Revitalizing the Computer Science Curriculum

Gary Nutt

Department of Computer Science
University of Colorado

Technical Report Number CU-CS-1020-06
December 2006

Abstract

Computer science educators and other stakeholders often lament the state of the curriculum in university computer science programs. Recently, NSF announced a program to attempt to stimulate change in the undergraduate education (see the CPATH announcement). There is wide concern that implications of an aging curriculum include decreasing (or at least fluctuating) enrollments in computer science programs, inadequate preparation of undergraduate students to enter the commercial workforce, and a general mismatch of the education with the state of information technology in the real world. This paper reviews the state of contemporary undergraduate curricula in U.S. universities, considers some important information technology trends (including how they are not supported by extant curricula), and suggests how educators can use the IEEE/ACM recommendations on curricula to influence reform.

1. Introduction

Information technology (IT) is essential to today's society, providing a means for conducting commerce, disseminating information, and entertaining us. Because of its importance to society, universities have an incentive to educate students about IT so that they are well prepared to enter the workforce, and ultimately to help the nation remain competitive in today's IT environments.

There is serious concern that there is a growing mismatch between the IT taught in colleges and universities and that needed to staff competitive workforces, e.g., see NSF's CPATH announcement calling for a national effort to revitalize the way IT is taught in the nation's colleges and universities [2]. This paper examines the models used to define and maintain undergraduate computer science curricula, the nature of a few contemporary software development models, and then it makes some suggestions for how computer science departments can update their curricula to better prepare students to work in commercial IT development environments.

2. Contemporary Computer Science Curricula

Consider the chestnut that "the curriculum has not changed for X years" (where X is on the order of 30-40 years). Many argue that the curriculum has not changed much from the first recommendations (1968 and 1978), even after the IEEE and ACM released their current recommendations on computer science curricula in 2001. That is, the curriculum structure in which information is offered to students today is similar to the one proposed in 1968 – a time when computer science technology was based on batch processing operating systems, mainframes with 32KB of memory and 50 MB disk, Fortran and assembly programming languages, no network, no distributed computing, no interactive computing (punched card input), no graphics, etc. Let's examine the brief history of undergraduate computer science curricula to understand why this feeling is so prevalent.

2.1. The Early Years

In the 1960s and 1970s, computer science was just forming as an academic discipline – Purdue University organized the first computer science department in the US in 1962. At about the same time, other departments were forming, often as adjuncts to the math department or electrical engineering, e.g., University of California at Berkeley and M.I.T. computer science were parts of their respective electrical engineering departments. In this timeframe, it was not uncommon for computer science to form as a graduate program that was part of the Graduate School, rather than being associated with either engineering or math, e.g, as was the case for the Universities of Illinois and Washington.

During the 1970s, most of these groups were focused on becoming viable *graduate* research departments, although they typically offered undergraduate service courses for majors in other disciplines. At that time, if an undergraduate wanted to major in computer science, he or she could study computer science as an adjunct to math or electrical engineering. Official undergraduate programs began to appear as early as 1972 (Illinois), but it was not until the early 1980s that the majority of the departments were offering an official undergraduate degree in computer science.

The 1968 ACM recommendation addressed both the undergraduate and M.S. curriculum [6]. The report partitioned computer science topics into *information* *structures and processes* (including data representations, programming languages, and models of computation), *information processing systems* (architecture, compilation, and operating systems), and *methodologies* (everything else – numerical mathematics, data processing and file management, text processing, graphics, simulation, information retrieval, artificial intelligence, process control, and instructional systems). It is not difficult to recognize this basic organization in contemporary undergraduate curricula – programming and data structures; systems and compilers; and specialized areas as they emerge.

In 1978, ACM updated their curriculum recommendation [7]; however the structure of this recommendation still seems to be based on the 1968 report. The 1978 report defined 8 core courses "… which should be common to all computer science undergraduate degree programs.":

CS 1. Computer Programming I
CS 2. Computer Programming II
CS 3. Introduction to Computer Systems
CS 4. Introduction to Computer Organization
CS 5. Introduction to File Processing
CS 6. Operating Systems and Computer Architecture I
CS 7. Data Structures and Algorithm Analysis
CS 8. Organization of Programming Languages

Additionally, the recommendation defined another 10 advanced elective courses that should be offered if the department's resources allowed it. These courses ranged across subjects such as Computers and Society, database management systems, AI, theory, and numerical math. As in 1968, the structure is built around programming and data structures; systems and compilers; and specialized areas as they emerge. File processing and algorithm analysis are the new areas compared to the 1968 report.

