Broadening Student Enthusiasm for Computer Science with a Great Insights Course

Marie desJardins
University of Maryland Baltimore County
Department of Computer Science & EE
Baltimore MD 21250
mariedj@cs.umbc.edu

Michael Littman
Rutgers University
Department of Computer Science
Piscataway NJ 08854
mlittman@cs.rutgers.edu

ABSTRACT

We describe the "Great Insights in Computer Science" courses that are taught at Rutgers and UMBC. These courses were designed independently, but have in common a broad, engaging introduction to computing for non-majors. Both courses include a programming component to help the students gain an intuition for computational concepts, but neither is primarily programming focused. We present data to show that these courses attract a diverse group of students; are rated positively; and increase students' understanding of, and attitudes towards, computing and computational issues.

Categories and Subject Descriptors

K.3.2 [**Computers and Education**]: Computer and Information Science Education—*computer science education, curriculum*

General Terms

Design, Human Factors, Measurement

Keywords

introductory courses, attitudes towards computing

1. INTRODUCTION

Most computer science degree programs include an introductory course sequence that focuses on teaching students how to program. Many colleges also have introductory courses designed for students majoring in other disciplines, and students who have not yet decided on a major. However, these courses are typically still programming centered.

Several studies over the past few years have identified the emphasis (some would say overemphasis) on programming in introductory courses as one of the problems with CS degree programs [17]. A heavy emphasis on programming often does little to attract non-majors into the major, and may "turn off" some students who had intended to major in CS. Women and underrepresented minorities are more likely to be affected by this situation [16]; one hypothesis is that these students are more motivated by a desire to help other people in their careers (rather than by a prior interest in programming and computers *per se*), and are therefore less likely to feel that the programming-heavy introductory sequence is relevant to their long-term goals.

The emphasis on programming in courses designed for non-CS majors may also unintentionally turn students off to CS. In the long run, because CS is inherently interdisciplinary, and because computing is becoming increasingly important across disciplines, it is essential—both for the field of CS and for society as a whole—that *all* college graduates have an appreciation for the capabilities and limitations of computing. Therefore, not only is it problematic if an exclusive focus on programming reduces students' interest in CS, but these programming-centered courses may not lead to the learning outcomes that are most important for a broad student population: namely, an understanding and appreciation of the basic and applied *science* of computer science. Recently, some institutions have begun offering courses that are "contextualized" to different interest areas, such as engineering or multimedia computation [12], or that offer a "breadth-first" approach to computer science rather than a purely programming-centric pedagogy [1, 2, 10]. However, while intended to be more appealing to non-CS students, these courses are still centered around programming.

At Rutgers University and UMBC, the authors have designed and taught courses for non-majors that focus on what we call the "Great Insights" of computer science. These courses have been very popular at both schools, have attracted a number of students to take additional programming-focused CS courses, and have had a positive effect on students' attitudes towards computing as a discipline. The courses have attracted a higher percentage of female and minority students than the traditional programming-centered introductory courses for majors and non-majors. While the courses we describe do include a programming component, we introduce "student-friendly" visual programming paradigms (Scratch and Alice) only to illustrate the idea of programming to the students, not with an intention to give them proficiency in programming *per se*.

One of the key aspects of the courses we describe is that they are not "lightweight" or "high-level": while they are meant to be engaging and accessible for students with a wide range of backgrounds, they do not shy away from topics that

SIGCSE'10, March 10–13, 2010, Milwaukee, Wisconsin, USA.

are conceptually challenging and mathematically rigorous. At the same time, the courses make no attempt to "weed out" students, but focus on exposing *all* students to the deep and interesting ideas underlying the discipline.

The idea that programming and technologies are not philosophically at the core of computing and computational thinking is not new. Denning [7] argues that we should move away from the "classic" division of computing into technology categories (networking, compilers, etc.) and towards developing a model that is focused on core scientific and engineering principles. Denning identified five such "core principles" of computing: computation, communication, coordination, automation, and recollection; in later work, two additional principles—those of evaluation and design—were added [8]. Wing [18] argues persuasively that understanding principles of computation and "computational thinking" is important for everyone, not just computer science. We were also partly inspired by "Great Ideas" and breadth-first programming-centered courses [2, 6, 10].

