation and enable major and non-major CS1 students to express the solution with their own words, which can be understood by all. Assessing students' design and problem-solving skills by using open-ended problems enable students to consider the concepts that are relevant to

the situation and to demonstrate their ability to work through an analytic problem solving process. Because the quiz is such a small part of the grade, the assessment is considered formative as it provides constructive feedback in an ongoing learning to the instructors.

This method of assessing students can only be beneficial if it is a reflective approach of teaching. The emphasis is on acquiring a solid understanding of the CS1 concepts while strongly discouraging memorization. This can be difficult to do. An inductive approach of teaching may be more efficient to help students learn to use core concepts for their particular value and how to use them as a foundation for advanced learning. For example, an instructor may begin the class with a problem and ask student to find out the concept that is critical to the problem. Based on their existing knowledge and experience, students attempt to solve the problem with possible cases based on the attributes and constraints given in the problem. As they work through this process, students become aware of the key components relevant in all the cases. Consequently, they build their knowledge based on the phenomenon observed. Building on the learning experience, the instructor introduces new cases to the students so they can identify fundamental components. Mathematical explanations and diagrams may be used as tools to help students refine their understanding of a concept. As they do acquire such knowledge, the instructor introduces the theory and reconnects it to the problem. This approach differs from the deductive approach - which is commonly used in CS1 courses where students 'listen, see, and do' as the instructor transfers the knowledge to novices through lecturing - by (1)

introducing the context first before the concept and (2) educating students to be more reflective about their own learning as their learning experience is more iterative. The role of the instructor is primarily to show students what to look for and then how to explain unclear situations. Using feedback and coaching, the instructor's goal for student learning is to formulate problems and solve problems using concepts. In this iterative process, the novice begins to learn from experience and thus students are able to develop their skills and confidence.

Future research will include additional core computational concepts to be assessed. After the introduction of each computational concept in class, the quiz (test) would be given to the students, and then, based on the outcomes of the tests, the instructor will do an in-class intervention to go over the common mistakes found in students' test and the reinforcing the concept learned. The test will be given again to the students. This second round testing would enable to compare students' written responses between the two tests, and thus, their progress for a given concept would be better evaluated. In addition, with computational thinking as a cognitive revolution, it would be of interest to design the tests specifically with computational thinking in mind. Since computational thinking is still broad and not clearly defined yet, the approach would be to look at the intersection between CS1 computational concepts and computational thinking. The design of the tests would still involve both algorithm design and problem solving. However, the problem solving part would not be programming language-specific but rather pseudo-code, i.e. a step-by-step procedure to solve a given problem. In this study, findings have shown a difference in scores between students programming in Java

and students programming in C++, which is not clear whether or not it is due to the programming language. Thus, this change in programming language in the coding section would allow a better assessment of students' thinking/explanation as they will be using a common language, which is plain English. Plus, this is in agreement with computational thinking should be understood by all individuals regardless of their field of study, background, and programming language. Students would no longer spend time on syntax while solving the problem, which was one of the parameters that may have limited students' thinking. Last but not least, the limitation in the number of participants in this current study has made the findings limited. For future research, the study will be open to all CS1 courses, which therefore, will involve more than one CS1 instructors. To take this into account, the assessment will look into any score distribution differences across the courses with different instructors, and thus instructors' teaching approach (deductive versus inductive). This may provide valuable data to improve CS1 teaching.

REFERENCES

[1] M. Ford and S. Venema, "Assessing the success of an introductory programming course," In *Journal of Information Technology Education*, vol. 9, pp. 133-145, 2010.

[2] M. Guzdial and E. Soloway, "Log on education: teaching the Nintendo generation to program," in *Communications of the ACM*, 2002, vol. 45, no. 4, pp. 17-21.

[3] E. Soloway *et al.*, "Cognitive strategies and looping constructs: an empirical study," In *Communications of the ACM*, vol. 26, no. 11, pp.853-86, 1983.

[4] D. Perkins *et al.*, "Conditions of learning in novice programmers," In *Studying the Novice Programmer*, E. Soloway and J. Spohrer, Eds. Hillsdale, NJ: Lawrence Erlbaum Associates, Publishers, pp. 261-279, 1989.

[5] M. McCracken *et al.*, "A multi-national, multi-institutional study of assessment of programming skills of first-year CS students," In *Working group reports from Innovation and Technology in Computer Science Education*, pp. 125-180, 2001.

[6] R. Lister *et al.*, "Further evidence of a relationship between explaining, tracing and writing skills in introductory programming," In *SIGCSE Bulletin*, vol. 41, no. 3, pp. 161-165, 2009.

[7] S. Fincher *et al.*, "Programmed to succeed?: a multi-national, multi-institutional study of introductory programming courses," In *Computing Laboratory Technical Report 1- 05*, Canterbury, UK: University of Kent, 2005.

[8] T. L. Friedman, *The world is flat 3.0: A brief history of the twenty-first century*, New York, NY: Farrar, Straus, and Giroux, 2006.

[9] J. Mead *et* al, "A cognitive approach to identifying measurable milestones for programming skill acquisition," In *SIGCSE Bulletin*, vol. 38, no. 4, pp. 182-194, 2006.

[10] D. Shaffer and J. Gee, *Before every child is left behind: how epistemic games can solve the coming crisis in education. WCER Working Paper No. 2005-7*, Madison, Wisconsin: Wisconsin Center for Education Research, 2005.

[11] Student Affairs Leadership Council, *The Data-Driven Student Affairs Enterprise: Strategies and Best Practices for Instilling a Culture of Accountability*, Washington, DC: The Advisory Board Company, 2009.

[12] L. Carter, “Why students with an apparent aptitude for computer science don’t choose to major in computer science.” In *Proc. SIGCSE Technical Symposium on Computer Science Education*, pp. 27-31, 2006.

[13] M. Clancy, “Misconceptions and attitudes that interfere with learning to program,” In *Computer Science Education Research*, M. Petre and S. Fincher, Eds. London, UK: Routledge Falmer, ch. 1, pp. 85-100, 2004.

[14] A. Eckerdal *et al.*, “Putting threshold concepts into context in computer science education,” In *Proc. on Innovation and Technology in Computer Science Education*, pp. 103–107, 2006.

[15] C. Schulte and J. Bennedsen, “What do teachers teach in introductory programming?” In *Proc. International Workshop on Computing Education Research*, pp. 17-28, 2006.

[16] E. Seymour, “The problem iceberg' in science, mathematics, and engineering education: student explanations for high attrition rates,” In *Journal of College Science Teaching*, pp. 230-232, 1992.

[17] E. Spertus, E, “Why are there so few female computer scientists?” In *MIT Artificial Intelligence Laboratory*, Technical Report: AITR- 1315, 1991.

