Software Engineering - Retrospect and Prospect

Harlan D. Mills
University of Florida
and
Information Systems Institute

Abstract

Software engineering is less than a human generation old, but its products are already critical in business and government operations on a broad scale. As a result, software engineering will emerge from a buzzword, like sanitary engineering, to a bona fide university discipline, like civil or electrical engineering. With better discipline, human performance in software engineering can be expected to improve in dramatic ways, much like human capabilities improved in long division in place notation compared with heuristic division in roman numerals. A key method in software engineering will be the design of software systems by stepwise refinement and verification using ideas of Dijkstra and Parnas.

Early Software Engineering History

Software engineering is a term in human use less than a human generation old. An early reference appeared in a NATO conference in 1968 [Buxton 1969]. The term software engineering was first voiced more as a hope than a reality. The 60's had seen many difficulties and disasters in attempts to produce significant software systems. The difficulties of the IBM OS/360 development were well known. As the leader of this development, Frederick Brooks, later recorded [Brooks 1975], OS/360 eventually survived as one of the most complex logical artifacts constructed by humans in its time. But other software developments, many involving hundreds of human years of effort, never saw the light of day as usable systems. These included attempted airline reservation systems, command and control systems, library management systems, among others.

Except where unavoidable, these software disasters were little admitted or used for lessons to be learned. Peoples career's were at stake, not only for the people leading such disasters, but for people sponsoring them, as well. For example, in one case, after a 50% overrun and a year behind, one multi-million dollars government project of the sixties was cancelled, in spite of the contractor's promise to complete in another year with just another 50%. One might have expected some lessons learned by an airing of the difficulties on the project. Instead, within a week, a government executive was explaining to congress that, while it was indeed cancelled, the project had been just an experiment, and the agency put the same group to work on a new multi-million dollar contract that turned out no better. The sponsor was as embarrassed as the contractor.

With these kinds of problems, it is little wonder that software engineering would be adopted as a buzzword. Compared with total chaos, a three day short course on software engineering offered some help. A whole new genre of practical consultants and trainers arose. It was a new human endeavour (at this age in civil engineering the right triangle was yet to be discovered) so there was a good deal of heuristic advice about developing software systems, but not many logical principles.

Early Professional Efforts

The mid 70's saw more organized professional efforts, with the beginning of the series of international conferences on software engineering, journals such as the IEEE Transactions on Software Engineering, and a rash of professional society tutorial offerings in one day courses and texts on many topics in software engineering. These efforts represented a start in moving beyond the
From the very beginning, computer scientists and engineers have brought deep ideas into software engineering from other fields and developed new ones. For example, John McCarthy discussed the mathematical correctness of software early [McCarthy 1962]. Edsger Dijkstra used the reduction in correctness proof sizes as the initial argument for structured programming [Dijkstra 1969]. David Parnas showed how to decompose software systems into precise logical modules with information hiding [Parnas 1972]. But these ideas are beyond the three day short course or one day tutorial.

Unlike the situation in civil or electrical engineering, which requires a four year university curriculum, people can declare themselves to be software or sanitary engineers after a three day short course with no examinations. However, businesses and governments compete in their fields more and more on the basis of their software, but not on their plumbing. The business news frequently reports late software projects and business executives are often quoted to the effect that software development is an art, not a science. But the businesses that discover first how to conduct software development under university level engineering discipline will be the ones that survive. So the high stakes involved will ensure that software engineering will emerge as a university curriculum with sound mathematical foundations and engineering discipline.

The mathematical foundations are made practical by the very nature of software in controlling computers, whose behaviour is entirely known by construction. In contrast, say in political science, mathematical foundations are much more tenuous - while some mathematics might be useful, a good deal more than mathematics is needed at the foundations. And what software should be for good human use requires more than mathematics as well. The engineering discipline is vital because of the complexity gap between what computers do, instruction by instruction, and what software systems do, transaction by transaction. Engineering discipline provides orderly process for large scale human efforts in building software.

For the large gap between computer instructions and software system transactions, this first generation of software developers seemed to have had little to go on. The very idea of system seems to many to be to think of everything at once, and to forget nothing. That is an error prone assignment, even for one person in a relatively simple system, let alone for a group of hundreds in a complex system. Although it seemed possible for intelligent, well motivated people to simply write everything down in some structured form, late and failed projects were the rule.