In this paper, we describe the shared structure of our courses, discuss some of the specific topics and teaching techniques that each of our courses has incorporated, and present data supporting the success of the classes in both schools. We conclude by discussing some of the ongoing improvements that are planned for both courses and our recommendations for other faculty who may be considering introducing similar courses at their institutions.

2. GREAT INSIGHTS COURSES

The two "Great Insights" courses that we describe here are Rutgers's CS105, "Great Insights in Computer Science," and UMBC's CMSC 100, "Introduction to Computers and Programming."[1] The intention of both courses is to give students from a variety of backgrounds a broad exposure to the scientific principles and mathematical aspects of the discipline of computer science.

The essence of the courses is a key principle of computer science: the notion of *abstraction* (or, viewed the other way, problem reduction). The message is "it's all just bits"; the key to computer science is that those bits can be composed in highly complex ways through abstraction layers, resulting in today's sophisticated hardware and software.

Both classes use Google and universal product codes (UPCs) as case studies that touch upon many of the "Great Insights" that drive computer science. Specific topics that both courses cover include logic circuits, Boolean algebra, binary numbers, algorithms (including top-down design and searching/sorting), computer organization, theory of computation (finite state machines, the halting problem, complexity classes, and NP-completeness), data compression (including Huffman codes), AI and heuristic search, and computer graphics. The differences between the two classes, and specific pedagogical techniques that each of the authors has used in their classes, are discussed in the next sections.

2.1 Great Insights at Rutgers

The Rutgers course was created in the Spring of 2006 to respond to a perceived need to reach out to non-majors and present the "science" side of computer science. Existing introductory classes at Rutgers were concerned with societal issues, programming, and software applications, but not academic computer science. (Table 1 summarizes the topics covered.)

Week	Topics
1	Course Overview; Barcodes
2–3	Bits; Boolean Algebra; Logic Blocks
4	Binary Arithmetic; State Machines
5	Programming in Scratch
6	Machine Language; Subroutines; Recursion
7–8	Randomness; Self-contradiction; Halting
9–10	Midterm; Decision Problems on Lists
11	Growth Rates: Songs, Algorithms; NP
12	Algorithms: Reachability, Sorts, Binary Search
13	Heuristics; Huffman Codes
14	Parallelism; Images; Learning
15	Robots; Reinforcement Learning; Genetic Algorithms

Table 1: Topics by Week for Rutgers's CS105

Everyday Examples. We used a short book by Danny Hillis [13] that provides an engaging account of the inner workings of computers, from logic gates up to parallel and learning systems. The book uses personal anecdotes to make the material more accessible. For example, the section on algorithms begins with a story in which Hillis's college roommate announces, "I have decided to use a better algorithm for matching my socks." The book then describes how two different solutions to the same problem can have different properties.[2]

The class elaborates on this example, carrying out a live demonstration, soliciting ideas for other approaches to the same problem, presenting the algorithms themselves as programs, considering different objectives for optimality, and going through the analysis of the different options.

Other examples in the class's algorithm-analysis section include analyzing song growth [5], decision problems on lists ("Who can be the first to tell me if the median of this list of 19 numbers is 47?"), and graph search in web spidering. The emphasis is on engaging the students to look at familiar problems from a (novice) computational perspective.

Programming in Scratch. Although the class does not emphasize programming, it is difficult to convey computational ideas without some concrete means for expressing algorithms. In the first year, we used Python for all in-class examples because, to our eye, it looked like executable pseudocode. When it became clear that beginning students still struggled with the syntax, we tried dropping the examples, but that made the class too vague. When we found Scratch (scratch.mit.edu), however, our search was over. Scratch was designed around three core principles: "Make it more tinkerable, more meaningful, and more social than other programming environments" [11].