[18] S. Hansen and E. Eddy, “Engagement and frustration in programming projects,” In *Proc. SIGCSE Technical Symposium on Computer Science Education*, pp. 271-275, 2007.

[19] N. Jacobson and S. Schaefer, “Pair programming in CS1: overcoming objections to its adoption,” In *SIGCSE Bulletin*, vol. 40, no. 2, pp. 93-96, 2008.

[20] R. Lister *et al.*, “Further evidence of a relationship between explaining, tracing and writing skills in introductory programming,” In *SIGCSE Bulletin*, vol. 41, no. 3, pp. 161-165, 2009.

[21] D. Cliburn and S. Miller, “Games, stories, or something more traditional: the types of assignments college students prefer,” In *Proc. SIGCSE Technical Symposium on Computer Science Education*, pp. 138-142, 2008.

[22] J. Gilbert *et al.*, “Learning C with Adam,” In *International Journal on E-Learning*, vol. 4, no. 3, pp. 337-350, 2005.

[23] N. Herrmann *et al.*, “Assessment of a course redesign: introductory computer programming using online modules,” In *Proc. SIGCSE Technical Symposium on Computer Science Education*, pp. 66-70, 2004.

[24] B. Woolf, *Building Intelligent Interactive Tutors: Student-centered strategies for revolutionizing e-learning*, Burlington, MA: Morgan Kaufmann, 2008.

[25] G. Novak *et al.*, *Just-in-Time Teaching: Blending Active Learning with Web Technology*, Upper Saddle River, NJ: Prentice Hall, 1999.

[26] J. Rountree and N. Rountree, "Issues regarding threshold concepts in computer science," In *Proc. Australasian Computing Education Conference*, pp. 139-145, 2009.

[27] L. Murphy *et al.*, "A multi-institutional investigation of computer science seniors' knowledge of programming concepts," In *Proc. SIGCSE Technical Symposium on Computer Science Education*, pp. 510-514, 2005.

[28] A. Postlewaite and O. Compte, "Confidence enhanced performance," In *Penn Institute for Economic Research*, vol. 94, no. 5, pp. 1536-1557, 2001.

[29] A. Seidman, *College Student Retention: Formula for Student Success*, Westport, CT: Praeger Publishers, 2005.

[30] D. Wortman and P. Rheingans, "Visualizing trends in student performance across computer science courses," In *SIGCSE Bulletin*, vol. 39, no. 1, pp. 430-434, 2007.

[31] I. Milliszewska *et al.*, "Improving progression and satisfaction rates of novice computer programming students through ACME – Analogy, Collaboration, Mentoring, and Electronic support," In *The Journal of Issues in Informing Science and Information Technology*, vol. 5, pp. 311-323, 2008.

[32] C. Ramamoorthy, "Trends and perspectives in computer science and engineering education," In *Proc. IEEE*, vol. 66, no. 8, pp. 872-879, 1976.

[33] R. Sloan and P. Troy, "CS 0.5: a better approach to introductory computer science for majors," In *Proc. SIGCSE Technical Symposium on Computer Science Education*, pp. 271-275, 2008.

[34] A. Pears *et al.*, "A survey of literature on the teaching of introductory programming," In *Proc. Innovation and Technology in Computer Science Education*, pp. 204-223, 2007.

[35] L. Sudol, "Forging connections between life and class using reading assignments: a case study," In *Proc. SIGCSE Technical Symposium on Computer Science Education*, pp. 357-361, 2008.

[36] D. Stevenson and P. Wagner, "Developing real-world programming assignments for CS1," In *Proc. SIGCSE Innovation and Technology in Computer Science Education*, pp.148-162, 2006.

[37] J. Stone and E. Madigan, "The impact of providing project choices in CS1," In *ACM SIGCSE Bulletin*, vol. 40, no. 2, pp. 65-68, 2008.

[38] R. McCartney *et al.*, "Commonsense computing (episode 5): algorithm efficiency and balloon testing," In *Proc. International Computer Science Education Research*, pp. 51-62, 2009.

[39] M. Biggers *et al.*, "Student perceptions of computer science: a retention study comparing graduating seniors vs. CS leavers," In *Proc. SIGCSE Technical Symposium on Computer Science Education*, pp. 402-406, 2008.

[40] R. Sperry and P. Tedford, "Implementing peer-LED team learning in introductory computer science courses," In *Journal of Computing Sciences in Colleges*, vol. 23, no. 6, pp. 30-35, 2008.

[41] L. Beck and A. Chizhik, "An experimental study of cooperative learning in cs1," In *ACM SIGCSE Bulletin*, vol. 40, no. 1, pp. 205-209, 2008.

[42] M. Chi, "Active-Constructive-Interactive: A Conceptual Framework for Differentiating Learning Activities," In *Topics in Cognitive Science*, vol. 1, pp. 73-105, 2009.

[43] J. McConnell, "Active and cooperative learning: final tips and tricks (part IV)," In *ACM SIGCSE Bulletin*, vol. 38, no. 4, pp. 25-28, 2006.

[44] L. Ma *et al.*, "Using cognitive conflict and visualization to improve mental models held by novice programmers," In *Proc. SIGCSE Technical Symposium on Computer Science Education*, pp. 342-346, 2008.

[45] W. Jin, "Pre-programming analysis tutors help students learn basic programming concepts," In *Proc. SIGCSE Technical Symposium on Computer Science Education*, pp. 276-280, 2008.

[46] J. Bonar and E. Soloway, "Uncovering principles of novices programming," In *Proc. ACM Symposium on Principles of Programming Languages*, pp. 10-13, 1983.

[47] E. Soloway *et al.*, "A goal/plan analysis of buggy Pascal programs," In *Human-Computer Interaction*, vol. 1, no. 2, pp. 163-207, 1985.

[48] E. Soloway, "Learning to program = learning to construct mechanisms and explanations," In *Communications of the ACM*, vol. 29, pp. 850-858, 1986.

[49] R. Rist, "Knowledge creation and retrieval in program design: a comparison of novice and intermediate student programmers," In *Human-Interaction Computer*, vol. 6, no. 1, pp. 1-46, 1991.

[50] S. Segelman, "A continuing study of intermediate programming errors," In *Computer Information Science,* 60.1 Senior Research Project, 2003.

[51] J. Anderson *et al.*, "Learning to program in LISP," In *Cognitive Science*, vol. 8, pp. 87-129, 1984.

[52] R. Jeffries *et al.*, "The processes involved in designing software," In J. R. Anderson (Ed.), *Cognitive Skills and their Acquisition*, pp. 255-283, Hillsdale, NJ: Laurence Erlbaum Associates, Inc, 1981.

[53] D. Perkins and F. Martin, "Fragile knowledge and neglected strategies in novice programmers," In E. Soloway & S. Iyengar (Eds.), *Empirical studies of programmers*, Norwood, NJ: Albex, pp. 213-229, 1986.