The 1976 ACM report was published at about the time that many computer science units were evolving into full-fledged departments, and at a time when many were preparing to install/tune their undergraduate programs. Why were so many departments inspired to launch an undergraduate program at this particular time? The workforce market demanded it: first and foremost, there was popular demand from enrolling students. Other factors included the fact that computer science as a research discipline was now well-established, and commercial software industry was in steep growth (in part because of the transition to open software models in which there was now significant commercial incentive to develop software). The raw number of people using computers was also growing rapidly. In the 1980s personal computers were beginning to appear in homes, and computers were generally poised to burst into public consciousness once the Internet became well established. By 1985, there were probably more than a hundred Ph.D. granting programs in the U.S., and most of them had either created an undergraduate program, or were planning on doing so. Most relied heavily on ACM Curriculum '78 as a guideline for establishing their program.

Consider the typical undergraduate curricula based on the 1978 report:

Freshman	Computer Programming I and II
Sophomore	Computer Systems and Computer Organization
	Data Structures and File Processing
Junior	OS and Programming Languages
Senior	Electives

2.2. Today's Curriculum

What has happened to the curricula since the 1980 time frame? While it is quite apparent that IT has changed considerably since that time, how has this technology been added to these curricula? Roughly speaking, it has been added one course at a time. That is, the structure of the typical curriculum has not changed very much since it was created. New technologies have been fit into individual courses – sometimes by changing the courses (e.g., to teach OO programming methodologies rather than C or Pascal), and sometimes by adding new elective courses, first at the graduate level and then as upper division courses. However it appears that the courses and their interrelationships remain largely as they were when the curriculum was defined for the department.

While the computer science curriculum is generally stale, other university units have begun to offer courses and programs to address changes in IT. In some cases math, electrical engineering, and business information system departments have continued to offer and refine their own IT programs. In other cases, new departments have been formed, e.g., some liberal arts departments have launched a program focused on the use of IT in the performing or publication arts. Universities also provide a plethora of academic certificate programs (e.g., network management, database administration, digital media, computer administration) offered by different academic units.

2.3. The Current IEEE/ACM Curricula Recommendation

IEEE-CS and ACM jointly published a recommendation on the undergraduate computer science curriculum in 1991, and then updated the recommendation in the current recommendation, Curriculum 2001 [9], and the associated Computing Curricula 2005 [12]. These reports recognize that there can several different kinds of undergraduate degree programs in computing – ranging from the study of information systems, to software engineering, to computer engineering, to information technology, and others as that may be arise in the future. As a result these reports partition IT into smaller *knowledge units* that can be combined to define a course (examples of knowledge units are programming fundamentals, algorithms and complexity, net centric principles and design, intelligent systems, E-business, software modeling and analysis, and digital media development). Courses are defined by defining collections of knowledge units; the course and curriculum specification are largely delegated to individual IT programs (although the report describes how these knowledge units can be weighted and combined to define a curriculum for the targeted degree programs).

The Computing Curricula 2005 report highlights the need for multiple IT curricula, the need for agility in any particular curricula through the use of knowledge units, market and the pressure from certificate programs (including training programs). The recommendation explicitly recognizes the diversity of requirements for different programs, and describes the idea of developing a portfolio of programs to address a college's strategic goals.

3. The Mismatch Between Commercial IT and the Curriculum

Let’s consider a few examples of contemporary commercial IT development environments. These examples illustrate the existence of computer science topics that are too large to simply add into the curriculum as another course. In some of the examples, other university units have stepped up to provide a curriculum (or certificate program) to address these educational needs. Computer science departments have difficulty competing in these markets – even though the markets are dominant in the commercial world and in their popularity among students – because of the outdated curriculum.

3.1. The Mobile Code Model

Object oriented (OO) programming appeared, first as a cult programming approach (recall the Simula ’67 and Smalltalk efforts); then as a buzz world (in the early 1990s); and ultimately as a fundamental programming methodology. Java was one of many OO language/systems that appeared in the 1990s. According to its developers, Java was originally conceived as an OO language that would be used in the context of digital cable networks (see http://java.sun.com/features/1998/05/birthday.html). Ultimately the developers turned their attention to transporting media content – and behavior – around the internet. By combining the OO technology with internet (and eventually WWW) technology, Sun was able to demonstrate the applet idea in 1994. This idea required that a web browser incorporate a Java Virtual Machine (JVM) to host the execution of a Java applet. Then when a web browser downloaded an HTML file with an embedded applet, the web browser could extract and interpret/execute the applet. In the paradigm, web servers distributed applets that would execute on behalf of the web server without explicit intervention of the web server. That is, the applet was a simple means of implementing mobile distributed programs as part of web content.

Academic researchers quickly recognized the power of mobile code, particularly in the context of object oriented programming, e.g., see the network objects papers [8]. Faculty began to incorporate information about remote objects and mobile code in their extant courses, but there was no ground swell modification of the curriculum to accommodate this new way of programming.

