Software - Quality or Quantity, That is the Question

Prof. Niklaus Wirth, ETH Zürich

Niklaus Wirth is professor of Computer Science (Informatik) at ETH Zürich since 1968. He has made significant early contributions to programming methodology, and he is the designer of the programming languages Pascal, Modula and Oberon. Niklaus Wirth is recipient of the Turing Award and member of the Ordre pour le Merite

When I accepted the invitation to speak at this event I thought it would be some sort of after-dinner speech. Now I see it is a fairly big conference with a lot of people and I hope to be able to accommodate for that. The point is of course that I was asked by Ruedi Schild and Helmut Sandmayr, who were both former PhD students of mine, to come and give a talk at this 10th anniversary of their company and talk about my views on software and what it's all about. My first reaction was "Hell no!" I've not been in software anymore for 8 years. I use software, I create software, but I'm not teaching about software methodology, I'm not creating new languages anymore, which seems to be what everyone expects. I have shifted my field rather into designing tools for digital circuit design. And software has become a sad topic to me, a frustrating topic. But then, of course, what won't you do for two former PhD students, so here I am.

And then of course I have a certain fame for giving talks where there is a lot of grumbling. Amusing grumbling, of course, and so I hope I can fulfil my anticipated role.

Yes, Software and Quality: you know, this combination of words is a little bit loaded. Of course, software has, like most other fields, a history and it has its triumphs and it has its failures. And it has become so important in technology and in our everyday life that software is also a darling for the journalists, and whenever something goes very bad they welcome it very much because they can make a big story out of it; the big successes are hardly ever reported.

I said I'm well-known for grumbling now and then, but I would like to begin my talk with a reminder that there is an enormous amount of software in this world, here and there and everywhere, that works perfectly well and does its job. I mean, think of all the railways and the airplanes and the reservation systems that have to work daily and this very minute and this second, and by and large they do work. You hear about the failures now and then, but these we honestly must regard as the exceptions. The banks and the governments, by and large they function – well, if you want to call it functioning. But certainly they don't dysfunction because of bad computers. If anything works then it's still the computers, better than the people.

So, software has had its triumphs. Think also of the space adventures. I mean, it's absolutely phenomenal what has been accomplished there. Missions to Venus and Mars, not even to speak of the Moon, all by automatic remote control, by computer path prediction and everything, under extremely hard conditions. And so I think the only thing we should actually do is applaud all these engineers who has been so clever and intelligent and persistent and meticulous to make it all happen.

As I said, of course the journalists rejoice, because it's the only time they get something to do when there is a big failure; the bigger it is the better. And we all remember that Ariane rocket explosion, the biggest triumph of the journalists there was, of course, that they could say it's one single little error that could have been avoided, an error with the name of "overflow".

What actually bothers me a lot more than these singular big spectacular failures which people like to talk about are the little annoyances. And those we all experience, not occasionally, but practically every day. Maybe not with the software that controls a rocket, but with software that controls our work. I talk of operating systems of personal computers, I talk of text editors, of computer aided design software which is cumbersome to use, which malfunctions now and then and makes us think we are really fools and idiots that can't even handle that simple thing. That's what I call the little annoyances.

It was just two days ago my wife was really getting into a fit, because somehow she touched the wrong button on one of these radios and the stations got erased and she couldn't manage. She was looking for the manual, finally found it, read it ten times and it still wouldn't make sense – the buttons didn't react right. So she gave up. Then, at two o'clock in the morning, in the middle of my sleep, I suddenly hear music coming out of the kitchen. That bloody thing had started automatically. So then the other thing was to find out how to stop that. It's just a marvellous example of an absolutely dismal human interface. It would be a marvellous recipe of how not to do it. It was not only the sequence of pushing buttons that was relevant, but the timing in that pushing was relevant. She was constantly timed-out.

These are the little annoyances, and where do they come from? I think it's mostly from people who design things and don't have experience in designing. As for example in the concept of operating this radio, of pushing the buttons. In software it's the same thing, and then also in structuring the software, how to make it function in a proper way, when it becomes more and more complex, when new functions have to be added. We all know that then everybody is always under high pressure and cannot really lean back as it should be done and think the whole thing over and say, well, this is the best way, let's try it. Oh no it wasn't, let's try another way. You don't have the time for all that. So either the design is right from the beginning – which it never is – and therefore you get concoctions, with the consequence of the cancerous growth of software, and with it, of course, the number of errors. Because a designer usually inherits a large system from other designers, he has not the time nor the patience to study it in depth. Perhaps it wouldn't even be possible because it's not systematic enough, and then he has to add a few things, and so for the n+1-st person it gets just a bit worse. But you can't afford to drop the software and so it's getting bigger and bigger with more and more errors. Now fortunately – I call it unfortunately – we have this fantastic new hardware which comes out with double the speed every 18 months, you know. And that helps us to cover up all these inefficiencies and minor and major flaws. Minor flaws each in itself, major flaws in their sum.