[54] J. Brown, and K. VanLehn, "Repair theory: a generative theory of bugs in procedural Skills," In *Cognitive Science*, vol. 4, pp. 379-426, 1980.

[55] K. VanLehn, "Bugs are not enough: empirical studies of bugs, impasses and repairs in procedural skills," In *Journal of Mathematical* Behavior, vol. 3, no. 2, pp. 3-71, 1981.

[56] D. Perkins and G. Salomon, "Teaching for transfer," In *Educational Leadership*, vol. 46, no. 1, pp. 22-32, 1988.

[57] M. McCracken *et al.*, "A multi-national, multi-institutional study of assessment of programming skills of first-year CS students," In *Working group reports from Innovation and Technology in Computer Science Education*, pp. 125-180, 2001.

[58] R. Lister *et al.*, "A multi-national study of reading and tracing skills in novice programmers," In *SIGCSE Bulletin*, vol. 36, no. 4, pp. 119-150, 2004.

[59] M. Lopez *et al.*, "Relationships between reading, tracing, and writing skills in introductory programming," In *Proc. International Workshop on Computing Education Research*, pp.101-112, 2008.

[60] J. Wing, "Computation thinking," In *Communications of the ACM*, vol. 49, no. 3, pp. 33-35, 2006.

[61] J. Lu and G. Fletcher, "Thinking about computational thinking," In *Proc. ACM Technical Symposium on Computer Science Education*, pp. 260-264, 2009.

[62] O. Astrachan *et al.*, "The present and future of computational thinking," In *Proc. ACM Technical Symposium on Computer Science Education*, pp. 549-550, 2009.

[63] Carnegie-Mellon University, "Center for computational thinking," http://cs.cmu.edu/~CompThink

[64] P. Seymour, "An exploration in the space of mathematics education," In *International Journal of Computers for Mathematical Learning*, vol.1, no.1, pp. 95-123, 1996.

[65] A. Bundy, "Computational thinking is persuasive," In *Journal of Scientific and Practical Computing*, vol. 1, no. 2, pp. 67-69, 2007.

[66] M. Guzdial, "Paving the way for computational thinking," In *Communications of the ACM*, vol.51, no. 8, pp. 25-27, 2008.

[67] National Academy of Sciences on Computational Thinking, *Report of a Workshop on the Scope and Nature Computational Thinking*, National Academies Press, 2010.

[68] P. Denning, "Great principles of computing," In *Communications of the ACM*, vol. 46, no. 11, pp.15-20, 2003.

[69] O. Astrachan and P. Denning, "Innovating our self image," In *SIGCSE Technical Symposium on Computer Science Education*, vol. 10, no. 1, pp. 178-179, 2008.

[70] J. Wing, "Computational thinking and thinking about computing," In *Philosophical Transactions of the Royal Society*, vol. 366, pp. 3717-3725, 2008.

[71] Committee for the Workshop on Computational Thinking; National Research Council. (2010). *Report on Workshop on the Scope and Nature of Computational Thinking*. Available: http://catalyst.fullerton.edu/library/Scope_and_Nature_of_Computational_Thinking.pdf

[72] O. Hazzan, "Reflections on teaching abstraction and other soft ideas," *Inroads*, vol. 40, no. 2, pp. 40-43, 2008.[73] J. Kramer, "Is abstraction the key to computing?," *Communications of the ACM*, vol. 50, no. 4, pp. 37-41, 2007.

[74] F. Olsen, “Computer Scientist says all students should learn to think ‘algorithmically’,” In *The Chronicle of High Education*, 2000.

[75] J. Gal-Ezer, T. Vilner, and E. Zur, “Teaching algorithm efficiency at CS1 Level: A different Approach,” In *Computer Science Education*, vol. 14, no. 3, pp. 235-248, 2004.

[76] M. Patton, *Qualitative Research and Evaluation Methods*, (3rd Ed.), Thousand Oaks, CA: Sage, 2002.

[77] N. Dale, “Most difficult topics in CS1: results of an online survey of educators,” In *SIGCSE Bulletin*, vol. 38, no. 2, pp. 49-53, 2006.

[78] J. Boustedt *et al.*, “Threshold concepts in computer science: do they exist and are they useful?” In *Proc. SIGCSE Technical Symposium on Computer Science Education*, pp. 504-508, 2007.

[79] T. Amabile, "The Social Psychology of Creativity: A Consensual Assessment Technique," In *Journal of Personality and Social Psychology*, vol. 43, pp. 997-1013, 1982.

[80] K. Neuendorf, *The content analysis guidebook*. Thousand Oaks, CA: Sage, 2002.

[81] R. Kolbe and M. Burnett, “Content-analysis research: An examination of applications with directives for improving research reliability and objectivity,” In *Journal of Marketing Research*, vol. 27, pp. 185-195, 1991.

[82] M. Lombard and J. Snyder-Duch, “Content Analysis in Mass Communication: Assessment and Reporting of Intercoder Reliability,” In *Human Communication Research*, vol. 28, no. 4, pp. 587-604.

[83] R. Bakeman, “Behavioral observation and coding,” In H. T. Reis & C. M. Judge (Eds.), *Handbook of research methods in social and personality psychology* (pp. 138-159). New York: Cambridge University Press, 2000.

[84] J. Landis and G. Koch, “The measurement of observer agreement for categorical data,” In *Biometrics*, vol. 33, pp.159-174, 1977.

APPENDIX A

IRB APPROVAL

ASU ARIZONA STATE UNIVERSITY
RESEARCH AND ECONOMIC AFFAIRS

Office of Research Integrity and Assurance

To:	Tirupalavanam Ganesh EDUC - I.
From:	Mark Roosa, Chair Soc Beh IRB
Date:	09/29/2009
Committee Action:	**Exemption Granted**
IRB Action Date:	09/29/2009
IRB Protocol #:	0909004380
Study Title:	How Computer Science & Engineering Freshmen Write Computer Programs?

The above-referenced protocol is considered exempt after review by the Institutional Review Board pursuant to Federal regulations, 45 CFR Part 46.101(b)(2) .

This part of the federal regulations requires that the information be recorded by investigators in such a manner that subjects cannot be identified, directly or through identifiers linked to the subjects. It is necessary that the information obtained not be such that if disclosed outside the research, it could reasonably place the subjects at risk of criminal or civil liability, or be damaging to the subjects' financial standing, employability, or reputation.

You should retain a copy of this letter for your records.

APPENDIX B

SURVEY ON CS1 CONCEPTS (SENT BY EMAIL)

How Computer Science & Engineering Freshmen Write Computer Programs?

