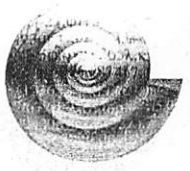
Mark-
I am cleaning my office - thought you could use this; else discard Gail 4-23-14

BY
KEITH DEVLIN,
Guest Editor

WHY UNIVERSITIES REQUIRE COMPUTER SCIENCE STUDENTS TO TAKE MATH

The main benefit of learning and doing mathematics is that it develops the ability to reason about formally defined abstract structures like those in computer science and its applications.



SOME YEARS AGO, I GAVE A LECTURE TO THE COMPUTER SCIENCE DEPARTMENT AT THE UNIVERSITY OF LEEDS IN ENGLAND. KNOWING MY BACKGROUND IN MATHEMATICS—MATHEMATICAL LOGIC, IN PARTICULAR—THE AUDIENCE EXPECTED IT WOULD BE FAIRLY MATHEMATICAL, AND ON THAT OCCASION THEY WERE CORRECT. AS I GLANCED AT THE ANNOUNCEMENT OF MY TALK POSTED OUTSIDE THE LECTURE ROOM, I NOTICED THAT SOMEONE HAD ADDED SOME RATHER TELLING GRAFFITI. OVER THE FAMILIAR HEADER "ABSTRACT" ABOVE THE DESCRIPTION OF MY TALK, THIS PERSON HAD SCRAWLED THE WORD "VERY."

ILLUSTRATION BY JEAN-FRANÇOIS PODEVIN

A cute addition. But it struck me then, and does still, that it spoke volumes about the way many computer science students view the subject. To the graffiti writer, operating systems, computer programs, and databases were (I assume) not abstract but real. Mathematical objects, in contrast, the graffiti-writer likely believed—and I have talked to many students who feel this way—are truly abstract, and reasoning about them is an abstract intellectual pursuit. Which goes to show just how good we humans are (perhaps also how effective university professors are) at convincing ourselves (and our students) that certain abstractions are somehow real.

The truth is, of course, that computer science is entirely about abstractions. The familiar sleek metal boxes don't, in and of themselves, compute. As electrical devices, if they can be said to do anything, it's physics. It is only by virtue of the way we design their electrical circuits that, when the current flows, obeying the laws of physics, we human observers pretend they are performing reasoning (following the laws of logic), numerical calculations (following the laws of arithmetic), or searches for information. True, it's a highly effective pretense. But just because it's useful does not make it any less a pretense.

Once you realize that computing is all about constructing, manipulating, and reasoning about abstractions, it becomes clear that an important prerequisite for writing (good) computer programs is the ability to handle abstractions in a precise manner. As it happens, that is something we humans have been doing successfully for more than three thousand years. We call it mathematics.

This suggests that learning and doing mathematics could play an important role in educating future computer professionals. But if so, then what mathematics? From an educational point of view, in order to develop the ability to reason about formal abstractions, it turns out to be largely irrelevant exactly which abstractions are used. Our minds, which evolved over many tens of thousands of years to reason (largely imprecisely) about the physical world, and more recently the social one, find it extremely difficult accepting formal abstractions. But once we have learned how to reason precisely about one set of abstractions, it takes relatively little extra effort to reason about any others.

But surely, you might say, even if I'm right, when it comes to training computer scientists, it makes sense to design educational courses around the abstractions they will actually use after graduating and going to work for IBM, Microsoft, or whoever. Maybe so (in fact no, but I'll leave that argument to another time). But who can say what the dominant programming paradigms and languages will be four years into the future? Computing is a rapidly shifting sand. Mathematics, in contrast, has a long history and is stable and well tested.

Sure, there is a good argument to be made for computer science students studying discrete mathematics rather than calculus. While agreeing with this viewpoint, however, I personally find it is often overplayed. Here's why.

> ONCE WE HAVE LEARNED HOW TO REASON PRECISELY ABOUT ONE SET OF ABSTRACTIONS, IT TAKES RELATIVELY LITTLE EXTRA EFFORT TO REASON ABOUT ANY OTHERS.

A common view of education is that its main aim is the acquisition of knowledge through the learning of facts. After all, for the most part that is how we measure the effectiveness of education, testing students' knowledge. But it's simply not right. It might be the aim of certain courses, but it's definitely not the purpose of education. The goal of education is to improve minds, enabling them to acquire abilities and skills to do things they could not do previously. As William Butler Yeats put it, "Education is not about filling a bucket; it's lighting a fire." Books and CDs store many more facts than people do—they are excellent buckets—but that doesn't make them smart. Being smart is about doing, not knowing.

A number of studies by education researchers have shown that if you test university students just a few months after they have completed a course, they will have forgotten most of the facts they had learned, even if they passed the final exam with flying colors. That doesn't mean the course wasn't a success. The human brain adapts to intellectual challenges by forging and strengthening neural pathways, and these pathways remain long after the "facts" used to develop them have faded away. The facts fade, but the abilities remain.

If you want to prepare people to design, build, and reason about formal abstractions, including computer software, the best approach is to look for the most challenging mental exercises that force the brain to

master abstract entities—entities that are purely abstract—and cause the brain the maximum difficulty to handle. Where do you find this excellent mental training ground? In mathematics.

Software engineers may well never apply any of the specific theorems or techniques they were forced to learn as students (though some surely will, given the way mathematics connects into most walks of life in one way or another). But that doesn't mean those math courses were not important. On the contrary. The main benefit of learning and doing mathematics is not the specific content; rather it's the fact that it develops the ability to reason precisely and analytically about formally defined abstract structures.

In this special section, six professors of computer science give their own particular slants on the reasons mathematics is an important component of a computer science education. Together, their articles make a strong case. Yes, we all know of individuals with no mathematics education beyond high school who have developed highly successful computer programs. That success does not imply mathematics is not important for computer science or to one's ability to write innovative or bug-free code. A more plausible inference is that with a more substantial mathematical background, these successful individuals might have been even more successful.

Kim Bruce et al. argue that knowledge of, and proficiency in, discrete mathematics, in particular, is essential for practicing computer professionals. Not so much because they are likely to have to apply any particular theorem or method—though the authors do provide some specific examples. Rather, because, they say, "One of the most important goals for a college or university education is to provide the foundations for further learning." Specific techniques, either in mathematics or in any other discipline, they say, can always be learned—and are arguably best taught—through on-the-job training as needed. University education, on the other hand, should aim to provide a sound base preparing the way for subsequent acquisition of specific skills. Or, as the authors themselves put it, "Traditional university education provides just-in-case learning rather than the just-in-time learning provided by on-the-job training." They conclude by saying, "We know that mathematical thinking will be of use; we just don't know exactly when or what form it will take."

Peter Henderson sets his sights on software engineering, which he defines as "an emerging discipline that applies mathematical and computer science principles to the development and maintenance of software systems." To Henderson, writing software is analogous to the more established physical engineering disciplines, including chemical, civil, electrical, and mechanical. This makes the importance of mathematics—or at least mathematical thinking—self-evident. As he observes, "All engineering disciplines require developing and analyzing models of the desired artifact... Abstract modeling and analysis is mathematical." As for Bruce and his co-authors, Henderson still must answer: "What math?" Like them, Henderson plumps for discrete mathematics, especially logic.