Although Scratch is pitched at a younger audience, it can be used successfully in a college-level course [15]. We found that the simplicity of the *appearance* of the language made it possible for students to absorb the basic ideas very quickly

[1]UMBC's original CMSC 100 taught students how to use Microsoft Excel and other application tools, and covered the basics of programming in various languages. The author completely redesigned the course in 2008, but the course title has not yet changed to reflect the revised course contents.

[2]The book's strength is its broad and engaging exploration of computer science concepts. Its primary weakness is that it is not actually a textbook; it lacks exercises and reviews that students could use to cement their understanding.

Week	Topics
1	Course Overview; Deconstructing Google
2	Deconstructing a Computer; "It's All Just Bits"
3	Huffman Codes; Introduction to Alice
4	Algorithms
5	Computer Organization; Computational Efficiency
6	Operating Systems and Networks
7–8	Algorithms; Presentations; Midterm
9	Data Abstractions and "Movie Day"
10–11	Databases, AI, and Game Playing
12	Graphics and Theory of Computation
13	Secure Computing
14–15	Computing Tomorrow and Student Presentations

Table 2: Topics by Week for UMBC's CMSC 100

and to be able to start thinking about simple programs right away. In the first year that we adopted Scratch, students began insisting that we use Scratch not just for the actual programming examples, but as the pseudocode as well.

Because of the ease with which Scratch control structures can drive simple media (sounds and animation), we were able to illustrate concepts such as loops and recursion in a way that was more perceptual and less abstract. As a concrete example, students were asked to determine whether a simple loop would halt. When this material was presented using Python, most students answered these questions correctly no more often than random guessing, even at the end of the semester. When we switched to a live demonstration using Scratch, and placed an annoying sound effect inside the loop (on each iteration, it would make a loud pop), students began taking a greater interest in distinguishing halting and non-halting loops ("No, don't start *that* one!"), resulting in increased student performance on these tasks.

Another benefit of Scratch is the community website (see http://scratch.mit.edu/users/cs105). Scratch programs can be posted on a YouTube-like website and shared with friends, even those who have not installed the Scratch development environment. The class included an assignment in which students used this facility to create an animated holiday card. We demonstrated the best cards in class for a "Scratch Off"—students voted to decide which program would receive a prize and be posted on the class Scratch site.

Videos. Another mechanism that we used in the class to engage students with ideas from computer science was the creation of music videos on several pertinent topics, roughly one per chapter of the book. The current videos (youtube.com/mlittman) include Octopus's Counting (binary numbers), The Sorter (sorting), Paintcan (recursion), The Answer Man (computability), and an end-of-class review. Some of these videos have been picked up for use in other classes, including at UMBC—the Octopus video has been viewed over 32,000 times.

The class has been offered a total of eight times, five of these by the second author (who was the original course designer). Each semester brings new experience and examples.

2.2 Great Insights at UMBC

UMBC's CMSC 100 uses as a textbook Brookshear's *Computer Science: An Overview* [3]. The textbook covers many of the topics in the course, but we omit much of the "technology-centered" material and add more of illustrations of the core scientific principles of computation (e.g., Huffman codes, finite state automata, minimax game trees, and Google as a case history). We also supplement the course material heavily with additional lecture notes, handouts, and relevant articles from the scientific and popular press. A high-level outline of the course topics by week is given in Table 2.

By way of comparison, UMBC's course is slightly more applied than the course at Rutgers, and has an added focus on CS and society, as well as applications of computing in a variety of domains.

CMSC 100 is currently in its second offering, with the author (and original designer) still teaching the class. In 2010, we expect the course to be taught by other instructors, so one of the main course-development activities this year is to create a "course package" that can easily be reused by other instructors.

Applications. The UMBC course spends time throughout the semester discussing how computing affects society. Specific topics include privacy, environmental impacts, security and computer viruses, validation of autonomous systems, and the ethical and philosophical issues surrounding AI and the possibility of true intelligent computers.

