A THREE SEMESTER INTRODUCTORY COMPUTER SCIENCE SEQUENCE

Kent White¹, Ray Giguet²

Abstract - ACM guidelines suggest that the introductory Computer Science sequence can be split into three courses, either to introduce more topics, or to cover topics in more depth. We have been applying the second approach for several years. We have found that covering introductory topics in depth greatly reduces the failure rate in upper level courses. We feel that without a minimal level of programming expertise, and the intuition that comes with it, students will have difficulty grasping the advanced concepts, and completing the large software projects, introduced in upper level courses. Our introductory courses therefore stress programming experience, problem-solving skills, object-oriented design, and software engineering principles. Our goal is not to weed out weaker students, but to allow them overcome their weaknesses early in the degree program. An important objective for all three courses is to insure that each introductory class prepares students for the next introductory class. This paper discusses the topics covered in each of the three introductory courses, including sample programming projects designed to give students the tools they need to succeed in upper level courses.

Index Terms – ACM guidelines, Introductory Computer Science Sequence Teaching Computer Science

INTRODUCTION

The introductory computer science sequence has traditionally been split into two courses, corresponding to CS1 and CS2. The new ACM guidelines, however, suggest that the introductory sequence can be spread over three courses, either to introduce more topics or to cover topics in more depth [1]. We have chosen the second approach because we feel that the added depth is necessary for student success. Spreading the introductory sequence over three courses not only allows subject matter to be covered more thoroughly, but perhaps more importantly it increases student “maturity” by an additional semester of work.

For several years, the CMPS program at Nicholls State has averaged close to 100 majors. However, the percentage of majors reaching the junior and senior levels has been relatively low. This was due largely to the low retention rate in our two-course introductory sequence. We felt the students were not getting enough programming experience to have success in the introductory courses and subsequently in the higher-level courses.

We have switched our introductory sequence to a three-semester sequence (CMPS 120, 221, and 222). Since switching to a three semester sequence our students are having a better pass rate in the introductory courses and more importantly the students that are passing the third computer science course have a high graduation rate. When we switched our program requirements to an accredited curriculum our graduation rate for majors was below 50%. Since the Fall of 1999, 80% of Computer Science majors who have passed the three semester introductory course sequence have either graduated or are still in the program. We feel, therefore, that CMPS 120, 221, and 222 are the pivotal courses that determine the number and quality of our graduates. It is therefore essential that students come out of these courses with not only a firm grasp of CS1 and CS2 concepts, but also a level of programming ability that allows them to succeed in later courses.

OVERVIEW OF INTRODUCTORY COURSES

Each of the three introductory courses serves a particular purpose. CMPS 120 is designed to get all our freshmen majors “on the same page”. Nicholls State University has an open admissions policy, and consequently many of our freshmen have poor math and logic skills. We therefore begin by introducing them to problems requiring simple math, logic, and organization. Students are introduced to programming and problem solving methods, and are given extensive guidance and in-class lab time. Overall goal: Students should be able to write a simple program that solves a simple problem.

CMPS 221 serves as an intermediate step, bridging the gap between CS1 and CS2. It allows students to gain maturity through further programming experience. In a sense, this is really the “third” class added to the introductory sequence. Students are given larger problems involving two or three components and are introduced to the concept of abstraction. They receive less guidance and have little in-class work. Overall goal: Students should be able to design and develop programs requiring the coordination of a few simple components.

We consider CMPS 222 to be our gateway course that ensures students are prepared for upper-level courses. Students are given larger, more complex problems requiring the coordination of several components with minimal instructor guidance. Students learn to manage

¹ Kent White, Nicholls State University, Department of Mathematics and Computer Science, PO Box 2168, Thibodaux, La 70310 cmps-kmw@nicholls.edu
² Ray Giguet, Nicholls State University, Department of Mathematics and Computer Science, PO Box 2168, Thibodaux, La 70310 cmps-rmg@nicholls.edu
complexity by abstracting away details to better understand the overall picture. **Overall goal:** Students should be able to analyze problems, design solutions, and implement programs on their own.

