The Role of Programming in IT

Dianne P. Bills
Information Technology Department
Rochester Institute of Technology
Rochester, New York 14623
(585) 475-6791
dpb@it.rit.edu

John A. Biles
Information Technology Department
Rochester Institute of Technology
Rochester, New York 14623
(585) 475-7453
jab@it.rit.edu

ABSTRACT
Early in its history as an academic discipline, depth in computer programming was a primary distinguishing factor between IT and older computing disciplines, such as computer science. Initially, IT was perceived, or misperceived, as being "computing without the programming." However, as IT has begun to mature as a computing discipline, computer programming is emerging as "the" foundation skill for information technologists. However, programming in IT is fundamentally different from programming in computer science or software engineering, and the tasks and requisite skill sets of IT professionals differ from those of other computing professionals.

The IT Department at RIT has changed the weight and delivery of programming in its curriculum several times since its inception in 1992. Today, programming is an essential foundation for other more advanced IT skills in all curricular knowledge areas, and it is a central outcome of the curriculum. This paper discusses the role of programming in IT, the types of skills necessary, how we see the need for this skill changing in the other "pillars" of this academic discipline, and the impact on programming curricula.

Categories and Subject Descriptors
K.3.2 [**Computer and Information Science Education**]: Curriculum

General Terms
Programming, design

Keywords
Computer Programming in IT, Curriculum, Information Technology Education, Programming Fluency, Application Domains.

1. INTRODUCTION
Since the inception of information technology (IT) as an academic discipline, the role of computer programming has been a contentious issue. In some cases, IT has been perceived – or perhaps misperceived – as being computing with less programming [5, 6]. Other programs do not differentiate between the programming needs of information technologists and the programming skills taught in older computing disciplines, such as computer science [7, 8]. However, programming in IT is different [9].

As defined in the current draft of the ACM computing curriculum, Computing Curriculum – Information Technology Volume, IT is a broad computing discipline with knowledge areas encompassing five content sub-disciplines or "pillars":

> The five pillars of an IT curriculum are programming, networking, web systems, information management, and human-computer interaction. We have recommended above that programming be covered in the introductory material; the remaining four pillars (also knowledge areas) should be covered in the intermediate material. It is the feeling of the committee that these four knowledge areas are best studied after students have been introduced to them briefly in the introductory material, and after the students have learned the basics of programming in an appropriate high-level language [1].

While programming is a tool for all computing professionals, programming in IT is fundamentally different from programming in computer science or software engineering because the programming tasks and requisite skill sets for IT professionals differ significantly from those of other computing professionals. IT professionals deal with issues at the interfaces between technologies. In IT, programming is the tool used to "glue" together technologies to create infrastructure solutions [10].

We agree with the ACM IT curriculum that programming is the foundation for the other essential skills in all of the IT curricular knowledge areas [2], and as such it should be a central outcome of any IT curriculum. This position paper discusses the role of programming in IT, how programming in IT is different, the programming skills necessary for IT professionals, and how we see the need for this skill changing in the other pillars of the IT discipline.

2. OUR EXPERIENCE
The Information Technology Department at the Rochester Institute of Technology (RIT) began offering its Bachelor of Science degree in information technology (BS/IT) in 1992. Currently, we enroll approximately 1200 full- and part-time

undergraduate students and admit over 200 new freshmen each year.

Since the beginning, computer programming has consistently been a key component of our IT curriculum. However, our perception of the role of programming and the skills needed by IT professionals has changed as the IT discipline has matured. We currently see programming as "the" fundamental technical skill for information technologists, and we believe that the importance of strong programming skills for information technologists will increase in the future. To understand how we have arrived at this conclusion, some discussion of our experience in teaching programming is necessary. Table 1 shows a chronology of the programming experiences we have offered in our baccalaureate IT program since the start of our degree.