As the basic ideas matured, commercial software developers recognized the power of this approach in supporting web-based distributed programming. The mobile program was moved to a remote environment where it would execute a client-side protocol on behalf of the web server. Today, remote objects (particularly applets) are widely used to perform blocks of intense computation in a remote environment, for establishing a surrogate computation in a remote environment, etc.

Fueled by early success, this approach to mobile programming is now firmly entrenched as a commercial programming methodology – it is the basis of both commercial Java development and of Microsoft .NET software. Distributed virtual machines (such as JVM and Microsoft’s CLI) provide a platform for migrating objects, marshaling information, dynamically downloading components of the program, authenticating the veracity and source of mobile code, ensuring execution of the code on a variety of hardware platforms, and so on.

This mobile code technique is an important revolutionary change in the way programming is done – a way that also extends to distributed computing. There are several unique features about such an environment:

- Java and C# are OO programming languages
- Java and C# depend on strong typing to implement a secure sandbox for execution in remote environments.
- The JVM and CLI – runtime systems – are an essential part of the approach
- The approach decouples compilation from the target hardware using JIT compiling techniques.
- Applet extensions to web browsers depend on the completeness of the sandbox security model.
- These virtual machines support a multithreaded model of computation.
- The JVM and CLI distributed computing models support network objects
- Distributed objects intercommunicate using a form of remote procedure call

Each of these features are fundamental programming concepts that can be covered in some course in the extant curricula, although none of the features are necessarily in the mainstream of such courses. For example, OO programming is part of the introductory course; strong typing is part of the programming language course; the runtime system and JIT compilation (and interpretation) are sometimes covered in a compiler class, or perhaps touched on in computer organization. Mobile code execution environments might be discussed in a network class, an OS course (or maybe not at all). JVM as a multithreaded computation model and as a means for supporting network objects might be covered in a network course. Remote procedure call is usually described as part of network and OS courses. Mobile code can be addressed in programming languages, networks, and/or OS courses. The sandbox security model doesn’t quite fit anywhere (except as an optional knowledge unit that could be put into a security course, a programming language course, an OS course, etc.).

Commercial organizations (especially vendors like Sun and Microsoft) view this approach as the preferred (or at least very important) commercial software development environment; however a student with a B.S. degree from a conventional undergraduate program will have, at best, only studied the approach as a collection of bits and pieces spread across 4-5 different courses. The new approach simply does not fit into the traditional curriculum, and students will be largely naïve about mobile code models when they earn an undergraduate computer science degree.

What might an idealized curriculum look for preparing a student to be well prepared to take a job as a mobile code developer? The crucial first order technologies include:

- Computer system organization (for JIT vs native machine languages)
- Strong typed languages and the sandbox programming model
- JIT compiling and interpretation
- Runtime systems
- Hierarchical address spaces
- Execution engine
- Process/thread models and their implementations
- Client server computing
- Network objects
- Security models
- Encryption

Despite the fact that this is an important commercial software development model – arguably the dominant model – students who pass through a typical computer science curriculum will likely never have seen the various aspects of the model as a uniform IT approach.

3.2. Ubiquitous computing

Ubiquitous computing is generally acknowledged as having been launched about a dozen years ago in a series of papers by Weiser, e.g., see [14]. Ubiquitous computing typically refers to cases where a small, communicating computer (SCC) is used to interact with a number of other computers (some large, some small). In the last five years, commercial, ubiquitous computing has crossed a threshold in which there is now a thriving market for such products – PDAs, cell phones, media playback units, sensor networks, embedded systems, etc. In most cases, these SCCs operate with limited resources – limited battery power, limited CPU cycle speed, limited executable memory, limited type and number of devices, no keyboard, limited/no secondary storage devices, low bandwidth network, etc.

System and software design for SCCs is radically different from that used with a conventional desktop computer. For example, it is not unusual for an SCC to design the OS and middleware as a single unit – this is one popular technique for effective soft real-time scheduling capable of supporting audio and video applications (such as mp3 and mpeg playback). As another example, since an SCC cannot store significant amounts of information on its local devices, it must make relatively heavy use of its network to move data back and forth between itself and mass storage servers (the SCC may not even have a file system or storage devices).

Application software is also of a different style than the corresponding code for desktop machines. First, the HCI may be based on touch sensitive screens, audio I/O, etc. Secondly, since display space is frequently limited, the applications limit the amount of user prompting, style of graphics, amount of text displayed, etc. Reasoning about HCI and cognitive issues takes on a completely different tenor for ubiquitous computers, e.g., compared to desktop computing.

All of this repackaging of technology is best served by a corresponding flexibility in the curriculum. Somehow, the software design and implementation that currently fits in OS, programming, HCI, and so on needs to be repackaged to correspond to the structure of the system components in ubiquitous computers. Otherwise, graduating students will not be familiar with the approach, its limitations, its strengths, or how to contribute to an organization that builds/uses ubiquitous computing.

