Computing Science Education: The Road not Taken

Niklaus Wirth

Only a few days after receiving the request to deliver the opening address at this Conference on Computer Science Education, I read a presentation given by a colleague in the USA. It immediately caused me to accept this kind invitation. Said presentation culminates in a paragraph on teaching, and it says:

It is instructive to compare a mathematics text book with a computer text book in a high school curriculum. I had the misfortune to do so at some length, and to put it simply: we suck. I assume that we are sending a strong message: don't even consider computer science as a career unless you are a masochist.

I happen to agree. Having devoted a substantial part of my professional life to put the art of designing programs into a state where it can be taught in an orderly and systematic fashion, I have become disappointed of the dominant, erosive trends. Although I am tired of the unpopular role of a perennial critic, the paper caused a flare to erupt once again, and I am here, since the afore mentioned presentation then continues:

As computing professionals, it is our duty to speak up against a culture that equates computer literacy with mastering the intricacies of a production programming language.

This reminds me of E. W. Dijkstra's tale of his worst night after reading the specifications of the new programming language PL/1 in 1965. He said he had the painful vision that in the future *programming* will be equated with *learning PL/1,* and *computer science* with *mastering OS/360 JCL*. Replace PL/1 by C++ or Java, and JCL by Windows or Linux, and you are miraculously transposed into the present time.

Thereupon I wrote to my colleague to signal my agreement. He answered with the following explanation:

My scathing remarks on education came from my total frustration in teaching C++ to my son, who was a high school sophomore. The design of the language is a mess, and the book was an atrocity. My son could not understand why x = y should differ from y = x. Dijkstra complained to me that a significant book on Java did not include a grammar for it.

Indeed, a formal syntax was specified with the fourth revision of the language only! Let me continue the quote:

I am disappointed that not only this happens, but serious computer scientists take this to be absolutely normal. I have yet to see a manual on UNIX/C++/Java that I can dare to read in about a week. Their manuals are unreadable, they expect you to belong to a cult whose incantations are not public, and you are not to expect much in terms of reliability, coherence or general elegance. My frustration was extreme when I was teaching my son how to program in C++, an elective course in high school! After one semester of painful experience --for father and the son-- I advised him to drop the class.

What I don't understand is the lack of outrage among the computer scientists. When the college board decided to adopt C++ in the mid 90s, I recall having sent a scathing letter to them. But I don't count.

Teaching in high school appears to be an excellent place to test not only one's own aptness for teaching, but also the clarity of a text book. Now and then I receive complaints from high school teachers, reporting about difficulties with modern tools and languages, (and not seldom asking for versions of old Pascal for new equipment). The following is an excerpt from a correspondence with a researcher in theoretical physics in Russia, who besides has endeavoured to teach programming to bright high school students. He wrote:

It is interesting how the lyceum course helps with understanding things about programming and about teaching programming. Roughly half of the improvements in the next edition of the university course will be due to the lyceum experience. I have also learned what a horrible mess the programming education is in -- both at the high school and university levels

These are harsh words, but not exaggerated. What went wrong? What can be done? The quoted colleague's humble "I don't count" comes to mind. We feel helpless. Some feel condemned to lamenting, but most turn to accepting the plain facts, and to cope with them. But this is hardly a convincing attitude for intellectual leaders. Let us face it: Is not the majority of educational institutions firmly in the grip of a few companies whose only professional goal is to increase profits, be they computer manufacturers, software houses, or publishers? Universities, who could exert some correcting influence, and whose members are the authors of trendy text books, are in fact mostly interested in flashy research, leaving education to their teaching assistants.

