Design Process for a Non-majors Computing Course

Mark Guzdial
College of Computing/GVU
Georgia Institute of Technology
801 Atlantic Drive
Atlanta, Georgia
guzdial@cc.gatech.edu

Andrea Forte
College of Computing/GVU
Georgia Institute of Technology
801 Atlantic Drive
Atlanta, Georgia
aforte@cc.gatech.edu

ABSTRACT

There is growing interest in computing courses for non-CS majors. We have recently built such a course that has met with positive response. We describe our design process, which includes involvement of stakeholders and identifying a context that facilitates learning. We present evaluation results on success rates (approximately 90% of the students earn an A, B, or C) and impact of the course on students over time (80% report that the class has influenced them more than a semester later).

Categories and Subject Descriptors

K.4 [**Computers and Education**]: Computer and Information Sciences Education
; H.5.1 [**Information Interfaces and Presentation**]: Multimedia Information Systems

General Terms

Experimentation,Design

Keywords

Multimedia, CS1, CS2, programming, non-majors

1. DESIGNING COMPUTER SCIENCE FOR THE NON-MAJORS

There is growing interest in the creation of computer science for non-CS majors. One reason for this increase is the recognition that computing now influences every aspect of our society, and it is a competitive advantage for students in other majors to know more about computing. A reason for the interest from CS departments is declining enrollment in the CS major, which inspires CS faculty to look elsewhere for customers [6].

At Georgia Institute of Technology (*Georgia Tech*), the faculty require that every incoming student must take a course in computing, including a requirement to learn and use programming. When that decision was made, the only class available was our majors-focused CS1 based on the TeachScheme approach[7], which did not adequately meet the needs of our liberal arts, architecture, and management students. The course saw low success rates, and the students and faculty were vocal in their dissatisfaction. We saw this dissatisfaction as an opportunity to create a new course and build toward Alan Perlis' vision of programming for *all* students—as a component of a general, liberal education [8].

The course that we developed, *Introduction to Media Computation*, is an introduction to computing contextualized around the theme of manipulating and creating media. Students really do program—creating Photoshop-like filters (such as generating negative and greyscale images), reversing and splicing sounds, gathering information (like temperature and news headlines) from Web pages, and creating animations. Details on the structure and content of the course are available elsewhere [20, 11].

The purpose of this paper is to use our course as an instance, and abstract from it a design process for a non-majors introductory computing course and a set of benchmark evaluation results. Our definition of success is:

- Non-CS majors students should have a higher success rate than in a traditional introductory computing course. If we are designing the course explicitly for that audience, we should be able to satisfy their needs and meet their interests better than we can in a course designed for our own majors.
- The course should have impact beyond the single term. If the course doesn't influence how non-major students think about computing, and they will only take a single computing course (probably), then we will have lost our opportunity to influence these students.

In this paper, we describe how we designed the Media Computation course as an example process for designing a non-majors computing course. A sketch of our process follows, and is detailed in the second section of the paper.

- *Setting objectives*: We set objectives for the course based on campus requirements for a computing course, on ACM recommendations, and on the existing computer science education research literature about what students find difficult about computer science.
- *Choosing a context*: We selected a context that allowed us to meet our curricular objectives and that we

SIGCSE2005 '05 St. Louis, MO, USA

believed would be motivating to non-major students. Having an explicit context helped them understand *why* they should care about computing, which is significant issue in introductory computing [13].

- *Set up feedback process*: We sought feedback from faculty in the majors that we planned to serve, as well as from students through multiple forums.
- *Define infrastructure*: An early challenge was to choose the language and programming environment, which are critical (and sometimes religious) issues. We found that the process of choosing a language for a non-majors course is as much about culture and politics as it is about pedagogy.
- *Define the course*: Finally, we defined lectures, assignments, and all the details of what makes up a course. Here our decisions were informed by research in the learning sciences [4].

2. DESIGN PROCESS

We detail below each stage of the development process for the Media Computation class. We believe that a similar process could be used to create other introductory computing courses targeting non-CS majors.

2.1 Setting Objectives

The Georgia Tech computing course requirements states that the introductory course curriculum has to focus on algorithmic thinking and on making choices between different data structures and encodings. There is an explicit requirement that students learn to *program* algorithms being studied. We also wanted to build upon the recommendations in *Computing Curricula 2001* [2] as a standard for what should be in a computing introduction, with a "CS1"-level course as our target.