Students are required to write a research paper on the applications of computing in a field of interest to them. This assignment includes a short proposal, a preliminary bibliography, an in-class one-minute topic presentation, a draft paper that is read and critiqued by two other students, a final paper, and a short (3-minute) in-class presentation of their findings. Topics in 2008 ranged from autonomous vehicles to theatrical lighting and pyrotechnics to motion-capture technology for testing bowling balls and bowling technique. This assignment not only gives the students important writing practice (with significant feedback to improve their writing and paper organization), but it also gives them a much broader and deeper exposure to the possibilities and importance of computing (particularly since they learn about what all of the other students have researched as well).

Programming in Alice. The courses uses Alice [4] to teach basic programming concepts. The first two assignments simply ask the students to download Alice 2.2 and run the three included tutorials, answering a few short questions about the tutorials to make sure they are paying attention. After that, there are two very open-ended programming assignments: the students create their own movie/story that includes a certain number of objects, classes, and methods. There are a few other requirements to make sure that students use the whole "toolkit" of programming constructs (e.g., they must at some point iterate using a "for" loop).

The resulting programs are shown in class, and the students have the opportunity to vote on their favorites. This "movie day" is very popular with the students, since it gives them the chance to "show off" their own work and to see what the other students were able to create with the Alice environment.

Given our experiences in the two classes, both Alice and Scratch seem to be good environments for beginning programmers to experiment and apply what they have learned about algorithmic techniques in practice. Scratch's interface is perhaps a bit easier to use (the drop-down menus in Alice for setting parameters, and mechanisms like the camera angle, are not always intuitive for the students to understand). Scratch's syntax is also intuitive enough to be used

as the pseudocode standard in Rutgers's CS105; since training students to read and understand pseudocode (or even the concept of a sequential or iterative algorithm) is quite challenging, we are considering trying out Scratch as an alternative to Alice in the 2010 offering.

Nifty Lectures and Assignments. Several of the lectures and assignments have been particularly effective and popular. Students were especially engaged by the "Deconstructing Google" and "Deconstructing a Computer" lectures. In the former lecture, we start from "how Google works" and peel away the layers of abstraction, bringing the discussion down to the level of server farms and the amount of storage required to crawl and index the web. In the latter lecture, we literally take apart a computer and talk about the components, how modular the design is, and how the pieces are configured to make a computer work.

As part of the final homework assignment, students are required to submit three proposed exam questions on the course Blackboard website. As an incentive to design good (but fair) questions, students are told that 30% of the final exam will consist of questions from the website. Students can see each other's questions, which helps them to prepare—and the instructor gains valuable insights into what students thought the course was actually about.

Honors Section. UMBC has an Honors College program for academically talented students who wish to pursue a more intensive and engaging option throughout their college career. All Honors College students are required to take a certain number of "Honors"-designated courses. CMSC 100H, the Honors section of CMSC 100, is offered as a weekly discussion session/lab in addition to the regular CMSC 100 lectures. The honors section alternates seminar-style discussion sessions with "programming boot camp"—a fast-paced introduction to Linux and Python (the programming language used in our CS 1 class) that is designed to prepare students for the introductory programming course for CS majors. (In 2008, all of the eight honors students continued into that course; one withdrew from the course for personal reasons, but the others all passed the course with a B or higher.) Over the course of the semester, as the students' algorithmic problem solving skills improve, they incrementally design and implement a player agent for the game of Mastermind. In the second year, we used pairs programming in some of the labs, which proved very effective in this accelerated learning environment.

"Social impact" discussion topics have included privacy and security, the environmental impact of Google's large-scale search operations, and Asimov's "Three Laws of Robotics" and whether it is ethically acceptable to treat intelligent robots as servants.

3. OUTCOMES

In the Fall of 2008, we engaged in some simple evaluations to assess how well our introductory courses were meeting our goals. We had introductory students at UMBC and Rutgers fill out entrance and exit surveys. These surveys collected some basic data about the students, their educational background, their familiarity with computers, and their attitudes towards the class, computer science, and computers more generally.[3] Surveys were offered in paper form for some classes and via an online site (surveymonkey) for others. Students were not required to complete surveys, and exit surveys in particular had a low rate of return. (As a result, we were unable to form strong conclusions about how the courses *changed* student perceptions—other than anecdotally—but have more substantial data about the populations that the courses attract.)

