Essay on Software Engineering at the Turn of Century

Władysław M. Turski

Institute of Informatics, Warsaw University
Banacha 2, 02-097 Warsaw, Poland
wmt@mimuw.edu.pl

Abstract. Presents a personal view of the development of software engineering and its theoretical foundations, assesses the current state, lists (some) unresolved problems and identifies (some) directions of fruitful research. The Y2K problem is ignored.

0 Disclaimer

When asked to prepare this presentation, I was given a broad description of its desired content: about software engineering, something appropriate for the year 2000, a look backward and few hints on future development, perhaps a little about important research topics.

It could have been a very boring presentation, full of references and technical jargon. Even if I decided to go this way, I am sure I would have missed some references and some technical terms very dear to some of my esteemed audience and/or readers. I do not need more enemies than absolutely necessary, therefore I decided not to include *any* references and limit the use of technical jargon to bare essentials. There is one exception: in several places I quote from the proceedings of the 1968 Garmisch Conference on Software Engineering[1]; it is there that it all began.

There is no widely accepted hierarchy of topics in software engineering, or if it exists, I am left out of the common knowledge. I decided to follow my own preferences (and if you wish to call them predilections, or even prejudices, I will not argue).

There is always the perplexing matter of style. I believe that even without references I could have produced a very learned-looking document, but I decided to follow another pattern. I have chosen the form of an essay: at least at first glance it seems easy to read.

I realise that writing this essay in the autumn of 1999 to be presented in the spring of 2000 I am facing the difficulty of guessing the computer events of the 99/00 changeover and their consequences for our systems/economies/lives.

[1] Software Engineering. Report on a conference sponsored by the NATO Science Committee. Garmisch, Germany, 7th to 11th October 1968. Editors: Peter Naur and Brian Randell. January 1969.

T. Maibaum (Ed.): FASE 2000, LNCS 1783, pp. 1–20, 2000.

I am no seer, I cannot tell the future, but I am sure that in the *long run* the whole Y2K affair shall have beneficial effects for software engineering practice because some very bad systems will have been discontinued (or destroyed), some bad systems will have been mended a little and — most of all — a few more practicing software engineers and their bosses would have realised that proper discipline in programming and documentation is not just for academic birds: it pays, handsomely.

Finally, all opinions expressed in this essay are mine and I accept the whole, undivided responsibility for them.

1 Background

The ancient Greeks were excellent mathematicians. Despite a relatively high level of mathematical education, (at least some) Greek engineering projects clearly exceeded cost and — one presumes — time limits.

To celebrate their victory over the Macedonian king Antigonus I in 306 B.C., proud citizens of the island of Rhodes decided to erect at the entrance to their main harbour a bronze statue of Apollo. The design and project management were entrusted to the famous sculptor and architect, Chares. When he submitted the cost estimates, the Rhodians asked what would it cost to build the statue twice as high. The answer "twice as much" accepted and corresponding budget allocated, Chares started the construction. According to Sextus Empiricus, when funds ran out *long before* the statue was completed, Chares committed suicide. Colossus, one of the Wonders of the Ancient World, was eventually completed (70 cubits high) by others. It was severely damaged by an earthquake in 224 B.C., never restored, and finally (in A.D. 672) sold for scrap to a merchant of Edessa, who carried it off piecemeal on 900 camels. One wonders if the drastic change of project management was responsible for structural heterogeneity of the monument: one part was destroyed barely 80 years after construction (albeit by an earthquake), another stood for nearly a millennium. This question can be answered only by speculation; the glaring error in estimation of the cost of change in specifications is a historic fact.

The Roman legionnaire, who in 212 B.C. killed Archimedes while the latter was engrossed in solving a mathematical problem, gained a dubious kind of immortality being the *only* ancient Roman ever mentioned in the history of mathematics. A very comprehensive textbook on the subject may add that an inscription prohibiting dogs and mathematicians adorned the doors of many a Roman tavern. Mathematics was not among the subjects dear to a Roman mind, yet Romans were excellent engineers.

And so it continued throughout the centuries. The contemporary structural analysis of medieval cathedrals bristles with mathematics which certainly was neither available nor even accessible to the inspired master masons who built them. Yet, very few cathedrals collapsed on their own. The relationship between mathematical and engineering excellence (and, indeed, proficiency) is not as simple and straightforward as the popular view would have it.

Software, or should I say "the software engineering product", is by no means unique in that it is less than completely satisfactory, reliable, trouble-free. In fact, no engineering product ever is. Yes, there were a few actual disasters caused by software, some — regretfully — cost human life. It is pointless and distasteful to run a scoreboard of tragedies, but software engineering surely is not a clear leader. A brief analysis of two catastrophes may throw some light on issues we are to discuss later on.

Owners of the White Star Line raising toasts in April 1912 as their new liner set course for New York were a little precipitate. Getting the Titanic launched and selling the tickets were not really the issue. In retrospect, knowing all we know today, what was the single most important "error"? I claim it was an incomplete specification in which hardly any attention was paid to life-saving features under conditions of the total failure of disaster-prevention features. Apart from considerations of cruising speed, cost-effectiveness and passengers' (somewhat graduated) comfort, designers worked to the best of their ability to make the ship seaworthy and unsinkable. It did not enter their heads that all precautions could fail under the impact of a North Atlantic iceberg, or, few years later, if a U-boot would score a hit. The Titanic's lifeboats did not have sufficient capacity.

The Chernobyl disaster is often quoted as an example of nightmares that the technology could unleash. To be sure, the reactor design in the Chernobyl power station was not the safest known. But what is conveniently overlooked in most analyses and — especially — in dramatic writings intended for popular consumption is that the actual catastrophe was due entirely to a *human operator error*, with control software intentionally switched off. There is absolutely no reason to suspect that had the software operated as usual on that fateful day in April 1986 there would have been an explosion. This realisation opens a very difficult problem, one that may be truly insoluble, to wit: in safety-critical situations, where computer-based *i.e.* software control is installed, should it be possible for a human operator to override it? (Switching off being a rather special instance of overriding.) The Chernobyl disaster points towards a negative answer; most car manufacturers seem to go in the same direction, some aircraft crashes perhaps could have been avoided if the pilot's reactions were not overridden. But the problem is deeper than this; the classical answer "it depends" is more than a little disturbing.