Surely, in this post-modern academic environment the professor has long ago ceased to be the wise, learned man, penetrating deeper and deeper into his beloved subject in his quiet study. The modern professor is the manager of a large team of researchers, the keen fund raiser maintaining close relationships with the key funding agencies, and the untiring author of exciting project proposals and astonishing success stories. It would be suicidal in this highly competitive business to waste time pondering about how to best present trivial subjects to a mass of beginners. When it comes to course material, software and tools, the obvious choice is what lies on the shelf and has been adopted by everyone else anyway. In this fight for success and survival it is best to join the bandwagon. Achievement is measured in the size of a team, the number of publications generated and citations discovered, conferences visited and resources consumed, but not in the devotion to teaching, which is not measurable anyway. Surely, this academic life style is often contrary to the better knowledge of the individual, but is enforced by pressure to convert places of learning into profit centers of high visibility, and it borders on prostitution.

These are harsh words. What can be done? It is very difficult indeed to counteract a global trend as an individual. I think of the trend to shift from long term planning to short term profits, from learning in order to understand to applying skills for immediate action. Considering our own subject, the field of computing and programming, an academic institution's ultimate goal must be much wider than the mastery of a language. It must be nothing less than the art of designing artifacts to solve intricate problems. Some call it the art of constructive thinking. It is in this context that the availability of an appropriate tool, a properly designed language, is of importance. It assumes the role of a theory on which we base our methods. How can anyone learn to design properly and effectively, if the formalism, the foundation, is an overwhelming, inscrutable mess? How can one learn such an art without master examples worth studying and following? Surely, some people are more, some less gifted for good design, but nevertheless, the proper teaching, tools, and examples play a dominant role.

I spoke of designing complicated artifacts. It is an obvious fact that our artifacts become increasingly complicated. For example, a car no longer consists of an engine and four wheels only. It also incorporates a computer to determine the optimal amount of fuel to be injected at the most appropriate moments. This is in order to reduce fuel consumption, to conserve energy. But the modern car also contains a dozen (or more) well hidden electric motors to move windows and antennas, a radar system to indicate the distance to hostile objects, a system bus to connect all the gadgets and gismos, and a control computer to operate them. These objects do not contribute to a car's original purpose, but they add complexity and increase maintenance and cost. This sentiment was expressed in a recent article reviewing "the ultimate driving machine" in the New York Times (titled *Behind the wheel, BMW 745i, dazed by a technical knockout*):

"People seem to misinterpret complexity as sophistication," Niklaus Wirth, the Swiss computer scientist, once said. In luxury cars that flaunt the latest technologies, gadgets and amusements, it can be hard to tell the difference, just as it is difficult to define the point where technology intended to aid the driver starts to get in the way of driving … Suffice it to

say that the 7 is surely the world's most advanced sedan, but it is not the most user-friendly.

This example translates smoothly into the realm of hard- and software systems. They have become inordinately complicated not so much because all their wonderful features and facilities were actually needed, but simply because they are possible. For this reason they are hoped to yield an edge over the feared competition. The edge, however, consists of added bulk, difficulty in use, and hampered reliability. All this, however, the duped customer realizes only after the purchase.

The conclusion is that unnecessary, home-made complexity has many drawbacks. Most importantly, it blurs the vision for what is essential (the optimal fuel injection) and what is ephemeral (the motors to move windows). The trouble is that it takes much more talent, insight and time to construct an economical, lucid, and effective system than a complicated and bulky one.

So it is proper design that must lie near the heart of our teaching. But how are we to teach exemplary design with tools and languages that make us a laughing stock? To our regret, the software industry has done little to help us teachers in our dilemma.

Let us therefore peek into the software industry which supposedly is incapable of providing the ideal tools. We discover that the software industry itself is suffering from the exaggerated, unwarranted complexity, and from the lack of regularity and reliability of its products. Industry would indeed be much more productive to develop systems from a sound basis rather than by grafting new pieces on top of rotting stems. I happen to know of a case in a large company where a project was launched to design from scratch a prospective replacement for a widely used application that had grown too bulky and maintenance resistant. After some time, however, the project was cancelled on the advice of the marketing forces. The decision was based on the overwhelming resistance of customers against any change, and thereby against improvement. Among the customers were also many educational institutions. They had to protect their earlier investments in courses and course material.

