A Brief History of Software Engineering

Niklaus Wirth (Wirth@inf.ethz.ch) (25.2.2008)

Abstract

We present a personal perspective of the Art of Programming. We start with its state around 1960 and follow its development to the present day. The term *Software Engineering* became known after a conference in 1968, when the difficulties and pitfalls of designing complex systems were frankly discussed. A search for solutions began. It concentrated on better methodologies and tools. The most prominent were programming languages reflecting the procedural, modular, and then object-oriented styles. Software engineering is intimately tied to their emergence and improvement. Also of significance were efforts of systematizing, even automating program documentation and testing. Ultimately, analytic verification and correctness proofs were supposed to replace testing.

More recently, the rapid growth of computing power made it possible to apply computing to ever more complicated tasks. This trend dramatically increased the demands on software engineers. Programs and systems became complex and almost impossible to fully understand. The sinking cost and the abundance of computing resources inevitably reduced the care for good design. Quality seemed extravagant, a loser in the race for profit. But we should be concerned about the resulting deterioration in quality. Our limitations are no longer given by slow hardware, but by our own intellectual capability. From experience we know that most programs could be significantly improved, made more reliable, economical and comfortable to use.

The 1960s and the Origin of Software Engineering

It is unfortunate that people dealing with computers often have little interest in the history of their subject. As a result, many concepts and ideas are propagated and advertised as being new, which existed decades ago, perhaps under a different terminology. I believe it worth while to occasionally spend some time to consider the past and to investigate how terms and concepts originated.

I consider the late 1950s as an essential period of the era of computing. Large computers became available to research institutions and universities. Their presence was noticed mainly in engineering and natural sciences, but also in business they soon became indispensable. The time when they were accessible only to a few insiders in laboratories, when they broke down every time one wanted to use them, belonged to the past. Their emergence from the closed laboratory of electrical engineers into the public domain meant that their use, in particular their programming, became an activity of many. A new profession was born; but the large computers themselves became hidden within closely guarded cellars. Programmers brought their programs to the counter, where a dispatcher would pick them up, queue them, and where the results could be fetched hours or days later. There was no interactivity between man and computer.

Programming was known to be a sophisticated task requiring devotion and scrutiny, and a love for obscure codes and tricks. In order to facilitate this coding, formal notations were created. We now call them *programming languages*. The primary idea was to replace sequences of special instruction code by mathematical formulas. The first widely known language, Fortran, was issued by IBM (Backus, 1957), soon followed by Algol (1958) and its official successor in 1960. As computers were then used for computing rather than storing and communicating, these languages catered mainly to numerical mathematics. In 1962 the language Cobol was issued by the US Department of Defense for business applications.

But as computing capacity grew, so did the demands on programs and on programmers: Tasks became more and more intricate. It was slowly recognized that programming was a difficult task, and that mastering complex problems was non-trivial, even when – or because – computers were so powerful. Salvation was sought in "better" programming languages, in more "tools", even in automation. A better language should be useful in a wider area of application, be more like a "natural" language, offer more facilities. PL/1 was designed to unify scientific and commercial worlds. It was advertised under the slogan "Everybody can program thanks to PL/1". Programming languages and their compilers became a principal cornerstone of computing science. But they neither fitted into mathematics nor electronics, the two traditional sectors where computers were used. A new discipline emerged, called *Computer Science* in America, *Informatics* in Europe.

In 1963 the first time-sharing system appeared (MIT, Stanford, *McCarthy*, DEC PDP-1). It brought back the missing interactivity. Computer manufacturers picked the idea up and announced time-sharing systems for their large mainframes (IBM 360/67, GE 645). It turned out that the transition from batch processing systems to time-sharing systems, or in fact their merger, was vastly more difficult then anticipated. Systems were announced and could not be delivered on time. The problems were too complex. Research was to be conducted "on the job". The new, hot topics were multiprocessing and concurrent programming. The difficulties brought big companies to the brink of collapse. In 1968 a conference sponsored by NATO was dedicated to the topic (1968 at Garmisch-Partenkirchen, Germany) [1]. Although critical comments had occasionally been voiced earlier [2, 3], it was not before that conference that the difficulties were openly discussed and confessed with unusual frankness, and the terms *software engineering* and *software crisis* were coined.

