Symbolic Programming vs. the A.P. Curriculum

Brian Harvey
University of California, Berkeley

A popular metaphor in recent years compares the process of writing a computer program to that of designing and building a bridge. The point of the metaphor is that it's not acceptable to debug a new bridge design by building the bridge, opening it to traffic, and waiting to see whether or not it collapses. People who use this metaphor argue that the analogous technique isn't acceptable in computer programming either. The phrase *software engineering* was coined to evoke the comparison with civil engineering and other, similar disciplines. The view of programming as software engineering has had a profound influence on computer science education. The purpose of this paper is to examine that influence, particularly at the high school level through the College Board Advanced Placement curriculum, and suggest an alternative view.

(I am focusing attention on the A.P. curriculum to make the point concrete, but it is just one of many similar influences. The College Board did not invent the software engineering view; it tries to follow the lead of its client colleges. At the college level, the ACM curriculum standard could be cited instead. I teach at both levels, but I feel most strongly about the inappropriateness of the software engineering approach in secondary education.)

Software Engineering: Programming as Discipline

What are the characteristics of the software engineering approach? One fundamental assumption is that *the goal of a project is given in advance.* It is not the civil engineer's job to decide where a bridge should be built, or whether a bridge should be built in the first place. Those decisions are made by someone else—politicians, traffic flow experts, perhaps railroad owners. The engineer is employed to carry out a predefined project, and is given a formal specification of the requirements.

A natural consequence of externally-provided goals is a *top-down design* technique. Ideally, beginning with the formal goal specification, the engineer maps out large subtasks, then develops each subtask into more and more detailed structure by a process called "stepwise refinement," one of the central elements of the *structured programming* methodology. Mastery of this technique is the main content, apart from programming language syntax, of most introductory computer science courses.

If a bridge falls down, its designer can be sued for damages. A similar standard of *professional liability* should apply to the authors of computer programs, if lives or property are endangered by their failure. Some professional societies have begun to propose uniform standards for licensing of software engineers. If programmers are held liable for their failures, it pays to use *conservative techniques* in the design of new programs. Just as civil engineers rely on a few well-understood designs for bridges, programmers should use proven algorithms to ensure program correctness. Mistakes should not be debugged by trial and error, but should be prevented in advance by redundant error checking, strict adherence to design standards, and formal proofs of correctness, techniques that one popular text calls *antibugging* [Cooper and Clancy, 1985].

One person can't build a bridge alone. Bridges are built by large teams, and it's important that each team member have a well-defined job. Similarly, software engineering emphasizes the needs of *programming teams* in which each programmer is assigned a subtask. The top-down design approach is particularly important in a team environment so that each program segment has a well-defined interface to the rest of the project. Rigid standards are required about things like scope of variables and documentation of subprocedures.

Finally, an *emphasis on product* has been implicit throughout this description. What is most important is the program, rather than the style of work of the programmers. A software engineer derives job satisfaction from seeing people use the product, more than from the programming itself.

(In a recently-published polemic, Edsger Dijkstra, one of the founders of the structured programming methodology, dissociates himself from the name "software engineering" because he thinks that it evokes a methodology based on testing, rather than on the formal mathematical reasoning that Dijkstra prefers. Still, from the point of view I am presenting, Dijkstra and the software engineers are in the same camp because they both focus on producing a program that meets a prior specification, rather than on the development of that specification.)

Artificial Intelligence: Programming as Art

Some computer programs actually do control processes in which a malfunction could threaten human life. In those cases, I agree that programmers should regard their work as closely analogous to other engineering disciplines, with the same professional standards and the same conservative techniques. Much programming, though, does not have this character. If a video game, for example, malfunctions, no lives are lost. Reliability is still desirable, of course, but other characteristics, such as creativity in the design of the game, are equally important. Unlike bridge-building, much computer programming is done by hobbyists for their own personal satisfaction, and professional liability does not apply in this case.