Admittedly, the importance of discrete mathematics already makes software engineering quite unlike the other engineering disciplines, with their heavy dependency on calculus-based, continuous mathematics. But is software development an engineering discipline? This question leads Henderson to start off by reformulating the original question. The bulk of his article is devoted to establishing an affirmative answer—or at least making the case that the answer should be affirmative. He has played an active role in the development and subsequent updating of the ACM/IEEE *Computing Curricula 2001* and, not surprisingly, draws on that background in making his case.

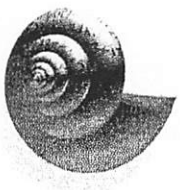
Finally, Vicki Almstrum shines a very different light on the issue. Drawing on a survey she administered to 500 computer professionals, she tries to tease out what motivates individuals to study computer science in the first place. Is it the gadgetry of computing and a desire to make things—put crudely, to write code that does stuff—or is it more an intellectual activity, akin to mathematics, even a branch of mathematics? For how many of us is the driving motivation to see our code work? For how many is the goal a desire to understand what's going on? Do these differences lead to distinctions between those who view mathematics as not important to computer science and those who do?

While almost 80% of Almstrum's survey participants were women, research by others has shown there is a gender difference in what attracts people to enter or remain in the computer field, with men tending to be attracted more by "building and doing" and women more by "understanding." Nevertheless, Almstrum's survey serves to raise awareness of the breadth and complexity of why individuals become computer professionals. Perhaps of greatest relevance to the focus of this special section, only 2% of those responding to the survey felt that a good understanding of mathematics was not helpful in computer science. ■

KEITH DEVLIN (devlin@csli.stanford.edu) is Executive Director of the Center for the Study of Language and Information and a founding member of the Media X program, both at Stanford University, Stanford, CA.

BY
KIM B. BRUCE,
ROBERT L. SCOT DRYSDALE,
CHARLES KELEMEN,
AND
ALLEN TUCKER

WHY MATH?



MATH REQUIREMENTS! THESE WORDS ARE ENOUGH TO SEND CHILLS DOWN THE SPINES OF A GOOD SHARE OF NEW COMPUTER SCIENCE MAJORS EVERY YEAR. EVIDENCE THAT EVEN SOME PRACTITIONERS AND EDUCATORS QUESTION THE VALUE OF MATHEMATICS FOR COMPUTER SCIENCE IS DISCUSSED IN [2]. THEY MIGHT CLAIM MATHEMATICS IS USED SIMPLY AS A FILTER—WEEDING OUT STUDENTS TOO WEAK OR UNPREPARED TO SURVIVE—OR JUST TO PARE DOWN THE HORDES OF POTENTIAL COMPUTER SCIENCE MAJORS TO A MORE MANAGEABLE SIZE. OTHERS might argue it is just another sign that faculty in their ivory towers have no clue what practitioners really do or need. Each of these views surely has its adherents, but we argue here that learning the right kind of mathematics is essential to the understanding and practice of computer science.

The mathematical thinking, as well as the mathematics, in a computer science education prepares students for all stages of system development, from design to the correctness of the final implementation.

What is the right kind of mathematics for preparing students for real-world responsibilities? In computer science, discrete mathematics is the core need. For applications of computer science, the appropriate mathematics is whatever is needed to model the application discipline. Software (and hardware) solutions to most problems, including those in banking, e-commerce, and airline reservations, involve constructing a (mathematical) model of the real (physical) domain and implementing it. Mathematics can be helpful in all stages of development, including design, specification, coding, and verifying the security and correctness of the final implementation. In many cases, specific topics in mathematics are not as important as having a high level of mathematical sophistication. Just as athletes might cross-train by running and lifting weights, computer science

ILLUSTRATION BY JEAN-FRANÇOIS PODEVIN

students improve their ability to abstract away from details and be more creative in their approaches to problems through exposure to challenging math and mathematically oriented computer science courses.

Discrete mathematics includes the following six topics, or discrete structures, the core in the ACM/IEEE computer science report *Computing Curricula 2001: Computer Science* [5]: Functions, relations, and sets; Basic logic; Proof techniques (including mathematical induction and proof by contradiction); Basics of counting; Graphs and trees; and Discrete probability.

We begin our exploration of the need for discrete mathematics in computer science with a simple problem whose solution involves its use. Vectors are supported in standard libraries of C++ and Java. From the programmer's point of view a vector looks like an extensible array. That is, while a vector is created with a given initial size, if something is added at an index beyond its extent, the vector automatically grows to be large enough to hold a value at that index.

A vector can be implemented in many ways (such as a linked list), but the most common implementation uses an array to hold the values. In such an implementation, if an element is inserted beyond its extent, the data structure creates a new array large enough to include the index, copies the elements from the old array to the new array, then adds the new element at the proper index. This vector implementation is straightforward, but how much should the array be extended each time it runs out of space?

Keeping things simple, suppose the array is being filled in increasing order, so each time it runs out of space, it needs to be extended by only one cell. There are two strategies for increasing the size of the array: always increase its size by the same fixed amount, F, and always increase its size by a fixed percentage, P%. A simple analysis using discrete mathematics (really just arithmetic and geometric series) shows that in a situation in which there are many additions, the first strategy results in a situation where the average cost of each addition is $O(n)$, where n is the number of additions (that is, the total of n additions costs some constant multiplied by n^2); the average cost for each addition with the second strategy results in a constant (that is, the total of n additions costs a constant multiplied by n).[1]

This simple but important example analyzes two different implementations of a common data structure. But we wouldn't know how to compare their quite significant differences in cost without being able to perform a mathematical analysis of the algorithms involved in the implementations.

Here, we aim to sketch out some other places where mathematics or the kind of thinking fostered by the study of mathematics is valuable in computing. Some of the applications involve computations, but more of them rely on the notion of formal specification and mathematical reasoning.

Determining Efficient Algorithms

Mathematics is central to designing and analyzing algorithms. We could discuss how to solve recurrence relationships, average-case analyses, and many other things everyone would agree are highly mathematical. But the argument could be made that only a handful of specialists need to do such things; everybody else can just look up the algorithms others have developed.

Still, evaluating and selecting algorithms is not simple. Consider a simple consulting job: Suppose the independent cab and limo operators in Salt Lake City had decided to contract with a consultant to write a program to help each of them schedule all the customers who wanted to ride with them during the 2002 Winter Olympics. Their first request might have been for the consultant to write a program into which customers could enter requests of the form: "I want a cab and driver from such and such a start date and time to such and such a finish date and time." As drivers are paid a flat rate per ride, the program would provide a driver the largest possible subset of requests that did not overlap in time.