Programming as a Discipline

In the academic world it was mainly *E.W.Dijkstra* and *C.A.R.Hoare*, who recognized the problems and offered new ideas. In 1965 Dijkstra wrote his famous *Notes on Structured Programming* [4] and declared programming as a discipline in contrast to a craft. Also in 1965 Hoare published an important paper about data structuring [5]. These ideas had a profound influence on new programming languages, in particular *Pascal* [6]. Languages are the vehicles in which these ideas were to be expressed. Structured programming became supported by a *structured programming language*.

Furthermore, in 1966 Dijkstra wrote a seminal paper about harmoniously cooperating processes [7], postulating a discipline based on semaphores as primitives for

synchronization of concurrent processes. Hoare followed in 1966 with his *Communicating Sequential Processes* (CSP) [8], realizing that in the future programmers would have to cope with the difficulties of concurrent processes. Obviously, this would make a structured and disciplined methodology even more compelling.

Of course, all this did not change the situation, nor dispel all difficulties over night. Industry could change neither policies nor tools rapidly. Nevertheless, intensive training courses on structured programming were organized, notably by H. D. Mills in IBM. None less than the US Department of Defense realized that problems were urgent and growing. It started a project that ultimately led to the programming language *Ada*, a highly structured language suitable for a wide variety of applications. Software development within the DoD would then be based exclusively on Ada [9].

UNIX and C

Yet, another trend started to pervade the programming arena, notably academia, pointing in the opposite direction. It was spawned by the spread of the UNIX operating system, contrasting to MIT's MULTICS and used on the quickly emerging minicomputers. UNIX was a highly welcome relief from the large operating systems established on mainframe computers. In its tow UNIX carried the language *C* [8], which had been explicitly designed to support the development of UNIX. Evidently, it was therefore at least attractive, if not even mandatory to use C for the development of applications running under UNIX, which acted like a Trojan horse for C.

From the point of view of software engineering, the rapid spread of C represented a *great leap backward*. It revealed that the community at large had hardly grasped the true meaning of the term "high-level language" which became an ill-understood buzzword. What, if anything, was to be "high-level"? As this issue lies at the core of software engineering, we need to elaborate.

Abstraction

Computer systems are machines of large complexity. This complexity can be mastered intellectually by one tool only: Abstraction. A language represents an abstract computer whose objects and constructs lie closer (higher) to the problem to be represented than to the concrete machine. For example, in a high-level language we deal with numbers, indexed arrays, data types, conditional and repetitive statements, rather than with bits and bytes, addressed words, jumps and condition codes. However, these abstractions are beneficial only, if they are consistently and completely defined in terms of their own properties. If this is not so, if the abstractions can be understood only in terms of the facilities of an underlying computer, then the benefits are marginal, almost given away. If debugging a program - undoubtedly the most pervasive activity in software engineering – requires a "hexadecimal dump", then the language is hardly worth the trouble.

The widespread run on C undercut the attempt to raise the level of software engineering, because C offers abstractions which it does not in fact support: Arrays remain without index checking, data types without consistency check, pointers are merely addresses where addition and subtraction are applicable. One might have classified C as being somewhere between misleading and even dangerous. But on the contrary, people at large,

particularly in academia, found it intriguing and “better than assembly code”, because it featured some syntax.

The trouble was that its rules could easily be broken, exactly what many programmers cherished. It was possible to manage access to all of a computer’s idiosyncracies, to items that a high-level language would properly hide. C provided freedom, where high-level languages were considered as straight-jackets enforcing unwanted discipline. It was an invitation to use tricks which had been necessary to achieve efficiency in the early days of computers, but now were pitfalls that made large systems error-prone and costly to debug and “maintain”.

