# Perceptions of Foundational Knowledge by Computer Science Students

Katharine Blanchard
Dept. of Communication
1280 Main Street West
Hamilton, ON. L8S4K1, Canada
McMaster University
blanchkg@mcmaster.ca

Michael Soltys
Dept. of Computing & Software
1280 Main Street West
Hamilton, ON. L8S4K1, Canada
McMaster University
soltys@mcmaster.ca

## ABSTRACT

In this paper we are concerned with computer science students' perceptions of foundational knowledge, understood as the mathematical underpinnings of the field. We review recent literature on the subject, propose an approach for teaching foundational knowledge, and finally present a case study where we analyze the merits of our approach. We make our observations based on experience and on a student survey.

## Categories and Subject Descriptors

K.3.2 [**Computers and Eduction**]: Curriculum

## General Terms

Theory, Human Factors

## Keywords

Curriculum, Foundations, Theory, Perceptions, Satisfaction

## 1. INTRODUCTION

Computer Science (CS) and Software Engineering (SE) have become fashionable fields of study due to the high employability of its graduates. From software development to system administration, the job market has an insatiable appetite for highly skilled individuals. However, after the dot-com bubble burst in 2001, CS enrollments decreased dramatically. Enrollment has since recovered, and while it never rose to the levels of its halcyon days[1], yet it is now encouragingly high.

[1]The U.S. Computer Research Association has tracked enrollments and graduation rates of CS students for the last 40 years. The Association's April 2011 report shows that during the dot-com era enrollments swelled—in 2001 the average enrollment in CS departments in U.S. universities was 398, and in 2007 it dropped by half. Since then enrollments average 253 students per department (see [21]).


*WCCCE'12,* May 4–5, 2012, Vancouver, British Columbia, Canada.


Expectations and perceptions of the field evolve with new trends in technology; as computer technology changes, perceptions of CS change as well. For example, the "Millennials," also called the "Nintendo generation" in [9], have grown up with Xbox and Nintendo, and often envision CS as a branch of the entertainment industry. The student generation arising from the "Millennials" is perceived to be less satisfied with the curriculum. These students are seen as surprised with the amount of mathematics and theory that is involved in the field. "Students often have misconceptions about the field of computer science... other students like to play video games, so their dreams are to become video game programmers. They often do not realize the mathematics and computer skills necessary for such endeavors" [3].

We propose that the view of CS and SE students as "math-averse" is not accurate. While many of the "Millennials" are surprised by the amount of foundational/mathematical material, a solid majority of the students are satisfied with the curriculum. The enrollment in CS, despite the ups and downs in the last decade, remains high. While many students express dissatisfaction with foundational material (enrollment being high, there will be a portion of dissatisfied students), our survey shows that more than three-quarters of the students are quite satisfied with the math content, and realize its importance. In turn, it is crucial for educators to realize two things: the intrinsic importance of theoretical material, and the fact that it is well received when properly taught and motivated.

Administrators and educators alike, when faced with dissatisfaction in the student ranks, may be too quick to diagnose the problem as "too much mathematics too early." But, for example, a 2006 study from the University of Washington CS and Engineering Department ([19]) reports that after the dot-com collapse of 2001 they were facing "a collective struggle" to figure out what to teach and how to teach it. They named several common problems that prompted the study: (i) a decline in student satisfaction and enrollment in introductory courses; (ii) a decline in applicants to the major, especially women; (iii) inconsistency in teaching among instructors; (iv) a lack of basic programming skills reported by upper-division instructors.

The study found that after redesigning their introductory CS courses to emphasize problem solving, procedural decomposition and mastery of basic skills, enrollments increased, satisfaction rates rose, and more women entered the program. While the curriculum changes were only in place for

four semesters at the time of the study, early results showed a marked increase in satisfaction and higher evaluations[2]. In fact, rather than reduce mathematics to increase student satisfaction, they increased the mathematical content; the two seemed *directly* rather than *inversely* proportional.