Have you ever thought of taking your software that you are running today and load it on a computer of ten years ago? Of course you have thrown that one away – that's too bad. It might have been better than the ones nowadays, more stable. But you would find that today's software would be utterly useless. You couldn't use one single piece of it, it would be absolutely dreadful, abysmally slow. And this is because software has been getting slower and you didn't notice this fact because hardware was getting faster. Unfortunately there is

> Reiser's Law:
>
> Software is getting slower more quickly
> than hardware is getting faster

Did you get that? It's true ... well, it's largely true, not always, of course.

Well, now you of course will say: Ha, that's old hat. I mean, we knew that ten years ago, even earlier, and indeed there was something that was called the software crisis. And that "software crisis" is a word that was created roughly thirty years ago. To be exact, in 1968 at the NATO Conference held, I think, in Garmisch-Partenkirchen, where people came together and

for the first time in history, the people from industry confessed that they are in trouble. And some of these large time-sharing projects almost brought big companies to the collapse. It was recognised that software design was not a trivial thing; however, it was not immediately evident how it could be solved. About as inevident as it is now, I'm afraid. And then of course the term "Software Engineering" was coined and it became a panacea. You know, if you don't know what to do, you find a correct word.

> Wo Begriffe fehlen, da stellt ein Wort zur rechten Zeit sich ein.
> Faust

That's an old story. Well, in Software Engineering, the meaning was that for the design of programs we should apply sound engineering principles, which is apparently something that exists in other disciplines, and the solution consists in making programming an engineering discipline. Thirty years later, we know what has been happening in Software Engineering, more or less; there were certain new subjects created such as, well, certainly *tools* became important. I'll talk a bit more about those afterwards. I'm doing *my* programming mostly without many tools. Object orientation, a methodology where 90% of the people that I meet and talk about don't really know what the essence of it is. So, a technical topic but usually very very very nebulous. Then software metrics came up, that's a very nebulous field as far as I'm concerned. One of the more straightforward measures, of course, is the number of lines a programmer produces per day. By that measure *my* most productive days are when I have a negative count. That means when I find places in my programs that I can cross off without any loss. That happens! And that's really an exciting feeling, but I would of course flunk and fail miserably on such a metrics test. I'm sure there are other measures but in my feeling it's a very futile attempt to put a metric on such complex things. Like putting a metric on the economic system of a country or something like that. The gross national product doesn't say much about its structure and how to improve it.

Quality Assurance: we're getting closer to the subject of our conference. Now of course, this is like motherhood, a good word, "quality", everyone wants it. I still have difficulty finding out how to put hard technical criteria on this subject. And then evidently there are large projects, which you can't do alone. Teamwork is required, project management is the unavoidable consequence. As soon as you have at least two people you have to have a third one who manages them.

Now the central point that I want make here and today and to you is that the nature of our problem in software is technical incompetence. You cannot produce good software with all the case tools in the world, with all the object orientation that you are capable of managing, with all the software metrics and quality assurance, and with a hundred teamwork managers, if you don't have competent programmers.

I always have a tendency in such talks to make some exaggerations because I feel that it sticks better in my audience afterwards. I'm running the danger of being misinterpreted, so this caveat is that I'm probably exaggerating. I'm not saying that all these topics are useless. But they are not the core of our problem, and, at least partly, they are buzzwords and used as excuses for avoiding the core problem. And if we look at that we sometimes come closer to a solution if we do not try to avoid what is really at the core. Now Dijkstra of course has a much harsher tongue

than I do. He said about Software Engineering that it is essentially how to program when you can't. That means you find a group of people who can't and then you manage them according to Quality Assurance Software Engineering and then it comes out o.k.

I said that the heart of our profession is – did I say programming? What I really mean, of course, is design, and since we deal with software it comes out partially as programming, but the *design* is the essence of an engineering activity, of a productive, of a creative engineer. And that simply is hard, whether you like it or not. And you can't expect a graduate to emerge at age 20 or 22 and be an experienced and excellent designer. Some have the gift of becoming such, but what you cannot dismiss is, of course, experience. In design you learn a lot by doing it. Now the thing which rather frightens me is the discovery that design is a topic that is by and large not taught in our universities. Particularly not in computer science. "Programming" has become a word that you look down upon. Programming? That may be done in some secondary schools, you know, but certainly not in academia, at the academic level, that's below us. You go and find a professor who does programming – well, you see one before you, but I think that's rather the exception. Its a hard business, it's hard, and as a professor you can earn easier brownie points doing other things.