Later, the drivers might have realized that instead of charging a fixed rate they could have customers bid for how much they were willing to pay for the requested period; the opening ceremonies and figure skating were, for example, more popular than the biathlon. The second version of the program scheduled the set of non-overlapping requests to maximize the amount of money the driver using the program would earn.

However, some customers might have wanted the same driver the whole time they were at the games. To accommodate them, a third version of the program could have been developed to take a set of time-period requests, along with a single bid for the whole set. A driver would have had to agree to drive for all requested intervals or refuse the request. The program would pick the sets of requests that maximized the amount of money the driver would receive without overlapping in time.

At first glance, it seems like the main difference between the three program versions would have been in the user interface. But that was not the case. The version with the flat-rate pricing can be solved by a simple greedy algorithm in $O(n \log n)$ time: sort the requests by finish time and at each step schedule the first request that does not overlap the last job scheduled. However,

[1] The constants depend on the values of F and P. A very simple analysis is possible when the algorithm starts with an empty array and $F = 1$ (add one new element when the array runs out of space) and $P = 100\%$ (double the size of the array when it runs out of space).

this greedy algorithm would not solve the variable-rate version, but a particular $O(n \log n)$ dynamic programming algorithm would solve it.

The third problem—handling sets of requests—is NP-hard. For practical purposes, this means the consultant wouldn't have found a substantially better solution than trying all the 2^n possible subsets of requests and so should have tried to find a good but not optimal solution rather than promise to find the best solution.

How would the consultant have known that a simple greedy algorithm solves the first problem (but not the second) and that a dynamic programming algorithm solves the second problem? The consultant would have had to prove it. How would the consultant know that the third problem is NP-hard? The consultant would have had to prove it by reducing a known NP-hard problem—Set Packing—to this problem. There would have been no way to do a professional job on this consulting assignment without doing these proofs. (See [3] for more on algorithms.)

We could offer many more examples where similar problems must be solved or where some are easy and others intractable. Mathematical proofs are the only way to distinguish among the alternatives.

Formal Specifications in the Real World

The term "formal methods" in hardware and software design means that precise mathematical specifications are used to define a product and that the product's implementation (code) is verified using mathematical proof techniques. The extent to which formal methods are used to design a particular product depends on many factors, including the cost of development, efficiency of the resulting code, skills of the developers, and safety-critical nature of the application.

There has been a great deal of practitioner interest of late in formal specification and verification of hardware, as well as of software. The potential cost of a mistake in the design of, say, a chip can be enormous, thus it can be financially beneficial to commit the resources to verifying a hardware design. Also, when designing a protocol that could be widely used, it is crucial to verify it has the required performance and security properties.

Most software engineers tend to think of these formal proofs of correctness when they hear the words formal methods, but we consider formal methods more broadly to encompass a variety of situations where there are benefits to using a specification and mathematical tools by computer scientists.

XML, recursion, and mathematical induction. The syntax of a programming language is formally specified via context-free grammar or syntax diagrams. This specification makes it clear to both compiler writers and programmers what is legal syntax.

A promising development with the same flavor as the formal specification of programming language syntax is the introduction of XML as a structured way to transmit information between programs and systems [1]. Data is presented using tags similar to those in HTML, but the tags indicate the semantic structure of the data, rather than its layout in a browser. Data type definitions (DTDs) provide a formal specification of the constraints on the structure of data similar to the way a static type system indicates constraints on legal programs in a particular programming language.

XML data can be parsed like programming languages, resulting in structures like parse trees. The data itself can be verified against DTDs using techniques similar to the ones used in type checkers on programming languages. However, rather than being restricted to the inflexible structure of a fixed programming language, groups sharing data with similar meanings can agree on different sets of tags and DTDs for representing different kinds of data.

If sender and receiver agree on the DTD for data, the sender can generate XML-formatted data, while the receiver can parse, verify, and transform it into a format easier for the receiver to use. All this processing can use technology originally developed for compiling programming languages. The technology has been one of the great triumphs of theoretical computer science, providing provable algorithmic connections between the formal description of languages and programs for processing the languages.

However, even if programmers ignore this technology and simply process the data directly using the equivalent of recursive descent compilers, the mathematical understanding of XML as formally specified data provides tools for working with XML. The DTD provides a specification of the structure of data similar to that of a regular expression. Simple algorithms based on finite automata derived directly from such specifications can verify that incoming data satisfy the specifications, while other data-directed algorithms parse and transform the data into other formats.

XML documents can be understood in their parsed form as trees. Recursive algorithms for working with trees are significantly easier to understand than equivalent iterative algorithms using a stack. (Most programmers find it a real challenge to write an iterative algorithm to do an in-order traversal of a tree.) While many programmers have tried to avoid recursive algorithms—some because they didn't understand them, others because they felt they were too inefficient—processing recursively specified or tree-structured data is much easier with recursion.

How might programmers best understand recursion and ensure their recursive programs satisfy the given

specification? The answer is mathematical induction—one of many reasons that proof by induction is such an important topic in courses on discrete mathematics. Programmers with a good understanding of mathematical induction find it easier to write and, more important, provide convincing arguments for the correctness of recursive algorithms.

We were careful to say "provide convincing arguments" rather than "prove" in the preceding paragraph. While there are circumstances where a careful formal proof of correctness is called for, most of the time it is sufficient to provide an informal argument for the correctness of an algorithm.

If programmers are able to write the specifications of the parts of an algorithm, it is generally relatively easy for them to also provide an informal argument of correctness by asking, and answering: Does the base case satisfy the specification? and Do complex cases eventually get down to a base case? If the programmer presumes that all embedded recursive calls do the right thing, does this case satisfy the specification? Moreover, rather than just using such a process to verify an existing program, the process can also be used to develop and verify a program at the same time.

Secure and safety-critical systems. Recent virus and other security attacks highlight the importance of and often critical need for secure and safety-critical systems. While most computer scientists do not write secure or safety-critical systems, they must still understand the existing and potential threats to their systems. Interesting work has been done on ways to verify that downloaded software from untrusted sources will not behave in ways that put a system at risk. Downloaded applets in Java (at least with the proper security policy included in the browser) are guaranteed to run in a "sandbox," which excludes reading from or writing to the local file system.

Other interesting research has focused on "proof-carrying code" [4]; programmers provide a machine-assisted proof that the program satisfies a given security policy (such as it won't write to memory outside a fixed set of locations or won't write to files). This proof is typically much easier to develop than a proof of correctness of the program. The proof may be downloaded with the code and checked (automatically) against the downloaded code to ensure it is correct and the downloaded code is secure.

While developing and sending a proof might be deemed too expensive for code intended to run only one time (for which restricting execution to a sandbox may be sufficient), it can provide great assurance against accidentally downloading viruses or other damaging code as part of major programs that will be used repeatedly on a system. Other techniques are also being developed, including compiling to assembly language with proof annotations, using mathematical proof techniques for the same purpose.

Conclusion

These arguments and examples give a sense of why mathematics and mathematical thinking are important in computer science. We could have cited many more, including the remarkable success of relational databases and model checkers for verifying hardware. The examples we selected are interesting in their own right and different enough from the ones usually cited to suggest that the tools and reasoning taught in mathematics courses, especially those covering discrete mathematics, are of great value later in practice.

