Computer Science, Software Engineering, and the Cost of Software

David Feign
Chapman College

For the past ten years all of the computer industry has been deeply concerned with the skyrocketing costs of software. In contrast, the dramatically falling costs of hardware have evoked predictions of the tremendous expansion of computer applications. In order to fulfill these optimistic predictions, it would seem that something will have to be done about the software problem.

Almost everywhere you look you can find articles about the software problem. Just a bibliography on this subject would fill this publication. The popularity of the subject has led to all sorts of myths, fables, and misunderstandings, but not to very much in the way of solutions. In order to solve any problem, we must first decide what the problem is and what it is not.

There are many ways to reduce the cost of anything. Getting better people to do the work is one. Ratios as high as 30 to 1 in "productivity" have been found among different programmers. The obvious solution is to get more intelligent, better trained, and more experienced people to do the work. Intelligence is limited, and the astrophysical growth of the computer industry prohibits finding sufficient numbers of trained and experienced people. If this solution to the software problem is to work, we must identify the kinds of training and experience we need, and then we must permit a year to go by so as to give someone a year's experience. I have found that the most useful software is prepared not by the computer specialist but by the specialist in the field of application with a little training in programming. An engineer with a working knowledge of Fortran or an accountant with some familiarity with Cobol can produce better software cheaper than can the computer scientist who is well versed in operating systems and compilers and is fluent in a dozen programming languages. To reduce costs still further, the applications specialist must get more training and experience in programming or the computer specialist must get more training and experience in the applications. Which is easier? It depends on the application.

We computer scientists are frequently reminded of the falling costs of hardware contrasted to the increasing costs of software. But let us look at what is meant by the "cost of hardware" and the "cost of software." In most cases, the hardware costs are the unit prices of computers for which the design and development costs have been amortized over hundreds or thousands of copies, while the software costs are those for developing one program. That's like comparing apples and locomotives! Let's get a little closer and compare apples and oranges.

If we want to compare like costs, let us compare the price of a "cheap" micro-computer with the "expensive" software available for it. Take, for example, Radio Shack's TRS-80. The basic price of a "usable" computer is $599. But by the time a useful system is put together, the total package price can go as high as $3874. Inexpensive? Certainly! And those prices include built-in software/firmware. Now let's take the prices of the available software packages. These range from games (Backgammon/Blackjack) at $4.95 to useful applications (statistical analysis) at $29.95. Adding all the software advertised by Radio Shack, we get a total of $197.40. So where is the high cost of software?

When amortized over an equivalent market, software turns out to be cheaper than hardware!

So why are we spending so much more money on software than on hardware? Estimates run as high as 10 to 1 for software over hardware. The reason is simple. We spend more on software than on hardware because software is so much cheaper than hardware! As the Radio Shack example shows, if you are willing to buy standard packages and stick with them, the hardware will cost about 20 times as much as the software. But if standard packages do not meet your needs; what will you develop? I submit that developing a hardware solution to a problem would cost at least an order-of-magnitude more than the corresponding software solution. To repeat, software costs so much only because it is so cheap!

And the cheaper and bigger computers become, the more we will be spending on software. When there were only dozens of computers that executed 1000 instructions per second and had "large" memories of 1000 or 2000 words, one or two programmers could keep a computer usefully occupied 8 hours a day. Hundreds of programmers could then serve the needs of the computing community. Today, there are thousands of computers, some of which can execute tens of millions of instructions per second with memories holding up to a million words. Simple arithmetic applied to these numbers might indicate that the total population of this planet could not keep the voracious appetites of these machines satisfied. But things are not nearly that bad. More complex
programs are running for longer periods on much more data than 25 or 30 years ago. Nevertheless, there is a much greater need than in the past for people to feed new programs to computers. The process of designing, developing, and testing these programs will cost more than in the past. Because the computers are bigger and faster, the programs will be more complex. And because of increasing familiarity with and dependence on computers, we are getting more ambitious with the applications to which we are putting these machines. All of this leads to the inevitability of increasing costs for software.

The question that remains is: given that we will be spending more for software, how can we minimize these expenses? Most proposals to minimize software costs involve spending more to develop better “tools.” The Department of Defense, for example, has been involved in a large-scale effort to develop a new, “universal” programming language to reduce software costs. The main problem with this approach is that it attacks only a small fraction of the problem, namely the coding problem—how to express the solutions to your application problems. This is, at most, 20 percent of the cost of software. Even if this newer, “better” language does save half the coding expenses, it will be, at most, a 10 percent saving. The real issue is not how to express solutions, but how to develop solutions to problems. And, in my experience with both large and small problems, the biggest stumbling block is defining and understanding the problem. As I said before, this has nothing to do with programming languages or with computer science, but has to do with training, experience, and the resulting discipline in the application field.

The best thing that computer scientists and software engineers can do is to educate people in the fundamentals of what computers are, what they can and can’t do, and how to communicate with them. Like mathematics, computer science provides a tool for specialists in other fields to use. An understanding of the fundamentals in math or computer science is the prerequisite to using the tools well. The big challenge in both fields is to understand what these fundamentals are and how to communicate them to others. The recent failure of “the new math” is attributable mainly to the failure to communicate to the “teachers” what they were trying to say and how to say it to children. It seems to me that the best thing computer scientists can do to reduce the future costs of software is to try and understand what digital computers really are, what they really can and can’t do, and how, in general, to use these tools. Then comes the more difficult and demanding task of communicating these ideas to others.

To end on an optimistic note: in watching the “play” efforts of children using computers and the developing hobby of building computers, I can see a much greater understanding of the design, operation, and use of these devices by the general public in 20 years than I see among most of my colleagues (and myself) today.