Now let me dwell a little bit on that topic that design is neglected. The other day – well, it was a few weeks ago to be honest – I got a letter. Unfortunately it is in German, so I'll translate it for you. It is from another former PhD student of mine. He writes: "Only very few universities still value and emphasise good programming methodology. Very few people know" – he means graduates; he is talking about his experience of interviewing computer science graduates – "very few of them understand concepts like modularisation, correct abstraction level, information hiding, things like that."

That is rather alarming, but he says he had interviewed at least 250 people recently and only very few understood these concepts. I'll come back to it in a minute, but let me go on a little bit. What is still worse: In some departments Microsoft started projects that were supposed to rest on very sound programming techniques. Some of the brightest architects of the company were put on these projects in order to make it right from the start. However, they failed dismally. They knew everything about design, about modularisation, interface specification, and so on. However, they made the big mistake not to concentrate on the essentials, and therefore they concocted gigantic buildings of thoughts which nobody was able to build anymore.

One of the most famous examples, the so-called Cairo Project, wasted several hundred man-years before the project was finally cancelled. Such disasters have caused the general attitude "Forget design, forget methodology, just program ahead, finish the product, and put it on the market." It is better to spend 70 % of your time debugging a product, rather than spending 70 % in designing a product that you'll never finish. Well, what can you say? This is information from an insider, how things go on in his company, you are under pressure, good successful design methods are supposed to be to costly, in time in particular, and then you have large groups, and you have to meet deadlines, and so on and so on. And you release products with thousands of known bugs. I know that. And we dumb customers accept it; worse, we pay for it. And if there is anything I have to say to you as managers or customers: Don't put up with that any longer. Send that stuff back before you pay for it.

I asked that fellow what his solution was, if he can't find any people who know these concepts that he considers essential, and he said: well, just recently there has been some glimmer of hope, you know. We can now employ people from the Eastern European countries

and from Russia. That's how, very quickly, large colonies of Russians grow in these hubs of Software Engineering developments. He said, well, they still are brought up with designing methods, they're not under time pressure, they have no computers and they have no salaries, but they have time, so they spend time exercising projects that ultimately are not realised. But they know the methods, they know the concepts for doing it, and in a rapidly expanding speed people from over there, were they have lousy salaries, are employed in the US. Indeed, let me tell you, it was not more than two or three weeks later that I met one of these Russian immigrants in the Bay area and I related this story to him, and he said, oh yes, you are absolutely right, we have also started a group here in, I think it was STI, and you know what it is? Our people were still brought up with Pascal and Modula.

Well, that was of course nice for me to hear, but it was also revealing, you know. That what America gets for their craze of programming with C, and forgetting essentially everything we have learned about programming and designing software systems in the last 30 years.

Now, by these last 30 years I mean concepts like abstraction, building clean abstraction layers, defining clearly specified interfaces between modules and between the various abstraction layers. Of modularisation, abstract data types, and so on and so on. Now, I have just written down three lines here as a reminder.

Structured programming - Pascal (1979)
Modular programming - Modula (1980)
Extensible programming - Oberon (1990)

Adequate tools would always have been available

We have, of course, been very active in this development of programming methodology. Very roughly we can characterise it with the three eras of Structured programming, Modular programming, Extensible programming. In every one of these eras we had our programming language ready which supported these notions. I mean, when we design we build abstract buildings, and we have to express them somehow; we don't do it with concrete nor with steel, we do it with a notation. And these are the notations for it that cater to these concepts that were evolving in these eras. Structured programming offers the notion that you build something as a hierarchical structure of data definitions, of statement definitions. Modular programming, then, adds that you separate these abstractions by hiding information that is needed locally only – David Parnas, information hiding – and extensible programming – Oberon – which brings in the possibility to build new buildings, new data types, with new operators on top of old ones, remaining compatible with the old ones. That's one aspect of object oriented programming that is much more explicit here.

So all I want to say is, adequate tools would actually all have been available at the right time, and there were times when it also seemed these ideas would be picked up. I mean, they were certainly picked up with Pascal.They were more or less picked up with Modula, but now in the last ten years the trend has rather reversed, and that is not only surprising, it is rather sad.