So another idea was to identify system structures in block diagrams, as in the data flow diagrams [DeMarco 1979], SADT or IDEF [Ross 1977]. Edward Yourdon claims "90% of the world-wide professional data processing community is at least superficially familiar with the basic concepts of structured analysis, design, and programming", but that "only 10% of the dp organizations in North America practice structured techniques in a disciplined fashion" [Yourdon 1986]. He gives three reasons, first, the need for "dozens, if not hundreds, of dataflow diagrams" in a sizable project and the "overwhelming tedium of drawing and redrawing" such diagrams, second "Inability to apply classical structured analysis to complex, real-time systems", and third, "People lured away from structured analysis by the promises of prototyping tools and fourth generation languages" [Yourdon 1986].

However, there is a deeper reason for the lack of effectiveness of block diagrams and structured analysis in formal disciplines. Block diagrams summarize out the exact effects of computers and software in executing individual transactions. Computers do not deal with the generic data flows of block diagrams, but only with specific data items, one transaction at a time. Block diagrams are useful in organizing general conceptions for communications with sponsors and users. But they irreversibly summarize the effects of various transactions into data flows. Yet it is still the individual transactions that must eventually be designed and programmed for computer execution. While block diagrams allow systems to be refined step by step through a hierarchy of data flows, the verifications of the transactions that make up these refinements must wait the program implementation some time later in the development.
This separation of refinement and verification in outlining system data flows for later implementation of the transactions introduces an inherent source of errors in software development. When transactions are only implicitly indicated in block diagrams, not precisely defined, exactly what to program is ambiguous. A programmer may interpret the transactions required for a block diagram differently than intended by the designer, because different program implementations can have identical block diagrams.

It turns out that Dijkstra showed how to keep refinement and verification together in constructing programs [Dijkstra 1969], and Parnas identified a usage hierarchy among well specified modules that can be used keep refinement and verification together in system design [Parnas 1974].

Our interest in stepwise refinement and verification, and the criticism of block diagrams as a basis for software design is not intended to negate the clear value of block diagrams in general descriptions of software system at executive levels, nor even for predesign analyses of possible system approaches to a new problem. One such use, advocated by DeMarco, is to model a current system and its operations before setting out to replace it [DeMarco 1979].

Stepwise Refinement and Verification in Arithmetic

There is a simple precedent in mathematics in dealing with the separation of refinement and verification, leading to one of mathematics greatest achievements, namely place notation and long division. There is a lesson in it for software engineering.

Before place notation and long division, say in roman numerals, numbers had to be treated as wholes, and very creative persons were needed for division problems. The verification of results is difficult and tedious, but still possible by multiplying the candidate quotient by the divisor and adding the candidate remainder to compare with the dividend. But this verification takes considerable knowledge and discipline just because whole numbers, rather than digits, are involved. It seems likely that roman scribes knew such methods as trade secrets.

But with long division in place notation, numbers could be treated at their digit level and the refinement and verification kept together. It took the best and brightest of mankind thousands of years to discover the place notation that made long division possible. The use of a new digit zero for nothing was a critical discovery. So breaking division into the small steps of long division took fundamental mathematical discoveries to make it possible.

Long division in place notation is a stepwise refinement and verification process that produces a quotient and remainder from a dividend and divisor. Each major step produces the next digit of the quotient by a creative estimate of its value followed by an immediate verification of its correctness: if the verification fails, a new estimate is provided and verified. The minor steps consist of digit by digit operations of arithmetic with no further invention beyond estimating this next digit of the quotient. A skilled school child can solve problems beyond the capability of Euclid and Archimedes in earlier days.

Had society tried to do as much arithmetic with roman numerals as software development and maintenance in the past 30 years, the consultants of the day would be involved in teaching people how to be better guessers in long division and square roots, and even trade secrets just in adding long columns of numbers. But one can be sure that the conventional wisdom of these experts would be very similar to that in the software scene today, namely that arithmetic is an error prone activity that takes good intuitive skills and is very difficult to manage to schedules and budgets. For example, the great Inventory after the Norman conquest was recorded in roman numerals, hamlet by hamlet, but never added up because of such wisdom.

