Designing a small-footprint curriculum in computer science

Allen B. Downey and Lynn Andrea Stein
Olin College of Engineering
Needham, MA 02492
{allen.downey, las}@olin.edu

***Abstract*— We describe an innovative computing curriculum that combines elements of computer science, engineering and design. Although it is tailored to the constraints we face at Olin college, it contains elements that are applicable to the design of a CS major at a small school, a CS minor, or an interdisciplinary program that includes computing. We present the core courses in the program as well as several courses that are meant to connect the computing curriculum to other fields. We summarize the lessons we have learned from the first few years of this program.**

I. Introduction

Necessity breeds invention. At Olin College of Engineering we had the opportunity to design a computer science curriculum from scratch, but we face several constraints. Like most small colleges, we have only a few faculty in members in computer science (CS); in addition, we have to keep our major requirements small to accommodate general requirements in engineering, science and math. And, like many computing programs, we are trying to make more room for soft skills like teamwork, communication and life-long learning.

This paper presents our Engineering with Computing curriculum, a concentration combining elements of engineering, design, and CS. We discuss the objectives of our curriculum and its tradeoffs. Although Olin's context is unique, much of what we have learned applies to the design of a CS major at a small school, a CS minor, or an interdisciplinary program that includes computing.

The core of our curriculum is three classes that cover program design, theory, and software systems. In both cases we have refactored the CS curriculum by combining material from several standard classes. Our theory class integrates elements of complexity/automata theory, algorithms, and programming languages. For example, we teach the logarithmic/exponential relationship of tree structure—the foundation of many data structures and algorithms—while showing how that same logarithmic structure underlies binary numbers, the short certificates of NP-complete problems, and the stack trace of expression evaluation in programming languages. The systems course covers operating systems, networks and database implementation, allowing students to understand recurring themes (like locality and caching) and appreciate the design of systems that blur conventional boundaries.

Refactoring gives us an opportunity to demonstrate connections among topics in a way that is clearer than in the conventional structure, but it also forces us to make decisions about what to omit. We mitigate these omissions by providing students with an overview of de-emphasized topics and by helping them develop capacity for life-long learning. For example, several classes include explicit instruction and practice at reading dense technical material.

At this point we have taught two complete iterations of our curriculum. This paper presents our experiences and observations, although for now they remain generalizations without formal evaluation. At this point we have anecdotal information about early outcomes: our undergraduates regularly receive internship and research offers over the summer, and the feedback from their employers is positive. Our first graduates have been accepted to prestigious graduate programs and have been offered competitive jobs. We believe that our students are differently educated from graduates of more traditional CS programs, but it appears that they are of at least equal interest to employers and academic institutions.

A. Olin College

Olin College of Engineering was founded in 1997 with the goal of exploring innovative approaches to engineering education. The first group of students matriculated in 2002 and will graduate in May 2006.

Olin's curriculum is intended to address problems identified by a number of agencies and organizations concerned with the directions of engineering education. Specifically, Olin emphasizes [1]:

- A shift from disciplinary thinking to interdisciplinary approaches;
- Increased development of communication and teaming skills;
- Greater consideration of the social, environmental, business, and political context of engineering;
- Improved student capacity for life-long learning; and
- Emphasis on engineering practice and design throughout the curriculum.

These values are shared by many other institutions; Olin was founded specifically to pioneer ways of incorporating them into an engineering curriculum.

In each year of the Olin curriculum there are several classes that involve project work. Some educational experiences follow the traditional pattern of acquiring knowledge and then

applying it ("learn then do"), but many others take the "do-learn" approach, allowing students to explore and learn as needed before receiving explicit instruction. We believe that this approach builds students' pragmatic skills (including diagnosis and debugging) and their capacity for life-long learning. At the same time, it necessarily reduces the guarantees that we can make about specific content delivery to our students.

Olin also emphasizes the context of problems and solutions. We stress that it is not enough to produce a technically feasible solution to a problem; the solution must also meet the needs of its human users, integrate into their social and organizational context, be pragmatically manufacturable and financially and organizationally sustainable. Olin's mission is to train engineers who understand the social and pragmatic aspects of problem-solving as well as the more technical aspects. This requires the ability to communicate in both technical and non-technical contexts and to work effectively in multidisciplinary teams.

