Why the High Attrition Rate for Computer Science Students: Some Thoughts and Observations

Theresa Beaubouef
John Mason
Department of Computer Science
Southeastern Louisiana University
Hammond, Louisiana 70402
{tbeaubouef, jmason}@selu.edu

Abstract
This paper investigates the possible causes for high attrition rates for Computer Science students. It is a serious problem in universities that must be addressed if the need for technologically competent professionals is to be met.

Keywords: attrition, retention, computer science education

1. Introduction
At our university, there are over four hundred declared majors in Computer Science. Each semester, however, only about fifteen to twenty students graduate in this field. The freshman courses comprise overflowing multiple sections, but the upper level courses make two sections at most, usually one, and the numbers are such that many elective offerings are not possible. During advising and pre-registration, it often seems that the advisors sign as many change-of-major forms as they do schedule approvals for registration.

What happens between the time that a student decides to major in computer science and the time he or she drops out of the program? There are many reasons why students drop, resign, or transfer to other universities. After all, every major suffers attrition from these causes. They may be based on personal relationships, illness, financial hardship, military service, or outside employment. Additional factors affect computer science retention, however, and several of these are discussed in following sections.

It should be noted that most of the attrition happens during (or between) the freshman and sophomore years. Drop rates as high as 30 to 40% are reported by many institutions, and are rapidly becoming the norm for computer science programs. Therefore, much of the following discussion is related to CS1/CS2 issues.

2. Poor Advising Before and During College
When asked why students choose to major in computer science, a variety of not necessarily good reasons are revealed. Sometimes students are directed into computer science because high school guidance counselors, parents, and the students themselves often have a less than perfect idea about the students' aptitudes. They feel that "everything is going to computers" so students "can't go wrong by majoring in computer science. Many times these same students were not advised to take more than the minimum required math, language, and science courses, and therefore, are less than prepared to begin college in this major.

Students often have misconceptions about the field of computer science. Many of them take a computer literacy course, do well in it, and believe that computer science is all about using word processors, spreadsheets, or web browsers. Because they do well, they are encouraged to take additional computer courses. Many other students like to play video games, so their dreams are to become video game programmers. They more often than not do not realize the mathematics and computer skills necessary for such endeavors.

Advising during college also plays a significant part in the student's ability to succeed. One of the authors taught a section of Computer Architecture in which one of the students had never taken a computer science course! His advisor had placed him in the course "because he needed a 300 level architecture course." Other students at a university where the authors worked previously were often advised to take a computer language in place of a foreign language because CS1 and CS2 could substitute for a foreign language, and learning a foreign language is difficult. Many were unpleasantly surprised to discover that CS1proved to be even more difficult.

3. Poor Math Skills and Problem Solving Abilities

Mathematics is an integral part of computer science and because of this, students must develop a good understanding of mathematics in order to be successful in computer science. It was shown in [1] how mathematics relates to a typical curriculum in computer science and in the professional area. Students who are adept at math tend to perform better in computer science. They are better able to understand relationships in data, scientific computations, and algorithm design. This allows them to be better at solving problems and generating good designs from requirements.

The ability to solve problems is often very weak in computer science students, and Beaubouef, et. al. [3] have investigated ways to improve problem solving skills in beginning computer science students. It most often manifests itself as the inability to solve basic word problems. While the ability to solve a word problem is a math skill, word problems play a vital role in most CS1 courses. The authors have found that students that might do very well on multiple choice tests may not perform nearly as well on tests that present a simple real world problem and ask the students to develop a basic C++ (or Java) program.

Careful retesting revealed that syntactical knowledge was not the problem. Students who did well on multiple choice tests could successfully "hand trace" syntactically correct code, spot syntax errors in incorrect code, and successfully implement a detailed algorithm given in simple English. The difficulty was the inability to form the algorithm in the first place.

Naturally, the ability to formulate a procedural algorithm (object oriented approaches are covered below) is independent of syntax; indeed, it is independent of the language chosen at the CS1 level. This is a skill that can be taught. It is alarming to realize that many entering computer science students do not have this skill at even a rudimentary level. The first one to three weeks of our CS1 courses are spent teaching basic problem solving.

This is a painful "Catch-22" situation. Donald Knuth has said that "CS = problem solving". Much of what we do is teaching students to teach a computer to solve a problem. Nonetheless, rudimentary problem solving is a prerequisite for CS1, therefore teaching it is outside the scope of the course. Nevertheless, failing to teach these skills will definitely result in more students dropping from the course, therefore exacerbating the problem of high drop rates. The authors suspect that requiring students to take a course in elementary problem solving and natural language algorithm design would result in better CS1 grades and higher retention rates in the first two years of the computer science curriculum.

4. Poorly Designed CS1 Lab Courses

Few people would argue that laboratory courses are an essential part of CS1 / CS2 courses. Denning, et.al. [5] espouse the view that "Computing sits at the crossroads among the central processes of applied mathematics, science, and engineering. The three processes are of equal - and fundamental - importance in the discipline, which is a unique blend of interaction among theory, abstraction, and design."