SURVEY

Dear [X],

I am a graduate student under the direction of Dr. Tirupalavanam Ganesh in the Fulton Institute and Graduate School of Education and Dr. James Collofello in the School of Engineering at Arizona State University. I am conducting a research study to identify skills that freshmen develop in their introductory computer programming course. This study will help instructors to understand and assess how their students design their algorithm (flowchart) and how their students write their algorithm (methods).

To conduct this study, I first need to identify the concepts that students in CS1 have difficulty with. You have been selected, because you are either an instructor or a teaching assistant in CS1 courses, to help us identify the most troublesome concepts in CS1 courses.

We thank you in advance for the information that you are about to share. If you have any questions concerning the research study, please call me at (480) 276-4188 or email me at EBillion@asu.edu.

Based on your teaching experience, please list below the most difficult concepts that students in your class encountered:

1.
2.
3.
4.
5.
6.

Comments:

Feel free to continue the list if you have identified more than six concepts. Please email your response to EBillion@asu.edu.

APPENDIX C

TEST I (FOR BOTH CSE 100 AND CSE 110)

School of Computing, Informatics and Decision Systems Engineering,
Arizona State University
Fall 2009. Quiz 1
10 + 4 Bonus Points, 30 Minutes

You have been asked to develop a banking application for the Bank of ASU. A customer's account should have their name, account number, and the current balance. In addition, your ASUBankAccount class should be able to support customers who would like to ***withdraw*** from and ***deposit*** to their bank account. They also must be able to ***check the current balance***. Check current balance method should return the current balance. Customers are NOT allowed to overdraw on their account. Finally, the ***constructor*** should take name, account number, and the initial balance at the time of object creation and set account instance variable values accordingly.

Part 1 - DESIGN [4 Points]:

Please draw the UML diagram that represents the ASUBankAccount class above.

Make sure to identify proper data types for attributes (data members).

Part II – CODING [6 Points]

Based on your UML diagram above, please develop the ASUBankAccount class.

Part III – BONUS [4 bonus Points]

Add a data member (s) to store last three transactions. Then add a method named **displayTransactions** that displays the last three transactions. (Hint: you can use string variable (s) to store transactions and can update them when you withdraw or deposit money)

1. Please describe how difficult this problem is
   ___ 1 Really easy
   ___ 2 Easy
   ___ 3 Ok
   ___ 4 Difficult
   ___ 5 I'm dying, man!

Important Note: You may get a request to participate in the study entitled "*How Computer Science & Engineering Freshmen Write Computer Programs?"* This study is voluntary and will not impact your grade in any way.

APPENDIX D

TEST II (FOR BOTH CSE 100 AND CSE 110)

School of Computing, Informatics and Decision Systems Engineering,
Arizona State University
Fall 2009. Quiz 2
10 + 4 Bonus Points, 30 Minutes

You have been asked to develop a Student class to store, say ASU student information. Student class should store the student name, class name (such as CSE110), letterGrade, final average, and exam scores. Assume that each student has 5 exam scores. Constructor of the ASUStudent class should take the student name, class name as parameters at the time of object creation. Then, it sets the letterGrade to 'F' and all the exam scores and the final average to zero. ASUStudent class should have following methods.

readExamScores: Ask the user to enter exam scores from the keyboard and set exam scores
calculateExamAverage: This function calculates the exam average. Assume that each exam can have maximum 100 and each exam has the same weight in the average calculation.
determineLetterGrade: This method determine the letter grade based on the following criteria

final average >= 90	A
80 =< final average <90	B
70 =< final average<80	C
Otherwise	F

Part 1 - DESIGN [4 Points]:

Please draw the UML diagram that represents the ASUStudent class above. Make

sure to identify proper data types for attributes (data members).

Part II – CODING [6 Points]

Based on your UML diagram above, please develop the ASUStudent class.

Part III – BONUS [4 bonus Points]

- Add the method getHeighestScore that returns the getHeighestScore test score.

- Add the toString method that return the following message

<student_name> , you have earned <letter grade> for <class name>.
For example, if the Student name is John, and he has earned B for CSE 110, then, the toString method should return the following string
John, you have earned B for CSE 110.

2. Please describe how difficult this problem is
___ 1 Really easy
___ 2 Easy
___ 3 Ok
___ 4 Difficult
___ 5 I'm dying, man!

Important Note: You may get a request to participate in the study entitled "*How Computer Science & Engineering Freshmen Write Computer Programs?"* This study is voluntary and will not impact your grade in any way.

APPENDIX E

TEST III (FOR CSE 100)

School of Computing, Informatics and Decision Systems Engineering,
Arizona State University
Fall 2009. Quiz 3
10 Points, 30 Minutes

1.

i) _____ allows us to create new classes based on existing classes.

A) Polymorphism B) Inheritance C) Function overloading D) The copy constructor

E) None of the above

ii.) What is the correct syntax for defining a new `class Parakeet` based on the superclass `Bird`?

a. `class Parakeet isa Bird{ }`

b. `class Bird extends Parakeet{ }`

c. `class Bird hasa Parakeet{ }`

d. `class Parakeet: public Bird{ }`

iii. Inheritance is an example of what type of relationship?

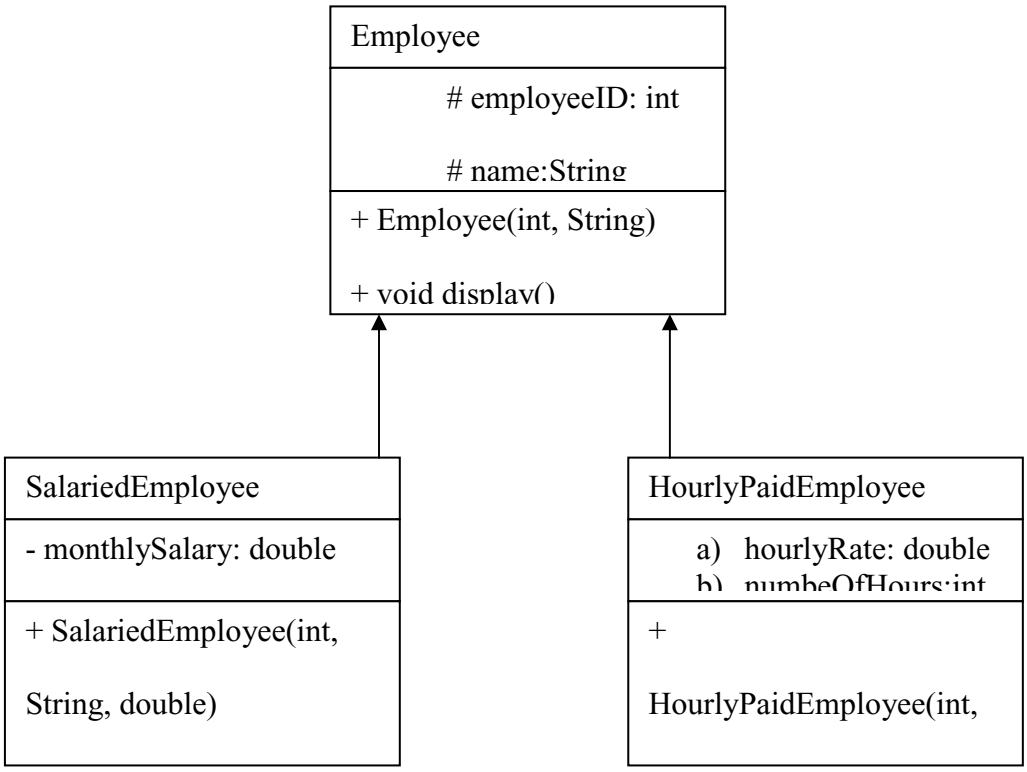
a. is-a

b. has-a

c. was-a

d. had-a

2. Consider the inheritance hierarchy given below and answer following questions



a) [1 points] What is the base (or super) class above?