Among professional programmers, research work is in some ways more like hobbyist programming than like civil engineering. Reliability may be less important than the flexibility to explore new techniques. (Of course, reliability is itself the topic of some computer science research, but even then the research cannot proceed by invoking already-known methods.) In particular, artificial intelligence research is characterized by the desire to extend the limits of what can be done with a computer; the researcher typically does not have a body of well-understood, reliable methods to follow. It is not surprising, then, that AI researchers have developed programming tools and approaches quite different from those of software engineering.

In a research project, *the goal of a project evolves in the work.* That is, the programmer starts with a broad idea, but often cannot specify the detailed behavior of the program until it's written. She starts with a partial understanding, attempts to program some better-understood corner of the project, and then interacts with the resulting program to see in what direction to proceed. The goals are chosen by the same person doing the programming.

If the goal is not precisely known in advance, a strict top-down design process is impossible. Researchers use *interactive design* techniques, which may include some top-down components but also may include a bottom-up approach in which the work already done provides a software toolkit that itself suggests new higher-level modules. Instead of a single programming methodology like stepwise refinement, the researcher uses *eclectic techniques.*

Since the result of a research project is rarely destined to be used directly in real applications, there is *no danger* and no liability. Instead of following conservative, well-understood patterns, the researcher is encouraged to *take risks* and invent new methods. Program debugging can be interactive; other members of the research community are often encouraged to try out the program, not only discovering coding bugs but also contributing to the incremental design process by suggesting improvements.

Research programs are rarely written by large teams. One reason is that you can't get a Ph.D. for writing one percent of a program; another is that it's hard to coordinate large teams without a precise, formal design document. But the most important reason is that large teams are rarely creative. Instead, *individual creation* is required to solve new problems.

To summarize, there must be an *emphasis on process* in the research programming environment. The resulting program may be demonstrated once and never used again, since it will probably be slow and clumsy, but by writing it, the researcher learns new ideas that can be refined and applied later. In education, too, the actual programs written by students are unimportant; the student, not the program, is the "product" of the teaching and learning activities. We should not be surprised if process-centered approaches to programming turn out to be most helpful in education.

Even when a program is developed for practical use, the development process may include steps that are more like research than like building bridges. Consider spreadsheet programs. The history of their development, starting from VisiCalc and continuing to current products like Lotus 1-2-3, has been a classic software engineering effort, with large development teams and precise goals. But the first step, starting from nothing and getting to VisiCalc, took two people, one of whom had the idea, "here is a new thing we can do with a computer," and the other of whom was a virtuoso computer hacker, trained in the MIT research environment.

(By the way, I do not mean to suggest that there is no research among civil engineers! No doubt new bridge designs grow out of a development methodology much like the one I'm describing for programming research. The contrast I am making is not between programming and civil engineering, but between research of any kind and the one-sided caricature of civil engineering that provides the metaphor for the software engineering school.)

The metaphor of bridge-building powerfully captures the spirit of the software engineering view of programming. As an equally powerful metaphor for this alternative view, I suggest considering programming as an art form, and the programmer as an artist.

How to Train an Artist

I am discussing the nature of computer programming to make a point about the nature of computer programming education. There are two very different kinds of programming activity, bridge-building and art. I suggest that our standard computer science curriculum does a good job of preparing students for the first kind, but not for the second. If we want to develop computer programming artists, we should explore the training of other kinds of artists.

(Both of these approaches are extremes. Some readers may feel more comfortable with an intermediate position, representable in metaphor by professions such as medicine and architecture that combine the need for individual creativity with careful attention to conservative techniques. Still, I am trying to call attention to an imbalance in our current educational practice, and in this context I think it is valuable to show what the opposite extreme would be like. I hope that even this extreme will not seem impractical, even if we ultimately settle on a compromise view.)