These findings corroborate the experience of the second author: a "problem solving" approach, where foundational knowledge is transmitted with examples of applications to current technology, yields high levels of satisfaction among students. It is important to stress to students that, if they learn fundamentals, they will have the skills to master the latest technology on their own. The latest technology should be presented, but as an example rather than to build the curriculum around it, as it will change by the time the students graduate. However, a solid foundational background will give them the ability to learn new technologies as they arise, for the rest of their careers.

A longitudinal study performed by the Rochester Institute of Technology ([11]) surveyed undergraduate students of CS over a three-year period. The study examined students' needs based on several measures including their previous computer background, knowledge of programming languages, what attracted them to a degree in CS, demography and finally personal learning style. The research found that students often had a different idea of what CS studies entailed prior to entering the program and this—among other factors—led to a high attrition rate in the program. No student enters a program as a *tabula rasa*, and to a certain extent universities have no control over incoming students' preconceptions. On the other hand, it is important to advertise programs honestly, so that students know what to expect, but also so that students who enroll for "mercenary reasons"—without a penchant for the field—have a chance to re-examine their decision.

There are many studies that cover the issues discussed in this introduction. For a different context see [22], a 2007 study conducted at the University of Edinburgh Business School. The purpose of the study was to investigate the educators' perceptions as to the primary purpose of undergraduate degrees: "theory or practice?" Other relevant studies are given in [10], [14] and [18]. An interesting article about the "cool factor" of computer science, especially in light of the 2010 movie about Facebook, "The Social Network," can be found in [15]. Finally, the issues discussed in this paper have been the subject of intense debate since a long time; see for example [7].

## 2. FOUNDATIONAL MATERIAL AND SATISFACTION STUDY

This paper is concerned with the teaching of foundational knowledge to computer scientists and software engineers, and especially with the perceptions that these students form of foundations. As was written in [17][3]:

> Every Engineer must understand the properties of the materials that they use. Whether it be concrete, steel, or electronic components, the materials available are limited in their capabilities and an Engineer cannot be sure that a product is "fit for use" unless those limitations are known and have been taken into consideration. The properties of physical products can be divided into two classes: (1) technological properties, such as rigidity, which apply to specific products and will change with new developments, (2) fundamental properties, such as Maxwell's laws or Newton's laws, which will not change with improved technology.

There is no doubt that a solid grounding in mathematics, especially discrete mathematics, or what is known as the "theoretical foundations of computer science" is necessary in any curriculum. Much has been written on this need, and on the methodology to teach the mathematical foundations of computation; an interesting perspective on imbuing computer science with mathematics can be found in [20].

However, comparatively little has been written on the subject of "student satisfaction," which, as we understand it in this paper, is not a way to "sell" a mathematically biased curriculum to our undergraduates, but rather a pedagogical *savoir faire* of transmitting to the students the value of foundations knowledge. How can we communicate to the "Nintendo Generation" the importance of propositional logic, partial orders or Turing machines?

If we are to focus on student satisfaction as in regard to their perception of the mixture of theory and practice we must examine the possible scenarios: whether the students, by and large, are satisfied with the foundational knowledge that they receive (in terms of both perceived relevance and amount), and whether a particular curriculum serves the right mix of foundations and practical examples, or does not. We have the four possible scenarios:

| | right mix | not right mix |
|---|---|---|
| satisfied | 1 | 2 |
| not satisfied | 3 | 4 |

1 and 4 are the "to be expected" cases. The cognitive dissonance is in cases 2 and 3. Lets consider case 3 first: suppose that the curriculum carries the right amount of theory and practice, but that the students are not satisfied. This is the case that interests us the most, as it is often seen as the most common situation. The reasons can be any combination of the following four.

(i) "Legacy prejudice." There is a certain undergraduate culture that disparages theoretical material. Previous year students repeat the mantra of "useless material" about a certain subject, and an attitude among current students is formed. It is difficult to break through it; an instructor might start a class with typical examples of the importance of the "propositional satisfiability" problem, but it is difficult to change a culture of perception with a speech.