A computer science education is not intended to teach what students need to know for their first job. Nor is it to teach what they will need to know for all the jobs they will ever have. On-the-job learning, reading, and short- and semester-long courses (whether online or in person) provide much of what is needed over the course of one's career.

One of the most important goals for a university education is to provide the foundations for further learning. We have heard it described this way: A traditional university education provides just-in-case learning rather than the just-in-time learning provided by on-the-job training. We know that mathematical thinking will be of use; we just can't always predict exactly when or what form it will take. ■

References

1. Bosak, J. and Bray, T. XML and the second-generation Web. *Sci. Am.* (May 1999).
2. Bruce, K., Kelemen, C., and Tucker, A. Our curriculum has become math-phobic! *SIGCSE Bulletin 33 (2001)*, 243–247.
3. Cormen, T., Leiserson, C., Rivest, R., and Stein, C. *Introduction to Algorithms, 2nd Ed.* MIT Press/McGraw-Hill, New York, 2001.
4. Necula, G. Proof-carrying code. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (1997), 106–119.
5. The Joint Task Force on Computing Curricula. Computing Curricula 2001. *J. Educat. Res. Comput. 1*, 3 (2001).

KIM B. BRUCE (kim@cs.williams.edu) is the Frederick Latimer Wells Professor of Computer Science in the Department of Computer Science at Williams College, Williamstown, MA

ROBERT L. SCOT DRYSDALE (scot@cs.dartmouth.edu) is a professor in and chair of the Department of Computer Science at Dartmouth College, Hanover, NH.

CHARLES KELEMEN (cfk@cs.swarthmore.edu) is the Edward Hicks Magill Professor of Computer Science and chair of the Computer Science Department at Swarthmore College, Swarthmore, PA.

ALLEN TUCKER (allen@bowdoin.edu) is the Anne T. and Robert M. Bass Professor and chair of the Department of Computer Science at Bowdoin College, Brunswick, ME.

Bruce's research was supported in part by National Science Foundation grant CCR-9988210.

BY
PETER B. HENDERSON

MATHEMATICAL REASONING IN SOFTWARE ENGINEERING EDUCATION

THE ENGINEERING PROFESSION IS A BRIDGE BETWEEN SCIENCE AND MATHEMATICS AND THE TECHNOLOGICAL NEEDS OF ALL PEOPLE. ALL ENGINEERING DISCIPLINES ARE FUNDAMENTALLY BASED ON MATHEMATICS AND PROBLEM SOLVING. TRADITIONAL ENGINEERING DISCIPLINES, INCLUDING CHEMICAL, CIVIL, ELECTRICAL AND MECHANICAL, RELY ON CONTINUOUS RATHER THAN DISCRETE MATHEMATICAL FOUNDATIONS. SOFTWARE ENGINEERING IS AN EMERGING DISCIPLINE THAT APPLIES MATHEMATICAL AND COMPUTER SCIENCE PRINCIPLES TO THE DEVELOPMENT AND MAINTENANCE OF SOFTWARE SYSTEMS, RELYING PRIMARILY ON THE PRINCIPLES OF DISCRETE MATHEMATICS, ESPECIALLY LOGIC.

Discrete mathematics, especially logic, plays an implicit role in software engineering similar to the role of continuous mathematics in traditional physically based engineering disciplines.

What role does mathematics play in software engineering? Consider the following two statements: Software practitioners do not use mathematics; and Software practitioners need to think logically and precisely. They represent an apparent contradiction in light of the similarity of the reasoning underlying software engineering and mathematics [3]. Perhaps software practitioners who say, I don't use mathematics, really mean, I don't use mathematics explicitly or formally. Many practicing engineers don't explicitly use calculus on a daily basis but do implicitly use mathematical reasoning all the time. Similarly, software engineers

ILLUSTRATION BY JEAN-FRANÇOIS PODEVIN

should learn to use foundational discrete mathematics concepts and logical reasoning at all times.

Ask traditional engineers if calculus should be eliminated from undergraduate engineering curricula; the answer would be no. In contrast, practicing software engineers have argued that mathematics is not that important in software engineering education since practitioners don't use it explicitly [4]. Was it continuous or discrete mathematics (or both) that software engineers considered less important? The answer was unclear. The role of discrete mathematics and logic in software engineering today is not well understood by either academic researchers or industrial practitioners. This lack of understanding will change as the discipline matures and academics and practitioners work together to develop that role, making it similar to the role of continuous mathematics in traditional engineering disciplines.

WHICH HUMAN ENDEAVOR WAS DEVELOPED TO DEAL WITH ABSTRACTION?

MATHEMATICS.

Physical vs. Software Engineering

Recent articles by software engineering educators have described the similarities and differences between traditional engineering disciplines and software engineering; for example, see [9]. One major difference is that traditional engineers construct real, physical artifacts, while software engineers construct non-real, abstract artifacts. The foundations of traditional engineering disciplines are mature physical sciences and continuous mathematics, whereas those of software engineering are less mature abstract computer science and discrete mathematics. In physical engineering, two main concerns for designing any product are cost of production and reliability measured by time to failure. In software engineering, two main concerns are cost of development and reliability measured by number of errors per thousand lines of source code. Both traditional and software engineering disciplines require maintenance but in different ways.

All engineering disciplines involve developing and analyzing models of the desired artifact. However, the methods, tools, and degree of precision differ between traditional and software engineering. Abstract modeling and analysis are mathematical in nature. They are very mature in traditional engineering and maturing slowly in software engineering. An example discussed in the following paragraphs demonstrates how mathematical reasoning is used in both traditional and software engineering.

In electrical engineering, the voltage decay as a function of time t of a resistor-capacitor (RC) circuit is specified by the function $V(t) = Vo\ e^{(-t/RC)}$, where R is the resistance, C the capacitance, and Vo the initial capacitor voltage. This model of the behavior of RC circuits is derived from principles of mathematical circuit design using foundational calculus and differential equations. Electrical engineering students learn this derivation in their electronic circuits course after studying calculus and differential equations. Here, mathematics-based reasoning is used to derive and understand a fundamental concept.

Iteration invariants represent a foundational concept few computer science or software engineering graduates understand, appreciate, or use effectively, even though they are important for deriving, understanding, debugging, and documenting algorithms. Every iteration has a predicate I(S) (S represents the

current state of the computation) called the iteration/loop invariant, that captures the underlying meaning of the iteration (such as sum the values in a list, search a tree structure, and compute the tax due by all taxpayers). Mathematical logic can be used to argue that the predicate I(S) satisfies logical constraints, as in Figure 1 for the `while-do` iteration; here, the red stuff in brackets represents logical assertions, and C(S) is a side-effect free Boolean condition. Upon termination, another mathematical issue, { I(S) and not C(S) are true }, must logically imply (=>) the desired post-condition. The use of this approach for linear search is discussed later.