But in retrospect, it is easy to see that the heart of the problem was the separation of refinement from verification as people tried to solve their problems. The errors in doing arithmetic in roman numerals were committed by people. But we know now that such errors were largely due to the methods they used rather than to human fallibility. People are certainly fallible, but at a very low level in long division compared to division in roman numerals.
Stepwise Refinement and Verification in Structured Programming

There is good evidence today that more than 90% of the errors made in software development are due to the immature methods people use rather than to inherent human fallibility. Of course, errors are always made by people, just as those in doing arithmetic in roman numerals. And without knowing about place notation and long division, there is no way to imagine arithmetic in roman numerals as other than an error prone activity.

There is also evidence today that there is a ten to one difference in productivity in both individuals and organizations in software development [DeMarco 1987]. But that difference is also due to immature methods rather than to inherent differences in people. For example, there would be a much wider difference in people's productivity in doing division in roman numerals than in doing long division. In roman numerals, the bulk of the time is spent in heuristic guesswork, but in long division, most of the time goes into writing digits down in the right places.

As already noted, in his first paper on structured programming, Dijkstra showed how to bring stepwise refinement and verification together [Dijkstra 1969]. It was to conceive a program under development as a "string of pearls", each pearl based on a virtual instruction set. The upper pearls are small programs with very powerful instructions, the lower pearls are larger programs with less powerful instructions, and the lowest pearl uses the instructions of the programming language that is available. The refinement of one pearl from its upper predecessor can be verified immediately. Are the virtual instructions being replaced by program parts at each stepwise refinement correctly implemented?

Structured programming is a stepwise refinement and verification process in contrast to previous methods of flowcharting and coding programs with little discipline over their control structures. However structured programming is very young in human understanding, compared with long division, so even today most people writing structured programs do so by heuristic invention, assembling ifthenelses, whiledos, dountils, etc. into structured programs to meet the objectives of a specification. There is a lot of advice in the trade literature and commercial short courses on the heuristics of how to assemble such programs.

But structured programming by stepwise refinement and verification [Linger 1979] is a quite different mental process of systematically deriving a structured program from a specification, not inventing a program to meet a specification. The difference is that the specification is always foremost, the program a product of the process. That is, programming becomes an iterated process of specification decomposition and reconnection, in the stepwise refinement of a specification into a program.

For most people writing structured programs by heuristic invention, programming performance is improved a little, but not by much. In assembling programs it is still easy and inevitable to forget exceptional cases, even some nonexceptional ones, and to get into a fix and try mode of programming. As a result, people writing structured programs by heuristic invention still spend more time debugging programs than writing them.

But for people doing structured programming through specification decomposition, programming performance improves dramatically. First of all, debugging practically disappears [Mills 1986]. It is the exception for a program not to be correct from its first execution. Second, productivity is typically increased by a factor of three or more, even though more time is spent in getting specifications into good shape and rewriting decomposed specifications during the structured programming process. But the best news is the reliability of the program in meeting specifications, because specifications have been the focus of the entire process.

The performance possible with structured programming allows a new form of software development under statistical quality control, as described in a case study in [Linger 1988]. In this form, called Cleanroom Software Engineering, programmers use stepwise refinement and verification with no unit or development debugging, sending system level increments down a statistical testing pipeline for quality control and certification. People are still fallible in the verification. But mathematical fallibility is quite different from debugging fallibility, both qualitatively and quantitatively.
First, from first execution on, people are capable of producing software with less than five errors per KLOC compared with more than fifty errors per KLOC typically expected. This reduction of errors by a factor of ten is due to keeping refinement and verification together in small steps, as in long division. The fifty errors per KLOC is no more necessary than the errors people would make in arithmetic in roman numerals. It is the method, not the people, responsible for 90% of the errors. Second, the fewer errors found are more easily fixed, qualitatively like simple blunders, than are the deeper system errors often produced by unit debugging [Mills 1987a].