B. Engineering with Computing

Olin offers three undergraduate degrees: in Mechanical Engineering (ME), in Electrical and Computer Engineering (ECE), and in Engineering. Computer Science comes in the form of the Engineering with Computing program (E:C), which is a concentration within with Engineering degree.[1] All Olin degree programs are designed to be completed in four years of coursework with no more than four courses in each semester.

Major programs at Olin minimize the number of course requirements in order to provide flexibility and to accommodate general requirements in Engineering; Science, Math; and Arts, Humanities, Social Sciences, Business and Entrepreneurship (AHSE).

All students are required to complete a foundation curriculum that includes 1.5 semesters (i.e., 6 courses) in engineering principles and practices, modeling, and design; 1.5 semesters of Science and Math; and 0.5 semesters in AHSE (followed by advanced elective coursework in each of these areas).

In the senior year students complete a "realization" component that involves a year-long team-based project drawn from industry; a semester-long capstone experience in AHSE; and a semester-long independent study of a topic of the student's choosing.

After these requirements and a small number of electives, what is left for the "specialization" component of the curriculum is approximately 1.5 semesters of coursework in the major. Of course, it is a challenge to fit a major program into this small footprint. Relative to other majors, the E:C program faces an additional challenge because the foundation courses provide little specific preparation for the computing curriculum. Also, because the number of students who choose the E:C major is small, we have an obligation to provide courses that appeal to a broader set of students.

[1]One E:C core course also serves as a core course for the ECE program, and several core courses in ECE and in E:C can be counted as electives for the other major.

C. Outline

The following sections describe the set of courses we have designed to satisfy these constraints.

Section II describes the classes we have designed as bridges between computing and other areas of interest to the Olin student body. These courses include the introductory courses, **Introduction to Programming** and **Software Design**, as well two advanced courses, **Human Factors and Interface Design** and **Computational Modeling**.

Section III describes our core courses, **Foundations of Computer Science** and **Software Systems**. These courses cover much of the traditional computing curriculum, though refactored and compressed.

Olin's computing offerings are augmented by classes in **Discrete Math** and **Computer Architecture** as well as more unusual electives including **Synchronization** and (a planned course in) **Computer Systems and Public Policy**.

II. Bridges to Computing

Because Engineering with Computing is only a small part of Olin's curriculum, most of our courses are designed with the whole student body in mind.

A. Introductory Programming

A perpetual challenge of the introductory sequence is dealing with the range of preparation in incoming students. About one third of our students did not write a line of code before college. Of the other two thirds, many are at the level of the Computer Science Advanced Placement Exam or beyond.

To mitigate this problem, we offer Introductory Programming, a 2-credit elective for incoming first-year students (2 contact hours per week plus 4 hours of preparation). This course is taught in parallel with an integrated course block that includes Calculus, Mechanics and a project-based introduction to modeling and control. It covers basic programming skills in MATLAB: variables and values, mathematical expressions, functions, and tools for solving linear and non-linear systems. Many of the example programs are based on work from other classes, so the motivation is more natural than in some other introductory classes.

Introductory Programming provides the basics students need in other classes and gives them a head-start if they go on to Software Design. It is partly self-paced; students work through a sequence of ten modules. Although the class runs on a schedule, students can complete any module at any time before the end of the semester.

Because the first semester is graded Pass/No Credit and this course is not required, students can fail to complete it without consequences. Some students who discover that they do not need the class drop out in the first few weeks. Other students who hit a time crunch in the middle of the semester drop out for a while, but some of them come back to complete the class. A few students who have a particularly hard time with programming are not able to complete all ten modules.

The goal of the course is to allow as many students as possible to get what they need with minimal risk. As a result,

we have seen students who might have been scared off by a conventional introduction, whose interest was piqued in this class, and who went on to success in Software Design.

B. Software Design

Software Design addresses the specialized problems of designing when your medium is software. It teaches students how to think about software artifacts and their construction/architecture. In addition, the course builds skills in programming, code reading, and debugging.

This course takes a novel approach to the subject of computer programming, differing from the traditional one both in the questions that are asked and in the territory that is covered. We provide a brief description here; this course is described in detail in a previous paper [2].