Finally, let us note that at least some disasters clearly emanating from faulty computer calculations can hardly be attributed to *software errors*. Stupidity, for example, needs no software to show its ugly face, even though software is by no means immune to it, as follows from a fairly recent piece of news:

The United States space agency Nasa admitted that the disastrous loss of its $130m Mars Climate Orbiter last month was due to faulty flight calculations based on the simple failure to convert imperial measurements into their metric equivalents. (The Guardian Weekly, October 7 - 13, 1999.)

2 A Bit of History

There is a tendency to consider the evolution of the software field as a kind of more or less systematic progression from individual machine code programs written in binary (or, indeed, wired by cables plugged into a switchboard) to contemporary mammoth systems and networks, heterogeneous in every respect: written in dozens of high (and very high) level languages, interconnected at several levels by message passing mechanisms, produced by a variety of tools, relying on vast depositories of non-uniformly structured information. This view is correct only insofar as the term "software" is used in the loosest possible sense of "all that makes computers run". For any analytical use such an interpretation of the term is inadequate because it blurs the essential distinction between programs which prescribe *computations* and those that prescribe *behaviours*.

Historically, computers were first used for computations or — in a slightly more abstract phraseology — for effecting well-defined state transformations, from a *given* to a *result*. Computations have been a part and parcel of human civilisation for several millennia; for most of that period they were performed by people. Algorithms were invented to formalise the process of human computations and thus to facilitate them, *i.e.* to make them performable by less qualified persons. The sequentiality of computation (and its particularly important variant: iteration), born from the human propensity to do one thing after another, has become a part of our cultural heritage. It was *not a consequence* of the von Neumann architecture, quite the opposite: *it dictated its principles*.

The essence of computation is captured by the familiar $\{P\}p\{Q\}$ triple; the fundamental problems of computation software are easily expressible in terms of this paradigm: termination and correctness nicely merged into notions of partial and total correctness. It took about 20 years of programming practice and its analysis by the more mathematically inclined to find this succinct representation; in another 10 years or so there emerged a fully-fledged discipline based on this paradigm. Originally concerned (almost) exclusively with proofs of program correctness, the discipline developed a number of logics (more or less) well-suited for this task. Subsequently, with the $\{P\}p\{Q\}$ triples embedded in predicate calculus (as predicate transformers), the discipline embraced algebra-like calculi of program derivation.

For software engineering, the most interesting aspects of the discipline were those in which $\{P\}p\{Q\}$ was considered as an equation in p, *i.e.* methods (or at least hints) for constructing an unknown program p for a given characterisation of the initial and final states. The big white hope was *automatic program construction i.e.* in fact an algorithmic solution for the program construction problem, a goal very hard to achieve in restricted circumstances and most probably impossible to reach in general. A number of limited-application solutions resulted in experimental systems implemented in academic and industrial research environments; none have passed into general use. A relaxation of the notion of algorithmicity, for example by admitting a heuristic trial and error approach or by exploiting interaction with a human operator, resulted in several more systems, some actually used, albeit primarily for educational purposes.

Insofar as programming remains essentially a human activity, the greatest impact of the $\{P\}p\{Q\}$ paradigm on software engineering comes through its influence on and absorption by programmers. As is usual with intellectual advances, the time needed for an innovation to reach its social fruition is measured in academic generations: first it is absorbed by peers, then it enters curricula, finally those educated with the innovation firmly established as an integral part of the "craft" (or "science", or "technology") they were taught apply it in their daily work. Assigning roughly five years for each generation, we arrive at the mid-80s as the epoch when the calculational view of program correctness and derivation (and, therefore, program manipulation) should have become a more or less established norm amongst educated programmers. Unfortunately, by that time the demand for programming bodies far outstripped the supply of programming graduates. The balance was made up by persons trained in program writing at various "intensive courses" and by self-taught amateurs, who attained a rudimentary ability to put down program texts with a reasonably small number of syntactic errors. (Easily available tools quickly convert such texts into syntactically perfect programs.) It is only in those establishments where the concentration of educated programmers is high enough that the impact of the calculational approach is felt. Such establishments tend to be found in the better part of academia and in the highly specialised industries. A very large part of the software industry has yet to discover the only tool (so far) for dealing with the *semantic problems of computations*, *i.e.* to reap the benefits of using the $\{P\}p\{Q\}$ paradigm in a calculational fashion.

The mentality that came up with and eagerly embraced the $\{P\}p\{Q\}$ paradigm was shaped by the computations culture, predominantly in science or in some other previously mathematicised fields. This mentality was also formed by the *scientific work ethic* in which criteria of success (although not the success itself!) were impersonal and Occam's razor was ever present. Thus there was little currency for "almost correct" programs, the value attached to "user satisfaction" (other than a perfect outcome of computations) was nil, and frugal use of computer resources was not only a necessity dictated by severe hardware limitations, but also a sign of one's maturity and professionalism. To be quite honest, this mentality could not really conceive of a million or more people running a particular program. Hence, for instance, a very cautious attitude towards program testing was taken: a pretty complex program *could not possibly* be thoroughly tested within a reasonable time by a programmer and his/her several friends/colleagues. In the realm of software for mass use, guided by market forces and paltry user expectations, firmly established by the end of the 80s, the fundaments of that world-view became decidedly outmoded.

For example, wide distribution of the so-called beta versions of popular packages increases the number of testers by several orders of magnitude. From the producer's point of view the beta version testing certainly is practically free, although collecting and processing the test results is not. An additional advantage of beta version testing by a large subpopulation of the intended users comes from an ample representation of the variety of use modes. Indeed, it is very

probable that the several hundred thousand testers using the beta version for several months will find nearly all bugs in a fairly large software product. That the product finally brought to the market still contains a number of errors is to be expected, but these should relate to "exotic" modes of use, or to rare configurations of interactions. Unfortunately, the challenges of managing the tidal wave of bug reports pouring in after the beta version release, and the time pressure to deliver the market version within a reasonable period after the beta version, cause a large proportion of information generated by the beta release to be wasted and/or misinterpreted, which — combined with the poor workmanship inevitable in a hasty patching-up — results in a market product much inferior to what could have been expected after such massive testing. The residual unreliability bears witness to the unmastered complexity and unresolved contradictions characteristic of the software process.