We explicitly decided not to prepare these students to be *software developers*, but instead, we focused on preparing them to be *tool modifiers*. We do not envision these students as professionals ever sitting down to program at a blank screen. Instead, we imagine them modifying others' programs, and combining existing programs to create new functionality. Based on our discussions with faculty in these majors, we realized that these students, as professionals, will very rarely create a program exceeding 100 lines. The implications of these assumptions and findings are that much of the design content and code documentation procedures that appear in many introductory computing curricula are less relevant for these students than for majors.

We also explicitly chose to use this class to attract students currently not being retained within computer science, especially women. Since our audience was non-majors, they clearly fit into the model of students not choosing computer science as a major. We used the 2000 AAUW report [1] and *Unlocking the Clubhouse* [13] as our main sources. We set three objectives based on these studies: Making the content relevant, creating opportunities for creativity, and making the experience social

Relevance: A frequently cited complaint about introductory computing courses is that they are too abstract and not anchored in a relevant context–students do not understand how introductory course content is useful or relevant to their goals or needs. We set an objective to make sure that all the assignments and lectures were relevant to the students' professional goals within that context.

One implication is that we decided to discuss issues of functional decomposition, how computers work, and even issues of algorithmic complexity and theoretical limits of computation (e.g., Travelling Salesman and Halting problems), but at the *end* of the course. During the first ten weeks of the course, the students write programs to manipulate media (which they do see as relevant), and they begin to have questions that relate to these more abstract topics. "Why are my programs slower than Photoshop?" and "Isn't there a faster/better way to write programs like this?" do arise from the students naturally. At the start of the course, the abstract content is irrelevant (from the students' perception), but at the end of the course, it is quite relevant.

Opportunities for Creativity: Comments made by female computer science graduates at a recent SIGCSE session on women in computing suggested they were surprised to find that computer science offered opportunities for creativity—it wasn't obvious in the first few courses, but was obvious later [19]. Providing more opportunities for seeing computing as a creative activity in early classes may help improve retention [1].

Making the Experience Social: We wanted students to see computer science as a social activity, not as the asocial lifestyle stereotypically associated with hackers—a stereotype which has negatively influenced retention [13].

2.2 Choosing a Context

Most introductory computing curricula aim to teach generalized content and problem-solving skills that can be used in any programming application. Research in the learning sciences suggests that, indeed, teaching programming tied to a particular domain can lead to students understanding programming only in terms of that domain. This is the problem of *transfer* [4, 5]. That's why it's important to choose a domain that is relevant to the students. This is not a problem only with students, though—most software experts only can program well within domains with which they are familiar [3]. However, there is strong evidence that without teaching abstract concepts like programming within a concrete domain, students may not learn it at all [12]. Contextualization may offer an important key to improved learning. By teaching for depth instead of breadth, we can teach more transferable knowledge [4].

The argument has been made that teaching programming, especially to non-majors, improves general problem-solving skills. Empirical studies of this claim have shown that we can't reasonably expect an increase in general problem-solving skills after just a single course (about all that we might expect non-majors to take), but transfer of *specific* problem-solving skills can happen [17]. Therefore, teaching programming in a context where students might actually use programming is the best way of teaching students something in a single course that they might use after the course has ended.

Within this context, we were able to address our learning objectives. Issues of data structuring and encoding arise naturally in media computation, e.g., sounds are typically arrays of samples, while pictures are matrices of pixel objects, each pixel containing red, green, and blue values. We were able to address the specifics of a CS1 course in the details of the course construction.

Media computation is relevant for these students because, for students not majoring in science or engineering, the computer is used more for communication than calculation. These students will spend their professional lives creating and modifying media. Since all media are becoming digital, and digital media are manipulated with software, programming is a communications skill for these students. To learn to program is to learn how the students' tools work and even (potentially) how to build their own tools. Our interviews with students suggest that they accept this argument, and that makes the class context relevant for them.

To create opportunities for creativity in assignments, we wanted students to have choices in selecting media to use in their homework whenever possible. For example, one assignment requires the creation of a collage where one image appears multiple times, modified each time. Students get to pick the required image, the modifications to use, and can include as many other images as they would like.

The media computation context also provided something to share which helped to encourage a social class setting. We encouraged students to post their media creations in a shared Web space, our *CoWeb* tool that we had used successfully in previous computer science courses [10]. Such sharing transforms the programming activity. Instead of completing the program for the TA to grade, students are completing the program in order to generate the artifact that can be shared with others. We use the same CoWeb every term of the class[1], so that the "*Galleries*" build up over time. A healthy sense of competition develops–one student told us that her collage "can't be beat by the others" from past terms.