Software engineering students can learn to use mathematical reasoning to model, derive, understand, debug, and document software systems. With enough practice, the underlying mathematical concepts become intrinsic to their thought processes, supporting rather than hindering their thinking.

```
{pre-condition}
Initialization code
{I(S) is true}
while C(S) is true do

    {I(S) and C(S) are true}
      <code for body of iteration>
    {I(S) is true}

{I(S) and not C(S) are true ⟹ post-condition}
```

Figure 1. Logic of a while-do iteration.

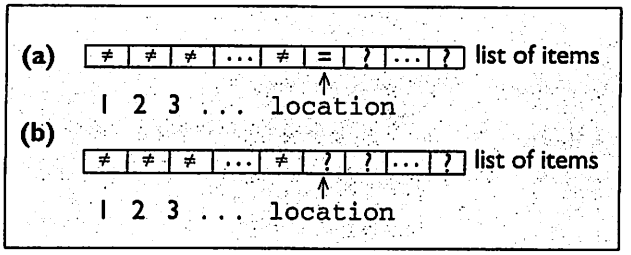


Figure 2. (a) Post condition and (b) potential iteration invariant.

Mathematics and Software Engineering

Key reasons for wanting to learn and use mathematical reasoning include:

Abstract software. Constructing non-real (abstract) artifacts requires abstract reasoning. Which human endeavor was developed to deal with abstraction? Mathematics. Hence one view of a software system is as a mathematically precise model of some desired process or computation. Mathematics is one tool for reasoning about software systems, as well as for practitioners' rigorous reasoning and analysis.

Notations, symbols, abstractions, precision. The expression `y = ax + b` is familiar from algebra, and `count == 0` is familiar from programming. Each uses notations and symbols and is precise, given the types of data and semantics of the operations, specified mathematically. Learning a formal notation is no more difficult than learning a programming language. Indeed, it is often easier, as the syntax and semantics are cleaner. Programming appeals to our innately process/imperative-oriented minds, and programming tools breathe life into programs. Mathematics tends to be declarative and static, though such tools as Axiom, Mathematica, and Maple help mitigate this perception.

Modeling software systems. A model, even a mental one, must be created before construction of any artifact can begin. Most software development is like creating art whereby an initial vision slowly takes form. Planned evolution, along with maintenance issues, is often ignored. This process may be acceptable for some software projects, but with many such projects the more the software engineer can learn and understand early, the better. Modeling is one vehicle for achieving this understanding, and mathematics is an important tool for building, checking, analyzing, and experimenting with models [2]. Moreover, developing precise models using specification languages, including (in order of popularity) Z, Larch, and Alloy, is important for identifying specification errors, which are very costly to correct once a software system is implemented [6].

Application domains. Software practitioners can use mathematics to communicate with their colleagues, including engineers, scientists, mathematicians, statisticians, actuaries, and economists. Mathematics is a rich, comprehensive, universal language for communication between such diverse groups.

Mathematical reasoning. One definition of mathematical reasoning, attributed to an informal working group of computer science and math educators (www.math-in-cs.org) [5] is: "Applying mathematical techniques, concepts, and processes, either explicitly or implicitly, in the solution of problems; in other words, mathematical modes of thought that help us solve problems in any domain." In the most general interpretation, every problem-solving activity is an application of mathematical reasoning. For example, consider the benefits of exercises requiring students to translate English statements into prepositional or predicate logic form; these "modeling" exercises help them be more precise and inquisitive about the interpretation of English statements. When clients or colleagues say, "A or B," do they mean "inclusive or" or

"exclusive or"? When they say, "for all ...," do they mean universal quantification? What is the intended meaning of "for all ..." when there are no elements over which to quantify?

Mathematics in Software Engineering

The article "Why Math?" by Bruce et al. in this section describes several example applications of mathematics in computer science. The following simple linear search problem illustrates the use of logical reasoning to derive an algorithm. The problem statement is: Find the location of the first instance of a specified item in a list of items; the specified item is known to be in the list. Develop an algorithm for this problem. Think about the following questions (hopefully the ones students would be asking themselves): How many items are in the list?; Can the list be empty?; What happens if the specified item appears more than once?; and What is meant by "first instance" and by "location"? Addressing these questions and using the given problem information enables software engineers to formulate representative pre and post conditions required to ensure the problem is well defined.

> IN THE MOST GENERAL INTERPRETATION, EVERY PROBLEM-SOLVING ACTIVITY IS AN APPLICATION OF MATHEMATICAL REASONING.

Most undergraduate students would love to see such a problem on a comprehensive exit exam.[1] Applying a familiar pattern, or template, leads to a few lines of code (simple). When developing computer programs, do students simply pattern-match to get an approximate program, then use extensive testing to refine it? Or do they really understand the underlying logic?

To understand the relevance of this issue, consider a variant of the problem as it was presented to professional programmers in a tutorial session [1]. They had to compose a program for binary search, a search strategy that iteratively divides the list in half. Approximately 90% of them got it wrong, unable to identify proper pre and post conditions, apply a familiar pattern, compose a correct algorithm, or express it in a programming language.

Returning to the linear search problem, consider a solution strategy based on the foundational concepts of logic and iteration invariants. First, the algorithm developer identifies the pre and post conditions to clearly specify the problem. They can be presented formally using predicate logic; for this discussion, I offer a picture of the post condition (see Figure 2a). That is, the desired item is not found (≠) in the list before `location` and the desired item is found (=) at `location` of the list. As the algorithm iterates, the not found (≠), knowledge accumulates by advancing the value of `location` from 1, 2, 3, ...; the iteration terminates once the item is found (=). These factors lead to a potential iteration invariant (see Figure 2b) and a potential iteration termination condition, that is, iterate as long as the desired item ≠ item in `location` of the list.

Figure 3 outlines a partially complete linear search algorithm with logical assertions and iteration invariants; the color blue denotes the item referenced by the variable `location`. Note that the iteration invari-

[1] Few colleges and universities give such exams.

ant is true prior to entering the iteration when location = 1, that is, it is vacuously true.[2] Indeed, this is often the case for iterations, another reason students must understand the logical concept of vacuous. (You may complete the algorithm using the red stuff to derive the `body of the iteration`.)

One may argue that the red stuff is useless clutter.