[3]The "attitudes" questions all used a five-value Likert scale: Strongly Agree, Agree, Neutral, Disagree, and Strongly Disagree.

	Programming	Insights	Programming	Insights
UMBC	17% (36)	30% (46)	24% (34)	33% (46)
Rutgers	14% (21)	50% (66)	10% (19)	84% (44)
Combined	16%	42%	19%	65%

Table 3: Percentage (and total survey responses, in parentheses) students who are female (left) and who feel they would not enjoy being a computer scientist (right) in two kinds of introductory courses (traditional programming-based and Great Insights) at the two schools.

In the results below, we distinguish between the new introductory Insights courses described in this paper—Rutgers-I (CS105) and UMBC-I (CMSC 100)—and two more traditional introductory programming courses offered at the same time—Rutgers-P (CS111/112) and UMBC-P (CMSC 104).

The composition of the classes themselves varied (Table 3). At both schools, the Great Insights courses attracted a substantially greater percentage of female students than the traditional programming-based courses (left part of table, 16% vs. 42%). (In the Rutgers case, the difference is significant, $p < .01$ by a two-tailed Fischer's exact test).

Although the survey did not ask for the respondents' race, we have statistics for the UMBC courses. There was a noticeably higher percentage of minority students in the UMBC-I course than in UMBC-P. The latter course usually has very few Hispanic and African-American students, typically less than 10%. (UMBC's undergraduate student population is 16.5% African American and 4% Hispanic.) By contrast, in 2008, 13% of the students in 100/100H fell into these categories; this year, 22% of the students are Hispanic or African-American. The course also attracts an extremely diverse group of students in terms of their majors: over the two years the course has been taught, we have had students from economics and financial economics, psychology, engineering, computer science, biology, visual arts, history, public policy, media communications, mathematics, and political science. UMBC-I also has a relatively large percentage of undeclared students (nearly 20%).

One of the questions on the survey asked students whether they agreed with the statement, "I think I would enjoy being a computer scientist." The percentage who *disagreed* with the statement (answering either "Disagree" or "Strongly Disagree") are shown in the right of the table. There are a number of interesting trends here. First, the traditional programming-style courses attracted students who were more convinced they would enjoy being computer scientists (with only 19% of the students disagreeing, compared with 65% of the Insights students disagreeing). The Great Insights students may have been interested in computer science, but not necessarily as a career. Second, the Rutgers class in particular seems to have reached a wider community, the vast majority of whom were not expecting to enjoy a career in

computer science. It is worth noting that the percentage of such students dropped in the post-survey from 84% to 67%, indicating that the idea of being a computer scientist seemed at least somewhat less unappealing.

Less quantitatively, our effort to include accessible and engaging material appears to be working. The number of students has grown with each offering of both classes. We have also received a number of very supportive and unsolicited statements from students via email, including:

- "*i'm a History and English buff, total opposite side of the spectrum...lol but i must say i truly LOVED this class.*"
- "*Thanks to you I'm minoring in CS now. You made it even more enjoyable and made it a lot easier so that even an art major like me could understand.*"
- "*I am currently a majorless sophomore who really enjoyed your class, however I'm not a huge math fan. Your class made me think of majoring in something computer related ... Thanks for a great class!*"
- "*Thanks to your wonderful teaching in CMSC 100H, not only am I in CMSC 100H, but my first day of MATH 301 [Analysis I, a notoriously difficult class required for mathematics majors] was a breeze.*" (CMSC 100H student)
- "*I would absolutely be taking more CS classes had I not graduated. Already, the small amount that I do know has given me tangible results both personally and professionally. I am considering following up my education with a CS program because so few classes bore such great results.*"

4. CONCLUSIONS / RECOMMENDATIONS

We have described two courses, at Rutgers and UMBC, that use the idea of Great Insights in computing to teach non-majors about computational thinking and the science of computation. These courses have been very successful at increasing student interest in and engagement with computing. However, the popularity and success of the courses also depend heavily on the degree of engagement and enthusiasm that the instructors bring to the courses. This is particularly true in a course of this type, because non-CS majors are not necessarily intrinsically motivated or predisposed to understand the importance or relevance of CS concepts to their own interests. Capturing the key pedagogical techniques and methods that make these courses successful is a focus at both UMBC and Rutgers as the courses begin to be taught by new instructors. We have also considered the possibility of writing a textbook to carry the ideas of these courses to other institutions.