The reversed trend signals a movement away from structure and of discipline, as this letter said "Forget structure, program ahead, and meet the difficulties when they arise, then see what you can do about it, make a few patches locally and you will see, somehow you can get on."

This, of course, is also strongly supported by this so-called language C which is by now almost 30 years old, too, like Pascal, and its so called descendant C++.

Have you ever heard of C+-? Well, I have; it's also pronounced "C more or less". I don't want make fun of it, but the big point is that with C you do abdicate the one big advantage that you have, namely that you put redundancy in your program, you can associate types with everything you handle in your program, and let your compiler check this redundancy. That's a very big help, I cannot often enough repeat that. And you can rely on it that these checks are sound and thorough. Now you can't do that in either C or C++; there's no support of the compiler possible in type consistency checking. Instead, of course, if you don't use the primary tool you use in programming – the compiler –, you have to use others, which simply come in later, and they are called "tools for debugging". And that is, of course, a thing one can't have enough of today. I mean, a so called software engineer is a guy who knows the maximum number of debugging tools and knows how to apply them well.

Well, there is no point in dwelling on this, but certainly the tendency is of doing a quick design, and then with all the things you have available, with all the work stations and the software to fight that devil which you have planted yourself, instead of trying to do a clean design where debugging would take little time afterwards only. This is not just something I state now, and I have stated that it's the wrong approach for the last 15 or 20 years, but I just looked around a little bit among the papers which were on my desk and, lo and behold, I have two examples here which were right at my fingertips.

Here is an article that appeared in this month's Software Practice and Experience issue, and its title – in a highly respected scientific journal – is "Low-Cost Concurrent Checking of Pointer and Array Accesses in C Programs." Now how the hell would you have to check pointer and array accesses? I thought about that, and of course we have been doing that in Pascal, Modula, and Oberon all the time, the compiler includes array bound checks. Now you can't do that reliably in C; because of the language's definition you can manipulate accesses and all kinds of things. So the article – I didn't read it but I glanced through it, and it's a very very sophisticated elaborate highly scientific method where you have concurrent processes that work in the background and of course produce garbage, which is by a third process collected again, and somehow, very ingeniously these processes at the side know what are array and what are pointer accesses and check whether they are legal. So you build a gigantic, sophisticated mechanism for something you could have had almost for free if you had done it right at the beginning. But that is how computer science nowadays produces famous papers.

The second point here is from a final report from one of the projects from the "Schwerpunktprogramm Informatik" of the Swiss National Science Foundation. On the first page of that proposal or report it summarises what the goals of the project were.

1 How can easy programming be achieved, in particular for non-experts?

2 How can software methods be realised which allow hiding the difficult parts of programming?

Thank you for laughing, but you are tax payers, you pay the Nationalfonds with your tax francs to support such projects. Now I'm standing here as an earlier member of that expert committee, and I thought it was nonsense. It went through anyway. So you see, instead of tackling the problems at the right place, it has become the custom of doing it wrong where it should have been done right, and then building large mountains of gadgets which nobody will understand anymore, but which everybody will be glad to pay because it supports research! I'm

a critic of these activities, and I think somewhere someplace somebody ought to put a stop to this nonsense.

Now, you of course ask me: who? and where? What can we do? I don't know. I make a big confession: I don't know. I have tried to find out. I have made contributions. We have built programming languages and supporting systems, which really lead to more meticulous programming, to more careful approaches, and we have had success with it. We have even gone to the trouble to build entire operating systems with file systems, viewer systems, really entire operating system. We showed that it can be done by few people rather rapidly, we showed that these systems are very efficient, we showed that they can get by with a few hundred kilobytes where modern systems require megabytes and dozens of megabytes. We showed that you can do much better if you want to. Somehow the trend is away from that. People are gladly paying the money for 24 megabytes if the software is so miserable that it is required. Is that because managers decide what to acquire without asking the programmers? I wonder.

Anyway I'm ending on a relatively somber note. I'm sorry for that, but I'm not going to laugh when I don't feel like it. Software Quality Engineering is perhaps a very catchy conference title. But it's not a topic in my experience that anyone is really seriously interested in. Because if they were seriously interested, they would stop buying that miserable software that it is around. Computer Science, is it a science? That's, of course, a question that has been with us for 30 years, and Al Perlis once said: Well, don't worry about that; computer science is simply what people do with it.

So my last line is: Is it a failure? Have we failed? I mean, if we have spent 30 years in trying to develop sound programming methods and software engineering, and we must say, basically we are worse off than we were 20 years ago, what else can it be but a failure?

I hope, Jochen, this is a good start for a lively discussion of the panel.