Apart from a few and, frankly, well isolated special cases, software is no longer expected to be *mathematically* correct. With this change, the whole edifice of mathematically-founded software technology appears irrelevant, a folly.

As early as the Garmisch Conference, the incipient change was noted: *If the users are convinced that if catastrophes occur the system will come up again shortly, and if the responses of the system are quick enough to allow them to recover from random errors quickly, then they are fairly comfortable with what is essentially an unreliable system.* (J. W. Smith)

Since the time of the Garmisch Conference the scope of the notion of software has vastly changed. The change is not only of size, even though the volume of software written and used is several orders of magnitude larger today, it is not only of kind, even though numeric computations and sequential file processing then dominant now constitute a barely discernible fraction of computer use, it is not even just the change of mode, even though interactive programs swept from a mere curiosity very prominently into the forefront. The most important change has occurred in the public perception of software.

Two equally powerful and interrelated factors contributed to the change of attitude: emergence of the mass market and an essential shift in the mode of computer use. To the great majority of users constituting the mass market any notion of scientific work ethic is quite alien: they are guided by (and expect) much more relaxed criteria, often "not too bad" taken literally (*i.e.* not as an understatement) means "good enough". Computers are not any longer used primarily as machines to execute computations, instead they are employed to perform a variety of functions defined in terms of contexts taken from everyday life. Windows is a bad idea whose time has come.

It is a serious educational fault that the software consumer population earnestly expects miracles. A vast majority of current PC users are "first-time" users. With no prior first-hand experience of pain and misery, they take an advertising copy for the gospel truth. For example, we talk of software maintenance, when in fact we mean software change. Nobody expects house maintenance to encompass addition of an extra floor or of an Olympic-size swimming pool in the attic. Accomplishments of this calibre are routinely expected of software

maintenance. Regrettably, in the realm of software, consumer education is just as neglected as are consumer rights.

Computer today is typically used as a multimedia communication device or as a glorified switchboard enabling multidirectional flow, transformation, storage and retrieval of discrete signals. There is a growing tendency to eliminate any visible barrier (such as, *e.g.* typing) between the computer and its environment. Some applications still do heavily depend on large volume of repetitive computations (for instance, many medical applications rely on extensive FFT computations), but even then the user is hardly aware of it. In many applications, the individual computations invoked in the course of performing a visible function are trivial (*e.g.* in word processing); the usefulness of an application obtains not from any particular computation, but from the available set of "applications" and the simplicity of their invocation[2]. This situation has profound consequences for the present state of software engineering.

Individual programs of the $\{P\}p\{Q\}$ paradigm are visible neither in the mass market, nor in the field of special applications. Computer systems and instruments with embedded computers are bought and sold for the visible functions they perform, these functions defined in terms of the environment in which the systems and instruments are to be used. Definitions of such functions are seldom mathematical or even simply precise. Correspondingly, the criteria of performance are often diffused. There is no clear relationship between the qualities of the software, as expressed in the framework of the $\{P\}p\{Q\}$ paradigm, and the users' assessment of the quality of the system or instrument. Moreover, the observable deficiencies may just as likely be due to the software buried in its bowels as to other components of the system or instrument. For instance, a poor quality image on the screen of a medical instrument may be due to a defective sensor, bad software, a glitch in the screen electronics or an imperfection in its coating, just to name a few causes with *identical* visible results; naturally, all these causes may be independently present, can interfere with each other, and (don't we all love it?) could be transient. Strict adherence to the $\{P\}p\{Q\}$ school dicta would make software engineering quite insensitive to demands posed by the marketplace; unfortunately even the best software will not sell an erratically behaving instrument, and no programmer can claim that the excellence of software implies an overall good quality of the system.

As resources that used to be scarce got cheaper and cheaper, the economic reasons for many good programming practices started to evaporate, soon to be replaced by their near opposites. As storage units grew in capacity to previously unimaginable sizes, and processing units grew very fast indeed, the premises for the programmer's frugality assumed the flavour of a sectarian ethic. Combined with the growing cost of waste disposal, this has led to an abominable practice of not removing obsolete parts of system software, just shunting them off in new releases. The percentage of dead wood in current releases of popular systems is quite large. So, of course, is the risk that the bypasses put in shall not always

[2] At the Garmisch Conference, A. Perlis prophetically observed: *Almost all users require much less from a large operating system than is provided.*

stay firm. But as long as the havoc resulting from an occasional activation of a shunted off program can be cured by hitting the famous combination CTRL ALT DEL, even with a loss of a file or two, the catastrophe is fully acceptable to most users, immunised by occasional failures of other appliances.

Thus even in that part of software engineering which is concerned with programming for computations, the frame of reference has changed dramatically since the epoch dominated by programs written by scientists for scientists. It is tempting to say that in $\{P\}p\{Q\}$ the emphasis has shifted from p to the two predicates, *i.e.* to specification. If this is indeed the case — and I believe it is — we ought to note a remarkable success of the research carried out under the banners of the $\{P\}p\{Q\}$ paradigm: its original and initially considered the most important goal, to wit: the development of technology for construction of correct programs satisfying specifications given by firm initial and final conditions on computational processes, has been reached.

3 A Linguistic Aside

In English, there is a bit of confusion about the meaning of the noun "model": it can denote either something that sets a pattern to be followed (*model for*, as in "model citizen"), or something that mirrors some other (real) entity (*model of*, as in "models of World War II aircraft"). Sometimes the noun is employed with both meanings simultaneously, which greatly adds to the confusion. Unfortunately, such is the case when speaking of software specifications one uses expressions like "model of real-world (relationships)".