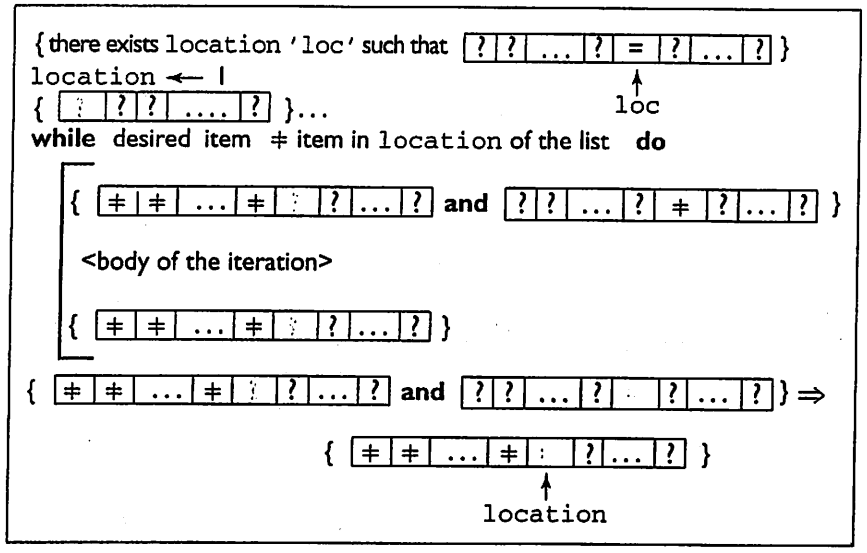


Figure 3. Linear search algorithm.

Perhaps, but it captures the natural reasoning our minds use. It is explicit here. This is the type of reasoning—from first principles—software engineering students should be able to perform. However, this does not mean they must use it all the time but be ready to apply it as needed, when, for example, a derived software system must be correct or when understanding, debugging, or documenting software systems.

What is an iteration invariant for binary search? Understood intuitively, it is that the location of the desired item is constrained between two other locations—low and high—and the algorithm adjusts to converge on the potential location of the desired item; the desired item may not even be on the list. This "intuitive" invariant is useful to software engineers for deriving a correct algorithm. What percentage of the programmers described in [1] would have solved the problem correctly if they had reasoned more mathematically?

Which Mathematics for Software Engineers?

Educational foundations are being identified as computer science and software engineering mature. For example, the ACM/IEEE *Computing Curricula 2001: Computer Science* [10], a guideline for undergraduate computer science programs required discrete mathematics in its core, recommending it be taken early in the undergraduate curriculum. Meanwhile the ACM/IEEE *Computing Curricula 2001: Software Engineering* [7] for undergraduate software engineering programs adopted and extended this foundational model. Included in its two-course mathematical foundations core (E = essential, D = desirable, and O = optional) are the following topics:

Functions, Relations, and Sets (E);
Basic Logic (prepositional and predicate) (E);
Proof Techniques (direct, contradiction, inductive) (E);
Basic Counting (E);
Graphs and Trees (E);
Discrete Probability (E);
Finite State Machines, regular expressions (E);
Grammars (E);
Algorithm Analysis (E);
Number Theory (D); and
Algebraic Structures (O).

Foundational core material included:

Abstraction:
- Generalization;
- Levels of abstraction and viewpoints;
- Data types, class abstractions, generics/templates; and
- Composition;

Modeling:
- Principles of modeling;
- Pre and post conditions, invariants;
- Mathematical models and specification languages;
- Model development tools and model checking/validation;
- Modeling/design languages (such as UML, OOD, and functional);
- Syntax vs. semantics (including understanding model representations); and
- Explicitness (make no assumptions or state all assumptions)

Though not explicitly in the core, a year of calculus was also required to: enhance mathematical maturity and thinking; provide a contrast with discrete mathematics concepts; and ensure sufficient background

[2] There is nothing to make it false, so it must be true "vacuously."

for various client and application disciplines. Statistics and empirical methods were also recommended; personally, I would also add linear algebra. Advanced mathematical courses, including graph theory, combinatorics, theory of computing, probability theory, operations research, and abstract algebra, might also be required, depending on the goals of the program and the needs of individual students.

Mathematically oriented software engineering courses today cover formal specifications, formal methods, mathematically rigorous software design, software verification and validation, and software models and model checking. As software engineering matures, the word "formal" will disappear; it is rarely used in traditional engineering where formal approaches are the norm.

Specific material varies by degree program. However, an important goal is to ensure foundational mathematical concepts are introduced early and used and reinforced in computer science and software engineering courses in the same way continuous mathematics is used and reinforced in traditional engineering courses. It will, however, take time, dedication, and rethinking the current software engineering curricula.

Conclusion

Mathematical reasoning is intrinsic to both traditional engineering and software engineering, though each discipline uses different foundational mathematics. Traditional engineers use continuous mathematics primarily in a calculational mode for modeling, design, and analysis, including to calculate, say, load on a bridge component, compute the wattage of a resistor, or determine the optimum weight of an automobile suspension system. Software engineers usually use discrete mathematics and logic in a declarative mode for specifying and verifying system behaviors and for analyzing system features.

The RC circuit and iteration invariants described earlier illustrate basic mathematical reasoning. However, engineering is significantly deeper and broader than these examples indicate. Engineers are systems architects who understand and apply the foundational principles of the discipline. Software engineers must therefore learn to use mathematics to: construct, analyze, and check models of software systems; compose systems from components; develop correct, efficient system components; specify (precisely) the behavior of systems and components; and analyze, test, and evaluate systems and components. They must therefore understand the theoretical and practical principles of programming and be able to learn and use new languages and tools.

One area where traditional engineering has an advantage is the number, variety, and maturity of tools for mathematical modeling, design, analysis, and implementation, including standard languages for communication in blueprints and schematic circuit diagrams and computer-aided prototyping, design, and analysis tools. Comparable software engineering tools are emerging as the discipline matures.

Evidence supporting the importance of mathematics in software engineering practice is sparse. This naturally leads to claims that software practitioners don't need to learn or use mathematics [4]. Surveys of current practices [8] reflect reality; many software engineers have not been taught to use discrete mathematics and logic as effective tools. Education is the key to ensuring future software engineers are able to use mathematics and logic as power{ful} tools for reasoning and thinking. ■

References

1. Bentley, J. Programming pearls: Writing correct programs. *Commun. ACM 26*, 12 (Dec. 1983), 1040–1045.
2. Clark, E., Grumberg, O., and Peled, D. *Model Checking*. MIT Press, Cambridge, MA, 1999.
3. Devlin, K. The real reason why software engineers need math. *Commun. ACM 44*, 10 (Oct. 2001), 21–22.
4. Glass, R. A new answer to 'How important is mathematics to the software practitioner?' *IEEE Software 17*, 6 (Nov./Dec. 2000), 135–136.
5. Henderson, P. et al. Striving for mathematical thinking. *SIGCSE Bulletin (Inroads) 33*, 4 (Dec. 2001), 114–124.
6. Hinchey, M. and Bowen, J., Eds. *Applications of Formal Methods*. Prentice-Hall, London, U.K., 1995.
7. LeBlanc, R. and Sobel, A., Eds. *Computing Curricula 2001: Software Engineering Volume (1st Draft)*, June 25, 2003; see sites.computer.org/ccse/volume/FirstDraft.pdf.
8. Lethbridge, T. What knowledge is important to a software professional? *IEEE Comput. 33*, 5 (May 2000), 44–50.
9. Parnas, D. Software engineering programmes are not computer science programmes. *Annals Software Engin. 6*, 1–4 (1998), 19–37.
10. Roberts, E., Ed. *Computing Curricula 2001: Computer Science Final Report*. IEEE Computer Society, New York, April 2002.