Also, universities tend to become business-like, offering what their customers request and pay for, rather than what would be more educational in the longer run. The students, however, focus on what will provide the best job prospects, and this is learning the skills required by current industrial practice.

Evidently we are confronted with a highly stable vicious circle: Teachers cannot change their ways, because they must attract and please students, students request what is practiced in industry, and industry generates and applies what their employees had learnt. This deadlock reminds me of what I had characterized in the introduction to the report on Pascal in

1970: Universities were apt to teach what was requested by industry, and industry practiced what its employees had learnt in universities.

Vicious circles exist in order to be broken. It must be done by those who discover their viciousness. The trouble today is that this long term viciousness is insufficiently recognized. However, programming, the art of constructive design, is too important to be sacrificed to short term commercial gains and habits. Programming is perhaps the most important new discipline of the post-industrial era.

Frequently, the powerful software industry is blamed for imposing its products on the powerless customer community, including the insignificant minority of teachers and students. However, the situation is worse: Teachers have largely submitted themselves voluntarily to the prevailing commercial trends, often feeling uncomfortable and openly lamenting about their self-inflicted plight. Many people in industry deplore this abdication of leadership and responsibility.

Only university teachers are in a position to break that vicious circle. It will neither be easy nor done over night. But if it were impossible, something would be profoundly wrong with teachers in their academic freedom. They simply must rise to be leaders.

The breaking of the vicious circle has happened once before, namely with the success of Pascal. With the support of equal-minded colleagues, and with sufficient obstinacy against conventionalists, Pascal spread through many schools and even into industry. It did so against powerful competition from industry and other large organizations, against PL/1, Algol 68, and Ada. However, Pascal's far superior successors, Modula-2 and Oberon, did not find the necessary attention among teachers, and they succumbed to the least deserving opponent: C. Least deserving, because it defied all criteria that had emerged for sound program design. It confused students by letting x = y and y = x mean different things, and coerced everyone to write x = = y for the usual x = y. For such sins alone it would have deserved being ignored by educational institutions. Instead, this ugly syntax was wholeheartedly adopted by Java, whose embrace by the academic community was at least partly thanks to this continuity.

At numerous universities the recognized split between the educationally desirable and the practice of the "real world" was answered by selecting a functional language for teaching programming in the first year. This escape, however, only widened the gap, because it taught a different paradigm of programming, of reasoning about program design and, as a consequence, required a conscious shift when turning from learning to practicing. As a result, the scientific discipline and the engineering practice of programming became perceived as separate entities with hardly any visible connection. The former ended in an art for its own sake, the latter emerged from continued refinement of heuristics and intuition, of hacking with sophistication. But thus can never be the foundations of an academic discipline, which increasingly lies at the heart of all technical machinery.

Yet the fight for breaking the vicious circle is not entirely hopeless. We have identified those who must ride the attack. But how?

Let me end this presentation with a bold suggestion to this illustrious audience of teaching professionals. I envisage an exemplary text book as a promising starting point. It should satisfy the following criteria:

1. It starts with a succinct introduction into the basic notions of program design.
2. It uses a concise, formal notation. This notation is rigorously defined in a report of no more than some 20 pages.
3. Based on this notation, the basic concepts of iteration, recursion, assertion and invariant are introduced.
4. A central topic is the structuring of statements and typing of data.
5. This is followed by the notions of information hiding, modularization, and interface design, practiced on exemplary applications.
6. The book establishes a terminology that is both intuitive and precisely defined.
7. The book is of moderate size.

Allow me to conclude with two further remarks. The colleague quoted at the beginning of this talk ended by saying: "It has been the guiding principle of my career in education and research that well-trained professionals are considerably more effective than inspired amateurs. There must be a gap in their performance, and the gap should be considerable. I think it should be our collective goal to widen this gap."

A few months ago I received a request to list a number of problems which I consider the primary challenges in Computer Science for the next decades. Perhaps this text book should be among these challenges, perhaps even on top of the list. It is certainly one that has not been met so far.

Opening Address at the ITiCSE Conference, Aarhus, 24. 6. 2002