Software Design presents computation as an ongoing interactive process. The programs that students work with are inherently concurrent and embedded in a context; they provide services or demonstrate emergent behavior. In a single semester students progress from simple expressions and statements to client/server chat programs and networked video games; these topics proceed naturally and straightforwardly from the interactive computational metaphor at the heart of the class.

Students are taught to phrase the programmer's questions in terms of the relationships between components (rather than algorithmic sequencing), so that topics like push vs. pull, event-driven programming, message passing, and network communication become integral aspects of this course. The curriculum exploits this shift in the fundamental story of programming to restructure what is basic and what is advanced material. In other words, this course does not go deeper into the curriculum than a traditional introductory course; it stands the traditional curriculum on its end.

Because the computations developed in this course differ from those encountered in traditional programming curricula, the course is suitable for students with no prior programming as well as those with background comparable to the AP Exam.

Instruction includes programming language syntax, but greater emphasis is placed on incremental design, development, and debugging; construction of conventional and specialized interfaces; and association of program behavior with particular patterns of code.

A major component of the class is a weekly two-to-three hour lab. Much of this time is spent in collaborative work on program development, with an emphasis on student-student interaction and student-student teaching, facilitated and enriched by the course staff. In addition, design and implementation work is supplemented with observational laboratory assignments, inviting students to consider not only how to build a program, but how to anticipate its behavior and how to modify that behavior.

To address the range of preparation students bring to this class, assignments include some elements that are required for all students and optional elements that challenge more advanced students. Optional work allows students to develop their potential, but does not affect their grade, so it is possible for a student with no prior experience to do well in the class by displaying mastery of the basics.

In the second half of the course, students work in small teams on a substantial project. Examples includes networked video games, a peer-to-peer file sharing system, and programs that analyze and generate text, sounds and images. For many of them, it is their first experience programming with a team. We emphasize the importance of learning to communicate about programs and the need for common vocabulary.

It is sometimes difficult to convince advanced students that the course has value for them. Some students are naturally focused on the task of getting a program to work (by any means necessary), and do not appreciate the importance of process and style. These students often choose ambitious projects that would be infeasible for a single person in the given time. Because they are motivated to make the project succeed, they often come to understand the need for design, style, teamwork and communication, in addition to raw programming talent.

This course sets up the idea of concurrency and introduces students to programming in a concurrent world, but it does not show them how to solve all of the problems that they will encounter there. Students see related material in Software Systems and have the option of pursuing it in depth in Synchronization (see Sections III-B and III-C).

C. Human Factors and Interface Design

Several classes in our curriculum combine students in the E:C major with other students who have less experience with software. One of these is Human Factors and Interface Design (HFID). While this class addresses aspects of human-computer interaction, it is not a heavily computational course. It relies as much on students' engagement with the design process as on their software skills.

Olin emphasizes design, with a special focus on understanding the users of a product or system as well as the context of its use. Every student takes a sequence of classes in design, including User-Oriented Collaborative Design, which requires students to choose a user group and engage those users as participants in a design process.

HFID continues the students' exploration of design and focuses them on the peculiar problems associated with usable software-centered systems. Because software enables us to accomplish so much, software users and especially programmers often overlook how difficult and unpleasant the experience of using software products often is. Too many software professionals focus on the problem of making the system work almost to the exclusion of making the system usable. This course, like other usability-focused curricula, emphasizes the need for a solution to work from a human, social, organizational perspective as well as from a purely technical standpoint.

Most students in HFID have taken Software Design, although it is possible to complete the class with only web programming experience. Over the course of a semester, student teams identify a software-centered application and

its user group, create an improved interaction/interface, then repeatedly refine their interface through user testing and formal evaluation. Students may choose to work on web services, standalone applications, or physical devices with software-driven interactions. Student projects have included an innovative music player/organizer, a location-aware PDA for tourists, and interfaces for campus information systems.

D. Computational modeling

Most physics is based on continuous models and differential equations, and most scientific computing is focused on numerical solutions of those equations. But the availability of computational power has led to different kinds of models and physical laws. This development is the subject of Computational Modeling, an elective class for students in E:C and ECE.