Languages appearing around 1985 (such as Ada and C++), tried to remedy this defect and to cover a much wider variety of foreseeable applications. As a consequence they became large and their descriptions voluminous. Compilers and support tools became bulky and complex. It turned out that instead of solving problems, they added problems. As Dijkstra said: They belonged to the problem set rather than the solution set.

Progress in software engineering seemed to stagnate. The difficulties grew faster than new tools could restrain them. However, at least pseudo-tools like software metrics had revealed themselves as being of no help, and software engineers were no longer judged by the number of lines of code produced per hour.

The Advent of the Micro-Computer

The propagation of software engineering and Pascal notably did not occur in industry, but on other fronts: *In schools and in the homes.* In 1975 micro-computers appeared on the market (Commodore, Tandy, Apple, much later IBM). They were based on single-chip processors (Intel 8080, Motorola 6800, Rockwell 6502) with 8-bit data busses, 32KB of memory or less, and clock frequencies less than 1 MHz. They made computers affordable to individuals in contrast to large organizations like companies and universities. But they were toys rather than useful computing engines. Their breakthrough came when it was shown that languages could be used also with microcomputers. The group of *Ken Bowles* at UC San Diego built a text editor, a file system, and a debugger around the portable Pascal compiler (P-code) developed at ETH, and they distributed it for $50. So did the *Borland* company with its version of compiler. This was at a time when other compilers were expensive software, and it was nothing less than a turning-point in commercializing software. Suddenly, there was a mass market. Computing went public

Meanwhile, requirements on software systems grew further, and so did the complexity of programs. The craft of programming turned to “hacking. Methods were sought to systematize, if not construction, then at least program testing and documentation. Although this was helpful, the real problems of hectic programming under time-pressure remained. Dijkstra brought the difficulty to the point by saying: *Testing may show the presence of errors, but it can never prove their absence.* He also sneered: *Software Engineering is programming for those who can’t.*

Programming as a mathematical discipline

Already in 1968 R.W. *Floyd* suggested the idea of *assertions* of states, of truths always valid at given points in the program [10]. It led to Hoare's seminal paper titled "An Axiomatic Basis of Computer Programming", postulating the so-called *Hoare logic* [11]. A few years later Dijkstra deduced from it the calculus of *predicate transformers* [12]. Programming was obtaining a mathematical basis. Programs were no longer just code for controlling computers, but static texts that could be subjected to mathematical reasoning.

Although these developments were recognized at some universities, they passed virtually unnoticed in industry. Indeed, Hoare's logic and Dijkstra's predicate transformers were explained on interesting, but small algorithms such as multiplication of integers, binary search, and greatest common divisor, but industry was plagued by large, even huge systems. It was not obvious at all, whether mathematical theories were ever going to solve real problems, when the analysis of even simple algorithms was demanding enough.

A solution was to lie in a disciplined manner of programming, rather than a rigorous scientific theory. A major contribution to structured programming was made by Parnas in 1972 with the idea of *Information Hiding* [13], and at the same time by *Liskov* with the concept of *Abstract Data Types* [14]. Both embody the idea of breaking large systems up into parts called *modules*, and to clearly define their *interfaces*. If a module *A* uses (imports) a module *B*, then *A* is called a *client* of *B*. The designer of *A* then need not know the details, the functioning of *B*, but only the properties as stated by its *interface.* This principle probably constituted the most important contribution to software engineering, i.e. to the construction of systems by large groups of people. The concept of modularization is greatly enhanced by the technique of *separate compilation* with automatic checking of interface compatibility.