We explicitly encouraged a social context in the traditional parts of the class, as well. We allowed for collaboration on most assignments, only designating two as "take-home exams" on which no collaboration was allowed. We also used in-class quizzes and exams for assessment, but encouraged collaborative studying including collaborative exam review pages.

2.3 Set Up Feedback Process

We frequently involved students in our course design process. When we first started planning this class, we created on-line surveys and asked teachers of freshman campus-wide classes (such as introductory English composition, Calculus, and Biology) to invite their students to visit the pages and address the survey questions. Later, as our questions became more specific, we had follow-up surveys just inviting non-CS majors in our introductory computing courses. These were important mechanisms for gathering impressions and attitudes, and then for bouncing ideas off of students. As the class was taking shape, we invited non-CS majors in our introductory computing courses to attend pizza lunches where we presented the class and got feedback on the course. The lunch forums helped create an interest in the course, and that spurred more discussion and feedback in the on-line surveys.

We also set up an advisory board of eight faculty from around campus who reviewed materials and give us advice on what they wanted for their majors. The advisory board was very helpful in several ways. In several cases, the advisory board told us specific content issues that they wanted to see in the course, e.g., one faculty advisor told us about the kinds of graphing that she wanted to see, and another from Architecture suggested a particular topic that is relevant to architects (the difference between vector and bitmapped representations) that he hoped we could include. The board was also helpful in creating local expertise in the course when it came time for the various academic units to vote whether or not to accept the new course for their majors. The advisory board members were advertised as the local experts who knew the course better than just what was in the course proposal, which helped to sell the course to the rest of the faculty.

[1] `http://coweb.cc.gatech.edu/cs1315`

2.4 Define Infrastructure

Our first choice for programming language for the course was Scheme, since it was what we were already using [7]. Scheme was resoundingly rejected by both students and non-CS faculty. Students saw it as "more of the same"—just like our existing introductory computing course. The faculty rejected it for more surprising reasons: *Because* Scheme is perceived as more serious CS. One English faculty member said that she found Scheme unacceptable for her students simply because it was the first language taught at MIT.

We explored several other languages after that, including Java and Squeak [9], since the media manipulation was simple and cross-platform in those languages. Java was unacceptable to the non-CS faculty because we used it in our upper-level courses. That branded it as too complex for non-majors. Squeak was simply unknown–it could not be vetted in the same way that other languages could.

In the end, we settled on Python—in particular, the Jython dialect, implemented in Java, in which we could access cross-platform multimedia easily [18]. Python was acceptable for two reasons:

- First, we could list a number of companies using Python that non-CS faculty recognized, such as Industrial Light & Magic and Google. Having such a list was quite important to them. Non-CS faculty want some measure of quality of materials and content provided for their students, but the non-CS faculty may not have much background in computer science themselves. How can they then vet a programming language for their students? By looking at who else uses it, we discovered.

- Second, unlike a more obscure language like Squeak, there are references to Python everywhere on the Internet, always associated with terms that the faculty members found consoling: Easy-to-use, Internet-ready, and simple for beginners.

While the choice of language was limited by external factors, we were happy with the choice of Python because of the opportunities it gave us to apply lessons from computer science education research–in particular, for teaching iteration and conditionals. We know that learning iteration is hard for students [21], but we also know that if that iteration is expressed as a set operation, novices find it easier to understand [14, 15]. Because of how Python defines a `for` loop, we were able to introduce pixel manipulations as a set operation, e.g., `for p in getPixels(picture):`. Later, we introduced a more traditional `for` loop where an index variable varies across a range of integers, but only after students were successfully programming and dealing with iteration at an easier stage.

Once we had chosen our language, we needed to provide tools for this language. We decided to build two sets of tools. The first would be a development environment for the students, JES (Jython Environment for Students) because no such simple development environment existed for Jython. Second, we developed a set of media tools (called MediaTools, implemented in Squeak) to enable students to look at sounds at the sample level, record new sounds, playback movies, and look at individual pixels in pictures. We viewed the MediaTools as important debugging tools for the students.

We have found weaknesses in our original plans. Observations of students programming revealed that the MediaTools were never used as debugging aids. Sitting in a separate application, the MediaTools were simply ignored while students worked on their programs. We have since implemented some of the media exploration functionality in JES, which do get used by the students to help them understand what their programs are generating. The lesson we draw from this is that the context selected for a non-majors course places demands upon the programming environment. A programming environment that works for majors may not be adequate for non-majors, especially if a relevant context is chosen as we advocate. The programming environment must support both context and computing learning objectives.