PETER B. HENDERSON (phenders@butler.edu) is a professor in and head of the Department of Computer Science and Software Engineering at Butler University, Indianapolis, IN.

BY
VICKI L. ALMSTRUM

WHAT IS THE ATTRACTION TO COMPUTING?

WHAT MOTIVATES ANYONE TO STUDY COMPUTING? A NATURAL CURIOSITY ABOUT THE UNDERLYING CONCEPTS? THE FIELD'S POTENTIAL USEFULNESS IN OTHER AREAS? PROGRAMMING? THE ABILITY TO ADVISE LESS TECHNICALLY LITERATE COLLEAGUES? BUILDING WEB SITES OR DESIGNING SYSTEMS, INCLUDING VIDEO GAMES? AFTER SEEING SUCH PREFERENCES CITED IN ARTICLES AND BOOKS AS COMMON KNOWLEDGE, I BEGAN SEEKING EVIDENCE THAT WOULD SUPPORT OR REFUTE SUCH STATEMENTS AND IDENTIFY THE FACTORS THAT INFLUENCE WHETHER OR NOT INDIVIDUALS CHOOSE TO ENTER AND STAY IN THE FIELD.

The strongest motivators include a sense of accomplishment from solving problems and programming; the weakest include being captivated by the Web and a passion for playing computer games.

In 2002, as I prepared to participate in a panel called "Women, Mathematics, and Computer Science" at the annual SIGCSE Technical Symposium [3], I heard and read a variety of assertions about the factors attracting and repelling students to and from the computing field. The striking lack of evidence supporting most of the assertions prompted me to design a survey to gather data that might shed light on the issue. The survey was exploratory in nature, with the core set of items based directly on statements derived from a variety of sources. I also included items that would allow me to profile the respondents and analyze the data based on various characteristics, including gender and highest academic degree completed.

Nearly 500 computing professionals from the U.S. and other countries completed the survey. The sampling method, though

ILLUSTRATION BY JEAN-FRANÇOIS PODEVIN

somewhat informal, quickly put me in touch with a large number of professionals directly concerned with these issues. I recruited participants through several online mailing lists, including Systers (www.systers.org) and the members mailing list of the ACM Special Interest Group on Computer Science Education (www.sigcse.org). I also encouraged respondents to invite their colleagues to participate, thus increasing the international perspective; about 11% of the respondents said they were employed outside the U.S. Highlights of the respondents' demographics included:

- Given this survey was for a panel discussion on women, math, and computer science, most of the respondents (78%) were women;
- Based on the year they reported finishing their precollege education (assuming it occurred about age 18), the average age of respondents was 39, with good representation in all age categories, from under 25 to over 60;
- Nearly 90% reported having at least a bachelor's degree, over 50% a master's degree, and 37% a doctoral degree;
- Over 50% were involved in education, either as students (17%) or instructors (36%). About 33% reported their principal job responsibility was in industry, including in software development, technical management, and consulting; and
- Over 50% reported having taught at the college level, with 24% teaching both computing and mathematics courses, 28% teaching only computing courses, and 3% teaching only mathematics courses.

Details of the development and content of the survey, along with the ongoing results from my investigation, are available at www.cs.utexas.edu/users/csed/sigcse/2002/women-math-cs/.

Attitudes Toward Math and Computing

One section of the survey focused on respondents' attitudes toward mathematics and computing by asking how strongly they agreed with several statements. About 75% of the respondents indicated that a good understanding of mathematics is important for computing professionals; the same percentage replied that this understanding is also important for being able to appreciate the concepts of computing. Only 40% expressed a preference for programming courses over theoretical courses. Practically all respondents agreed that mathematics is a key foundation for computing. Only 2% disagreed with the statement: "A good understanding of mathematics helps one better understand and appreciate the concepts of computer science."

Three statements focused on perceptions of the role of math anxiety in undergraduate student success in computing. Only 15% of the respondents agreed with either of the two statements (one about women, the other about men): "Undergraduate [women/men] who suffer from math anxiety cannot succeed in a computing field." The nearly perfect positive association between responses to these statements suggests a view that math anxiety can affect a student's ability to succeed in computing, irrespective of gender. Responses to the statement: "At the undergraduate level, men are as likely to suffer from math anxiety as are women," showed little relationship to the responses to any of the other statements related to math anxiety. The patterns of responses to the math anxiety statements suggest a relationship between respondents' personal experience with math anxiety and their opinions about the likelihood of success for both women and men at the undergraduate level.

About 80% of the respondents agreed with statements concerning whether they enjoyed mathematics or were good at it. The responses shared substantial negative association with the responses to statements about whether undergraduate men and women suffering math anxiety are able to succeed in a computing field. There was also a positive association between responses about enjoyment of and personal skill in mathematics to the statement about whether men and women are equally likely to experience math anxiety.

Attraction Factors

To investigate why individuals choose computing as a profession, I included in the survey a set of 18 possible attraction factors suggested by the literature, refining them through a series of pilot runs with earlier versions of the survey. Respondents rated each such factor as to whether it had been important, minor, or irrelevant in their personal histories.

The two most influential factors influencing whether or not respondents pursued computing were "the sense of accomplishment that comes from solving the problem" and "using a computer to solve problems," as cited by 96% and 93% of the respondents, respectively. The least-cited factor was "a passion for playing computer games," influencing only 18% of the respondents. The second least-cited attraction factor was "captivated by the Web," influencing only 30% of the respondents. Each of the other 14 attraction factors was cited as influential by over 50% of the respondents (see the table here).

In analyzing the responses, a complex pattern of strong, mostly positive, associations emerged among

the 18 factors. To search for relationships among them, I built a descriptive model using the technique of primary component analysis, a statistical method related to factor analysis [4]. Using these techniques, a good model should result in a small number of underlying dimensions, each composed of two or more of the observed variables—in this case, the attraction factors—such that the dimensions have a meaningful interpretation. The primary component analysis resulted in a model of four dimensions, with each of the 18 factors contributing to various degrees across these dimensions.

Interpreting this model, I determined which factors contributed most to each dimension. The combinations of factors led me to label the dimensions as follows (adapted from dictionary.com):

Factors attracting survey respondents to computing.

Dimension	Weight*	Factors from Survey	Influence†
Benefit	[0.84]	*creating something that will benefit others*	85%
	[0.82]	*using the computer to solve problems*	93%
	[0.77]	*usefulness in supporting other areas/fields*	85%
	[0.74]	*sense of accomplishment from solving problems*	96%
	[0.55]	*ability to create "exact" solutions*	69%
Science	[0.87]	*logic involved*	90%
	[0.72]	*mathematics at its foundations*	60%
	[0.67]	*beauty and elegance one can achieve*	76%
	[0.56]	*programming*	81%
Experiment	[0.92]	*prospect of something to play with*	76%
	[0.87]	*possibility of experimentation*	85%
	[0.81]	*"gadgetry"*	55%
	[0.66]	*having something to push to its limits*	55%
Vanguard	[0.89]	*captivated by the Web*	30%
	[0.70]	*passion for playing computer games*	18%
	[0.65]	*being in a position to advise the less technically literate*	60%
	[0.59]	*being on the cutting edge*	69%
	[0.47]	*linguistic foundations*	52%

* Standardized regression coefficient, which is functionally related to the partial or semipartial correlation between a variable and the dimension when the other dimensions are constant [3]. A value closer to 1.0 indicates a stronger contribution to that dimension.
† Reports the percent of respondents choosing the corresponding factor as either an important factor or a minor factor in attracting them to computing.