(ii) The instructors have not provided the right justification; the right focus, or the right examples. In short, they have not made a case for the importance of the theoretical material that is being presented.

(iii) The nature of education is to bring students out of "ignorance" into "enlightenment." The students are not satisfied to the degree that they are ignorant, i.e., they do not know what is good for them. This is an elitist view, held by some but rejected by most instructors.

(iv) "Customer mentality," that is, we—the university—offer a service, and the students are our customers. As any salesman we woo them with promises, and it is in the nature of a customer to "whine" a little bit, that is, to place the

[2]In a personal communication, the author expects to work on a follow-up, but the statistics can be seen at `http://www.cs.washington.edu/homes/reges/stats.pdf`

[3]The longer version of this appeared in [16].

psychological burden of the inconvenience of having to work intellectually on the professor, who after all is getting paid "because I toil." This is a cynical view, embraced by some professors as a reaction against the view of universities as just a business.

We believe that (i) and (ii) are the best ways to focus on the problem. There is little, in the short term, that can be done about (i), but theoretical knowledge can be justified with excellent examples of relevance. For example, instead of presenting the dynamic programming problem of finding the shortest path in a graph, focus on computing routing tables in a TCP/IP protocol; the mathematical content is the same, but the practical focus makes all the difference to the students. This is discussed more in depth in the next section.

The second case is (ii) where the mix is not right; there are two possibilities here: too much theory, or too much practice. Universities by their very nature will likely sway in the direction of too much theory. Academics are interested in the way things are, in generalizations and theories; elegant thinking is more important than the necessarily more "messy" world of industrial applications. Most curricula offer a good mixture of both; however, as [19] points out, a common problem is the inconsistency of what is being taught (inconsistency between different instructors, and inconsistency between instructors and the curriculum).

In 2010/11 [4] conducted a satisfaction study of 100 students in the department of Computing & Software at McMaster University. These were second year students in a software development course, and fourth year students in a networks & security course, both courses taught by the second author. A correlation was found in that the highest level of satisfaction came from the students who intended to go to graduate school (60% "very satisfied" and 20% "satisfied") while the lowest level of satisfaction came from students whose intended career path was to become a consultant; see figure 1. Interestingly enough, the study [22] mentioned in the introduction had almost identical findings. These findings were to be expected as the curriculum for both courses had a lot of foundational/theoretical material, and it is natural that students with a penchant for theory would form a desire to stay longer in academia.

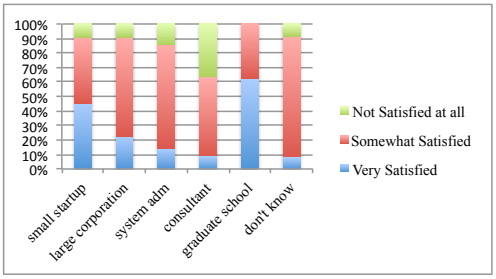


**Figure 1: Correlation between program satisfaction and intended career path (100 students).**

In terms of the perceptions of the program that the students formed prior to entering university, [4] found that the students who felt that they had the most accurate mental picture of what to expect, were students who visited the campus and spoke with a person (a professor, a counselor, a student). The students who felt that they had the least accurate picture of the program were those who formed their opinion of CS and/or Software Engineering based on the material on the departmental web site.

Of course, it would also be interesting to measure the accuracy of students' perception regarding their future career path. That is, do students understand what graduate school is about, what is involved in working for a large corporation such as Google, or what it means to be a system administrator? Fourth year students are more mature, and one hopes that their expectations are realistic, but there is no data to support this. We have some anecdotal evidence (mostly emails from former students) that they were surprised by the usefulness of foundational knowledge once they entered the work place, but unfortunately, it is not enough to form a robust opinion. Nevertheless, [4] also measured the perceptions of the theoretical content by asking students "how relevant they felt foundations were to their intended professions." Surprisingly, 53% agreed that mathematics was relevant to their intended profession, while 21% disagreed. 62% stated that the foundational knowledge they learned was "very valuable," while 33% believed that it was "partially valuable." These are encouraging findings.