2.5 Building the Course

We developed the course lectures and assignments to achieve the objectives within the given context and infrastructure. The syllabus[2] for the course walks through each media type, with some repetition of concepts so that conditionals and loops can be re-visited through exploration of different media types.

We only briefly address the issue of inefficiency—it's mostly a distraction in a first course [1, 13]. We do, however, address encoding issues, such as the number of bits per red, green, and blue channel in the pixel, and the theoretical number of colors that such an encoding provides. We consider this a *relevant* technical detail since representations of color are part of the communications focus of the course, and it allowed us to address the Institute requirements of discussing encoding and data structuring. We similarly discuss the number of bits in a sound sample and sampling rate, and relate that to the limitations of sound recording (e.g., the Nyquist theorem).

Student programming assignments built upon the media in relevant communications tasks. As mentioned earlier, one open-ended programming assignment required creation of a collage. A later assignment on text manipulation and HTML was to write a function to generate an HTML index page for all sound and picture files in a given directory. The students' final programming assignment was to write a program to fetch the index page of a news website, find the top three headlines from the page, then generate a ticker-tape movie of those headlines. The programs reached a level of computing and domain complexity that the students seemed to find satisfying.

3. EVALUATION: SUCCESS AND IMPACT

We identified at the start of the paper that our measures for success were (a) improved success rates (percentage of students earning an A, B, or C) and (b) impact of the course over time, after the class was completed. We found that the course has had remarkable success rates and is showing signs of having an impact on students after they leave the course.

[2]The syllabus for the course is summarized in [11].

Average for CS1 (2000-2002)	72.2%
Media Computation Spring 2003	88.5%
Media Computation Fall 2003	87.5%
Media Computation Spring 2004	90.5%

Table 1: Success rates of the original CS course compared with three offerings of Media Computation course

Table 1 lists the success rates (defined as the proportion of students who earn an A, B, or C in the course–the additive inverse of those who withdraw or earn a D or F) for our introductory computing course from 2000 to 2002, when all students at Georgia Tech took the course; then the three offerings of the Media Computation course. Around 90% of the Media Computation students succeeded, compared with 72% of the students across all of campus (including CS, science, and engineering majors). Our best-practice comparison is with non-CS majors in a pair-programming offering of CS1 which had a 66.4% success rate[16]. It should also be noted that over half of the students in the Media Computation class have been women [20].

In Spring 2004, we conducted an online survey all the 425 students who took the course in Spring 2003 ($n = 120$) and Fall 2003 ($n = 305$) semester. We sent an email invitation to participate in the survey, which was available as a web form. Sixteen of those students had since graduated, leaving us 409 potential subjects. We had 59 respondents, for a response rate of 14%–not tremendous, but not unreasonable considering we were asking for feedback from non-majors on what was a service course.

The results suggested that students were still engaged with computer science since the course ended. We asked students to report on how much they had talked about computer science with their friends or relatives before and after the course. 64% of the respondents indicated that they engaged in computer science discussions more often since taking the course. Eleven of the 59 respondents (19%) indicated that they had written programs in Python since completing the course. We asked students whether they had edited pictures, sounds, or videos before taking the course, and since. 27% of the respondents indicated that they had edited media since taking the course although they hadn't previously. Only one of the students had taken another CS course.

We asked for open ended responses to the question, "How would you say that CS1315 has changed the way you interact with technology, if at all?" 47 of the 59 students (80%) did indicate that CS1315 had an impact on the students' relationship with computing technology.

- That result implies that 20% did *not* indicate that CS1315 had an impact on them. A typical response is, "*No, I think CS was completely irrelevant to my college career. Python is a language I will never remember because it is likely I will never use it again.*"

- Others, however, offered a detailed explanation on how the course had impacted them, such as "*It made me understand more how computers work so I can use*

them better. Helped me use the normal programs like email and internet better. And I know how picture editing works, which is cool." and "*I have learned more about the big picture behind computer science and programming. This has helped me to figure out how to use programs that I've never used before, troubleshoot problems on my own computer, use programs that I was already familiar with in a more sophisticated way, and given me more confidence to try to problem solve, explore, and fix my computer.*"

We realized that we phrased our last question incorrectly. For many students, what changed after the Media Computation class was not how the students *interacted* with technology, but how they *thought* about technology–in a sense, it served as a "computing appreciation" course. Some example student statements include "*Definitely makes me think of what is going on behind the scenes of such programs like Photoshop and Illustrator.*"

4. CONCLUSIONS

The process that we describe in this paper is not specific to designing a Media Computation course for non-CS majors. Rather, we feel that this process is appropriate to follow whenever creating a CS course for non-majors.