Given the critical importance of lab courses for the beginning programmer, many universities do a poor job developing labs. Walker [8] notes that, for many universities, CS1 lab periods are "used for student homework assignment completion." This may seem reasonable at first glance, especially since beginning programmers need all the practice they can get.

Nonetheless, this approach is often detrimental. Walker further notes that "homework-centric labs tend to degenerate to teacher-assisted debugging sessions that are frustrating to the students and teachers. These debugging sessions do not develop programming skills and are not conducive to close teacher-student interaction." It is possible that one student can "consume the teacher's attention for the majority of a lab period." Meanwhile, other students are forced to wait for their turn to speak to the teacher or lab assistant. The natural conclusion is that "programming skills are not developed in these debugging sessions. Program planning and logic take a backseat to code-compile-debug cycles." Walker provides several excellent suggestions for improving current CS1 labs.

5. Lack of Practice / Feedback

Almost every academician would agree that practice really does make perfect, yet in a typical seventeen-week semester, it is rare for an introductory course to include more than twelve to fifteen programming assignments. There are a variety of reasons for this, the predominant one being that CS1 courses typically have very large sections. Grading a large number of programs requires a large amount of time, or an automated grading system.

While many instructors prefer automated grading systems, some instructors (including the authors) are philosophically opposed to them. Students have a right to have a knowledgeable programmer look at their code and provide meaningful feedback. Automated systems can't provide this kind of feedback. Moreover, it cannot provide the beneficial feedback to the instructor so that he/she gains understanding of those areas in which students may need additional practice or instruction.

By the same token, many instructors would give more tests and "pop quizzes" if doing so did not require a substantial time investment. While many tests and quizzes can be made multiple choice, other types require time intensive "hand grading". As class sizes increase to save university dollars, the amount of time taken per student for grading must, by necessity, decrease. This is less of a problem at larger institutions that have graduate students available to faculty, since graduate students are generally more than qualified to grade CS1 / CS2 programs and quizzes, even if they are not yet ready to teach them.

6. Graduate Student Teachers

Many universities have absolutely no compunction about allowing graduate students to teach low level CS courses. In fact, many graduate students *are* capable of teaching CS1 / CS2 courses, but many graduate students are thrown into a large section of introductory programming courses with little or no training. While every graduate student in computer science should possess the technical knowledge to teach the course, he may be completely unprepared to speak in front of a classroom, especially if he suffers from shyness or struggles with the English language. Spoken and written language is important for everyone, and mastery of language is essential for all in the computer science profession [2]. If the person instructing the class has language problems, then the quality of instruction is compromised, and students are missing out on essential learning opportunities. Furthermore, many undergraduates do not show graduate students the same respect they would show an instructor or a professor. Few would argue that being an effective teacher requires more than mastery of the subject matter.

Large universities are especially guilty of this shabby treatment of students and graduate students. One of the authors was a second semester junior (i.e. 6 semesters into the computer science degree) before having a professor in the classroom. Every CS course below the 4000 level was taught by a graduate student. This can sometimes send a message to students that they are not important, which when added to the stress of a demanding major may be too much for some to endure.

7. Poor Project Management Skills

Most students who leave computer science programs do so in the first or second year. These younger students have often not developed adequate study habits. However, even those students having good study habits often lack some important project management skills, which are necessary to get programming assignments completed in a timely manner.

A major problem students (and quite a few professionals) have is in estimating the time required to successfully develop a piece of software. This time minimally includes the processes of analysis, design, coding, testing, and documentation. Software projects developed by professionals are notoriously behind schedule in the real world [9]. It should come as no surprise that software developed (programs written) by students will tend to be even more behind schedule.

Unsuccessful students often want to skip analysis and design and begin typing in code immediately. Documentation is an afterthought at best, and little or no testing is performed. Because the student planned to attack the assignment in this manner from the beginning, he will often wait until the last minute to begin and work until its done or time runs out. These students set themselves up for frustration, unnecessary rework, and failure.

Better prepared students, although still not good judges of time and effort involved, begin projects earlier and usually complete them on time. These students, however, may still feel overwhelmed and frustrated when projects are not completed as quickly as they had anticipated due to poor understanding of the requirements or poor design.

In addition to having good time management skills, computer science students must also learn to effectively manage resources. They must have the required CDs, disks, folders, etc. on hand when they are needed. They should back up important work to avoid loss of effort. Additionally, students who require computers in the lab must be able to schedule their work when the computers are available and the lab has minimal distractions. Those students who work from home have additional equipment management responsibilities. Whether in the lab or at home, computers break, networks go down, and power and telephone outages occur. Students must develop skills to best manage their time and resources in less than perfect working environments.