4 A Short Treatise on Model Theory with Applications

The confusion of the natural language usage of "model" is avoided in mathematics, where two terms are assigned to its two roles: *theory* and *model*. Pure mathematics deals with abstract entities which possess only such properties as by accepted rules of reasoning follow from their definition; *e.g.* it is meaningless to ask about the colour of the number π. Some entities are known as *domains*, usually they have a structure: elements, relations, functions etc. A *fact* is a property "observed" in the domain, perhaps involving its structure. Thus in the domain of natural number arithmetic it is a fact that all even numbers divisible by 3 are divisible by 6.

Theories are sets of sentences generated by application of listed rules of inference to a listed set of axioms. It is said that theory T *is satisfied* in domain D, $\mathrm{sat}(D, T)$, or that D *is a model* of T, $\mathrm{mdl}(T, D)$, iff there is an interpretation of sentences by means of facts, such that to each true sentence there corresponds an observable fact. It is important to note that even if $\mathrm{sat}(D, T)$ there very well may be facts in D which under the chosen interpretation correspond to no sentences in T. Indeed this is what Gödel's famous theorem on incompleteness is all about. In this sense a *model is richer than its theory*. For example, the domain of natural number arithmetic is richer than Peano's axiomatisation (a theory

expressly designed to succinctly capture the arithmetic of natural numbers). It follows that a given theory T can have different models, *i.e.* domains which in addition to "core" facts (all corresponding to true sentences of the theory, albeit perhaps under different interpretations, specific for each domain) exhibit their own "additional" facts not necessarily convertible from one domain to another. Thus two perfectly valid models of the same theory need not be very similar (isomorphic). Peano's axiomatisation, for instance, has two well-known models: arithmetic of natural numbers and arithmetic of transfinite numbers. If a theory is consistent (*i.e.* not self-contradictory) it is guaranteed to have a model, and *vice versa*, a theory that has a model is consistent; the latter is often used to prove consistency.

In exact sciences it is usually the case that a theory has two important models: the physical world, W, and a suitable mathematical model M. The theory and its mathematical model being both artefacts, it is (at least in principle) possible to *prove* the $\mathrm{sat}(M, T)$ relation. On the other hand no proof of $\mathrm{sat}(W, T)$ is ever possible; the scientist use *experiments* to check if (important) statements of T correspond to facts in W. Long series of confirmatory experiments increase the likelihood that the theory is OK, a single irrefutable failure is enough to shoot it down. A time-honoured scientific practice is to select for experimental verification the most implausible statements of T.

So far we considered an idealised static picture. In practice, a theory is often the last-to-arise element of the trio. The physical world precedes all man-made artefacts, but in scientific analysis it is usually presented by means of a class of observations which filter out most aspects of the reality. Thus in theory formation and subsequently in verification, if $\mathrm{sat}(W, T)$ holds, W is seldom the whole wide world but instead a specific view of it chosen by the scientist. It is also possible first to construct an elegant mathematical structure, then to invent a theory for which it would serve as a model, and only then look for an aspect of the physical world that could be considered as the physical model; it is rumoured that some Nobel Prize winners in physics have worked exactly in this manner.

5 Specifications

In software engineering, it is the specification that acts as the theory, the corresponding software as one model, the application domain as another. As in science, two elements of the trio are artefacts (specification and software), one is (an aspect of) the real world. Thanks to the discipline of the $\{P\}p\{Q\}$ paradigm, the relations between (properly presented) specifications and programs (software) are calculable, at least in principle. This means we can prove that a program satisfies its specification (if it indeed does so), or — which is much better — given a non-contradictory specification, we can construct a program that provably ("by virtue of construction") satisfies it.

The relationship between the specification and the application domain is much harder to deal with. The chief problem with software for non-formal application domains is that no matter how well-educated and conscientious are the

specification builders, the informality of the domain precludes any strict verification of the abstraction process that leads from the application model to the specification. This plain fact is sometimes masked by the nature of specification-making tools which are increasingly more sophisticated and whose use yields specifications that are formal entities with desirable pragmatic properties. Thus, as far as formal criteria are concerned, we are getting excellent specifications from which it is increasingly easy to design correct programs, indeed, to obtain them automatically. Nevertheless, from the application (*i.e.* ultimate user's) point of view, the quality of the program is determined by how well the informal abstraction process semantically captures the intended application model. (Let us stress that we assume all subsequent steps in program construction and implementation to be faultless, hence the program given to a user is correct wrt informally derived, but itself formally structured and impeccably formal, specification.)

Thus, even in the simplest cases (just as in exact sciences) there is no finite calculational procedure to establish the satisfaction; rapid prototyping and on-site tests — which in this context play the role of experiments — are not conclusive when positive. When negative, following the pattern of exact sciences, the first failure should *invalidate the specification*, but in many application domains the specifications are so rickety that such a clean cut decision is seldom taken. A negative result may simply be dismissed ("it was not really that important"), the specification could be mended in an *ad hoc* fashion ("it really should have been the other way"), or the view of the domain could be changed in a way that invalidates the test ("don't worry, in practice it never happens"). There could (and often is) a powerful incentive for such a cavalier attitude, notably when large sums of money and a considerable effort have been expended on construction of the other model, *i.e.* on software: a slight mismatch between the specification and reality is not reason enough to throw that effort away[3], especially when the specification is a bit woolly and the view of reality a bit foggy. Needless to say, the tenuous link between the software and the application does not get any firmer by employing such practices.

In addition, in many instances software is written for application domains which do not have the intransigence of the physical world . Often, especially in the world of business applications, but also in a plethora of services and entertainment applications, the time span needed to produce software satisfying a given specification exceeds the life-span of the particular world view that served as the specification's other model. This gives rise to the phenomenon of *evolving specifications*: theories that evolve to reflect an ever changing world-model.