The media computation course has been a success at Georgia Tech. There are now non-CS majors asking for *more* CS courses! We are creating a follow-on course, that introduces data structures in a media context. We have also now defined a CS minor option, so that students interested in computing can go into more depth without leaving their own majors.

5. ACKNOWLEDGMENTS

This research is supported in part by grants from the National Science Foundation (CISE EI program and DUE CCLI program), from the Al West Fund at Georgia Tech, and by the College of Computing and GVU Center. We wish to thank all the students who helped create JES and the Media Computation class, and all the students in the class who volunteered to participate in our studies. Our thanks to Adam Wilson who has shepherded development of JES for the last few terms, and to Bob Amar who designed and implemented the follow-up survey.

6. REFERENCES

[1] AAUW. *Tech-Savvy: Educating Girls in the New Computer Age*. American Association of University Women Education Foundation, New York, 2000.

[2] ACM/IEEE. Computing Curriculum 2001. `http://www.acm.org/sigcse/cc2001`, 2001.

[3] B. Adelson and E. Soloway. The role of domain experience in software design. *IEEE Transactions on Software Engineering*, SE-11(11):1351–1360, 1985.

[4] J. D. Bransford, A. L. Brown, and R. R. Cocking, editors. *How People Learn: Brain, Mind, Experience, and School*. National Academy Press, Washington, D.C., 2000.

[5] J. T. Bruer. *Schools for Thought: A Science of Learning in the Classroom*. MIT Press, Cambridge, MA, 1993.

[6] E. Chabrow. Declining computer-science enrollments should worry anyone interested in the future of the u.s. it industry. *InformationWeek*, 2004.

[7] M. Felleisen, R. B. Findler, M. Flatt, and S. Krishnamurthi. *How to Design Programs: An Introduction to Programming and Computing*. MIT Press, Cambridge, MA, 2001.

[8] M. Greenberger. *Computers and the World of the Future*. Transcribed recordings of lectures held at the Sloan School of Business Administration, April, 1961. MIT Press, Cambridge, MA, 1962.

[9] M. Guzdial. *Squeak: Object-oriented design with Multimedia Applications*. Prentice-Hall, Englewood, NJ, 2001.

[10] M. Guzdial. Use of collaborative multimedia in computer science classes. In *Proceedings of the 2001 Integrating Technology into Computer Science Education Conference*. ACM, Canterbury, UK, 2001.

[11] M. Guzdial. A media computation course for non-majors. In *Proceedings of the Innovation and Technology in Computer Science Education (ITiCSE) 2003 Conference*, New York, 2003. ACM, ACM.

[12] J. Kolodner. *Case Based Reasoning*. Morgan Kaufmann Publishers, San Mateo, CA, 1993.

[13] J. Margolis and A. Fisher. *Unlocking the Clubhouse: Women in Computing*. MIT Press, Cambridge, MA, 2002.

[14] L. A. Miller. Programming by non-programmers. *International Journal of Man-Machine Studies*, 6:237–260, 1974.

[15] L. A. Miller. Natural language programming: Styles, strategies, and contrasts. *IBM Systems Journal*, 20(2):184–215, 1981.

[16] N. Nagappan, L. Williams, M. Ferzil, E. Wiebe, K. Yang, C. Miller, and S. Balik. Improving the CS1 experience with pair programming. In D. Joyce and D. Knox, editors, *Twenty-fourth SIGCSE Technical Symposium on Computer Science Education*, pages 359–362, New York, NY, 2003. ACM.

[17] D. B. Palumbo. Programming language/problem-solving research: A review of relevant issues. *Review of Educational Research*, 60(1):65–89, 1990.

[18] S. Pedroni and N. Rappin. *Jython Essentials*. O'Reilly and Associates, 2002.

[19] S. L. Pfleeger, P. Teller, S. E. Castaneda, M. Wilson, and R. Lindley. Increasing the enrollment of women in computer science. In R. McCauley and J. Gersting, editors, *The Proceedings of the Thirty-second SIGCSE Technical Symposium on Computer Science Education*, pages 386–387. ACM Press, New York, 2001.

[20] L. Rich, H. Perry, and M. Guzdial. A CS1 course designed to address interests of women. In *Proceedings of the ACM SIGCSE Conference*, pages 190–194, Norfolk, VA, 2004.

[21] E. Soloway, J. Bonar, and K. Ehrlich. Cognitive strategies and looping constructs: An empirical study. *Communications of the ACM*, 26(11):853–860, 1983.