It often takes several semesters of programming and dealing with real world issues before students attain the necessary judgment to be able to manage time and resources wisely. Compared to coursework in other disciplines, computer science can seem very demanding, time consuming, and frustrating. Students that aren't completely dedicated will often become discouraged and switch to less demanding majors.

8. Choice of Language and Objects Early vs. Objects Late

On the surface, this may seem like a ridiculous consideration, since whether we teach objects early (before traditional procedural programming) or later (after procedural programming) is surely merely a matter of personal choice. Nonetheless, there is increasing evidence that where (and how) we teach objects is a matter of great importance. McConnell and Burhans [7] studied introductory CS textbooks over a three decade period. They noted that older textbooks (on procedural programming languages) averaged under 500 pages. For example, the average Fortran text was 379 pages. More modern textbooks are considerably larger, with the average Java text weighing in at a whopping 866 pages.

Perhaps more alarmingly newer textbooks contain less coverage (on average) on simple data types, arithmetic expressions, relational and logic expressions, repetition statements, subprograms, and arrays. McConnell and Burhans conclude that they "can see a disturbing trend in the coverage of both basic programming constructs and subprograms. There has been about a 40% decrease in the coverage of basic programming constructs and a 62% decrease in the coverage of subprogram issues. At the same time, there has been a close to 300% increase in input/output issues, due solely to the inclusion of GUI issues. An increase this large must surely have an impact on what can reasonably be taught in a CS1 course."

Further evidence that many professors are concerned about whether objects should be taught early or late (or even at all) in a CS1 course came in the form of a series of exchanges on the SIGCSE mailing list in March of 2004. The discussion was so intense that Bruce [4] wrote an article that summarizes much of the academic debate.

Much of the discussion began when Elliot Koffman posted a message stating that many faculty refuse to adopt methods that emphasize teaching objects very early in introductory courses. Koffman compares teaching Java to the "new math" experiment of the 1960s. He opined that the major cause of this refusal is simply that many faculty do not posses the proper background to teach object-oriented programming. Thus they fall back on what they do know, and teach procedurally. On the other hand, he notes that "weaker" CS1 students struggle to master the additional layers of abstraction that result from using objects and GUIs, but states that "good" students have no difficulty with the objects first approach.

Stuart Reges made a post entitled "The new CS" that picked up on Koffman's "new math" analogy. Reges wonders when the "the discussion … switched away from WHETHER to teach objects early … to the question of HOW to teach objects early." He challenged proponents of the objects early camp to demonstrate that:

1. A broad range of teachers can teach objects early well.
2. A broad range of students can master it, and
3. The object early approach solves more problems than it creates.

As might be expected, this post led to considerable debate among CS1 teachers. Many teachers are convinced that teaching objects early poses more problems than it solves. While Bruce himself is in the objects early camp, even he notes that "the one thing that there was near universal agreement on during the discussion is that it is a challenge to teach objects early." Clearly this is a highly controversial topic that isn't likely to go away soon.

9. Conclusion

In this paper we discuss the very real problem of attrition in computer science programs. There are several reasons for poor retention of students. We discussed many of the areas that we consider relevant to all university computer science programs. It should be noted however, that other factors such as the transferring of students to other nearby universities, poor high schools, unpopular faculty members, administrative reorganizations, etc., may also be significant factors for some institutions.

While we address some important issues for computer science retention in this paper, we do not discuss solutions to these problems here. It is hoped that by identifying some of the major reasons for high attrition rates among students, these areas can be further studied, and efforts can be made to reduce the high attrition rates.

References

[1] Beaubouef, T., "Why Computer Science Students Need Math," *SIGCSE Bulletin (inroads*), vol. 34 No. 4, December 2002.

[2] Beaubouef, T., "Why Computer Science Students Need Language," *SIGCSE Bulletin (inroads)*, Vol. 35, No. 4, December, 2003.

[3] Beaubouef, T., R. Lucas, and J. Howatt, "The Unlock System: Enhancing Problem Solving Skills in CS1 Students," *SIGCSE Bulletin (inroads)*, vol. 33, no. 2, pp. 43-46, June, 2001.

[4] Bruce, K. "Controversy on How to Teach CS1: A Discussion on the SIGCSE-members Mailing List". *SIGCSE Bulletin (inroads)*, Dec. 2004 (36:4). pp. 29-34.

[5] Denning, P. (ed) "Computing as a Discipline". *Communications of the ACM*, Jan. 1989 (32:1). pp. 202-210.

[6] Kendall, K. and Kendall, J. *Systems Analysis and Design*, Prentice Hall, New Jersey, 1999.

[7] McConnell, J. and Burhans, D. "The Evolution of CS1 Textbooks. Proceedings of Frontiers in Education. 2002. pp. T4G-1-T4G-6

[8] Walker, G. "Experimentation in the Computer Programming Lab". Inroads, Dec. 2004 (36:4). pp 69-72.

[9] Whitten, J. and Bentley, L. *Systems Analysis and Design Methods*, Irwin McGraw-Hill, Boston, 1998.