Just as structured programming had been the guiding spirit behind Pascal, modularization was the principal idea behind the language *Modula-2*, the successor of Pascal, published in 1979 [15]. In fact, its motivation came from the language Mesa, an internal development of the Xerox Research Lab in Palo Alto, and itself a descendant of Pascal. The concept of modularization and separate compilation was also adopted by the language *Ada* (1984), which was also based largely on Pascal. Here, modules were called *packages*.

The Era of the Personal Workstation

However, another development influenced the computing field more profoundly than all programming languages. It was the *workstation*, whose first incarnation, the *Alto*, was built, once again, in the Xerox Research Lab in Palo Alto (1975) [16]. In contrast to the mentioned micro-computers, the workstation was powerful enough to allow serious software development, complex computations, and the use of a compiler for an advanced PL. Most important, it pioneered the bit-mapped, high-resolution display and the pointing device called *mouse*, which together brought about a revolutionary change in computer usage. Along with the Alto the concept of *local area network* was introduced, and that of central servers for (laser-) printing, large scale file storage, and e-mail service. It is no exaggeration at all to claim that the modern computing era started in 1975 with the Alto. The Alto caused nothing less than a revolution, and as a

result people to-day have no idea, how computing was done before 1975 without personal, highly interactive workstations. The influence of these developments on software engineering cannot be overestimated.

As the demand of ever more complex software grew persistently, and as the difficulties became more menacing, as some spectacular failures demonstrated that problems were serious, the search for panaceas began. Many cures were offered, sold, and soon forgotten. One of them, however, proved fruitful and survived: *Object-oriented programming (OO)*.

Up to 1980 the commonly accepted model of computing was transforming data from their given state to the result, gradually transforming input into output. In its simplest abstract form this is the *finite state machine*. This view of computing stemmed from the original task of computers: Computing numerical results. However, another model gained ground in the 1960s: It originated from the simulation of complex systems (supermarkets, factories, railways, logistics). Their abstraction consists of actors (processes) that come and go, that pass phases in their lifetime, and that carry a set of private data representing their current state. It proved natural to think of such actors with state as a unity, as an *object*. Some programming languages were designed on the basis of this model, their ancestor being Dahl and Nygaard's Simula in 1965. But they remained confined to the field of simulation of discrete event systems. Only after the emergence of powerful personal computers did the OO-model gain wider acceptance. Now, computing systems would feature windows, icons, menus, buttons, toolbars etc., all easily identifiable as visible objects with individual state and behavior. Languages appeared supporting this model, among them *Smalltalk* (Goldberg and Kay, 1980), *Object-Pascal* (Tesler, 1985), C++ (Stroustrup, 1985), Oberon (Wirth, 1988), Java (Sun, 1995) and C# (Microsoft, 2000). Object-orientation became both a trend and a buzzword. Indeed, choosing the right model for an application is important. Nevertheless, one must not overlook the fact that there exist applications for which OO is not the appropriate model.

Abundance of Computer Power

The period since 1985 up until a few years ago has chiefly been characterized by enormous advances in hardware technology. Today, even tiny computers such as mobile telephones, have a hundred times more power and capacity than those 20 years ago. It is fair so say that semiconductor and disk technologies have recently determined all advances. Who, for example, would have dreamt in 1990 of memory sticks holding several gigabytes of data, of tiny disks with dozens of gigabyte capacity, of processors with clock rates of several gigahertz?

This speedy development has vastly widened the area of computer applications. This happened in particular in connection with communication technology. It is now hard to believe that before 1975 computer- and communication technologies were considered separate fields. Electronics has united them, and has made the Internet pervasive. It features a bandwidth that appears to be unlimited. I am overwhelmed, when I compare this with the first, stand-alone minicomputer that I worked with in 1965, a DEC PDP-1: Clock rate, < 1 MHz, memory of 8K word of 18 bits, drum storage of some 200 KB. It

was time-shared by up to 16 users. It is a miracle that some people insisted in believing that one day computers would become powerful enough to be useful.