b) [2 Points] Briefly explain two benefits of inheritance

c) [1 points] How many data members does the HourlyPaidEmployee have?

3. The following program. Assume that all the programs are correct.

```
class Book
{
  protected:
      int pages;

   public:
   Book ()
   {
         pages = 0;
   }

   Book (int numPages)
   {
      pages = numPages;
   }

   void setPages (int
numPages)
   {
      pages = numPages;
   }


   int getPages ()
   {
      return pages;
   }
};
```

```
class Dictionary: public Book
{
  private:
      int definitions;

   public:
  Dictionary(int numPages, int
numDefinitions):Book(numPages)
   {
       definitions =
numDefinitions;
   }

  double computeRatio ()
   {
     if(pages > 0)
      return definitions/pages;
      else
      {
       setPages(900);
       return definitions/pages;
      }
   }

 void setPages(int p)
   {
         pages = p + 100;
   }
};
```

```
int main ()
   {
     Dictionary Dic1 (500, 10000);
     cout<<"Definitions per page: " << Dic1.computeRatio();//--
1


     Dictionary Dic2 (0, 10000);
     cout<<"Definitions per page: " << Dic2.computeRatio();//--
2

     return 0;
   }
```

- [1 points] What is the purpose of "`Dictionary(int numPages, int numDefinitions):Book(numPages)`" statement in the constructor of the Dictionary class above?

- [1 Points] What is the output generate from statement 1 in the main() program above? Explain your answer.

- [1 points] What is the output generate from statement 2 in the main() program above? Explain your answer.

Important Note: You may get a request to participate in the study entitled "*How Computer Science & Engineering Freshmen Write Computer Programs?"* This study is voluntary and will not impact your grade in any way.

APPENDIX F

TEST III (FOR CSE 110)

School of Computing, Informatics and Decision Systems Engineering,
Arizona State University
Fall 2009. Quiz 3
10 Points, 30 Minutes

1.

i) _____ allows us to create new classes based on existing classes.

A) Polymorphism B) Inheritance C) Function overloading D) The copy constructor

E) None of the above

ii.) What is the correct syntax for defining a new `class Parakeet` based on the superclass `Bird`?

a. `class Parakeet isa Bird{ }`

b. `class Bird defines Parakeet{ }`

c. `class Bird hasa Parakeet{ }`

d. `class Parakeet extends Bird{ }`

iii. Java supports both single and multiple inheritance…….(T/F).

2. Ship and CargoShip

Design a Ship class that has the following members

- A member variable for the name of the ship (a string)
- A member variable for the year that ship was built (an int)
- toString method that returns the name and year built

Design the CargoShip class that is derived from the Ship class. The CrargoShip should have the following members.

ii. A member variable to store the max capacity (an int)

iii. A constructor that takes three parameters for the ship's name, year built, and the capacity and sets ship's name, year built, and the capacity at the time of object creation

iv. Redefine the toString member function that returns the name, year built, and the capacity(this technique is called the function overriding)

- Draw the UML diagram that represents the above Inheritance Hierarchy

i. Then, implement the Ship class and CargoShip classes

Important Note: You may get a request to participate in the study entitled "*How Computer Science & Engineering Freshmen Write Computer Programs?"* This study is voluntary and will not impact your grade in any way.

APPENDIX G

THINK ALOUD PROTOCOL – TASK LIST

- Introduction to the purpose of the study - explain goals of this activity

- Think aloud Warm-up exercise – explain the concept of think aloud. Ask the participant to tell everything they are thinking about from the moment they read the task and when they complete it. They do not need to plan/think out what they want to say. Just act as if you are by yourself, talking to yourself. The important thing is to keep talking. Perform a sample think aloud. For example:

  > Think aloud as you count how many windows are in your house.
  > Now, ask the user to try just as you did. Another example follows.
  > "Please think aloud as you name how many doors are in your house?"
  > or
  > "Please re-count your actions in your morning routine before you came to work."

A. Establish some rules during the session
   1. You will not be able to answer any questions during the observation
   2. If you have questions, go ahead and ask them, but you won't respond until after the session is complete.
   3. Remind them that if they're silent for more than 5-10 seconds, you will ask them to "Please keep talking"

- Reaffirm that they agree with being audiotaping

- Explicitly mention in-room observers and/or videotaping

- Describe the exercise being presented – three sections

- Explain that you are not testing them

- Reassure users about what will happen if they encounter difficulties – to continue and do what they can

- Clarify tasks if confusing

- Confirm ending time and reassure them that they can stop at any time – 30 minutes

APPENDIX H

PRE-SURVEY QUESTIONNAIRE

Interviewee ID: _____ Date: _____ Time: _____

Hello. Thank you for taking the time to meet with me and answer questions related to logical reasoning and programming skills. Before you begin, I want to remind you that you can skip questions if you wish. If you choose not to participate or elect to withdraw from the study at any time, there will be no penalty. It will not affect your grade in any way. Do you choose to continue? Yes or No

1. Can you tell me about X*?
2. How would you describe X to another freshman student?
3. How easy do you think X is?
    1) Very Easy
    2) Easy
    3) Ok
    4) Hard
    5) Very Hard
4. How did you gain a better understanding of *X* (e.g. books, websites, discussion with a peer, instructor's notes, teaching assistant ...)?

Thank you very much for participating in this study. Your time and insights are greatly appreciated.

* X refers to a specific threshold concept

APPENDIX I

POST-SURVEY QUESTIONNAIRE

Interviewee ID: _____ Date: _____ Time: _____

Hello. Thank you for taking the time to meet with me and answer questions related to logical reasoning and programming skills. Before you begin, I want to remind you that can skip questions if you wish. If you choose not to participate or elect to withdraw from the study at any time, there will be no penalty. It will not affect your grade in any way. Do you choose to continue? Yes or No

1. How would you go about assisting other students who might be struggling with X?
2. What concepts better helped you understand X?
3. What concept(s) do(es) X help you better understand?
4. Has X come up in other contexts? Where?
5. Is there something more you want to share with me about X?
6. Are there any other concepts you struggled with early in the course that became clearer at the end?