## 3. A CASE STUDY

The results in the previous section were gathered while the second author was teaching two courses in the second term of 2010/11: software design and networks & security. In this section we offer some practical observations based on the experience with these two courses, supported by a survey administered to this group of students, and the resulting conclusions in [4].

The teaching philosophy that we propose is that of a "problem solving" approach, where foundational knowledge is transmitted with examples of applications to current technology. We believe that this is the best way to teach theoretical material, as it yields high levels of satisfaction among students, and prepares them for the job market by giving them the tools to master new material and technologies on their own. We list below the assignments given in these courses, which were designed to reflect this philosophy.

Mentioning current events related to the lecture material is also a fantastic stab at the elusive "relevance." For example, the movie *The social network* (2010), i.e., the "Facebook movie," came out last year and it prompted a discussion of social networks, search engines, and gave an opportunity to "defend" graphs, and incidence matrices and eigenvalues, as so much of the technology is based on that concept (see, for example, "The Anatomy of a Large-Scale Hypertextual Web Search Engine" by Sergey Brin and Lawrence Page—the founders of Google [5]). Another event was the IBM's "Watson" that pitted man versus machine on Jeopardy. These vast projects can be used to illustrate almost every aspect of the theoretical underpinnings of CS: for example, social networks are best explained with graph theory[4], and IBM's "Watson" can be used to motivate parsing (and many other topics)[5]. It is very surprising that graph theory appears to

[4]See, for example, [13], pages 75–76.

[5]IBM has produced fantastic video segments explaining the science behind "Watson":
`http://www-03.ibm.com/innovation/us/watson/what-is-watson/science-behind-an-answer.html`

be a lost art, at a time when it is the conceptual framework for Internet data mining and social networks.

In the following case study, each of the two courses were built around six tests and three programming assignments. The tests were given every two weeks, and they consisted of three questions: one question about concepts and definitions, and two questions where the students had to solve problems (which were simple, as they had only 50 minutes for each test). The frequency of the tests prompted the students to stay on top of the material, and gave them ample practice in solving problems, as they were given practice problems first—and they were motivated to do them in order to prepare for the tests.

The assignments had a strong algorithmic flavor, and consisted of three main components: (i) a simple user interface, (ii) an algorithm, and (iii) displaying an output. We used Python 2.6. In the past we selected one of C, C++, Java and Perl as the programming language, but this year we decided to use Python. The advantage of Python is that it is an easy language to learn, and there is a magnificent book, with an early version (1.1.22) available for free as a PDF file on the web [8]. The disadvantage of Python, as we found out, was that it is very much version dependent (2.6 versus 2.7 or 3.0) and platform depended (a Windows machine versus Mac OS X or Linux). The same program would not run with different versions or platforms. This resulted in a lot of headaches for the teaching assistants and for students who would submit a version confident that their program runs, and get back a grade of zero. The resulting regrading was a massive drain on the time resources of the teaching assistants (3 assistants per course).

There were three technical issues that in hindsight should have been addressed proactively. It was assumed that the students would install and learn Python on their own, choose a text editor to work with, and also learn subversion for team-work and submitting assignments (we use subversion on a UNIX server for submitting assignments)[6]. A good technical tutorial given by the teaching assistants at the beginning of the course would have saved everyone a lot of time. Even fourth year students find these technical aspects of the course challenging. We also propose to set up an online discussion group for the students, where they can help each other with technical difficulties. The teaching assistants could monitor the discussions.

Many of the issues discussed in the above paragraph came to the surface once the course was over, and the instructor and the teaching assistants engaged in a SWOT ("Strengths, Weaknesses, Opportunities, Threats") analysis at the end of the course. This year was the first time that the instructor engaged in such an exercise, and it cannot be overstated how fruitful it was.