Table 1. Programming Chronology[1]

Calendar Year	Student Year	Fall	Winter	Spring
1992-3	*First*	C++	C++	C++
1993-6	*First*	-	ToolBook[2] HyperCard[3]	-
	Second	C++	C++	C++
1996-7	*First*	-	HyperCard	-
	Second	C++	VB	C++
1997-8	*First*	C++	C++	VB
1998-9	*First*	VB	VB	-
1999-01	*First*	VB	VB	Director[4]
2001-05	*First*	Java	Java	Java
	Second	Director Flash[4]	-	-

When we deployed our first IT curriculum in 1992, we taught C++ in our freshmen programming sequence. We offered one year of C++ programming in three (3) courses: an introductory programming course, an introduction to OOP concepts, and an introduction to GUI interfaces and events. In 1993 we began offering a "pre-programming course" to provide a fundamental understanding of events and event handling, first in ToolBook, then in HyperCard, and finally in Visual Basic. By 1996 we had eliminated event-driven programming in C++ in favor of Visual Basic because freshmen had understandable difficulty with event-driven GUI concepts in C++.

In 1998, we evaluated Java as a possible foundation language for teaching introductory programming but found it too unstable. We felt at that time that the Java programming environment was inappropriate for novice programmers who could not tell the difference between their own mistakes and bugs in the compiler. So we settled on Visual Basic as our foundation language and eliminated C++ from our core curriculum.

Beginning in 1998, then, we offered a two-course Visual Basic (VB) freshman programming sequence for the next three years. Although the VB programming interface is very good for illustrating objects and events, and is a fine "glue" environment, we found that it was a poor pedagogical tool for novice programmers because they had difficulty seeing the "whole" program. Having the code distributed across multiple objects too early in the learning process tended to confuse some students. We found that, on average, retention of programming concepts was not as strong as we felt necessary, and we found that students also had problems transferring their knowledge to later courses that used programming languages without GUI interfaces.

In addition, we became concerned that six months of programming was not a sufficient foundation for the programming skills necessary for our upper-level IT curriculum. Although we added a Director-based authoring and animation course to the core in 1999 and had already added programming assignments to the other courses in the BS/IT core to help students retain their programming skills, we were still not seeing the overall level of proficiency that we wanted.

So we turned our attention back to Java in 2001 and decided that it had stabilized sufficiently for our freshmen. We currently teach our foundation programming sequence as three Java courses: a first course in introductory programming concepts, a second course in OOP concepts with I/O, error handling and the basic GUI interface classes, and a third course on advanced GUI concepts, data structures with threads and sockets, utilities, reusability, and software project management concepts. We teach the sequence in specially-designed classrooms that support active learning. As previously reported [3][4], this sequence has been very successful. Student feedback has been positive; retention through the first year programming sequence has increased; and the faculty is more satisfied with students' demonstrated skills in downstream courses.

We feel that an understanding of software objects and the firing of and response to events within and between software/hardware systems is critical for IT professionals. At its most basic, the ability to program provides the capability of interpreting events and "thinking like the machine" that is so critical for successful problem solving within the computing domain. This ability enables students to see the synergy between the IT knowledge areas that is necessary for successful IT professionals.

3. OUR CURRENT PERSPECTIVE

IT is a diverse computing discipline, with a wide variety of rapidly emerging sub-disciplines. This makes it difficult to identify the common core competencies in programming for all IT professionals. However, we feel that there are identifiable expected outcomes for programming that span the IT spectrum. These common outcomes not only help sharpen the focus of IT programming curricula; they also help define IT as a unified discipline.

IT programming outcomes differ from the expected programming outcomes in Computer Science (CS) and Software Engineering (SE). Some of these differences are fundamental, and some are more subtle, but they help delineate IT curricula from CS/SE curricula. RIT is currently the only institute of higher education to have professionally accredited Bachelors programs in Information Technology, Computer Science, Software Engineering and Computer Engineering (CE). Consequently, we feel we have a unique perspective on the differences between these computing

[1] RIT's academic calendar is based on four three-month quarters per calendar year.