At UMBC, we are discussing the possibility of offering a variation of CMSC 100 that is specifically designed for majors. On the practical side, this course would likely include somewhat more programming (probably in Python) and an exposure to Linux. On the conceptual side, the topics would likely stay the same, but in some cases with more depth and/or mathematical rigor. The CS majors that have taken the Insights classes have been extremely positive about them, since many of the concepts discussed in the class are otherwise not mentioned until late in the degree program, if at all.

We are enthusiastic about "spreading the word" to other institutions about the success of the Great Insights approach to introductory courses for non-majors at Rutgers and UMBC. We encourage other faculty to consider introducing similar classes into their curriculum as an alternative to traditional programming-centered computing classes for non-majors. Further information and many course materials are available on both course websites [14, 9].

Acknowledgements

The course development at UMBC was completed during Prof. desJardins's sabbatical leave, supported by UMBC and the CS&EE department. Prof. desJardins's time was also supported by NSF CAREER #0545726. Prof. Littman's time was supported in part by NSF ITR-0325281.

Thanks to Chaitra Satharayanara, who organized and helped to analyze the student response data.

5. REFERENCES

[1] A. W. Biermann. *Great Ideas in Computer Science*. MIT Press, 1997.

[2] A. W. Biermann and D. Ramm. *Great Ideas in Computer Science with Java*. MIT Press, 2001.

[3] J. G. Brookshear. *Computer Science: An Overview (10/e)*. Addison Wesley, 2008.

[4] Carnegie Mellon University. Alice website, 2009. Alice.org.

[5] D. Chavey. Songs and the analysis of algorithms. *ACM SIGCSE Bulletin*, 28(1):4–8, 1996.

[6] T. J. Cortina. An introduction to computer science for non-majors using principles of computation. In *SIGCSE'07*, pages 218–222, 2007.

[7] P. Denning. Great principles of computing. *CACM*, 46(11):15–20, November 2003.

[8] P. Denning and C. Martell. Great principles of computing website, 2009. http://cs.gmu.edu/cne/pjd/GP/.

[9] M. desJardins. CMSC 100 Syllabus, Fall, 2009. http://www.cs.umbc.edu/courses/undergraduate/100/Fall09/.

[10] Z. Dodds, C. Alvarado, G. Kuenning, and R. Libeskind-Hadas. Breadth-first CS 1 for scientists. *SIGCSE Bulletin*, 39(3):23–27, September 2007.

[11] M. R. *et. al.* Scratch: Programming for all. *Communications of the ACM*, 52(11), November 2009.

[12] M. Guzdial. Teaching computing to everyone. *Communications of the ACM*, 52(5):31–33, 2009.

[13] D. Hillis. *The Pattern on the Stone: The Simple Ideas that Make Computers Work*. Basic Books, 1998.

[14] M. Littman. CS105: Great Insights in Computer Science, 2009. http://www.cs.rutgers.edu/m̃littman/courses/cs105-08/.

[15] D. J. Malan and H. H. Leitner. Scratch for budding computer scientists. In *SIGCSE'07*, 2007.

[16] J. Margolis and A. Fisher. *Unlocking the Clubhouse: Women in Computing*. MIT Press, 2001.

[17] E. H. Turner, E. Albert, R. M. Turner, and L. Latour. Retaining majors through the introductory sequence. In *SIGCSE'07*, pages 24–28, 2007.

[18] J. Wing. Computational thinking. *CACM*, 49(3):33–35, March 2006.