What would you want a person to know if you were going to hire them to work on your ubiquitous computing project? Here are some ideas:

- Alternative HCI models that matched with device characteristics
- New application paradigms that matched with alternative operating system services and programming languages
- New programming language and runtime models that reflect limited resources in an SCC.
- New OS technology with an extreme emphasis on distributed computing
- A system infrastructure to support (or at least be cognizant of) real-time computation.

What would an ideal curriculum look like for a student preparing to focus on ubiquitous computing?

- Basic programming and data structures
- Core computer organization, but with a twist on small computer organization, power management, alternative I/O devices
- Address spaces
- Coroutine/thread/process models and their implementations
- Client server computing
- Cross compile environments
- Minimal runtimes, but rich compile environment
- No OS environments
- Remote file systems
- RPC
- Maybe new HCI for handhelds
- Real-time computing

3.3. Web-based Computing

The World Wide Web (WWW or "the Web") computing models focus on moving diverse forms of information across geographically large distances. Scaling is extremely important in web-based computing, since successful enterprises must provide computational services to (hundreds of) thousands of computer clients. The diversity of information types that can be distributed over the web are mind boggling – ranging from simple static alphanumeric data (such as a catalog), to dynamic information (such as stock market quotes), to graphic images (JPEG images), to audio streams (MP3 songs), streaming audio/video content (MPEG video clips).

The web is built on the ISO OSI model, meaning that it depends on the Internet Protocol (IP) to route information throughout the internet. Streaming information distribution typically depends on time-critical transmission, in order to provide a constant flow of information to a playback device, or at the very least web caching. For example, it you wanted to watch a concert that was broadcast over the web, it would be important for the audio to be of relatively high quality in order to realize the value of the webcast. This technology depends on hardware, network, operating system, and application technology that supports predictable content delivery. In today's curriculum the technology is likely to be taught in many different courses, so graduates of a classic curriculum are not likely to have any significant appreciation for the use of the technology to support web-based computing.

Besides the system and network layer concerns for supporting web-based computing, the web applications themselves are radically different from typical desktop computer applications. For example, the web creates an environment for publishing audio/video clips (consider the current success of the You Tube web site), for artists to experiment with new art forms, for electronic commerce, etc.

At the University of Colorado (like many other universities), web-based computing is so important that there are a number of special topic courses addressing various facets of the technology – but no significant change in the computer science curriculum to accommodate such courses. Most significantly, there is a new ATLAS (Alliance for Technology, Learning, and Science) program that is explicitly intended to support learning about modern computing methodologies including web-based computing, prominently led by the certificate program in technology, arts and media (the TAM certificate) – see http://www.colorado.edu/ATLAS/. The TAM certificate program is offered through ATLAS at least partly because the courses that provide the essential content for the certificate are not consistent with the computer science curriculum.

- TAM: Future of Technology Arts, and Media
- TAM: Intro project course
- TAM: Capstone project course
- TAM: History and social implications of TAM
- TAM: Theory and Foundations of TAM
- TAM: Invention and Practice of TAM

This IT area focuses on:

- HLL programming and design basics
- Applets, CGI scripts, etc.
- XML
- HTML
- Multimedia
- Graphics programming
- Streaming media programming
- Client server computing
- Security models
- Real-time computing

3.4. Collaboration Technology

In 1980, Ellis and Nutt published a paper to entice mainstream computer scientists to consider the IT needed to support computer supported cooperative work [9]. A significant thrust of the paper was that "office information systems" was an emerging area that used technologies from many parts of the discipline. In the intervening 25 years, collaboration technology has continued to be a persistent (if not major) part of the commercial and research worlds, since people typically expect computer systems to be able to provide assistance when they work in groups.

Collaboration technology is cross-disciplinary in nature. It has a strong distributed computing component (e.g., touching networks, operating systems, and programming languages), significant middleware components such as databases, window management, email and messaging, and event-based programming). But it also is heavily influenced by workflow modeling, business process reengineering, human-computer interface considerations, cognitive modeling, group behavior, etc.

While there is no large and important commercial collaboration technology domain, it is safe to say that the computer science curriculum is not prepared to provide a comprehensive, integrated educational program to prepare people to work in this area. Important knowledge units in collaborative computing include:

- Programming and design basics
- Graphics programming
- Modeling procedures (workflow)
- Multimedia
- Client server computing
- Unstructured work environments
- Social models of interaction
- Collaborative work
- Security models
- Real-time
- Same/different time/place meetings
- Virtual environments

As in the other examples in this section, the curriculum is not well-prepared to teach undergraduates about the relevant technologies for building collaboration technology, in part because of the interdisciplinary nature of the relevant IT. Successful collaboration technologies projects typically have as much effort in cognitive science as in mainline IT.