[2] Platte Canyon Multimedia Software Corporation, http://www.plattecanyon.com/

[3] Apple Corporation, www.apple.com

[4] Macromedia Corporation, http://www.macromedia.com/

disciplines. While there are differences in expected outcomes among CS, SE and CE, they are less pronounced than those between any of them and IT. For this paper, then, we'll lump CS, SE and CE together and call the aggregate CS/SE, in deference to the fact that CS, SE and CE students at RIT all take the same five-course programming sequence from the CS and SE departments. IT students take a different programming sequence offered by the IT department, which reflects our beliefs that IT programming skills differ fundamentally from CS, and that IT students are not well served by a "standard" CS programming sequence.

So what is programming in IT, and how does it differ from programming in CS/SE? We'll answer these two questions concurrently, since it is difficult to describe what characterizes IT programming without contrasting it with the well-known benchmark of CS/SE programming.

Probably the biggest distinction is that IT professionals don't build large systems from scratch. An essential outcome in CS/SE is the ability to build large software systems from scratch in a team setting, in other words, classic software engineering. IT professionals, on the other hand, are not software engineers. They may build large systems, but not from scratch.

This distinction arises from one of the fundamental differences between IT and CS/SE/CE as academic disciplines – CS, SE and CE focus on creating new technology, while IT focuses on making effective use of existing technology [10]. In the programming arena this means that CS/SE must be able to build large systems from scratch; that's what it means to "create" new software technology. IT, on the other hand, tends to build systems from existing components. IT systems can be very large, to be sure, but they are built by integrating existing functionality that has been identified as useful to a targeted user community.

Consider, as an example IT application, a Web-based, multi-user game developed by a team of four students for a Web-database integration course offered at RIT in the spring, 2005 quarter. The gaming domain is perfect for illustrating the IT application development process because today's multi-user games rest firmly on all five of the IT pillars and, therefore, are essentially a microcosm of the IT discipline. Game design and development is also one of our most popular IT concentration areas and is the career most frequently asked about among our entering students.

Our example application is a Web-based, multi-user, exploratory game, where users wander through a virtual space of connecting rooms and interact both with objects in the rooms and with other players that they encounter. The interactions with other players take the form of mini-games, with each room supporting a different mini-game. The interactions in this game lead to two different kinds of data communication – asynchronous, for loading room information when a player enters a new room, and synchronous, for interacting with other players and with objects within a room. A database (MySQL) stores everything used in playing the game: room information and methods, character information (including avatars), user account information, images, sounds, graphics, and animations. The client side is implemented in Flash and communicates with the database via a PHP-based middle layer that accommodates interface inconsistencies between Flash and MySQL and filters information for appropriate use by destination processes. In summary, this is a classic three-tier application with real-time interactive multimedia.

There are significant design issues in this application. For example, PHP can pull, but it can't push, so the client process must poll the server periodically in order to reflect changes in the room. Polling too frequently can swamp the server, which degrades the real-time illusion. Polling too infrequently also degrades the experience. Another design issue is the choice of when to use time-based animation (the movie metaphor) and when to use code-driven animation (sprites moving under program control). Each method offers different advantages and disadvantages, and each method raises different integration issues. In short, applications like this, which are typical in the IT world, offer significant levels of complexity and demand careful design in order to successfully integrate disparate components.

It is tempting to say that CS does programming in the large, while IT does programming in the small [11]. It is true that most IT applications are built by individuals or small groups and that these applications are often systems of scripts that glue together existing components and provide a usable interface to the integrated functionality of those components. However, these applications can be quite large and complex, as the previous example illustrates.

The distinction we would make is that CS focuses on designing architectures where the components are mutable, and IT focuses on building architectures that both accommodate and take advantage of existing components. In other words, CS/SE gets to design the components themselves and the interfaces among components, while IT has to work with the interfaces that others develop. The issue here, obviously, is reuse, something that good software engineering is supposed to facilitate. However, most software engineers will gladly build a new component if the existing component doesn't fit the architecture. In IT, the needed skill is to make the component fit, often by building a filter or middle layer to integrate disparate components. The outcome in CS/SE is to design *for* reuse; the outcome in IT is to design *by* reuse.