The training of artists is not the exact opposite of the training of engineers; art schools do not merely throw the student into a room full of materials and say, "go to it!" There is a discipline to art also, including a collection of well-understood, conservative techniques.

I suggest that the training of an artist can be divided into three categories of learning. There is *technical knowledge*, such as the differences between oil paints and water colors, or the rules of perspective. There is *technical skill*, such as the ability to draw a straight line freehand without wiggling, or the ability to stretch a canvas evenly before painting on it. And there is *inspiration*, seeing the world with an artist's eye and finding something worth painting at all.

All three of these are essential to a great artist, but inspiration comes first! I am not an artist, and my personal experience of art instruction has been limited to a couple of introductory courses when I was in high school. In those courses, though, the teachers made a big fuss about staying away from technique. "Never mind what it looks like" was a frequent comment. One beginning exercise was to draw a chair with our eyes closed; the point was to overcome our anxiety about photographic exactness and get us to enjoy moving a pencil over paper in broad strokes. The teachers promised that technical instruction, for those of us who did develop a deeper interest in art, would come later.

Art instruction that began with endless practice at drawing straight lines and perspective cubes would chase away any true potential artists. At best it would produce "commercial artists," a classic oxymoron. This is the risk we run in computer science education. (Commercial artists are socially useful, and so are the members of software engineering teams. But there could be no commercial art without true creative artists to invent the language and the tools of art.)

The Programming Language Paradox

Software engineers believe in constructing a program top-down. That is, they start with the broad ideas of the program, and fill in the details later. You would think their design tools would support programming in terms of broad ideas. Yet the programming languages of software engineering, of which Pascal is the prototype, focus attention on low-level details. They use explicit storage allocation, so the programmer must think about the size, location, and extent in time of each data aggregate. They use strong typing, encouraging the programmer to concentrate on the precise encoding of symbolic information rather than on the meaning. They provide half a dozen specific, hardwired control structures, emphasizing the sequence of events in a computation rather than the input-output behavior of functions. Complex data structures are manipulated with explicit pointers, a common source of confusion and bugs.

Some of these low-level properties of software engineering languages are the result of their early history as merely an abbreviation for machine language instructions. But, for example, the transition from C (a hacker's language) to C++ (a software engineer's language) has brought more attention, not less, to these low-level details. The improvement is that in C++ the details can be represented more rigorously, with less need to rely on *ad hoc* escape hatches such as type casting.

Software engineers recognize the inadequacy of their programming languages as design tools. That is why they develop their programs using "pseudocode," essentially English with indentation to indicate block structure. English is a good high-level design language precisely because it does not require the programmer to think about memory allocation, variable types, and so on. The difficulty is that if one really designs in English, it's possible to write an instruction that has no obvious translation (or even no translation at all) into any programming language. To end up with useful pseudocode, the programmer must actually cheat, writing the program mentally in something like Pascal and then translating into English and leaving out most of the details.

There are programming languages that don't require such extreme attention to low-level details as the ones software engineers love. These symbolic programming languages can readily express overall design as well as details. Why don't the software engineers choose higher-level languages? The reason is that they overemphasize their concern with reliability, and in particular with possible low-level bugs. For example, although a language with explicit variable typing forces the programmer to be concerned with low-level details, it also may help catch bugs in which an incorrect value is assigned to a variable.

The educational cost of the software engineering languages is that much of the time and effort in introductory courses goes into syntactic details, and into such low-level semantic issues as binary representation, word length, ASCII codes, and pointer arithmetic. By contrast, an introductory computer science course that uses Lisp [Abelson and Sussman, 1985], coming from the artificial intelligence research tradition, can focus attention on such high-level semantic issues as functional programming, object-oriented programming, and logic programming methodologies, data abstraction, and higher levels of abstraction such as the design and implementation of a programming language. There is no pseudocode in this text; a Lisp procedure written in terms of subprocedures that may not yet be defined provides the right level of abstraction, while retaining technical rigor.