Topics include graph algorithms, small-world graphs and scale-free networks; cellular automata and turmites; self-organized criticality, long-tailed distributions, Zipf's Law, Pareto's Law, long-range dependence; spectral analysis, fast Fourier transform, $1/f$ noise; Bayesian statistics; event-driven simulation, the heap implementation of a priority queue; agent-based simulation, emergent properties; philosophy of science, realism and instrumentalism, holism and reductionism.

This class includes several data structures and algorithms that are usually covered in a Data Structures class. One problem with conventional Data Structures is that the ideas are presented without motivation. In Computational Modeling we are able to present each data structure in a context with an immediate application. For example, in the small world module, students implement Dijkstra's shortest path algorithm and use it to measure the decrease in average path length as "long links" are added to a random graph.

This class is a survey, by necessity, because most of the material is relatively new; at this point it is hard to evaluate which topics are fleeting curiosities and which will be important parts of science and engineering in the 21st Century. To give the students the big picture, we started the semester by surveying the treatment of these topics in popular non-fiction. Books included Watts's *Six Degrees*, Wolfram's *A New Kind of Science*, Kaufman's *At Home in the Universe*, and Surowiecki's *The Wisdom of Crowds*.

In the second part of the semester, the students read academic articles on each topic, in many cases the seminal paper where each major idea was presented; for example, Watts and Strogatz's paper on small world networks and Bak, Tang and Wiesenfeld's paper on self-organized criticality. We believe that using popular non-fiction as a prelude to more detailed technical material is an effective motivational tool.

III. The Computing Core

Software Design is the entry point to Olin's computing core. The remainder of the core consists of two courses intended to help students "learn to learn" in computer science: Foundations of Computer Science and Software Systems.

A. Foundations of Computer Science

Foundations of Computer Science (FOCS) covers material that is typically found in courses on automata and complexity theory, algorithms, and programming languages. By presenting a cross-section of this material, the class draws out connections that may otherwise be less apparent. It also contrasts the kinds of questions and approaches that are common within these subdisciplines.

Clearly, a single class cannot cover the same amount of material that is usually found in three separate classes. Although we gain some efficiencies by combining presentation, most of the compression comes at explicit loss of detail. Still, this class is not a survey of the three areas or a shallow introduction. The goal is not to teach students everything they might need to know about theoretical computer science; instead, it is to prepare them to learn anything that they might need to know.

Specifically, this means giving them the language and intuition to understand automata and grammars, algorithms and data structures, complexity and computability. In addition, the course introduces two programming languages that are different from those most students will have encountered: Scheme, to teach the idea of programming without relying on assignment, and Prolog, to demonstrate that it is possible to separate programming from control flow.

For example, the course covers algorithms for sorting, but focuses on the theoretical lower bound and the contrast between naive sort algorithms and those that exploit the logarithmic structure of trees (either in divide-and-conquer or answer-accumulating fashion) to achieve that theoretical bound. We compare the use of tree structure in sorting to its application in search and identify the importance of balancing trees to maintain the logarithmic property. We also look at ways to break the assumptions of sorting's theoretical lower bound, such as knowing the range of objects to be sorted (in a bin sort).

Scheme provides a nice start to the class, showing students an approach to programming that is different from what they know (typically Java or Python). Through Scheme, the students learn to think in terms of functional programming, with special emphasis on how to program without assignment. Scheme also provides an opportunity to explore the role of the stack in recursive procedure execution. After a brief side excursion into array and linked list implementations of the stack abstract data type, we return to contrast the stack-based recursive pattern of execution with (stackless) iteration in the form of tail recursion.

These patterns recur in other parts of the course. When we discuss finite state automata, we find that precisely the same distinction—between stack-based and stackless control—exemplifies the limitations of finite languages when contrasted with push-down automata. Further, the process of repeatedly pushing onto and popping off of a stack is the fundamental operation behind a depth first tree walk. This same process of stack-based tree-walking is performed by expression evaluation in Scheme as well as by parsers, Prolog

programs and logic reasoners, and the brute force computation of solutions to NP-complete problems.

The final section of the course introduces the fundamental limits of computation. This includes the introduction of the Turing machine and demonstration of its universality as a computational model, followed by the Halting theorem and Godel's Incompleteness Theorems.