Some in the CS/SE community see the IT aversion to building things from scratch as a lack of ability to deal with complexity or as just laziness [16]. This criticism is incorrect. IT professionals come at an application from the user's perspective rather than the computer's perspective, and their priority is to identify and meet user needs. This requires a more flexible approach to application development than the traditional waterfall model allows and demands that maximum use be made of existing functionality in order to be productive. To be fair, the increasing popularity of agile computing methodologies [14] in the SE community is a good response to this issue.

4. PROGRAMMING IN THE PILLARS

Historically when a new IT technology has emerged, direct programming has initially tended to play little or no role. However as a technology advances and its functionality is enhanced, it tends to become more powerful and then requires programming. Web pages are a classic example. Early Web pages were built from static HTML scripts. Web pages today are dynamic, with programming functionality on the client side, on the server side, and for connectivity with backend databases.

We assert that as technology continues to advance, programming will become an increasingly important part of the responsibilities of the IT professional, regardless of specialty area. So we looked at where programming exists within our curriculum and talked with colleagues from each of the pillar areas to get their perspectives on how computer programming is and will be used

within their areas. Table 2 shows the occurrence of programming in RIT's current upper-level, or post-core, BS/IT curriculum.

Table 2. Programming Weight in BS/IT Post-Core

BS/IT Pillar	# Courses	Weight
Programming (including games)	7	100.0%
Database	4	50.0%
Networking	19	26.3%
Web Technologies	11	72.7%
HCI	6	33.0%

IT students at RIT complete two three-course concentrations chosen from 12 that are currently available. Some of these concentrations fit neatly into one of the five IT pillars. For example, Network Administration, Wireless Networking, and System Administration all fit into the Networking pillar. Game Design and Development, on the other hand, fits well in both programming and Web technologies, with a heavy dose of HCI and definite needs from networking and database. As we have noted before, it is the synergy of among the pillars that defines IT [13].

The "# Courses" column in Table 2 reflects a somewhat arbitrary assignment of each advanced course to one of the five pillars. For example, our games courses were assigned to the programming pillar. The "Weight" column reflects the percentage of post-core courses in that pillar in which programming is a primary activity. This is usually manifested in one or more projects that require significant programming, often a "final" project. We'll turn now to the five IT pillars and briefly discuss the kinds of programming that are typical in each.

4.1 Programming

It's not surprising that all of the advanced courses in the programming pillar require programming. Our switch from VB to Java as the language used in our introductory courses four years ago has led to a corresponding migration from .NET to Java-based development in our advanced programming courses in the last two years. The advanced IT programming courses focus on advanced application development spanning multiple languages, working with component models and security models, and distributed programming using various APIs. The specific languages used in these courses are only tools, but we've gotten greater traction from Java as the base language.

We grouped the three-course gaming concentration in this pillar because the focus in those courses is on "heavyweight" games developed in C++ and running as standalone applications, the current gaming industry standard. Web-based, "lightweight" games tend to come out of our interactive media group, and line up best with the Web technologies pillar. The term lightweight, however, can be misleading, as our multi-user, Web-based game example described above illustrates.

4.2 Database

Programming is important with databases because the types of interfaces through which we currently access information, client/server and the web, are not expected to change for the foreseeable future. These applications employ connectivity through JDBC or .Net technology and require data manipulation at the transaction level. For these interfaces, writing an SQL statement is not sufficient.

While SQL alone may suffice for simple reporting needs, IT database professionals need to be able to handle data transfer, conversion, and cleansing as well as changes to the design of data systems. This means using programmatic interfaces – typically a combination of traditional and DBMS programming languages as well as scripting languages, such as Perl, which are useful for ad hoc data extraction. The interpreted nature of scripting languages makes them perfect for "quick and dirty" data manipulation tasks. However, in situations where the "right" component for a task is not readily available, database professionals need the ability to code.