A closer analysis of the situation in exact sciences shows that the phenomenon is not entirely absent there. As the views of the physical world evolved, so did their theories and the computation programs that were their models. Today we compute planets' positions according to algorithms quite different from 2000

[3] This is not to say that I advocate the other extreme, vividly described at the Garmisch Conference by R. M. Graham: *We build systems like the Wright brothers built airplanes - build the whole thing, push it off the cliff, let it crash, and start over again.*

years ago; indeed, to compute positions of some planets and most comets we use programs modelling (mildly) relativistic dynamics, while for most other planets and asteroids the plain Newtonian dynamics will do very nicely. The point is that in exact sciences the theories tend to stay fixed for periods much longer than the software life-cycle.

Note, however, that *in mathematics* the phenomenon of evolving specifications is practically unknown. There, a theory once formulated stays unchanged forever, because the wholly artificial world of mathematical structures knows no internal evolution whatsoever. (Its expansion is an altogether different thing.) A circle for Archimedes was the same as for Gauss, even though to compute its circumference to diameter ratio (*i.e.* the number π) they would have used quite different algorithms. The assumption of immutability of theories permeates mathematical culture. The mathematically minded founders of the $\{P\}p\{Q\}$ discipline established a didactic paradigm in which the specifications are considered as given once and for all; they are so firmly fixed that any question of the kind "what would happen if the specification changed?" is easily dismissed as totally irrelevant. Much too easily!

Programming methods cultivated in the $\{P\}p\{Q\}$ discipline — and they are the best there is! — are often spurned on the grounds of frivolity of the examples used to convey the methods. (From the *Problem of Dutch National Flag* to the *Problem of Welfare Crook*, they all are "toy" examples, are they not?) The issue is not in the examples being too simple, because they are not *that* simple and no sane teacher would use much more challenging ones, but in the tacit yet very convincing acceptance of the sacro-sanctity of the specifications (problem statement). Students reared on the exclusive diet of such examples, particularly, the bright students, are likely to consider the problems entailed by evolving specifications as a can of worms carried around by simpletons unaware that good programming practice requires unambiguous and (of course!) fixed specifications. Therein lies one of the reasons for the *mutual* mistrust between software specialists and programmers brought up in the $\{P\}p\{Q\}$ discipline. It is deep and divisive, its background is cultural, and therefore it will not be easy to remove.

It would be untrue to say that the theoreticians have totally disregarded the problem of changing specifications or, to use a more elevated terminology, the problem of theory manipulation. The research under this heading took two main directions: logic-based and functional-programming-based. Despite significant internal achievements, none of them has yet had an appreciable impact on software making. Because of the novelty of approaches developed in this research, it is only to be expected that their social acceptance would take a few more years (*cf.* the academic generations phenomenon discussed earlier). I am afraid, however, that this is only a part of the explanation. The other part is related to some inherent features of the research, features that in the context of a desired wide acceptability appear to be weaknesses.

The logic-based theory manipulation has two such weaknesses. In logic itself the theory manipulation is quite awkward and computationally expensive. Per-

haps yet more importantly, the required model manipulations following from theory manipulation are seldom straightforward, indeed often are non-algorithmic. As a matter of fact, the only known instances of simple model manipulations restoring the satisfaction relation (after theory manipulation) correspond to rather uninteresting kinds of theory manipulation (such as renaming or definition unfolding). In addition, the logic-based approach has a strange relationship with mundane programming practice: its programming vehicle of choice, programming in logic, has obviously missed its chance of becoming the working tool of programming community (very active special interest groups notwithstanding), while the direct use of logic for specification of imperative programs leads to difficulties that the $\{P\}p\{Q\}$ community avoided by using algebraic-like specifications.

The use of algebraic specifications has also been the choice of the functional-programming school. With this choice, a very rich source of mathematical inspirations has been tapped and a lot of energy spent on establishing mathematical credibility of the school. While this activity generated a large number of elegant papers and created a legion of Ph.D. students and graduates, it did nothing for the working programmer and her boss[4]. If anything, the use of forbiddingly mathematical jargon (a PR fault assiduously avoided by the logic-based research community) alienated the functional-programming school from the software industry. Recently there are signs that functional-programming-based specification manipulation is making some inroads into industrial programming practice, where the object-oriented programming mania created a receptive ground.

Time will show how deep the fertile layer for cross-breeding is. The imperative programming habits of a working programmer seem as firmly established as ever, even as the quality of the programming languages in common use is rapidly deteriorating. On the other hand there is a growing tendency to use "fourth generation" and similar "very high level" languages even for professional work in software firms, despite their gargantuan appetite for computer resources. (Is it plausible that in the near future the professional programming languages will degenerate into two classes: one being machine language with object syntax, another — a catalogue of pictographically invoked "intelligent" components?)

Yet, if we accept that software is produced for applications, and applications are increasingly frequent in poorly formalised (or quite informal) domains, there is no escaping the problem of evolving specifications, not least because the (views of) application domains change as a result of implementation of computer systems. Paradoxically, the more successful — in terms of an application — is an

[4] For the vast majority of software engineers, the perennial question of whether (*undefined* = *undefined'*) ≡ **true** or (*undefined* = *undefined'*) ≡ **false** is of precious little significance and even less consequence. There, what matters is that *undefined* should never be encountered and — if it happens — should raise all sorts of alarms. The same, of course, goes for similar concerns with other errors of design: while they could be viewed as a fertile ground for subtle considerations, in the bread–earning community the need to *avoid* them is dominant, and the guarantee that errors, if made, will not remain undetected is a prime concern.

implementation, the more profoundly it changes the application domain, and, therefore, the less valid becomes the original specification.

The only proper way to proceed in case of a changed application model and an existing program is to modify the specification and then see how to modify the program so that it remains a model of the changed specification-theory. The specification's "resistance" to change (expressed in the effort needed to do so properly) is not an *obstacle*, but a *warning* about the real magnitude of effort required to accommodate a change in the application domain. Attempts to make it easier remind me of the policy of taking ever increasing doses of painkillers in order to avoid visiting a dentist when a tooth aches. Follow this policy, if you wish, but then do not complain that your teeth are unreliable and fail the simple test of taking a bite of a nice, hard, juicy apple.