Data Abstractions and Usage Hierarchies

Just as Dijkstra showed how to deal with programs early, Parnas showed how to deal with systems through modularity [Parnas 1972]. In quite a different deconstruction strategy from that of using block diagrams, Parnas modules, or data abstractions, are described at the transaction level. Instead of summarizing out transactions, Parnas summarizes out internal data storage and processing, leaving only the external behaviour of a module for specification and analysis. Parnas organizes the calling relations among modules in a usage hierarchy rather than a parts hierarchy of the modules themselves. As a result, all the transaction behaviour is maintained among modules in the usage hierarchy. Therefore, at each refinement, verification is immediately possible, just as for structured programming.

For example, a queue abstraction may have entries for start (initialize the queue empty), put (an item at the back), get (the item from the front, if any), query (length of the queue?). Any such queue transaction identifies a specific entry with specific data, if called for, and produces a specific return. In turn, the implementation of the queue data abstraction may call on other abstractions, say a data storage abstraction, with its own set of entries, such as store (at an address), fetch (from an address). And the queue abstraction will itself be called by higher level abstractions, say by an airlines reservation system, or an operating system. Each individual call of one data abstraction in the implementation of another defines a usage hierarchy among the data abstractions of a software system. Each usage in the system will appear as a distinct node in the usage hierarchy. For example, if the storage abstraction appears three times, say as two stores, one fetch, in the implementation of the queue abstraction, there will be three nodes in the usage hierarchy.

The data flow diagram of a usage hierarchy collapses all instances of a single abstraction, such as the queue or storage abstraction, into a single node, and all arcs between two abstractions into a single arc. As a result, the mapping from usage hierarchies to data flow diagrams is many to one. Conversely, the mapping from data flow diagrams to usage hierarchies is one to many. So, while data flow diagrams summarize the uses of abstractions, additional information, such as the patterns of usage, is required to recover the usage hierarchy. In particular, when programs are written on the basis of data flow diagrams, additional information is required to specify what the programs must be.

The usage hierarchy of Parnas provides a place notation for the stepwise refinement and verification of software systems. For example, at the design of the queue abstraction, the refinement will identify not only the data storage abstraction, but also the circumstances of each usage, so that the proper usages can be verified immediately.

Stepwise Refinement and Verification in Box Structured Systems

Box structures of data abstractions provide a finer grained stepwise refinement and verification of systems through their usage hierarchies [Mills 1987, 1988]. Each data abstraction has three descriptions, as a black box, a state box, and a clear box. The black box describes the abstraction externally, as a mathematical function from the stimulus history of previous usages of the abstraction to its next response. The state box provides an intermediate description by use of an internal state and internal black box defined by a mathematical function from stimulus and state histories to the response and next state. The clear box redescribes the state box's internal black box as a sequential or concurrent process in usages of other black boxes.

The black box defines the traces of Parnas [Parnas 1977] and Hoare [Hoare 1985]. The state box encapsulates stimulus history between the internal state and the internal black box. It generalizes the state machine, which requires that all history be encapsulated in the state; in a state box, history can be split between the state and the internal black box. The
clear box identifies the usages of black boxes at the next level of the usage hierarchy, and the exact form of use at the transaction level.

Box structures provide a simple but rigorous framework for system specification and design over the continuum from informal problem domains to formal computer domains. A typical system problem begins in a problem domain of natural language and the context of the system application. But it ends in the computer domain of formal languages, both for the developers and the users. Programming languages are completely formal, as are user languages for the developed system. Even though a user language is deemed to be user friendly, it is still a formal language. So system developers start with requirements in natural language and end with systems in formal languages in the software and user interfaces.

There are many routes from informal requirements to formal systems. In traditional practices, the system design is given in a formal language. Even more advanced, the system specification is given in a formal language preceding design and programming in formal languages as well. While desirable in many ways, formal specification languages may not be practical for a sequential effort from specification to design to code. Developers may not know enough about the problem domain to completely formalize the specifications at the outset, and sponsors and users may not know enough to agree to formal specifications ahead of time. So a spiral approach between the informal problem domain and formal computer domain is usually wise.