In the 1990s, a phenomenon started to spread under the name of *Open Source*. The distrust against huge systems designed in industrial secrecy became manifest. A wide community of programmers decided to build software and to distribute their products for free through the Internet. Although it is difficult to recognize this as a sound business principle – making the idea of patents obsolete – the bandwagon turned out to be rather successful. The notions of quality and responsibility in case of failure seemed irrelevant. Open Source appeared as the welcome alternative to industrial hegemony and abrasive profit, and also against helpless dependence.

It is often difficult in software engineering to distinguish between business strategies and scientific ideas. On the latter ground, Open Source appears to be a last attempt to cover up failure. The writing of complicated code and the nasty decryption by others is apparently considered easier or more economical than the careful design and description of clean interfaces of modules. The easy adaptability of modules when available in source form is also a poor argument. In whose interest would a wild growth of varieties of variants ever be? Not in that of high-quality engineering and professionalism.

Wasteful Software

Whereas the incredible increase in the power of hardware was very beneficial for a wide spectrum of applications (we think of administration, banks, railways, airlines, guidance systems, engineering, science), the same cannot be claimed for software engineering. Surely, software engineering has profited too from the many sophisticated development tools. But the quality of its products hardly reflects signs of great progress. No wonder: After all, the increase of power was itself the reason for the terrifying growth of complexity. Whatever progress was made in software methodology was quickly compensated by higher complexity of the tasks. This is reflected by Reiser's "law": "Software is getting slower faster than hardware is getting faster". Indeed, new problems have been tackled that are so difficult that engineers often have to be admired more for their optimism and courage than for their success.

What happened in software engineering was predictable and inherent in a field of engineering, where the demands rise, work is done under time pressure, and the cost of resources are almost disappearing. The consequence is waste of cheap resources – processor cycles and storage bits – resulting in inefficient code and bulky data. This waste has become ever-present and represents a grave lack of sense for quality. Inefficiency of programs is easily covered up by obtaining faster processors, and poor data design by the use of larger storage devices. But their side effect is a decrease of quality – of reliability, robustness, and ease of use. Good, careful design is time-consuming, costly. But it is still cheaper than unreliable, difficult software, when the cost of "maintenance" is not factored in. The trend is disquieting, and so is the complacency of customers.

Personal Reflections and Conclusions

What can we do to release this log-jam? There is little point in reading history, unless we are willing to learn from it. Therefore, I dare to reflect on the past and will try to draw some conclusions. A primary effort must be education toward a sense of quality. Programmers must become engaged *crusaders against home-made complexity*. The cancerous growth of complexity is not a thing to be admired; it must be fought wherever possible [17]. Programmers must be given time and respect for work of high quality. This is crucial and ultimately more effective than better tools and rules. Let us embark on a global effort to prevent software from becoming known as *softwaste*!

Recently I have become acquainted with a few projects where large, commercial operating systems were discarded in favor of the Oberon System, whose principal design objective had been perspicuity and concentration on the essentials [18]. The project leaders, being obliged to deliver *reliable, economical software*, had recognized that they were unable to do so as long as they – even most carefully – built their work on top of complex base software – a platform - that was neither fully described nor dependable. We know that any chain is only as strong as its weakest link. This holds also for module hierarchies. Systems can be designed with utmost care and professionalism, yet they remain error-prone if built on a complex and unreliable platform.

The crazy drive for more complexity – euphemistically called sophistication – long ago had also seized the most essential tool of the software engineer. Modern languages like Java and C# may well be superior to old ones like Fortran, PL/I, and C, but they are far from perfect, and they could be much better. Their manuals of several hundred pages are an unmistakable symptom of their inadequacy. Engineers in industry, however, are rarely free from constraints. They supposedly they must be compatible with the rest of the world, and to deviate from established standards might be fatal.