In the realm of software for humdrum applications, the major research challenge is to develop a usable specification calculus equipped with corresponding algorithmic transformations of software models. In short, for a class of specifications Σ we need the following:

1. a set of meaningful and useful monadic operations $U : \Sigma \to \Sigma$
2. a set of meaningful and useful diadic operations $B : \Sigma \times \Sigma \to \Sigma$
3. an indexed set of algorithms $\{\alpha\}_{i \in U \cup B}$

such that given specifications $s_1, s_2 \in \Sigma$ and their software models p_1, p_2, $\mathrm{mdl}(s_1, p_1)$, $\mathrm{mdl}(s_2, p_2)$, we would have $\mathrm{mdl}(u(s_1), \alpha_u(p))$ for any $u \in U$ and $\mathrm{mdl}(b(s_1, s_2), \alpha_b(p_1, p_2))$ for any $b \in B$.

It is a very tall order indeed, one that certainly cannot be fulfilled in its full generality, because it is not true that *any* meaningful and useful operation on a satisfiable specification yields another satisfiable specification. On the other hand, restricting the sets U and B to such operations that are guaranteed to preserve satisfiability for *all* operands would be self-defeating as such sets would consist of very few and mostly uninteresting operations. Yet, I believe this challenge to be most important for software research well into the next century.

6 Production of Software

The idealised view of software being produced by a programmer who was given precise and complete specifications is very far from the prevailing reality. We have already analysed one aspect, *viz.* specification evolution. There are two more that are best considered jointly:

- software very seldom is written "from scratch"
- software is most often produced in large organisations

The existence of large software systems and — in many instances — continuing dependence of users on their performance mean that a very large proportion of software is produced as enhancements (updates, extensions, new releases etc.)

of systems already in use. This entails additional constraints on writing software: not only must it satisfy whatever specifications are provided but it must also allow "seamless integration" of the fresh code with the existing body of the system code. The exact meaning of "seamless integration" varies somewhat from case to case, but in most instances it means that it should be possible to insert the fresh code without seriously interfering with the productive use of the existing system, and it always means that the writer(s) of the fresh code must take into consideration all kinds of interactions between the old and new parts of the system; needless to say, such interactions fall into two categories: expected (well documented, desirable etc.) and unexpected. It is the latter that cause royal headaches. Not infrequently "seamless integration" also implies that the actual users' procedures and habits evolved during their work with the old system should be respected, *i.e.* preserved as much as possible.

Of course all this is nothing new if software engineering is considered as a provider of utilities. Most other kinds of engineering are thoroughly familiar with such requirements: neither a new bridge, nor an additional water reservoir bring about major disruption to services provided (although the construction of a new bridge may occasionally create local havoc!). But implications for software engineering are far reaching. A very substantial part of the effort involved in a successful design and implementation of a piece of software needs to go not into the conversion of specifications into working code, but into all sort of peripheral activities that ensure the "seamless integration". Some of this extra activity is of a programming kind, some is closely related to programming; some, however, belongs to entirely different fields (such as, *e.g.*, public relations or ergonomics).

This alone would indicate that to provide software one needs an organisation which, in addition to programmers, employs other kinds of specialists. (Remember, we are not selling programs any more, we provide software services!). Add to it the sheer volume of contemporary software systems, bloated by prevalent practices, but also necessitated by the scope of activity covered, and the need for large teams becomes pretty obvious.

It is important to note that quite a few of existing large systems were not conceived as large systems, they just grew by repeated extensions, by adding features and functions, each, perhaps, of a moderate size, and by a permissive attitude to waste disposal. Thus, it is not necessarily the case that S. Gill's Garmisch warning: *It is of utmost importance that all those responsible for large projects involving computers should take care to avoid making demands on software that go far beyond the present state of technology, unless the very considerable risks involved can be tolerated* went unheeded.

One way or another, large and very large software systems exist[5], are being updated, extended, modified etc. Other large systems are manufactured, often from large components, much less often - from scratch. Work on a large software system requires a large organisation, a large organisation requires a managerial structure and a defined *modus operandi*. The question, whether the management

[5] *It is large systems that are encountering great difficulties. We should not expect the production of such systems to be easy.* — K. Kolence at the Garmisch Conference

of a software-making organisation is essentially different from that of any other complex-product making concern or not, has not been conclusively resolved[6].

Superficially, the differences seem dominant. The distribution of costs between design and actual production seems very peculiar in software-making: no matter how expensive an architect, his fee is minuscule as compared to the cost of steel, concrete, pipes, cables and other supplies that go into construction of a largish building; even the cost of construction labour is often larger than the architect's fee. In software construction the supplies and raw materials cost next to nothing and cost of labour is mostly subsumed in costs of design, particularly so if modern tools are used for churning out the actual lines of code. Many large software systems are unique (for application on a single site), hence the famous economies of scale hardly matter. And so on.

A closer look, however, casts some doubts on the simple conclusion. In any large organisation a fair share of costs is consumed by the infrastructure, both physical (offices, energy, various services etc.) and managerial (personnel, meetings, reporting etc.); these costs are a function of organisation size and structure regardless of what it does. Economy of scale *does* matter for software making, although in a somewhat different sense: a company supplying its tenth banking system spends less effort per line of code than it did for the first one, even if all ten systems are truly different, because it gained experience[7] in making systems for banks. And so on.

A very substantial volume of national expenditure on software in nearly all developed countries has been a powerful incentive for seeking managerial solutions to the high cost of software construction and its odious companions: budget and deadline overruns. A number of modelling techniques have been proposed for representing the software-making process in a variety of metrics. Not surprisingly, the predictive capability of these techniques is not very impressive: insofar as software-making remains chiefly a creative process, modelling based on statistics is bound to produce results applicable — at best — to collections, and quite inadequate for individual instances. Thus, even if we can predict that *on the average* projects of a certain kind are likely to cost X dollars and last for T months, any particular project of this very kind may cost $2X$ or $0.5X$ dollars and last $0.5T$ or $2T$. To get better predictions we should make the software process more homogeneous, less dependent on human idiosyncrasies; but this is not going to be an easy task. Actually, I am not sure it is possible at all, because as soon as technical means (tools) are introduced to automate (*i.e.* algorithmise) an aspect of the software process, the process itself is extended on "the other end"

[6] *cf.* K. Samelson at the Garmisch Conference: *By far the majority of problems raised here are quite unspecific to software engineering, but are simply management problems. ... Perhaps programmers should learn management before undertaking large scale jobs.*

[7] In a hard to describe way it is indeed the organisation, as distinct from individuals it employs, that gains the relevant experience. Of course, the employees, each in her/his special way, also gain experience from the completed tasks, but there seems to be a clear synergy effect, making the *combined* gain larger than the sum of individual ones.

by inclusion of tasks previously considered outside the software process proper, for example, by processing preliminary requirements.