Even if a database professional should never needs to write programs, he or she still must be able to interact with programming professionals during software development projects and help formulate the solution to problems. This fact alone necessitates a solid understanding of computer programming principles (private conversation with Prof. Kevin Bierre, Information Technology Department, RIT, 6/21/05).

4.3 Networking

IT networking and system administration professionals need solid programming skills to support their understanding of network protocols at the transport layer. Abstract manipulation skills are important here because they must be able to operationalize algorithms on the TCP/IP stack and manipulate system tables algorithmically.

In the future, networking appliances will incorporate more direct administrator-programming capabilities. Currently network hardware is primarily programmed by the manufacturer, often at the hardware level. However, in the near future, we expect these devices to be directly programmable by the network administrator. User-configurable network processors will be tunable so that networking functions, such as routing, can be adjusted to special purposes and security needs. Plus, we expect networking and system administration functionality to be more integrated in future protocols. The ability to develop customized software solutions will be beneficial because since they are directly modifiable by the end user, sites will be able to deploy their own protocols.

Programming within this pillar is different from computer science programming in its depth. However, network and system administrators must be willing, capable, and unafraid to program (private conversation with Prof. Pete Lutz, Information Technology Department, 6/13/05).

4.4 Web Technologies

In the early days of the WWW, it was sufficient to build static Web pages that displayed content as text and images. Interactivity was limited to following hyperlinks to other pages, which was, and still is, highly useful, but requires no real programming prowess. Today, Web sites, especially those that people actually pay to have built, are dynamic, interactive, media-rich, and highly adaptable. In other words, they actually *do* things, which requires programming.

The game example described above is a typical Web-based application. Web applications are increasingly the preferred deployment choice in many situations because platform, distribution and maintenance issues often can be dealt with more

easily. As is true in the other pillars, scripting largely replaces traditional programming as the primary activity, and the trend is toward more distributed applications and meta-approaches like XML (private conversation with Prof. Chris Egert, Information Technology Department, June 22, 2005).

4.5 Human-Computer Interaction

In the IT HCI pillar, programming once again takes a pivotal role, specifically in rapid prototyping activities that underlie the usability engineering lifecycle [12]. This approach to development focuses on building a useful and usable interface first, using a spiral process of prototyping and usability testing until the user community buys into the prototype interface. This process helps identify and define functionality by giving users a clearer view of what the system will do, and more importantly, what it can do. Once the interface is solid, it's time to identify and define the functionality required in the system. In a sense, this development method is the opposite of the classic waterfall model. Instead of identifying functionality first and building the user interface last, it uses the development of the interface as a tool to identify needed functionality.

5. THE MYTH OF THE COMMON CORE

So what's wrong with the typical CS/SE programming sequence for IT students? One might argue that implementing the classic data structures and algorithms from scratch and building entire large systems from the ground up (or the top down, as the case may be) is a good preparation for any computing professional. If you can build a huge system from scratch, then you surely can build a smaller system from components, or as the song says, "If you can make it there, you can make it anywhere."

There are two questions that need to be answered, however. First, is the standard CS programming sequence *necessary* in order to prepare IT students for the kinds of programming tasks they need to perform? Second, if it is not necessary, is it at least *sufficient* for preparing IT students. Our answer to both questions is "No."

It is necessary for IT students to be able to use stacks, queues, lists, trees, graphs and other data structures appropriately, but it is not necessary that they implement them from scratch. For example, a software engineer working for Oracle needs to worry about implementing external data structures like B-Trees in order to make the Oracle engine efficient, but an Oracle application developer doesn't need to know how the tables they define are mapped to the disk. Similarly, the folks at Macromedia who design and implement the scripting environment in Flash need to know something about designing language syntax and semantics, which they would have picked up writing a compiler in their CS curriculum, but the application developer who uses Flash to build a client agent in a multi-tier system need not know how the parser works. In other words, it is not necessary to know how the underlying technology was built in order to use it effectively, provided the underlying technology was built well.