The title of this paper is "Symbolic Programming vs. the A.P. Curriculum." It could equally well have been "High-Level Semantics vs. the A.P. Curriculum" or "Programming as Art vs. the A.P. Curriculum." For my purposes, what's important about symbolic programming is that it allows us to step back from technical details, in a way somewhat analogous to blindfold drawing exercises. There are technical ideas in the curriculum, but they are high-level ideas, and they don't deaden the hacker spirit (that is, the artistic spirit). In fact, as an introductory college-level course, I think the Abelson and Sussman course works better than the traditional curriculum even from a bridge-building perspective, because even software engineers should start with the big ideas. Data abstraction, for example, is part of the structured programming approach as well as the artificial intelligence approach to program development.

I want to address a possible misunderstanding of the point I'm making. The choice of programming language is not the central issue in designing an educational process. It would be possible to teach a software engineering curriculum in any language, including Lisp. (Indeed, these curricula generally make a point of not specifying a language, and the authors of the curricula respond to accusations of language chauvinism by pointing that out.) Such a curriculum would be just as bad as the same course taught in Pascal. Rather, my point is that an examination of the programming languages that the adherents of each approach actually do choose sheds light on how these approaches work out in educational practice. The principle of top-down design sounds great when expressed in the abstract, but the real teaching done in its name too easily gets bogged down in low-level technical details.

What High School Students Need

In the United States at present, high school computer science education is strongly affected by the accidental fact that there is no official secondary curriculum, but there *is* an official college-level curriculum for secondary schools in the form of the College Board Advanced Placement exam. The result is that most high school students who are interested in programming are now taught from a model that was designed (and badly designed, as I have argued above, at that) for a college population. The problem is not the College Board's fault; a lack of creative leadership at the secondary level has allowed what should be an advanced curriculum for a minority of students to set the tone for all programming instruction.

At the college level, where we are engaged in the training of professional programmers, the software engineering approach is sensible, even if not the whole story. I have not suggested abandoning concerns about professional liability, conservative design, and teamwork in the context of programming projects in which lives or property are at risk. Rather, I have argued that software engineering concerns should be balanced by a lively sense of the artistic, creative side of professional programming, and that large ideas should come before technical details in the curriculum.

At the high school level, I want to argue for a more extreme view. High school students are not about to enter the job market as professional programmers. Rather, they are exploring a possible interest in programming, and preparing for further instruction at the college level. *They can learn discipline later.* Now they need a kind of apprenticeship: real tasks to work on, the freedom to experiment and to risk failure, but in an environment that models respect for commitment. They need teachers who serve as experts available for consultation, rather than lecture on technical knowledge or set exercises of technical skills.

(I am concerned here with the needs of those students who are interested in computer programming. The design of a "computer literacy" curriculum for all students is a separate issue. I have argued elsewhere that it is really a non-issue; students should be quite comfortable with word processors and other utility programs long before they leave elementary school. The widely perceived need for a secondary school "literacy" curriculum is a temporary result of the fact that computers have only recently entered the schools in significant numbers.)

Not every teenager wants to program computers, but many do. Computers are powerful, responsive tools. Apart from reading, few intellectual skills appeal so strongly to young people, especially not formal, technical ones. We educators are lucky that computer programming ranks almost as high as skateboarding and playing the guitar! We should teach programming in a way that nurtures this excitement, not turning it into one more academic subject full of facts to be paraded on exams. Instead, high school programming instruction should be project-based, with projects chosen by the students themselves as much as possible. The A.P. curriculum is particularly bad, because of its emphasis on conservative technique and correspondingly low-level details, but *any* fixed curriculum would interfere.