Another direction taken in an attempt to improve the management of the software process in order to make it (*i.e.* the process) more predictable, consists in developing organisational patterns which supposedly impose best structure on the process and its managerial infrastructure. The success of this approach is the greatest in the most chaotic organisations, where imposition of *any* order, rules, procedures and standards cannot fail to improve co-ordination and internal communication, which in turn is bound to improve productivity and work discipline. Thus, as a means of combating chaos, such patterns are very good; whether they can produce improvements in a disciplined environment is open to some doubt. Needless to say, in this approach there is very little truly specific to software; it is a sad reflection on the state of many software companies that such simple "law and order" prescriptions yield appreciably positive results. What makes it even sadder is the realisation that the essence of the proper approach was clear at the time of Garmisch Conference: *The ability to estimate time and cost of production comes only with product maturity and stability, with the directly applicable experience of the people involved and with business-like approach to project control* (R. McClure) Why this simple message did not reach all concerned during more than 30 years and why substantial organizations can prosper doing very little other then embellishing this message with irrelevant charts and tables remains a psychological puzzle.

Because managerial remedies often do bring positive results in terms of better productivity and more realistic scheduling, and because these effects are eagerly sought after, and because — being managerial in nature and packaged as any other "product" on the managerial market — they look appealingly familiar to managers, such remedies sell very well to the top management of software companies. No wonder then that there is a fierce competition between various schools and institutions offering managerial remedies. The evidence for an objective assessment of individual competitors is (at best) scant, it would be foolish to attempt any ranking; the more so because it is very probable that any one of them is just as good as any other: very helpful to a chaotic organisation, and of little value where a stable and workable *modus operandi* has been established. If the latter hypothesis is correct, we should observe an industry-wide conversion to regimented production units structured on conveyer-belt-like principles, accompanied by a painful demise of traditional loose co-operatives of artist-programmers, followed by a rather rapid loss of interest in all-encompassing "managerial solutions".

7 Programming for Behaviour

As mentioned earlier in this essay, the interest in computer applications is shifting from calculations to behaviours. To be sure, computers still perform calculations and shall continue doing so, if only because they cannot do anything else. The shift occurs in the *external perception* of computing activity: it is per-

ceived less and less as a purposeful combination of calculations, and more and more as an unordered collection of *reactions to external stimuli*. A computer is no longer expected to be turned on to achieve a (calculational) result and then be switched off, instead it is expected to be on all the time and, while being on, to behave in conformance with its (independently and often unpredictably) changing environment.

What does this shift spell for software? There is a level at which the simple answer is: nothing new. Each individual reaction needs to be programmed just as any old fashioned procedure, except for one significant difference: The execution of any routine takes time, during this interval the environment, which now is assumed to be quite independent from the computation, can change in a way which makes the reaction being computed obsolete, inappropriate, unwanted or even plainly harmful. This is not an entirely new problem, we have faced it ever since programming concurrently running processes started. However, there is an important *novum*: we are not in control of all processes. Even if there could arise a harmful interference, we are not allowed the luxury of mutual exclusion of critical sections, the environment will not patiently wait suspended at a semaphore nor languish inactive in a monitor until our routine completes its critical section and releases the catch.

In other words, we have two problems

- a theoretical one: how to deal with continuous concurrence?
- a practical one: how to implement the theoretical solution on discrete (digital) computers?

There are other problems which programming for behaviours brings to the surface:

- how to express (specify) and program (implement) modalities: **do** *something* **as long as** R , where predicate R does not depend on *something*, indeed, may be totally outside our control, and: **do** *something else* **before** S, where *something else* does not influence S ?
- how to cope with situations in which several behaviours are indicated in the same state of the environment and there is no reason to expect that choosing one (or, for that matter, any subset) could be justified? Note that the collection of behaviours required in a state need not to recur in any other state and that its components may belong to other collections indicated in other states.

All these problems share one essential property: absence of a granularity of "time" common to all participating computational processes. This vitiates most (if not all) classical approaches to concurrence. Indeed, the principle of a common discrete time-like dimension permeates the temporal logic approach and its variants. Sometimes the common time is replaced by a common synchronisation principle, where the pattern of interactions (no matter whether synchronous or not) weaves a braid of time-like progression. Even in seemingly time-less mutual exclusion co-ordination there is an implicit pattern of sequencing which

corresponds to a time-like dimension[8]. There are two reasons why the granular time-like dimension (whether explicit or implicit) was chosen as the framework for (nearly) all research on concurrent programming and — by extension — for (nearly) all work on software systems: (i) it is a simple discrete version of "smooth" time which was and remains a basis of Western religions, philosophies and science, (ii) it allows us to play down the role of the interrupt, a phenomenon all too common in hardware and singularly difficult to deal with in theories based on logic, mathematics or computations.

A simple example should convince us about how eager we are to invent time-based solutions to problems in which time plays no role at all. Consider the problem of boiling a breakfast egg. To solve it means to bring the yolk and the white to specific conditions, *e.g.*, the white set, and the yolk semi-liquid. Assuming we start with a reasonably fresh egg, the solution will be achieved by heating the egg until the desired condition is reached. Simple, isn't it? But in deference to our scientific tradition (and also because measuring the consistence of white and yolk without breaking the shell is a little difficult), we invent the notion of a "three minute" egg and proclaim that under average conditions most eggs immersed for three minutes in boiling water *are* proper breakfast eggs. And so we cook breakfast eggs not to reach a satisfactory condition, but for three minutes. Indeed, most people consider this a proper (scientific?) solution to the original problem. In many instances, programming for behaviour is just restoring the original sense to problems falsified by simplifications.