**Skills Stressed**

The problems students have in the initial courses are due in large part to the introduction of abstract concepts such as algorithm design, runtime complexity, object-orientation, recursion, and the notion of “abstraction” itself. To many students this requires a new way of thinking, something not called for in their other freshmen and sophomore classes.

Teaching students to think abstractly is difficult without first stressing concrete examples. For instance, after programming several loops, a student is better able to generalize the concept of “repetition”. This means that programming experience, though not the main emphasis of the introductory courses, is vital to understanding higher-level concepts. In addition, upper-level courses require relatively large programs designed to teach complex subject matter. If juniors and seniors do no have adequate programming skills, such assignments are not feasible.

Our students use Java as a tool to implement problem-solving strategies. We find, however, that students have difficulty learning how to program without first developing strong problem-solving skills. In other words, in order to abstract, students must have programming experience, and in order to program well they must be able to solve problems.

We therefore attempt to build student skills in three stages:

1. **Problem-solving skills:** including problem definition and solution design, and the ability to create efficient algorithms.
2. **Software engineering skills:** including proper programming style, design, development, testing, debugging, and general programming experience.
3. **Abstraction skills:** including understanding object-orientation, developing abstract data structures, and the ability to generalize problems and solutions.

Each of the introductory courses stresses all three skills, but at a higher level than the previous course. Thus the problems assigned in 120 are designed to help students to learn the programming concepts and understand the abstractions introduced in that class.

**DETAILED COURSE DESCRIPTIONS**

**CMPS 120**

**Problem-solving skills**

The students are first given problems that involve simple arithmetic calculations (AC). They are then introduced to the topic of decision statements in solving more complex problem definitions (DS). Next, the students are shown two basic looping constructs using the counting loop and the sentinel loop (LC). We then introduce the students to the concept of modularity (MD). The students are given problems that reinforce the skills learned earlier but now are required to break a program up into simple modules. The last problem solving skill introduced to the students in CMPS 120 is the concept of arrays (AR).

**Software engineering skills**

Students are introduced to a program development environment (JBuilder), using basic I/O, test cases, and pseudocode, then they lean how to develop a program-structure template, and the importance of good programming style, top-down design, and modularity. The first student assignments are strictly written in pseudocode. They are then shown how to translate their pseudocode solutions into Java.

**Abstraction skills**

Abstraction skills learned include the development of algorithms and flowcharts, programming constructs such as sequence, selection, repetition, and modules, and the use of memory, variables and arrays.

**Assignment Examples**

Sample problems include calculating the area of a rectangle (AC), finding the distance traveled given the rate and time (AC), determining if a year is a leap year (DS), calculating a person’s pay including overtime (DS), summing the numbers from 1 to N (LC), calculating N factorial (LC), finding the smallest common divisor of two integers (LC), finding if a number is prime (LC), sorting an array using bubble sort (AR), and finding the difference each score in a set is from the average (AR)[2].

For example, the students are required to determine the day number for a particular date. The month, day, and year are given as integers. The output for the problem requires them to output the name of the month, the day, the year, what day of the year is it, and whether or not the year is a leap year. The students are required to use methods to help them in solving the overall problem. Following is an example of the pseudocode solution to the problem:

```java
Get Date(month, day, year all integers)
leap = is leap year(year)
dayNumber = calculate day number( month, day, leap)
monthStr = get month name(month)
display(monthStr, day, year, dayNumber, leap)
```

0-7803-7961-6/03/$17.00 © 2003 IEEE

November 5-8, 2003, Boulder, CO

33rd ASEE/IEEE Frontiers in Education Conference

T4C-7
CMPS 221

Problem-solving skills

The following topics are covered in CMPS 221: Single and multidimensional arrays (SM), searching (SE), user interface design (UD), object-based design (OB), object oriented design (OO), polymorphism (PM), and singly linked list (SL).

Software engineering skills

Student’s software engineering skills developed in this class include using multiple classes, understanding variable scope and the importance of local variables, exceptions, parameter passing, creating modular programs using functions and methods, algorithm efficiency, and object-oriented design.