The box structures framework of mathematical functions from stimuli histories to responses, expanded to state box functions, then the denotational, functional semantics of sequential or concurrent processes in clear boxes, provides a common basis for both informal and formal development. A set or a function can be described by three lines of mathematical notation or by three hundred lines of English. There is more fallibility in three hundred lines of English, but it is a matter of degree not kind, and group inspections can reduce fallibilities in dramatic ways. There is also room and need for the informal and formal to cohabit in system development.

For example, if the three hundred lines of English are produced first, then reduced to three lines of mathematical notation, the three hundred lines is still a good comment to reference from the three lines.

The CASE Explosion

Computer aided software engineering is not new, but has recently exploded as a commercial activity, much as consulting and training exploded earlier. The early aids were at the back end of the development process, beginning with program assemblers, then compilers. These were historic events, with programming language development high in achievements of computer science in support of software engineering. The development of effective programming environments, in library systems and program analysis came a little later, but still early. Software development management systems for large projects were also important, but much more ad hoc are variable than programming systems and environments.

The current CASE explosion seeks to provide system development aids earlier in the life cycle, in design, back to specifications, even further back to requirements. There is no question that such aids can and will be important. Getting specifications right the first time, rather than after releasing a system for use, can save much effort, schedule, and frustration. CASE also sets standards and conventions for uniform and systematic work by many people in large projects and organizations. There are large factors in productivity possible with less waste motion, less cross purposes in software development.

However, the current CASE explosion is on much less firm ground than earlier efforts, say in programming languages and compilers. Compilers automate a process of translating a procedure design from one formal language to another more detailed formal language. Much is already known as fundamentals in both the syntax and semantics of such languages. But current CASE tools address areas of specification and design where much more ad hoc are variable than programming fundamentals are known. Instead, the background is more methodological, ways of thinking about inventing specifications and designs, namely a subject new to the human race without the centuries of modern mathematics or physics.

The lessons of Roman numerals and place notation are relevant in the CASE.
explosion. We see aids for arithmetic in place notation in hand calculators and computers today. But with a little thought, we see that automatic aids would be possible for addition, subtraction, multiplication in Roman numerals. For example, to add two numbers in Roman numerals, say XXVIII and CXVIII, juxtapose them into XXVIIIICXVIII, then sort the symbols by value, as CXXXVIII, then rewrite any five I's as V, any two V's as X, and so on, to get CXXXVIII as a final result. Subtraction can easily be done in reverse, by rewriting in the opposite direction where needed and deleting symbols from the two operands together. Multiplication can be done by repeated addition, with modern algorithms than require only in 2 of the smallest operand additions in multiples of the largest operand. It seems likely that Roman scribes knew how to speed up multiplications this way as trade secrets.

But there seems no such easy way to do division in Roman numerals, even though clever heuristic tricks can be imagined to help. The lesson here is that even with automatic aids, school children who know long division in place notation would do better with pencil and paper. So while CASE is indeed important, it is also important to apply CASE to sound and effective methodologies.

Software Engineering in the Future

As software engineering emerges from buzzword to the standards of a four year university curriculum, human performance will grow in corresponding ways. In the early days of software, most effort went into development of stand alone systems. Today most effort goes into maintenance of interconnected systems. But most of software methodology and CASE is still directed to the development of stand alone systems. A minority of today's maintenance is devoted to correcting faulty software. With what is possible with human performance, that minority should become miniscule. The majority of today's maintenance is devoted to changing requirements, but is typically hamstring by poor documentation. That must change, too. In other fields of engineering, the need for up to date blueprints is well established. For example, building requirements change, with new walls coming in or out. When walls to be removed, the blueprints for electricity and plumbing need to be updated.

It is not new for human performance to grow in remarkable ways. Seventy years ago, engineers in Detroit could predict that automobiles would go seventy miles an hour, but few, if any, would have predicted that seventy year old grandmothers would be driving them. Twenty years ago it was predicted that computers would be chess champions of the world, but little was predicted about the ability of programmers to program computers. It seems easier to predict what machines can do than what people can do in the future. Right now, it seems to many that software development is an error prone, schedule and budget breaking process. It seems a black art very difficult to manage. But as software engineering becomes a professional reality, it will become more productive and manageable in astonishing ways.

References


[Yourdon 1986] E. Yourdon, Whatever Happened to Structured Analysis?, Datamation, June 1, 1986