It seems that there are two major avenues of approach to behavioural programming. In one of them, we decide to design perennially watchful programs, *i.e.* programs which after each atomic action would evaluate the state of the environment and progress according to the outcome of the evaluation. In the other approach we may decide to advance individual computations in larger chunks, but be prepared to roll them back if they progress beyond the limits of validity determined by the environment. Both solutions appear quite expensive in terms of control and/or restorative computations; the second approach is not unlike some techniques used in fault-tolerance and distributed database updating. A common aspect of both approaches is an attempt to simulate an event-driven behaviour within the framework of traditional computations. It may thus appear that these approaches are implementation oriented, or even implementation motivated. Indeed, with the prevalent computer architectures there is hardly any other way to implement behavioural computations.

An entirely different approach would result from taking an event-driven structure of computations as the basic paradigm, and all actions triggered by an observed event as atomic. The latter assumption implies that an action produces no externally observable events while it is being carried out (any possible visible effects occur only when the action terminates) and cannot be influenced by external events occurring while it is being executed. In recognition of the fact that the execution of any atomic action takes some time, we should equip an

[8] I am using the expression "time-like" in order to avoid any suspicion that what is implied is the well-behaved, uniform time of classical physics and theology.

action with two guards: one, call it the preguard, describing the state of the environment in which the action is fired, another, call it the postguard, describing the state of the environment in which the effects of the executed action are acceptable. Operationally speaking: As soon as the environment enters (creates) the state in which action is to be fired, its execution is initiated (on a *private copy of the universe*). As soon as the execution terminates, the state of the environment is tested by (evaluating) the postguard; if the postguard is satisfied, the action effects (if any) instantaneously update the environment, otherwise the action's execution and possible effects are completely ignored. This model can be completed by an assumption on the number of agents able to execute any action, for example, by stating that there are sufficiently many agents, or that their number is infinite[9].

The simplicity of behavioural specifications obtained in this fashion is very enticing. For instance, the specification for a fork-picking action in the *Dining Philosophers Problem* may look as follows

(*has left fork* **and** *right fork on table, right fork on table*)→ *pick right fork*,

where the two guards are separated by the comma. This specification corresponds to verbal behavioural instruction: "if you have the left fork and the right fork is available for taking, take it, provided you can pick the right fork before anybody else grabs it". In the same problem, the deadlock is prevented by a pair of specifications for each philosopher:

(*has left fork* **and** *right fork on table, right fork not on table*) → *release left fork*,

(*has right fork* **and** *left fork on table, left fork not on table*) → *release right fork*

Note that the deadlock-breaking actions are purely local to each philosopher.

Unfortunately, even this unorthodox approach cannot cope with the limitations unavoidably introduced by the discrete nature of digital computing. In specification $(P, P) \rightarrow \alpha$ there is no way to distinguish between the environment *unchanged* during α's execution and one changed but *restored* to a state satisfying P just in time for α's termination. A very careful formulation of guards may include terms that would reflect some "tracing" information, in which case the two guards above will not be identical, but the generality of this solution remains doubtful. Indeed, if the "observer" has a time constant $\tau > 0$ such that two consecutive observations must be separated by at least τ, then any cycle (*change, restore*) in the environment with period much less than τ is likely to be missed or miscounted by the "observer".

The problem of providing software for behaviour-oriented systems has been recognised long ago. So far, however, nearly all attempts to solve it have been based on extensions of the sequential calculational paradigm. Some extensions were brilliant, others — less so. A number of very interesting disciplines have been created, among them various schools of parallel programming, especially for

[9] On closer scrutiny, the apparently insurmountable implementational difficulties of this approach turn out much less forbidding.

numeric calculations, where the power of specially designed parallel processors obviously needed a corresponding development on the programming side. None of these addressed the main issues of behavioural programming. A pessimistic conclusion appears inevitable: the possibilities of extending the calculational paradigm towards a behavioural one have been exhausted without ever getting near the goal.

Thus the challenge is there, clearly visible to any impartial observer. It seems possible that in order to meet this challenge, we shall have to make an entirely fresh start, perhaps discarding not only much of what we learned about programming (and accepted as "natural"), but also largish parts of logic (in particular, some classical rules of reasoning, such as *modus ponens* and *tertium non datur*). It will be interesting to see if the programming/software community can take the dare. Modern physics took a similar step in the first quarter of this century; ever since then, the common sense and science remain at odds on many issues. In the next century, shall *we* take the plunge?

8 Conclusions

The basic elementary step in software construction, the derivation of a program correct wrt its fixed and consistent specification, has been fully intellectually mastered and can now be performed orders of magnitude faster and safer than 50 years ago. The mathematical clarity of these developments enabled the construction of tools which greatly facilitate the mechanics of programming. The process of making larger software constructs from smaller ones can be carried out safely in limited, highly constrained circumstances. The problem of adopting existing software to evolving specifications remains largely unsolved, perhaps is algorithmically insoluble in full generality. Development of a realistic specification calculus is badly needed and does not seem impossible. Managerial aspects of large-scale software production are important insofar as many software companies are still managed in an amateurish way, which certainly leaves a lot of room for improvement. Whether good management practices for large-scale software production differ in an essential way from those for any other large-scale team effort or not remains an open question. Prevailing criteria of commercial success in the software market are shifting away from strict notions of correctness towards much vaguer notions of user satisfaction; this tends to de-emphasise the role of hard science in programming. Computers are increasingly perceived not as calculating machines, but instead as elements of larger systems whose purpose is to react, or enable other system elements to react, to stimuli provided by the environment. This creates a need for a new programming paradigm, oriented towards behaviour rather than towards calculations. It is likely that such a new paradigm shall be radically different from what we consider as familiar.

I am not sure there exists a software engineering at all.