3.5. Game and Entertainment Technology

We conclude our list of significant computing areas by mentioning game and entertainment technology – an area that is receiving considerable attention as a poster child for inadequate computer science curricula. Within the last half dozen years, many trade schools and universities have added this field of study to their repertoire, but not necessarily as an integral part of the computer science department. For example see descriptions of programs at the University of Denver [5], University of California at Santa Cruz (http://www.cs.ucsc.edu/~ejw/gamedesign/ComputerGameDesignProposal.pdf), and the University of Southern California (http://www.usc.edu/dept/publications/cat2006/schools/engineering/computer_science/undergraduate.html)

Like the other examples described above, the computer science curriculum is not structured to provide an education to prepare people to work in this area. Game developers focus on a particular slice through computer science, but also learn aspects of digital media studies, electronic arts, and media design [5]. Within IT, the curriculum might address real-time computing, alternative programming environments, virtual reality, extreme graphics, multimedia playback, human-computer interfaces, security models, cognitive modeling etc. A Bachelor’s degree in this area is also likely to provide students with legal, ethical, and moral educations that are not part of the conventional computer science curriculum.

4. Targeting a Curriculum

Computer science departments have traditionally not been prepared to handle all of the attention that they have received in the last dozen years. Computer science is a young discipline, and its faculties are typically highly focused on research. As IT has become so highly visible in the commercial world, computer science departments are often overwhelmed: other units in the university often respond to the demand for contemporary educations more quickly than does computer science – perhaps offering certificate programs, minors, or even alternative degrees in IT sub disciplines.

Essentially, IT areas have grown more rapidly than have computer science faculty. A department cannot possibly provide curricula for all emerging IT areas. It must consider its resources, and then carefully pick and choose new areas that it can support.

4.1. The Perfect Curriculum

It is easy to criticize curricula, but the criticisms from different parties who have a stake in the curriculum are not necessarily consistent with one another. There is currently a significant decrease in enrollments in computer science programs, causing considerable concern for almost all of the involved parties [1][4][11]. Of course there are many possible causes for this decrease: The ebb and flow of the economy probably has the largest impact on enrollment: either a significant increase or decline in the economy can increase enrollments for the short term, but it can also decrease enrollments if students lose faith that education will influence their ability to get a desirable job. The dot com bust of 2000 is an example of how the economy initially increased enrollments, then influenced a significant decrease in the longer term enrollment.

Most also agree that the perception of the curriculum is one of the most influential controllable parameters affecting enrollments. If a curriculum were "perfect," then enrollments would still tend to track the economy, but computer science departments could then have a framework in which it would be possible to provide the most useful education to the student (and thereby to future employers).

How does one determine that a curriculum is "perfect?" There are many interested parties in computer science education, including students, department administrators, faculty and other instructors, and prospective employers. Students are ultimately influenced by prestige of the university (and perhaps the department), public perception of the jobs in the discipline, and by the sizzle in the nature of the work. For example, creating special effects for movies has a lot of sizzle, but being an employee working in a Dilbert-like cubicle maze on programs to manipulate data does not have a high excitement factor. Most students do not aspire for a job in the latter environment.

College/department administrators are focused on department prestige and the budget. Prestige is generally earned by having the department being internationally recognized for its research, meaning that its faculty publish widely and are well-funded by grants. Most CRA[1] departments are recognized for graduating students that are especially well-prepared to enter graduate school, rather than to contribute in the private sector. A department can be recognized for preparing students to quickly contribute in the private sector public rankings of various institutions, e.g., the U. S. News and World Report rankings (see http://www.usnews.com/usnews/edu/grad/rankings/rankindex_brief.php). In these reports ranking criteria can vary widely, meaning that a university or department could be ranked highly because of its extraordinary football team or available social activities.

[1] The CRA (Computing Research Association) is a consortium of approximately 250 organizations – primarily Ph.D. granting departments in universities. "CRA's mission is to strengthen research and advanced education in the computing fields, expand opportunities for women and minorities, and improve public and policymaker understanding of the importance of computing and computing research in our society." (see http://www.cra.org/)

Faculty want to teach topics in which they are most interested. In Ph.D. granting institutes, these interests are likely to revolve around the faculty members' research interests. Faculty are often willing to expend considerable effort to organize a course to represent their notion of best content. It is more rare that they have time to focus on the larger problem of curriculum refinement.

For-profit employers want to hire students who are achievers (there is a high probability that such students are enrolled in a prestigious university), and who are familiar with the technologies used in that particular employers work products. Employers recognize that they will need to provide on-the-job training for each new employee, but an achieving employee who is already familiar with the technology will become productive more quickly than one who is unfamiliar with the technology. The curriculum provides this familiarity.