**Assignment 1.** Write a Python program that takes as input the description of a grid, and outputs its minimum cost spanning tree. An $n$-grid is a graph consisting of $n^2$ nodes, organized as a square array of $n \times n$ points. Every node may be connected to at most the nodes directly above and below (if they exist), and to the two nodes immediately to the left and right (if they exist). An example of a 4-grid is given in figure 2.

[6]Students who worked with Windows OS on their home computers were especially confused when installing SVN; an excellent resource for them is `http://tortoisesvn.tigris.org` .

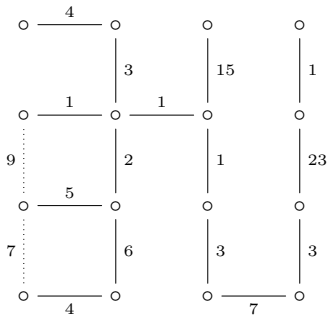


**Figure 2: An example of a 4-grid.**

The students were supposed to write a program which, when given as input a list of triples describing a graph (a triple consisted of two nodes and the value of the edge between them), the program had to check whether the list describes a proper grid, and compute a minimum cost spanning tree, and display (as a text-based graphic) this tree. The students struggled with parsing the input; the input is a text based list of triples, with many possible interpretations: how do we interpret spaces, unbalanced parenthesis, the same triple repeated, or given once as $(i, j, n)$ and then a second time as $(j, i, n)$ (the grid was assumed to be undirected). Thus, establishing whether a given set of triples is a valid grid was more difficult than implementing Kruskal's algorithm. We found that in general parsing the input tends to be the most challenging part of the assignment.

**Assignment 2.** Write a program that implements a dynamic routing policy mechanism; a routing table management daemon, which maintains a link-state database according to the OSPF (Open Shortest Path First—described in `RFC 2328`) interior routing protocol. The program had a user interface which added routers and networks, deleted them, as well as created connections. It also had a command for displaying the routing table, and another command for computing a tree of shortest paths for a given node $x$. One group was asked to implement Dijkstra's greedy algorithm for shortest path, and the other group the Bellman-Ford "dynamic programming" algorithm. Part of the assignment was to research RFC (Request for Comment) documentation on the web, explaining the implementation of such a daemon. This assignment is a great way for introducing "shortest-path" algorithms in a way that is very relevant to the students: via Internet protocols.

**Assignment 3.** Write a Python program that implements DES, the "Data Encryption Standard," following the description of DES in section 3.3 of [12]. Part of the challenge was to find a good source of "S-boxes" on the web—they were given in the textbook, but transcribing such a large amount of data would surely result in typos. The other challenge was to come up with a testing mechanism; how did the students know that their code was working correctly? The program was command driven and was supposed to take a 64-bit string as input (the plaintext) as well as a 54-bit string (the key), and output the corresponding 64-bit output (the DES ciphertext).

It was valuable to hear the students' observations regarding team-work on a large programming project. It seems that the biggest challenge in the end was not technical; it

was not conceptual either. While there is no data to support this observation, it seems that the greatest source of difficulty for the students was task management. Many teams found themselves writing the project the night before it was due, dismissing years of drilling on the proper software design cycle: design, prototyping, testing, etc. Work management is a difficult issue for them, and many are going into the industry with poor work habits. It has been a goal of the second author to develop material for a sequence of two or three lectures where time management issues are discussed. There are two sources that can be given to the enterprising students, but a synthesis of them could also be given to the entire class at the beginning of the year. The material could be prepared based on [6] and [1].

Finally, many students perceive CS to be a lonely field, where workers toil on code in isolation; this is not true. Most IT specialists work in teams, where communication skills are essential. Emphasizing the aspect of team-work and community in CS can be a way of retaining students. An excellent source for students to read about the community aspects of CS, especially in light of the tremendous success of the "Open Software" movement, can be found in [2].