Benefit. Something promoting or enhancing well-being;
Science. The observation, identification, description, experimental investigation, and theoretical explanation of phenomena;
Experiment. Trying something new, especially in order to gain experience; and
Vanguard. The foremost or leading position in a trend or movement.

The contributing factors generally made sense in the context of the dimensions. The only factor for which placement was somewhat puzzling was "linguistic foundations," which had its greatest loading in the Vanguard dimension, though the loading was also the lowest maximum among the 18 factors. Had I tried to predict the placement of this factor in advance, I would have placed it in the Science dimension. Perhaps the phrasing of the term led to inconsistent interpretations by the respondents.

Composite Profile

The composite profile of survey respondents helps guide interpretation of the survey results. Because about 33% of them classified their principal job responsibility as educational and about 33% gave their current employment responsibility as being in an industrial setting (such as software developer or manager), the data did not appear overly focused on either education or industry. However, nearly 80% of the respondents were women. Because extensive research indicates important differences in how men and women become interested in, enter, and remain in the computing field (see, for example, [2]), we can now ask whether the four dimensions of attraction to computing would hold in professional and educational settings with greater proportions of men. Future work is needed to validate the four-dimensional model by considering whether it holds for various subgroups, as well as for the general population.

The four-dimensional model does have potential for explaining why individuals are attracted to the computing field. Considering the composite scores, respondents with higher scores on the Experiment dimension also had higher scores on the Benefit and Vanguard dimensions. Science had the weakest associations with the other three dimensions, indicating that when individuals viewed the Science dimension as important, they were less likely to view the other dimensions as important, too. This outcome suggests that the Science dimension is a more unique rationale for entering computing than are the other three dimensions.

These results suggest that computing professionals have a strong sense of the field's mathematical roots and tend to be unhampered by math anxiety. It also appears that a love of programming, commonly cited as a key reason for entering the field, may not in itself be an especially strong motivator. Instead, the derived dimensions of Experiment, Benefit, and Vanguard may better explain what attracts most people to computing.

Implications for Students and Teachers

What do these results imply for students and teachers of computing courses? The relative strengths of the four dimensions in the attraction model suggest curricular strategies teachers can use to help encourage individual students to consider entering or continuing in the computing field.

Benefit (strongest motivator). The Benefit dimension was the strongest motivator for the respondents in the survey, with four of the five component factors selected as influential by 85%–96% of the respondents. The most influential factor within this dimension was "the sense of accomplishment that comes from solving the problem." A curriculum that emphasizes the factors of the Benefit dimension is thus likely to encourage people to consider entering and staying in the field.

Science (second motivator). The Science dimension emerged as the next-strongest motivator. The two most influential component factors were "the logic involved" and "the programming." The other two factors—"the beauty and elegance one can achieve" and "the mathematics at its foundations"—were influential factors for 75% and 60% of the respondents, respectively. A curriculum blending an appreciation of the underlying theory with an understanding of the practical skills of programming would support this dimension.

> THE RESULTS PROMISE TO HELP TEACHERS, CURRICULUM DESIGNERS, AND ADMINISTRATORS FINE-TUNE THE WAYS THEY RECRUIT AND RETAIN TALENTED STUDENTS IN THE COMPUTING DOMAIN.

Experiment (relatively weak motivator). The Experiment dimension, a relatively weak attractor for these respondents, had as its most influential factor "the possibility to experiment." The two component factors—"gadgetry" and "having something to push to its limit"—were influential factors for just over 50% of the respondents. This suggests that a computing curriculum with a strong experimental aspect might help attract people to computing and keep them in the field, but that "tinkering" and "tweaking" activities may have less appeal. On the other hand, [2] observed that boys/men tend to tinker with ideas that interest them, while women tend to focus on concepts and skills that will help them in class. This suggests that for some people, tinkering may help them better understand concepts not presented in class. Thus, while fostering a willingness to experiment and try new things may not initially attract people to computing, engaging in these practices may make a difference in some students' success over time.

Vanguard (least important motivator). The least important of the four dimensions was the Vanguard dimension. Among its five component factors, the two strongest were "being on the cutting edge" and "being able to advise less technically literate individuals." In the context of a computing curriculum, this suggests that

including concepts from the forefront of technology and making them explicit can help attract and retain students. The two weakest factors in this dimension were "captivated by the Web" and "a passion for playing computer games," suggesting that such specific current activities may not be primary motivators in the long term.

Conclusion

A key insight from this exploratory study is that much remains to be done. The four-dimensional model of attraction to computing suggests priorities for curricular elements that might pull individuals into the field. At the same time, these dimensions may play different roles in different segments of the population: female vs. male, academic vs. industry, experienced vs. novice. The respondents' written comments suggest that additional factors, including money, opportunity, and job security, also attracted them to computing.

To build on this work, it would make sense to redesign the survey and distribute it more broadly in several target settings; for example, the wording of certain factors could be improved and additional factors added. Another issue is the reliability of self-reports regarding motivating factors from earlier in the respondents' lives; perhaps some respondents answered in ways they wish had been true at the time they made their decisions concerning their pursuit of computing.

The results, including the model of attraction to computing, promise to help teachers, curriculum designers, and administrators fine-tune the ways they recruit and retain talented students in the computing domain. The model may also suggest how to advise young people who are considering entering, staying in, or even leaving the field. C

References

1. Academic Computing and Instructional Technology Services. *Factor Analysis Using SAS PROC FACTOR (Usage Note: Stat-53).* ACITS, The University of Texas at Austin, June 26, 1995; see www.utexas.edu/cc/docs/stat53.html.
2. Gürer, D. and Camp, T. *Investigating the Incredible Shrinking Pipeline for Women in Computer Science (Final Report).* National Science Foundation Project 9812016, summer 2002; see www.acm.org/women/ pipeline-final-report_ver_2.doc.
3. Henderson, P. (moderator). Panel on Women, Mathematics, and Computer Science. At the SIGCSE Technical Symposium 2002 (Cincinnati, KY, Feb. 27–Mar. 3, 2002).
4. StatSoft, Inc. Principal components and factor analysis. In *Electronic Statistics Textbook.* StatSoft, Inc., Tulsa, OK, 2002; see www.statsoft.com/textbook/stathome.html.

VICKI L. ALMSTRUM (almstrum@cs.utexas.edu) is a lecturer in the Department of Computer Science at the University of Texas at Austin.