Thank you very much for participating in this study. Your time and insights are greatly appreciated.

* X refers to a specific threshold concept

APPENDIX J

INTERVIEW QUESTIONS

Interviewee ID: _____ Date: _____ Time: _____

Hello. Thank you for taking the time to talk with me today about your thought process on writing program. Before we begin, I want to remind you that I am planning to record our conversation today so please speak clearly. Do I still have your permission to make the audio recording? [Note response] _____

I want to assure you that your identity will be kept strictly confidential. I will be asking you a number of questions so please feel free to discuss your ideas and views. Are you ready to begin?

- What was your initial idea for solving this problem when you first read the problem?
- Please share the thought processes you used when solving the problem.
- Please describe how you went about solving the problem.
- How did you feel when you were done solving the problem? What did you think?
- Describe any challenges you may have had when you attempted to solve the problem? [If yes,] How did you approach the challenge?
- Did you notice any other areas where a student might face challenges when solving this problem? Please describe them and discuss your reasons.

Thank you very much for participating in this study. Your time and insights are greatly appreciated.

APPENDIX K

EXAMPLE OF A TRADITIONAL CS1 QUIZZ

CSE 110: Principles of Programming with Java
School of Computing, Informatics and Decision Systems Engineering, Arizona State University
Fall 2009, Quiz
10 Points Point

Student Name:.

Part – I :

Identify the choice that best completes the statement or answers the question

1. What is the output of the following Java code?

```
int num = 0;
while (num < 5)
{
  System.out.print((5 - num) + " ");
  num = num + 1;
}
System.out.println();
```

a. 5 4 3 2 1 0
b. 5 4 3 2 1
c. 4 3 2 1 0
d. None of these

2. Suppose sum and num are `int` variables, and the input is

```
20 25 10 18 -1
```

What is the output of the following code? (Assume that `console` is a `Scanner` object initialized to the standard input device.)

```
sum = 0;
num = console.nextInt();
while (num != -1)
{
    if (num >= 20)
        sum = sum + num;
    else
        sum = sum - num;
     num = console.nextInt();
}
System.out.println(sum);
```

a. 17
b. 28
c. 45
d. None of these

3. In a `for` loop, which of the following is executed first?
   a. initial expression
   b. logical expression
   c. update expression
   d. for loop statement

```
int x = 0;
for (int i = 0; i < 4; i++);
    x++;
    if (x == 3)
        System.out.println("*");
```

4. What is the output of the code above?
   a. `*`
   b. `**`
   c. `***`
   d. There is no output

5. What is the output of the following Java code?

```
int j;
for (j = 10; j <= 10; j++)
     System.out.print(j + " ");
System.out.println(j);
```

   a. `10`
   b. `10 10`
   c. `10 11`
   d. `11 11`

```
int x = 27;
int y = 10;

do
    x = x / 3;
while (x >= y);
```

6. If `y = 0`, how many times would the loop above execute?
   a. 1
   b. 2
   c. 3
   d. 4

7. What is the output of the following Java code?

```
int num = 10;
boolean found = false;

do
{
    System.out.print(num + " ");
    if (num <= 2)
        found = true;
    else
        num = num - 3;
}
while (num > 0 && !found);

System.out.println();
```

a. 10 7 4
b. 4 7 10
c. 10 7 4 1
d. None of these

Part - II

8) [3 points] What is the output of the following program? Briefly explain your answer.

```
public class Question5
{
public static void main(String args[])
{
  String myString ="Winter is arriving!!!";
  String newString=" ";

  for (int i = myString.length() -1 ; i > 0 ; i -=2)
  {
    newString += myString.charAt(i);
  }
  System.out.println(newString); }
```

Answer:

APPENDIX L

CSE 110 DATASET

ID#	MAJOR	GENDER	ETHNICITY	PRIOR PROGRAMMING EXPERIENCE	UML1	UML2	UML3	CODING1	CODING2	CODING3
191307	CS	Male	White	Yes	0.75	0.00	0.75	0.92	0.83	0.83
191308	CS	Male	White	Yes	0.50	0.88	0.75	0.50	1.00	1.00
191310	CS	Male	White	Yes	0.88	0.88	1.00	1.00	1.00	1.00
191315	CS	Female	White	No	0.75	0.88	0.75	1.00	1.00	0.83
191317	CS	Male	White	No	0.88	0.38	0.88	0.92	0.92	0.83
191318	CS	Male	White	No	0.88	0.75	0.50	0.75	1.00	0.50
191319	CS	Male	non-White	Yes	0.50	0.63	0.88	0.83	1.00	1.00
191320	CS	Male	White	No	0.75	0.63	0.75	1.00	1.00	0.67
191321	CS	Female	non-White	No	1.00	1.00	1.00	0.58	0.67	1.00
191322	CS	Male	White	No	1.00	1.00	0.88	1.00	1.00	1.00
191324	CS	Male	White	Yes	1.00	1.00	0.88	0.67	1.00	1.00
191326	CS	Male	White	No	0.75	0.88	0.88	0.17	1.00	0.67
191328	CS	Male	White	Yes	0.50	0.38	0.25	0.67	0.67	0.67
191330	CS	Male	White	Yes	0.50	0.50	0.63	0.50	0.83	0.50
191331	CS	Female	non-White	Yes	1.00	0.88	0.88	0.17	1.00	1.00
191333	CS	Female	White	No	1.00	0.88	0.88	0.67	1.00	0.83
191334	CS	Male	White	Yes	0.50	0.88	1.00	0.83	1.00	1.00
191335	CS	Female	White	No	0.50	0.63	0.75	0.67	0.42	1.00
191336	CS	Male	non-White	Yes	1.00	0.88	0.75	0.67	1.00	1.00
191338	CS	Male	White	No	0.00	0.38	0.63	0.17	0.83	0.58
191339	CS	Male	White	Yes	0.75	1.00	0.88	1.00	1.00	0.83
191340	CS	Male	White	Yes	0.50	0.75	0.75	0.50	0.75	0.83
191342	CS	Male	White	No	0.25	0.50	0.75	0.00	0.67	0.50
191343	CS	Male	White	No	0.75	0.75	1.00	0.58	1.00	1.00
191344	CS	Male	non-White	No	0.75	1.00	1.00	0.33	0.83	1.00
191345	CS	Female	White	No	1.00	0.75	0.88	0.33	0.92	0.83
191346	CS	Female	non-White	No	0.50	0.38	0.75	0.50	0.33	1.00
191347	CS	Male	non-White	No	0.50	0.50	0.75	0.00	0.50	0.67
191349	CS	Male	non-White	No	0.00	0.25	0.25	0.00	0.58	0.33
191350	CS	Female	non-White	No	0.88	0.75	0.88	0.50	0.50	0.83
191351	non-CS	Male	White	No	0.50	0.75	0.75	1.00	1.00	0.67
191352	CS	Male	non-White	Yes	0.50	0.75	1.00	0.00	1.00	1.00
191353	CS	Male	White	Yes	0.50	0.63	1.00	0.67	0.42	1.00
191354	CS	Male	non-White	Yes	0.75	0.63	0.88	1.00	1.00	1.00
191355	CS	Male	non-White	No	1.00	1.00	0.88	0.67	1.00	0.67
191356	CS	Male	non-White	No	0.00	0.38	0.50	0.92	0.83	0.33
191359	CS	Male	non-White	Yes	0.00	0.25	0.50	0.92	0.00	0.50
191361	non-CS	Female	non-White	No	0.88	0.50	0.25	0.08	0.17	0.00
191362	CS	Male	non-White	Yes	0.25	0.63	0.75	0.58	0.67	0.67
191363	non-CS	Female	White	No	0.25	0.63	0.88	0.33	0.33	0.67
191364	CS	Male	White	Yes	1.00	0.50	0.75	0.33	0.67	0.83
191365	CS	Male	White	Yes	0.00	0.50	1.00	1.00	0.67	0.83
191367	CS	Male	White	No	0.75	0.63	0.88	0.00	0.92	0.83
191368	CS	Female	White	No	0.50	0.50	0.50	0.67	0.83	0.50