An optional concurrent course gives students the opportunity to apply the material covered in Foundations of Computer Science. The projects are brief, generally two weeks for each one. They include building small parsers, evaluators, finite state animations, and other programs that reinforce the ideas behind the techniques covered in the main class.

Students leave FOCS conversant with the major categories of results in theoretical computer science and ready to enroll in advanced coursework in these areas.

B. Software Systems

Software systems combines material from conventional classes in networks, operating systems, and database implementation. In compressing three semesters into one, we have to make decisions about what to leave out and what to cover more quickly, but the volume of the material is not entirely conserved, because there are cases where we can pack it more efficiently.

For example, we start the semester with a two-parameter model of communication: transmission time as a function of latency and bandwidth. This model applies to networks, of course, but it also applies to disk drives, buses, etc.

This combination of material also provides a broader view of some topics. For example, most operating systems classes talk about the implementation of general-purpose file systems. In Software Systems, we are able to approach database implementation as an example of a file system designed to support a particular set of operations. This view leads students to see the connection between data structure design and file system design: in both cases the designer chooses the structure that provides the best performance for the expected workload; the primary difference is the performance profile of the underlying hardware.

A major theme of this class is experimental design, a skill that is critical in graduate school and often important in industry, but seldom taught explicitly. Students work on a series of exercises that require them to characterize the performance of various network and operating system features, infer information about their implementations, and present their findings in the form of short technical papers. These exercises are described in more detail in a previous paper [3].

Combining networks and operating systems in one class also reflects the technological trend toward distributed systems. Where networking used to be a feature of an operating system, it is now an integral part. In our approach it is natural to address the design of distributed systems throughout the semester, not at the end or as an advanced topic. This class is also one of several places in the curriculum where we give students explicit instruction in reading research papers.

So what gets left out? Although students write C code that exercises various operating system features (for example, constructing and sending a UDP packet), they don't implement or modify features at the kernel level. We don't cover deadlock detection or recovery, and synchronization is covered briefly because we offer a half-class on the topic that goes into more depth (see below). Finally, students only work with one operating system (Linux) so they don't see a wide range of solutions.

In the second half of the semester, students work in teams on projects of their own choosing, so they have an opportunity to get into at least one of these topics in depth.

C. Synchronization

Synchronization is the study of software tools for managing concurrent threads. It is usually a module in an Operating Systems class, and usually includes semaphores, condition variables, monitors, and a suite of problems that can be solved with these tools. The classical synchronization problems are abstractions of common patterns that occur in system software and some applications. For example, the readers-writers problem is an abstraction of the constraints on concurrent access to a mutable data structure.

This material is challenging, and it takes time for students to gain facility with it. In a conventional operating systems class, students may come to understand solutions to classical problems, but few of them are able to formulate new problems or construct solutions.

This course is an attempt to address these limitations by extending the time students have to absorb and work with this material. It is based on *The Little Book of Semaphores*, which was written by one of us in an attempt to identify the patterns that underlie synchronization code and assemble those patterns into solutions to a suite of problems [4]. The book is organized as a sequence of increasingly-difficult puzzles, where each puzzle is followed by a hint and then a solution.

In our experience, most students are able to solve each problem on their own, and the rest are able to understand the solution when we discuss it in class. By the end of the semester, students are able to assemble solutions to the most complex synchronization problems we have found, and they start to invent new problems, often based on synchronization patterns they see in the real world. The latest edition of the book includes several problems invented by students.

This topic gets more weight in our curriculum than in most, in part because we think it is important and challenging material that develops useful thinking skills, and in part because it is a personal interest of one of us. More generally, this course is an example of an approach we think is effective for material that does not fit well in a sequence of modules; we spread it over a longer interval to give students more time to absorb it.

IV. Why should anyone else care?

Olin is not alone in feeling pressure to shrink the footprint of the core computing curriculum. Many small CS departments

are in a similar position. (See, for example, the Computer Science Small Department Initiative Report [5] and model curricula for computer science at liberal arts colleges [6], [7], [8].) Even in larger departments, there is pressure to add new material as the field develops, to make connections to other fields, and to give more attention to soft skills [9], [10].

A. Soft Skills and Learning to Learn

Industry continually reminds us that students need training in teamwork, project work and communication as well as opportunities to engage in meaningful projects that reinforce existing knowledge and prompt further learning.