# 4. CONCLUSION

Neither faculty nor administrators should be afraid of a healthy dose of theoretical material in undergraduate CS courses. A "problem solving" approach, where the mathematical underpinning of computer science are mixed with relevant examples results in a high student satisfaction. This is confirmed by studies such as [19], and in the case of McMaster University by [4] and the experience of the second author. Just as in the humanities students respond well to the "Great Books" programs, so in CS students respond very well to learning timeless principles, and seeing how they arise in practice. We also suggest that foundational knowledge be taught as early as possible, so that students may have the opportunity to observe how it arises in different domains of the field of CS and SE, and profit from their insight.

# 5. ACKNOWLEDGMENTS

We are grateful to Dave Scholz for his comments, feedback and guidance on [4]. We are also grateful to the students who participated in the study.

# 6. REFERENCES

[1] D. Allen. *Getting Things Done: The Art of Stress-Free Productivity*. Penguin, 2002.

[2] J. Bacon. *The art of community*. O'Reilly, 2009.

[3] T. Beaubouef and J. Mason. Why the high attrition rate for computer science students: Some thoughts and observations. *The SIGCSE Bulletin*, 37(2), June 2005.

[4] K. Blanchard. Undergraduate computer science students: measuring perception, marketing and satisfaction. A case study undertaken for a course in "public relations research" (MCM 712), at the DeGroote School of Business, McMaster University.

[5] S. Brin and L. Page. The anatomy of a large-scale hypertextual web search engine. In *Proceedings of the seventh international conference on World Wide Web 7*, WWW7, pages 107–117, Amsterdam, The Netherlands, The Netherlands, 1998. Elsevier Science Publishers B. V.

[6] S. R. Covey. *The 7 Habits of Highly Effective People*. Free Press, revised edition, 2004.

[7] P. J. Denning. A debate on teaching computing science. *Commun. ACM*, 32:1397–1414, December 1989.

[8] A. Downey. *Think Python: How to Think Like a Computer Scientist*. Green Tea Press, 2008.

[9] M. Guzdial and E. Soloway. Teaching the nintendo generation to program. *Communications of the ACM*, 45(4), April 2002.

[10] D. Hagan and S. Markham. Does it help to have some programming experience before beginning a computing degree program? In *ITiCSE*, pages 25–28, 2000.

[11] T. Howles. Preliminary results of a longitudinal study of computer science student trends, behaviors and preferences. *J. Comput. Small Coll.*, 22:18–27, June 2007.

[12] C. Kaufman, R. Perlman, and M. Speciner. *Network Security: Private Communication in a Public World*. Prentice Hall, 2 edition, 2002.

[13] J. Kleinberg and É. Tardos. *Algorithm Design*. Pearson Education, 2006.

[14] C. Lewis. Attitudes and beliefs about computer science among students and faculty. *SIGCSE Bull.*, 39:37–41, June 2007.

[15] C. C. Miller. Computer studies made cool, on film and now on campus. *New York Times*, June 2011.

[16] D. L. Parnas and M. Soltys. Basic science for software developers. SQRL 7, McMaster University, October 2002.

[17] D. L. Parnas and M. Soltys. Basic science for software developers. In *Formal Methods 2006, Educational Workshop*, 2006.

[18] R. Rashid. Image crisis: Inspiring a new generation of computer scientists. *Commun. ACM*, 51:33–34, July 2008.

[19] S. Reges. Back to basics in CS1 and CS2. *SIGCSE Bull.*, 38:293–297, March 2006.

[20] E. Sekerinski. Teaching the unifying mathematics of software design. In R. Brouwer, D. Cukierman, and G. Tsiknis, editors, *WCCCE '09: Proceedings of the 14th Western Canadian Conference on Computing Education*, pages 109–115, Burnaby, British Columbia, Canada, May 2009. ACM.

[21] P. Thibodeau. Computer science enrollments rebound, up 10% last fall. *COMPUTERWORLD*, April 2011.

[22] A. Tregear, S. Dobson, M. Brennan, and S. Kuznesof. Critically divided? How marketing educators perceive undergraduate programmes in the UK. *European Journal of Marketing*, 44(1):66–86, 2010.