APPENDIX M

CSE 100 DATASET

ID#	MAJOR	GENDER	ETHNICITY	PRIOR PROGRAMMING EXPERIENCE	UML1	UML2	CODING1	CODING2
201301	non-CS	Male	White	No	0.63	0.75	0.50	0.33
201302	CS	Female	non-White	No	0.75	1.00	1.00	0.92
201303	CS	Female	non-White	No	0.75	1.00	0.75	1.00
201304	CS	Female	White	No	1.00	0.88	1.00	1.00
201305	CS	Male	non-White	No	0.75	0.88	0.50	0.50
201306	non-CS	Female	non-White	No	0.88	0.88	0.33	0.42
201307	non-CS	Male	White	Yes	1.00	0.75	0.83	0.83
201311	CS	Female	White	No	0.88	1.00	1.00	1.00
201312	CS	Male	White	No	0.25	0.75	0.00	0.25
201313	CS	Female	non-White	No	1.00	1.00	0.75	0.83
201314	non-CS	Female	non-White	No	0.75	1.00	0.33	1.00
201315	CS	Male	White	No	0.75	0.75	0.58	0.83
201319	CS	Male	White	No	0.88	0.75	1.00	0.75
201320	CS	Male	White	No	1.00	1.00	1.00	0.75
201321	non-CS	Male	non-White	No	1.00	0.75	0.67	0.83
201322	CS	Male	White	No	0.75	1.00	1.00	1.00
201323	CS	Female	White	No	1.00	1.00	1.00	0.92
201324	CS	Male	White	No	0.25	0.50	0.50	1.00
201325	CS	Female	non-White	No	0.88	1.00	1.00	1.00
201326	non-CS	Male	White	No	1.00	1.00	0.83	0.83
201328	CS	Male	non-White	No	0.63	0.50	0.33	0.50
201330	CS	Male	non-White	No	1.00	0.75	0.67	0.83
201331	CS	Female	non-White	No	0.75	0.50	0.42	0.33
201332	CS	Male	White	No	1.00	0.75	0.83	1.00
201334	CS	Male	White	No	1.00	0.75	0.67	0.50
201335	CS	Female	non-White	No	1.00	0.88	0.83	1.00
201336	CS	Male	White	No	0.88	1.00	0.83	1.00
201337	CS	Male	non-White	No	1.00	0.88	0.58	0.83
201338	CS	Male	White	No	0.88	1.00	1.00	1.00
201339	CS	Male	non-White	Yes	0.00	0.75	0.33	1.00
201341	CS	Male	non-White	No	0.75	1.00	0.17	0.33
201342	non-CS	Female	non-White	No	1.00	0.75	0.50	0.58
201343	CS	Male	non-White	No	1.00	1.00	1.00	1.00
201344	CS	Male	White	Yes	0.50	1.00	0.67	0.42
201345	CS	Female	White	No	0.88	0.75	0.50	0.58
201347	CS	Male	White	No	0.88	0.88	0.50	1.00
201348	non-CS	Female	White	No	0.88	0.50	0.42	0.33
201349	CS	Male	non-White	Yes	0.50	0.50	0.33	0.25
201350	CS	Male	White	No	0.88	0.75	0.42	0.67
201351	CS	Male	White	No	0.38	0.50	0.33	0.33
201353	CS	Male	White	No	0.75	0.75	0.33	0.33
201356	CS	Male	White	No	0.75	1.00	0.42	1.00
201357	CS	Male	non-White	Yes	0.75	1.00	0.50	0.83
201358	CS	Male	White	No	0.75	1.00	0.75	0.83
201359	CS	Female	White	No	1.00	1.00	0.67	0.83
201360	non-CS	Male	White	No	0.75	0.88	0.17	0.42
201362	CS	Male	non-White	No	0.75	1.00	0.33	0.58
201367	CS	Male	White	No	1.00	1.00	1.00	1.00
201368	CS	Male	White	No	0.88	0.88	0.58	0.17