Of course the student's apprenticeship can't begin without an initial period devoted to learning the rules of a programming language. But I think we should look upon this early work as an unfortunate necessity, rather than as the focus of our curriculum, and we should try to get it over with smoothly and quickly.* To this end, we should choose a high-level language such as Lisp to minimize the time spent on syntax and machine representation issues. (Abelson and Sussman use the Scheme dialect of Lisp in their text; for high school

* In practice the division into stages is not so clear-cut. Students can be working on independent projects in parallel with formal instruction in the language. Still, the two are very different in style, and will probably be experienced by students as separate activities.

students I prefer the Logo dialect, which provides many of the same high-level mechanisms along with a user interface that is less intimidating to a beginner.) I also like to minimize formal lectures, relying instead on a self-paced format and scheduling advanced students into the lab at the same time as the beginners to allow for informal one-on-one tutoring.

In this early stage, I have come to think that debugging (and antibugging) should *not* be a large part of the student's experience. It's true that debugging is excellent mental exercise, but real beginners tend to make uninteresting mistakes. Spending 15 minutes struggling with an unnoticed punctuation error just teaches frustration, and so I will often debug a beginner's program myself and encourage the student to get on with the big idea that the bug interrupted. Only if the bug seems to indicate a serious conceptual misunderstanding do I make any fuss about it.

The second stage of the student's apprenticeship focuses on projects. This effort should be supported structurally with a curriculum-free lab course that can be repeated for credit; interested students may enroll in the lab throughout half a dozen semesters. (One problem with the "back to basics" movement is that it has encouraged a uniform set of courses for all students. One semester of programming is more than many students need, but not nearly enough for many others.) If the school will allow the course to be taught without assigning grades, so much the better. The teacher's role at the beginning of the semester is to ensure that each student takes on a project that is challenging but not overwhelming; later, the teacher serves as a consultant, looking over shoulders when that feels appropriate. It is also appropriate to make concrete suggestions about programming style, pointing out how a particular student program could be organized more clearly, but I think teachers should not insist on universal adherence to broad *a priori* rules.

For most high school students, this second stage can last through the senior year. If the programming projects are selected well, each student will learn certain universally important ideas (such as recursion), and also each student will learn many other ideas that happen to be relevant to his or her projects but are different from another student's learning. This project laboratory will be excellent preparation for a more rigorously structured computer science curriculum in college. Some students, however, will eventually get tired of projects and will ask for more formal instruction about meatier ideas. For them, a third-stage course can introduce computer science topics such as automata theory or compiler construction; alternatively, a true Advanced Placement course can be offered using the Abelson and Sussman text.

I have written a series of three texts, using Logo, to support these three stages [Harvey, 1985, 1986, 1987]. The first (introducing the language) and the third (computer science topics) are meant to be read sequentially, like most texts. The second is a collection of projects, in no particular order, intended to serve as an inspiration and to provide starting points for student projects rather than to be read from cover to cover. A series like this has the advantage of a uniform style and the ability to rely on earlier learning in the later volumes, but alternatives are available for each of the three stages. In particular, there are many collections of programming projects available, and the computer lab should have a wide assortment. In the third stage, even a Pascal-based text covering the official Advanced

Placement curriculum wouldn't be too harmful; my concern is that that curriculum has become, by default, the only programming experience for many high school students.

Conclusion

To summarize, I am presenting for computer science an argument that has come up before in many other areas of science and mathematics education. Our official curricula, I think, run the same risk as the official physics curriculum that caused Albert Einstein to drop out of school. Such rigidity would be particularly regrettable in a subject like computer programming, which lends itself so readily to a flexible, experimental approach.

References

Abelson, Harold, and Gerald Jay Sussman. *Structure and Interpretation of Computer Programs*, MIT Press, 1985.

Cooper, Doug, and Michael Clancy. *Oh! Pascal!* Second Edition, Norton, 1985.

Harvey, Brian. *Computer Science Logo Style*, MIT Press.
Volume 1: *Intermediate Programming*, 1985.
Volume 2: *Projects, Styles, and Techniques*, 1986.
Volume 3: *Advanced Topics*, 1987.