Each interested party in important to the education, and each has its own set of criteria by which it measures the quality of a curriculum and the resulting computer science education. While the criteria overlap across the parties, they certainly do not coincide. There is no perfect curriculum.

4.2. What Should a Department Teach?

Universities and departments are not intended to compete with trade schools. A trade school should *train* a person by teaching them to use targeted programming environments to produce commercial software. For example, a trade school might prepare students for certification (e.g. by Microsoft's Certification programs [3]) that verifies the student's ability to write programs for specific computing environments. By contrast, universities have a duty to *provide an education* that enables its students to understand the important concepts of a discipline in such a way that they can be applied to any particular environment or situation. But more importantly, a university education should stimulate students to continue a life-long learning process that enables them to keep abreast of changes in the chosen (and related) disciplines. In the context of programming, a computer science student should learn concepts that enable them to design and analyze information technology (IT) solutions in many different contexts, and to be prepared to comprehend subsequent evolution in IT. This suggests that the curriculum support much more than training people to write Java programs (even though it is useful if the student learns to write Java programs along the way).

Each department faces a fundamental *strategy* question of "how should a department organize the education it provides in today's IT environment?" Obviously the curriculum must address the needs/requirements of a spectrum of interested parties (or IT domains), yet still provide a long lasting education (as contrasted with targeted training). An essential aspect of the strategy selection is the identification of the "interested parties" – is the department focusing on preparing students to enter graduate school, a particular IT domain, or some combination that includes multiple domains. Once a strategy has been selected (that is consistent with the department's resources), the department can consider tactical question for implementing such a strategy of "how should the curriculum be defined?" Such decisions will influence whether or not various IT areas (e.g., introduction to theory or computer organization) are optional or essential in the curriculum.

5. Changing the Curriculum

A department can make significant progress toward updating its curriculum by first reading the IEEE/ACM curriculum reports (particularly the 2001[9] and 2005 [12] reports). But then the real work begins – determining the educational strategy based on sound goals and the resources that the department possesses. Early in this process the department must decide on its specific educational goals and then it can determine a strategy for achieving the goal. Once the fundamental groundwork has been established, the department can consider tactics for implementing the strategy, e.g.: (1) do little/nothing to the curriculum, but instead focus on changing course content; (2) take revolutionary action by throwing out the existing curriculum and starting the next freshman class on an entirely new curriculum; (3) devise a plan for incrementally changing classes and the curriculum so that they collectively move toward the desired goal; and (4) start a new program. Alternative (1) is the default approach, so it is the most often used – which explains why departments still use the same basic curriculum that they adopted in 198x. In cases where a small group of individuals decide to revolutionize the curriculum, it is likely that alternative (4) will be employed. All things considered, probably the only feasible path for significant curriculum change is to plan for evolution. For that reason, the recommendations in this paper are consistent with alternative (3), given that the department has invested the effort to determine goals and strategy that guide evolution.

5.1. Stating the Obvious: Barriers to Change

Even with a clear strategy and tactics, choosing a curriculum is necessarily a compromise across several competing constraints. For example, inertia is a significant barrier to changing the curriculum. The curriculum is made up of a collection of individual courses, each course of which is taught by a collection of instructors. An instructor generally makes a substantial time investment in the content and organization of the course, e.g., carefully considering the course content; choosing a textbook; understanding the prerequisites for the course; preparing lecture notes, supplementary notes, assignments, quizzes, and exams; and choosing an organization for teaching relevant information. Once the course content has been solidified, each class requires a substantial time investment – providing coaching and individual guidance, addressing individual students problems, organizing and guiding teaching assistants, etc. The cost of administering each class may be factored into the teacher's choice of course content. Most university teachers attempt to amortize all of these costs over several classes by teaching the course periodically.

Now suppose that it is desirable for a stable curriculum to change in a significant manner, e.g., by changing, say, half of the undergraduate courses in a curriculum. The first problem is that the new curriculum must define new interfaces among course in sufficient detail for each instructor to create the detailed materials for each course. An instructor will normally be inspired to teach new courses that overlap their own expertise. Once assigned to a new course, the instructor must invest the startup effort for each new course. Because of the lack of precision in course interface specification, the courses will have mismatches about where smaller topics are covered until the new curriculum has been used for a few iterations, i.e., over a period of years that may be as long as 4 years per iteration.

How long does it take “the market” to notice that a department changed its curriculum? There are at least two parts to “the market” – students and employers. Assuming that we focus on students who are inspired to enroll in a program because they believe it improves their job prospects, then it could take a decade for them to notice any real change in the curriculum. This follows because the product of the revised curriculum – the nature of the education – is not visible until a group of students have gone through the entire curriculum, which is normally 4-5 years (perhaps less if only part of the curriculum really changes). Employers are not likely to detect differences in career preparation for another few years, then another few years before that knowledge is perceived by prospective students.