APPENDIX N

CS1 DATASET

ID#	COURSE	MAJOR	GENDER	ETHNICITY	PRIOR PROGRAMMING EXPERIENCE	UML1	UML2	CODING1	CODING2
191307	Java	CS	Male	White	Yes	0.75	0.00	0.92	0.83
191308	Java	CS	Male	White	Yes	0.50	0.88	0.50	1.00
191310	Java	CS	Male	White	Yes	0.88	0.88	1.00	1.00
191315	Java	CS	Female	White	No	0.75	0.88	1.00	1.00
191317	Java	CS	Male	White	No	0.88	0.38	0.92	0.92
191318	Java	CS	Male	White	No	0.88	0.75	0.75	1.00
191319	Java	CS	Male	non-White	Yes	0.50	0.63	0.83	1.00
191320	Java	CS	Male	White	No	0.75	0.63	1.00	1.00
191321	Java	CS	Female	non-White	No	1.00	1.00	0.58	0.67
191322	Java	CS	Male	White	No	1.00	1.00	1.00	1.00
191324	Java	CS	Male	White	Yes	1.00	1.00	0.67	1.00
191326	Java	CS	Male	White	No	0.75	0.88	0.17	1.00
191328	Java	CS	Male	White	Yes	0.50	0.38	0.67	0.67
191330	Java	CS	Male	White	Yes	0.50	0.50	0.50	0.83
191331	Java	CS	Female	non-White	Yes	1.00	0.88	0.17	1.00
191333	Java	CS	Female	White	No	1.00	0.88	0.67	1.00
191334	Java	CS	Male	White	Yes	0.50	0.88	0.83	1.00
191335	Java	CS	Female	White	No	0.50	0.63	0.67	0.42
191336	Java	CS	Male	non-White	Yes	1.00	0.88	0.67	1.00
191338	Java	CS	Male	White	No	0.00	0.38	0.17	0.83
191339	Java	CS	Male	White	Yes	0.75	1.00	1.00	1.00
191340	Java	CS	Male	White	Yes	0.50	0.75	0.50	0.75
191342	Java	CS	Male	White	No	0.25	0.50	0.00	0.67
191343	Java	CS	Male	White	No	0.75	0.75	0.58	1.00
191344	Java	CS	Male	non-White	No	0.75	1.00	0.33	0.83
191345	Java	CS	Female	White	No	1.00	0.75	0.33	0.92
191348	Java	CS	Female	non-White	No	0.50	0.38	0.50	0.33
191347	Java	CS	Male	non-White	No	0.50	0.50	0.00	0.50
191349	Java	CS	Male	non-White	No	0.00	0.25	0.00	0.58
191350	Java	CS	Female	non-White	No	0.88	0.75	0.50	0.50
191351	Java	nonCS	Male	White	No	0.50	0.75	1.00	1.00
191352	Java	CS	Male	non-White	Yes	0.50	0.75	0.00	1.00
191353	Java	CS	Male	White	Yes	0.50	0.63	0.67	0.42
191354	Java	CS	Male	non-White	Yes	0.75	0.63	1.08	1.00
191355	Java	CS	Male	non-White	No	1.00	1.00	0.67	1.00
191356	Java	CS	Male	non-White	No	0.00	0.38	0.92	0.83
191359	Java	CS	Male	non-White	Yes	0.00	0.25	0.92	0.00
191361	Java	nonCS	Female	non-White	No	0.88	0.50	0.08	0.17
191362	Java	CS	Male	non-White	Yes	0.25	0.63	0.58	0.67
191363	Java	nonCS	Female	White	No	0.25	0.63	0.33	0.33
191364	Java	CS	Male	White	Yes	1.00	0.50	0.33	0.67
191365	Java	CS	Male	White	Yes	0.00	0.50	1.00	0.67
191367	Java	CS	Male	White	No	0.75	0.63	0.00	0.92
191368	Java	CS	Female	White	No	0.50	0.50	0.67	0.83
201301	C++	nonCS	Male	White	No	0.63	0.75	0.50	0.33
201302	C++	CS	Female	non-White	No	0.75	1.00	1.00	0.92
201303	C++	CS	Female	non-White	No	0.75	1.00	0.75	1.00
201304	C++	CS	Female	White	No	1.00	0.88	1.00	1.00
201305	C++	CS	Male	non-White	No	0.75	0.88	0.50	0.50
201308	C++	nonCS	Female	non-White	No	0.88	0.88	0.33	0.42
201307	C++	nonCS	Male	White	Yes	1.00	0.75	0.83	0.83
201311	C++	CS	Female	White	No	0.88	1.00	1.00	1.00
201312	C++	CS	Male	White	No	0.25	0.75	0.00	0.25
201313	C++	CS	Female	non-White	No	1.00	1.00	0.75	0.83
201314	C++	nonCS	Female	non-White	No	0.75	1.00	0.33	1.00
201315	C++	CS	Male	White	No	0.75	0.75	0.58	0.83
201319	C++	CS	Male	White	No	0.88	0.75	1.00	0.75
201320	C++	CS	Male	White	No	1.00	1.00	1.00	0.75
201321	C++	nonCS	Male	non-White	No	1.00	0.75	0.67	0.83
201322	C++	CS	Male	White	No	0.75	1.00	1.00	1.00
201323	C++	CS	Female	White	No	1.00	1.00	1.00	0.92
201324	C++	CS	Male	White	No	0.25	0.50	0.50	1.00
201325	C++	CS	Female	non-White	No	0.88	1.00	1.00	1.00
201328	C++	nonCS	Male	White	No	1.00	1.00	0.83	0.83
201328	C++	CS	Male	non-White	No	0.63	0.50	0.33	0.50
201330	C++	CS	Male	non-White	No	1.00	0.75	0.67	0.83
201331	C++	CS	Female	non-White	No	0.75	0.50	0.42	0.33
201332	C++	CS	Male	White	No	1.00	0.75	0.83	1.00
201334	C++	CS	Male	White	No	1.00	0.75	0.67	0.50
201335	C++	CS	Female	non-White	No	1.00	0.88	0.83	1.00
201336	C++	CS	Male	White	No	0.88	1.00	0.83	1.00
201337	C++	CS	Male	non-White	No	1.00	0.88	0.58	0.83
201338	C++	CS	Male	White	No	0.88	1.00	1.00	1.00
201339	C++	CS	Male	non-White	Yes	0.00	0.75	0.33	1.00
201341	C++	CS	Male	non-White	No	0.75	1.00	0.17	0.33
201342	C++	nonCS	Female	non-White	No	1.00	0.75	0.50	0.58
201343	C++	CS	Male	non-White	No	1.00	1.00	1.00	1.00
201344	C++	CS	Male	White	Yes	0.50	1.00	0.67	0.42
201345	C++	CS	Female	White	No	0.88	0.75	0.50	0.58
201347	C++	CS	Male	White	No	0.88	0.88	0.50	1.00
201348	C++	nonCS	Female	White	No	0.88	0.50	0.42	0.33
201349	C++	CS	Male	non-White	Yes	0.50	0.50	0.33	0.25
201350	C++	CS	Male	White	No	0.88	0.75	0.42	0.67
201351	C++	CS	Male	White	No	0.38	0.50	0.33	0.33
201353	C++	CS	Male	White	No	0.75	0.75	0.33	0.33
201356	C++	CS	Male	White	No	0.75	1.00	0.42	1.00
201357	C++	CS	Male	non-White	Yes	0.75	1.00	0.50	0.83
201358	C++	CS	Male	White	No	0.75	1.00	0.75	0.83
201359	C++	CS	Female	White	No	1.00	1.00	0.67	0.83
201360	C++	nonCS	Male	White	No	0.75	0.88	0.17	0.42
201362	C++	CS	Male	non-White	No	0.75	1.00	0.33	0.58
201367	C++	CS	Male	White	No	1.00	1.00	1.00	1.00
201368	C++	CS	Male	White	No	0.88	0.88	0.58	0.17