Teaching soft skills takes time and—to the extent that students engage in self-directed exploration—yields less predictable coverage of core content. For example, when students research and present topics to the class, they gain independent learning skills and practice communication, but the particular topics covered may vary and the extent of coverage may not be as great as if the instructor taught the material directly.

Our experience suggests that a small footprint curriculum can mitigate these problems: reducing the size of the core leaves time for greater exploration; refactoring content from multiple courses ensures coverage of fundamental material.

B. Teaching Multidisciplinary Students

Computer Science departments are increasingly facing declining CS enrollments and simultaneously increasing interest in hybrid programs such as Computational Biology or Interactive Media. Dickey's survey of model curricula includes a number of examples [11].

Students in these new disciplines need exposure to the key ideas of computer science, but computing coursework competes with coursework in other disciplines. Curricula in these emerging areas must reduce the footprint of the computing component to allow students to build background in complementary disciplines and in hybrid courses. For example, students in an interactive media program need to understand computational thinking, but they also need to learn art and technology. Computational biologists need grounding in both biology and computer science.

A refactored core allows students to get a complete but less detailed picture of computer science, rather than a subset of a larger deeper curriculum. Additional coursework can be tailored to the needs of the specific multidisciplinary program or to more advanced study in computing itself.

V. Conclusions

This paper describes the constraints that drive our curriculum and the courses we designed to satisfy those constraints. A number of lessons emerged from this process that we believe are applicable to other programs in computer science:

- Compressing the core of the CS curriculum is a necessity at many schools, but may be a virtue at others. By relieving the obligation of coverage, it facilitates other kinds of innovation.
- Combining material from several standard classes creates opportunities to make connections that are less apparent in other curricula, and to move students to the research frontier more quickly.
- Teaching introductory computer science in a low-risk environment allows students to get over the initial hurdle and achieve a level of programming skill appropriate for their interests and needs.
- Teaching concurrency early in the introductory sequence makes it easier to serve students with a wide range of preparations, and develops a kind of thinking they are likely to find valuable.
- Courses that serve non-CS majors along with majors can help departments deal with variability in enrollments, enrich the classroom experience, and foster the development of computer science as an interdisciplinary field of study.

References

[1] M. Somerville, D. Anderson, H. Berbeco, and et al., "The Olin curriculum: Thinking toward the future," *IEEE Transactions on Education*, vol. 48, no. 1, pp. 198–205, 2005.

[2] L. A. Stein, "What we've swept under the rug: Radically rethinking CS1," *Computer Science Education*, vol. 9, no. 2, pp. 118–129, 1998.

[3] A. B. Downey, "Teaching experimental design in an operating systems class," *SIGCSE Bulletin*, vol. 31, no. 1, pp. 316–320, 1999.

[4] ——, *The Little Book of Semaphores*. Green Tea Press, 2005, available from http://greenteapress/semaphores.

[5] "The Computer Science Small Department Initiative (CS SDI) Report," *ACM SIGCSE Bulletin*, vol. 36, no. 1, pp. 332–333, March 2004, http://cssdi.org.

[6] N. E. Gibbs and A. B. Tucker, "Model curriculum for a liberal arts degree in computer science," *Communications of the ACM*, March 1986.

[7] "Revised model curriculum for a liberal arts degree in computer science," *Communications of the ACM*, December 1996.

[8] K. Bruce, A. Brady, and et al, "The 2003 model curriculum for a liberal arts degree in computer science," in *SIGCSE*, 2004, special Session, http://www.lacs.edu/model-curriculum.pdf.

[9] J. Kurose, B. Ryder, C. Kelemen, and et al., "Report of NSF Workshop on Integrative Computing Education and Research, Northeast Workshop," November 2005.

[10] *Computer Science: Reflections on the Field, Reflections from the Field*. Committee on the Fundamentals of Computer Science: Challenges and Opportunities, Computer Science and Telecommunications Board, National Research Council, National Academies Press, 2004, http://www.nap.edu/catalog/11106.html.

[11] M. Dickey, "Model curricula for undergraduate programs in computer science and related fields," http://www.cs.washington.edu/homes/dickey/curricula/.