This points out a new problem: if one could create a perfect curriculum for “the market” in 2007, in a rapidly changing industry like computer science, isn’t the market likely to have gone through extensive evolution (or even revolution) by the time the prospective students see changes in a curriculum in 2015?

5.2. Grassroots Curriculum Development

Suppose you are a faculty member in a department with an aging (if not outright obsolete) curriculum and course work. You know that the curriculum should address different areas such as those described in Section 2.3. What can you do?

Before you can make any significant progress, your department must create a set of educational goals and a strategy for the curriculum – possibly attempting to accommodate several specialize curricula within the general curriculum. If there is no such plan, then your first task is to work toward establishing one.

As a single person working on curriculum reform, it will take you a long time to make enough changes to have a significant effect, and an even longer time to convince your colleagues that your ideas are the right ones. You will be most effective if you are able to marshal help from colleagues – the more the better.

Consider what is involved in transitioning a traditional curriculum so that it supports a particular contemporary area, say, the Mobile Code Model. The part of the existing curriculum that directly overlaps the mobile code curriculum includes:

- Computer programming 1 (Introduction to Programming).
- Computer programming 2 (Data Structures and Complexity)
- Computer organization
- Programming languages
- Compilers
- Operating Systems

In order to create the ideal curriculum, the content of these courses would be modified, either by providing new versions of the course that focus on specific technologies, or by rearranging material across courses. An example of providing a new version might be for Computer Science 1 to be modified so that it prepared students for studying the mobile code model by introducing them to the Java or C# object model while they are learning the fundamentals of programming (in the department’s language of choice). This would prepare students to learn about mobile objects in a later course. An example of rearranging content across courses might be to modify the programming language and compiler courses so that one or the other explicitly addressed dynamically downloaded code modules (with authentication), strong typed languages, and the sandbox model.

For the mobile code curriculum, new material would appear in one or more new courses:

- Managed code execution (the sandbox model)
- The JIT translation/interpretation model
- Distributed programming
- Network objects
- Security models
- Encryption

The trick to providing new courses is to be able to present new material without overlapping older courses in the curriculum. And of course new courses require that new materials, including textbook materials, be developed for each such course. In the mobile code oriented curriculum, it is probably possible to incorporate much of this in three new courses. That is, the mobile code curriculum superficially looks like a conventional certificate program offered within the existing curriculum.

Suppose that the extant courses are tuned to accommodate the mobile code model, and three new courses are added for the mobile code emphasis (this sample curriculum is based on the University of Colorado sample curriculum):

Freshman:	(Fall) Computer programming 1 (using an object-oriented language)
	(Fall) Other humanities, natural sciences, etc.
	(Spring) Computer programming 2 (data structures & complexity)
	(Spring) Other humanities, natural sciences, etc.
Sophomore:	(Fall) Other required CS courses (e.g., algorithms, computer organization)
	(Fall) Other humanities, natural sciences, etc.
	(Spring) "*Language and compiler course for mobile code components*"
	(Spring) Other required CS courses (e.g., algorithms, computer organization)
Junior:	(Fall) "*OS and network course for programming with network objects*"
	(Fall) CS electives (database, software engineering, theory, AI, etc.)
	(Spring) "*Distributed virtual machines*"
	(Spring) CS electives (database, software engineering, theory, AI, etc.)
Senior:	Capstone project

The core of this new curriculum is in modifications to existing courses, and repackaging of old and new information into three new courses (most of these topics for the mobile code example are addressed to some degree in [12]):

- **Language and compiler course for mobile code components**. This course uses materials from a traditional undergraduate programming languages course and from a compiler course. The material focuses on mobile code languages, assemblies and components, dynamic object loading, compilation/interpretation environments, using thunks, and the sandbox model.

- **OS and network course for programming with network objects**. This course uses materials from a traditional OS course, a network course, and other information about runtime environments. The material focuses on client-server computing, protocols such as http, authentication, digital certificates, execution engines, marshaling/unmarshaling, RPC/RMI, and network objects.
- **Distributed virtual machines**. This course uses material from an OS course, a computer organization course, a security course, and runtime systems. The material focuses on supporting a hierarchy of address spaces, stack support, thread and object execution models, IPC, and kernel security mechanisms,

The capstone project should be coordinated with the mobile code model by selecting projects that depend on mobile code IT.

In some ways, this new curriculum and courses resemble a conventional academic certificate program, i.e., they augment a core curriculum with specialized knowledge. More extensive IT areas will also reuse parts of the extant curriculum, with appropriate tailoring of low level courses, and creation of new courses at the sophomore level and above. Others' experience with games curricula suggest that the modifications might be substantial when compared to the sample curriculum for mobile code, thus creating a radically different curriculum rather than a certificate-like option for a standard curriculum.