Abstraction skills

Abstraction skills developed include algorithm efficiency, 2d arrays, objects & inheritance, introduction to recursion and recursive data structures (such as a linked list), encapsulation, and re-usability

Assignment Examples

Some assignment examples are: Using selection, insertion, and quicksort compare the actual and predicted times of sorting array sizes of 100 to 1,000,000 items (SM). This assignment requires students to analyze various algorithms on their efficiency at solving a problem. This assignment is usually followed by having the students compare linear and binary search on various size arrays with various number of items to search for (SM). Again, the students are required to analyze the complexity of their solutions. The next assignment given students in CMPS 221 involve using array addition, subtraction, and dot product (SM). The students are then introduced to multidimensional arrays and are required to add, subtract, and multiply matrices using the methods derived in the previous assignment (OB). Simple graphics programming to teach interface design and the basics of drawing graphical objects is given next (UD). The students are then introduced to the concept of polymorphism by writing a simple drawing program to draw various types of graphic primitives (PM)[3]. Students then extend the basic drawing program to include the ability to add lines, rectangles, ovals, polygons, text, and free hand lines (AR, UD, OO, PM, SL).

CMPS 222

Problem-solving skills

CMPS topics include exhaustive searching (ES), backtracking (BT), storing and retrieving data (SR), organizing large amount of data (OD), and searching through partially (PO) and completely ordered (CO) data[4].

Software engineering skills

Skills learned include program specification, algorithm design, object-oriented design, and strategies for program development and testing.

Abstraction skills

Abstraction skills include understanding why and how object-orientation is useful, O. O. concepts such as data protection, localization of effects, the definition of abstract data structures such as stacks (ST), lists (LL), queues, graphs (GR), trees (TR), heaps (HP), and hash tables (HT), and other skills such as the use of recursion, pre-and post-conditions, and the ability to “abstract out” details to see similarities in different problems.

Assignment Example

Assignments include the following: create a palindrome by adding and deleting from both ends of a sentence, then read the sentence backward and forward to make sure a palindrome has been created (LL); Create a family tree and then determine if one person is a direct descendant of another (GR, ST, ES, BT); Create a priority queue and use it to store and retrieve information about a singing group’s albums (HP, PO); Guess a particular 4-letter code, consisting only of B’s and W’s, by systematically generating all possible 4-letter permutations of Bs and Ws (TR, SR); Create a large student directory then add, delete, and locate information on particular students (HT, OD).

In one assignment, students play a game in which they guide an explorer through a maze filled with traps. Their explorer can do three things: move to an adjacent room, search a doorway for a trap, and remove a trap (using a stick of dynamite). The explorer has a map and three sticks of dynamite.

After playing the game, students write a specification describing its functions and requirements. Next they define a winning strategy, first in English, then in the form of a general algorithm. Next they design the necessary Java classes (they are required to have a Graph class that uses an adjacency matrix). Finally they code and test their program. The student’s program must mimic the game, but requires no input, and outputs only text.
Most of the methods are simple, but implementing a clever strategy can be challenging. A smart explorer, for instance, will minimize backtracking and will be able to calculate the shortest path between any two previously explored rooms.

The student’s program requires four classes: a Graph class, a Maze class, an Explorer class, and an application class. Each class except the application must be application-independent, to increase its reusability. The Graph class, for instance, is used in a subsequent assignment to implement a family tree. The Graph class is a sub-class of the Maze class, and the Maze class is a sub-class of the Explorer class, so students must be able to create and manage a layered program structure.

Students must code and test each component (i.e., object class) separately, one method at a time. This requires creating “toy” applications consisting of student-defined test cases for each method. Only after all components have been separately tested can they be combined to test the completed implementation.

CONCLUSION

Since switching to the three-semester sequence our students are better prepared for the rigors of the upper level computer science courses. They gain valuable experience and maturity by being able to work on a wide variety of projects ranging from the very simple to complicated.

BIBLIOGRAPHY