So if the standard CS programming sequence is not necessary for IT students, is it at least sufficient? After all, many, if not most, computing students begin their college careers not knowing the differences between the various computing disciplines, and IT, as the newest, is often the biggest mystery. Wouldn't it be advantageous to give new students time to decide whether they want to pursue IT, CS, SE, CE or even Information Systems by having all computing students take the same introductory sequence? Even though the standard CS isn't an ideal fit for IT, isn't it close enough?

There are two reasons why the standard CS programming sequence is not adequate for IT students. First, the expected outcomes from the CS sequence are not the outcomes expected of IT students, as we've hopefully made clear. At RIT the first CS programming course and the first IT programming course have similar outcomes, using the standard data types and control structures, and introducing objects. In the second course, the two course sequences begin to diverge. CS presents object-oriented development by implementing classic data structures and algorithms, while IT presents object-oriented development aiming toward GUI development and component integration.

In its third and last programming core course, IT focuses on threads and synchronization, inter-process communication with sockets, and building moderate sized applications using available components. From this point on, programming in IT becomes pillar-specific. In the third CS programming course, students build moderate-sized systems from scratch to prepare for the large systems with external data structures they will build in the fourth CS course and to prepare them for the introduction to software engineering, which is the fifth course. CS, SE and CE students at RIT are expected to complete this five-quarter sequence by the middle of the sophomore year. Thus, the expected outcomes from these two programming sequences are very different.

The second reason why the CS/SE sequence is not sufficient for IT students is that it mandates a build-it-from-scratch mentality among students. Clearly both IT and CS/SE students must be able to write a complete program from scratch. However, the first instinct of the IT professional should be to write a script to integrate existing components rather than to write new components. The nature of user-centered design is that the design must change fundamentally during the development process to meet the shifting needs of users. The spiral development process briefly described in section 4.5 requires *rapid* prototyping; building components from scratch is simply not appropriate unless and until useful functionality is identified that does not already exist.

Finally, there are the cultural differences between IT and CS/SE. Most who have taught programming for any length of time will agree that programming seems to be an unnatural act for most students, if not for most human beings. Retention in programming sequences is notoriously low, with a resulting low retention of students in CS majors. This is a significant problem in meeting the needs of society for competent professionals across all the computing disciplines [15].

For good or bad, the programming sequence in many CS programs is seen as a mechanism for weeding out weak students. Our experience is that many students who struggle in the CS/SE programming sequence do well in the IT programming sequence, not because the IT sequence is less difficult (it isn't), but because it is more focused on the kinds of computing tasks they thought CS would prepare them to do. Indeed, we still receive a significant number of internal transfer students who switch into IT from CS, SE and CE, despite our efforts to help potential students make informed decisions as to which computing major best fits their career goals. Students who did well in the CS programming sequence migrate fairly seamlessly into the IT sequence with no loss of credit. Serious students who struggled in the CS

programming sequence tend to find the IT sequence a better fit and also tend to succeed.

Students switching from IT to CS or SE are rare, but they seem to make the transition successfully, particularly if they switch early in the course sequences and don't have to "retake" CS courses corresponding to the IT programming courses they have already taken.

The bottom line is that programming in IT is fundamentally different from programming in CS/SE, and that IT students are not well served by a CS/SE sequence.

6. CONCLUSION

The ability to handle complex programming tasks is emerging as a defining characteristic of an information technologist regardless of specialty area. For other computing professionals, specifically CS, SE and CE, the focus is the computer – i.e., the computer itself is often the problem. IT professionals, however, are closer to the end user. Therefore, for IT professionals, the focus is on *using* computers to solve problems [13].

This perspective influences the kinds of programming tasks that IT professionals perform, which, in turn, should influence the programming courses IT students take. In short, the IT programming curriculum differs from the standard CS programming curriculum, and those differences become wider as the curricula progress.

Our hope is that this paper can generate discussion, both about the role of programming in IT and the best ways to prepare IT students to fill those roles.