Within the context of a strategic plan and an evolving curriculum, they provide the focus for a specific IT area. The degree of revolution in the curriculum depends on the number of such new courses in any given IT area. For example, a degree in game and entertainment programming would have more than 2 new courses, and require more interdisciplinary courses (presumably at the cost of elective courses in the traditional CS curriculum).

6. Conclusion

Computer science and information technology have gone through radical change, particularly in the last dozen years (with the emergence of personal computing and the internet in the consumer world). While academic computer science programs are usually organized around the 1978 ACM curriculum model, they can become ancient history rather than modern IT. The 2001 and 2005 IEEE/ACM curriculum recommendations provide a model for flexible curricula based on knowledge units and flexible course specification. The recommendation is based on the notion that academic unit have a clear set of goals, a strategy, and a plan for implementing the strategy. The devil is in the details: evolving the extant curriculum into different sub curricula that address contemporary IT areas requires significant investment in course refinement – often involving reorganization of material across 2-4 courses – as well as a wealth of new courses with new materials. Even with this plan for incremental change, it is a daunting barrier to all but the most ambitious of departments. Nevertheless, the long term health of academic computer science depends on constant evolution of the curriculum to address the evolution in IT.

Acknowledgements

Bryan Barnett (Microsoft) and Maité Suarez-Rivas (formerly at Addison Wesley) encouraged me to begin thinking about curricular change in designing a lab manual that focuses on distributed virtual environments. Bryan recognized the importance of teaching new models for programming, and the barriers that the new technology were likely to meet in academia. Maité's perspective was that of an editor. She realized the importance of publishing the work, even though there was no market for the book. It was her idea to sponsor the project, and to make the book available online at no charge.

One result of my experience with the distributed virtual memory lab manual was to make me acutely aware of the difficulty in matching the curriculum with what was going on in commercial software development. This led me to think about other ways that individuals and small groups of faculty might work toward evolution in the curriculum.

References

[1] –, "ACM Curricula Recommendations," ACM, see http://www.acm.org/education/curricula.html.
[2] -, "CISE Pathways to Revitalized Undergraduate Computing Education (CPATH)," National Science Foundation Program Solicitation NSF 06-608, 2006.
[3] -, "Microsoft Certification Overview," Microsoft Corporation, 2006, see http://www.microsoft.com/learning/mcp/default.mspx.
[4] – "More than Fun and Games: New Computer Science Courses Attract Students with Educational Games," Microsoft Press Pass web site, September 12, 2005 http://www.microsoft.com/presspass/features/2005/sep05/09-12CSGames.mspx.
[5] Argent, Lawrence, et al., "Building a Game Development Program," IEEE Computer, (June 2006), pp. 52-60.
[6] Atchison, William F., et al., "Curriculum '68: Recommendations for Academic Programs in Computer Science: A Report of the ACM Curriculum Committee on Computer Science," *Communications of the ACM*, Vol. 11, No. 3 (March 1968), pp 151-197).
[7] Austing, Richard H, Bruce H. Barnes, Della T. Bonnette, Gerald L. Engel, and Gordon Stokes, "Curriculum '78: Recommendations for the Undergraduate Program in Computer Science – A Report of the ACM Curriculum Committee on Computer Science," *Communications of the ACM*, Vol. 22, No. 3 (March 1979), pp 147-166).
[8] Birrell, Andrew, Greg Nelson, Susan Owicki, and Edward Wobber, "Network Objects," *Proceedings of the Fourteenth ACM Symposium on Operating System Principles*, 1994, pages 217-230.
[9] Chang, Carl, et al., (Curriculum 2001 Joint Task Force), "Computing Curricula 2001 Computer Science," IEEE-CS and ACM, December 15, 2001.
[10] Ellis, C. A. and G. J. Nutt, "Office Information Systems and Computer Science," *ACM Computing Surveys*, Vol 12, No. 1 (March, 1980), pp. 27-60.
[11] Chabrow, Eric, "Declining Computer-science Enrollments Should Worry Anyone Interested in the Future of the U.S. IT Industry," *Information Week*, August 16, 2004. See http://www.informationweek.com/story/showArticle.jhtml?articleID=29100069&tid=13692.

[12] Nutt, Gary, *Distributed Virtual Machines: Inside the Rotor CLI*, Addison Wesley, 2005 (this book is available at no charge at http://wps.aw.com/aw_nutt_dvm_1/.)
[13] Shackelford, Russell et al., (The Joint Task Force for Computing Curricula 2005), “Computing Curricula 2005,” ACM, AIS, and IEEE-CS, September 30, 2005. (Available at http://www.acm.org/education/curric_vols/CC2005-March06Final.pdf).
[14] Weiser, Mark, “Some Computer Science Problems in Ubiquitous Computing,” Communications of the ACM, Vol. 36, No. 7 (July 1993), pp. 75-84.