But this cannot be said about academia. It is therefore a sad fact that it has remained inactive and complacent. Not only has research in languages and design methodology lost its glamour and attractivity, but worse, the tools common in industry have quietly been adopted without debate and criticism. Current languages may be inevitable in industry, but for teaching, for an orderly, structured, systematic, well-founded introdution they are entirely mistaken and obsolete.

This is notably in accord with the trends of the 21st century: We teach, learn, and perform only what is immediately profitable, what is requested by students. In plain words: We focus on what sells. Universities have traditionally been exempt from this commercial run. They were places where people were expected to ponder about what matters in the long run. They were spiritual and intellectual leaders, showing the path into the future. In our field of computing, I am afraid, they have simply become docile followers. They appear to have succumbed to the trendy yearning for continual innovation, and to have lost sight of the need for careful craftsmanship.

If we can learn anything from the past, it is that computer science is in essence a methodological subject. It is supposed to develop (teachable) knowledge and techniques that are generally beneficial in a wide variety of applications. This does not mean that computer science should drift into all these diverse applications and lose its identity. Software engineering would be the primary beneficiary of a professional

education in disciplined programming. Among its tools languages figure in the forefront. A language with appropriate constructs and structure, resting on clean abstractions, is instrumental in building artefacts, and mandatory in education. Home-made, artificial complexity has no place in them. And finally: It must be a pleasure to work with them, because they enable us to create artefacts that we can show and be proud of.

References

1. P. Naur and B. Randell, Eds. *Software Engineering*. Report on a Conference held in Garmisch, Oct. 1968, sponsored by NATO
2. E.W. Dijkstra. Some critical comments on advanced programming. *Proc. IFIP Congress*, Munich, Aug. 1962.
3. R.S. Barton. A critical review of the state of the programming art. *Proc. Spring Joint Computer Conference*, 1963, pp 169 – 177.
4. E. W. Dijkstra. Notes on structured programming. In *Structured Programming*. O.-J. Dahl, E. W. Dijkstra and C.A.R. Hoare, Acad. Press, 1972.
5. C.A.R. Hoare. Notes on data structuring. In *Structured Programming*. O.-J. Dahl, E. W. Dijkstra and C.A.R. Hoare, Acad. Press, 1972.
6. N. With. The Programming Language Pascal. *Acta Informatica 1*, (1971) 35 - 63
7. E. W. Dijkstra, Cooperating sequential processes. Sept. 1965. Reprinted in *Programming Languages*, F. Genuys, Ed., Acad. Press, New York, 1968, 43-112.
8. C.A.R. Hoare. Communicating sequential processes *Comm. ACM, 21*, 8 (August 1978) pp. 666 - 677.
9. J.G.P. Barnes. An Overview of Ada. *Software - Practice and Experience, 10* (1980) 851 – 887.
10. R.W. Floyd. Assigning meanings to programs, *Proc. of Symp. in Applied Mathematics., 19* (1967), pp. 19-32
11. C.A.R. Hoare**.** An axiomatic basis for computer. *Comm. ACM, 12*, 10 (October 1969), pp. 576 - 580
12. E. W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Comm. ACM, 18*, 8, pp. 453–457, August 1975.
13. D. L. Parnas. Abstract types defined as classes of variables. *ACM Sigplan Notices II 2*, 149 - 154 (1976)
14. B. Liskov and S. Zilles**.** Programming with abstract data types**.** *Proc. ACM SIGPLAN symposium*, Santa Monica, 1974, pp. 50-59.
15. N. Wirth. *Programming in Modula-2.* Springer, 1974. ISBN 0-387-50150-9.
16. C.P. Thacker et al. Alto: A personal computer. Xerox PARC, Tech. Rep CSL-79-11
17. N. Wirth. A plea for lean software. IEEE *Computer*, Feb. 1995, pp. 64-68.

18. M. Franz. Oberon: The overlooked Jewel. In L. Boszormenyi, J. Gutknecht, G. Pomberger. *The School of Niklaus Wirth*. ISBN 1-55860-723-4 and 3-932588-85-1.