Acknowledgements

Our thanks to our colleagues: Professors Kevin Bierre, Chris Egert and Pete Lutz for their willingness to share their perspectives on programming in their curricular areas, as it relates to IT.

7. REFERENCES

[1] ACM Computing Curriculum – Information Technology Volume, April 2005 Draft, Chapter 7, p. 24; retrieved June 5, 2005, from sigite.acm.org/activities/curriculum/.

[2] IT Body of Knowledge, 3/2005 Draft; retrieved June 5, 2005, from sigite.acm.org/activities/curriculum/.

[3] Hill, L., Bills, D., and Biles, J. A Studio Model Approach to Teaching Introductory Object-Oriented Programming and Problem-Solving Using Java. In *Proceedings of the 3rd Annual Conference for Information Technology Curriculum*, Rochester, NY, Sept. 19-21, 2002.

[4] Whittington, K., Bills, D., and Hill, L. Implementation of Alternative Pacing in an IT Introductory Programming Sequence. In *Proceedings of the 4th Annual Conference on Information Technology Curriculum* (Lafayette, IN, Oct. 16-18, 2003). ACM Press, New York, NY, USA, 2003, 47-53.

[5] Dougherty, J., et al. Information Technology Fluency in Practice. *ITiCSE Conference 2002*, Working Group Report (Aarhus, Denmark, June 28-30, 2002). ACM Press, New York, NY, USA, 2003, 153-171.

[6] Chenoweth, J. Lessons Learned in the Development of an Information Technology Concentration. *Journal of Computing Sciences in College*, Oct. 2001, 17(1), 218-223. Consortium for Computing Sciences in Colleges, USA.

[7] Prasad, C., Li, X. Teaching Introductory Programming to Information Systems and Computing Majors: Is there a Difference? In *Proceedings of the Sixth Australasian Computing Education Conference* (Dunedin, New Zealand, 2004). Australian Computer Society, Darlinghurst, Australia, 2004, Vol. 30, 261-267.

[8] Spooner, D. A Bachelor of Science in Information technology: An Interdisciplinary Approach. In *Proceedings of the 31st Technical Symposium on Computer Science Education* (Austin, Texas, 2002). ACM Press, New York, NY, USA, 2000, 285-289.

[9] Taffe, W. Information Technology: a Degree in Computing. *Journal of Computing Sciences in Colleges*, Feb. 2002, Vol. 17, Issue 3, 183-189. Consortium for Computing Sciences in Colleges, USA.

[10] Ekstrom, J. and Lunt, B. Education at the Seams: Preparing Students to Stitch Systems Together; Curriculum and issues for 4-Year IT Programs. In *Proceedings of the 4th Annual Conference on Information Technology Curriculum* (Lafayette, IN, Oct. 16-18, 2003). ACM Press, New York, NY, USA, 2003, 196-200.

[11] DeRemer, F. and Kron, H. Programming-in-the-large versus programming-in-the-small. In *Proceedings of the International Conference on Reliable Software* (Los Angeles, CA). ACM Press, New York, NY, 1975, 114-121.

[12] Nielsen, Jakob. *Usability Engineering*. Morgan Kaufmann, San Francisco, 1994.

[13] Biles, J. The importance of Synergy: Integrating Curricular Components in IT. In *Proceedings of the 3rd Annual Conference for Information Technology Curriculum*, Rochester, NY, Sept. 19-21, 2002.

[14] Cockburn, A. *Agile Software Development*. Addison-Wesley, Boston, MA, 2002.

[15] Forte, Andrea. Programming for Communication: Overcoming Motivational Barriers to Computation for All. In *Proceedings of the IEEE Symposia on Human-Centered Computing Languages and Environments*, 2003, Auckland, New Zealand, 285-286.

[16] Dougan, Cort. Good Programmers are Not Lazy. Unpublished manuscript retrieved June 30, 2005, from http://hq.fsmlabs.com/~cort/publications/lazy/lazy.pdf