Engineering and Software Engineering

Michael Jackson

Department of Computing
The Open University
Milton Keynes MK7 6AA
United Kingdom

Abstract. The phrase 'software engineering' has many meanings. One central meaning is the reliable development of dependable computer-based systems, especially those for critical applications. This is not a solved problem. Failures in software development have played a large part in many fatalities and in huge economic losses. While some of these failures may be attributable to programming errors in the narrowest sense—a program's failure to satisfy a given formal specification—there is good reason to think that most of them have other roots. These roots are located in the problem of software engineering rather than in the problem of program correctness. The famous 1968 conference was motivated by the belief that software development should be based on "the types of theoretical foundations and practical disciplines that are traditional in the established branches of engineering." Yet after forty years of currency the phrase 'software engineering' still denotes no more than a vague and largely unfulfilled aspiration. Two major causes of this disappointment are immediately clear. First, too many areas of software development are inadequately specialised, and consequently have not developed the repertoires of normal designs that are the indispensable basis of reliable engineering success. Second, the relationship between structural design and formal analytical techniques for software has rarely been one of fruitful synergy: too often it has defined a boundary between competing dogmas, at which mutual distrust and incomprehension deprive both sides of advantages that should be within their grasp. This paper discusses these causes and their effects. Whether the common practice of software development will eventually satisfy the broad aspiration of 1968 is hard to predict; but an understanding of past failure is surely a prerequisite of future success.

Keywords: artifact, component, computer-based system, contrivance, feature, formal analysis, normal, operational principle, radical, specialisation, structure.

1. Software Engineering Is About Dependability

The aspiration to 'software engineering' expresses a widely held belief that software development practices and theoretical foundations should be modelled on those of the established engineering branches. Certainly the record of those branches is far from perfect: the loss of the space shuttle Challenger, the collapse of the Tacoma Narrows bridge, and the Comet 1 crashes due to metal fatigue are merely the most notorious of many engineering failures. But—rightly or wrongly—these failures are seen as local blemishes on a long record of consistently improving success. Failures of software projects and products seem to be the rule rather than the exception. Sardonically, we compare the frustrations of using Windows with the confident satisfactions of using a reliable modern car. At a more serious level, software engineering defects often play a large part in critical system failures. A nuclear power station control system shut down the reactor when a software update was installed on a connected data acquisition system [Krebs 08]. In the years 1985–1987, software defects of the Therac-25 radiotherapy machine caused massive radiation overdoses of several patients with horrendous results [Leveson 93]. Nearly twenty five years later, more modern radiation therapy machines were involved in a much larger number of very similar incidents [Bogdanich 10]. Software defects, of the kind that were responsible for the Therac disasters, were a major contributory factor in many of these contemporary incidents.

These and the multitudinous similar failures reported in the Risks Forum [Risks Digest] are evidence, above all, of the lack of dependability [JacksonD 07] in the products of software engineering. Regrettably, we are not astounded—perhaps we are no longer even surprised—by these failures. As software engineers we should place dependability foremost among our ambitions.

2. A Software Engineer’s Product Is Not the Software

A common usage speaks of the physical and human world as the ‘environment’ of a computer-based system. This usage is seriously misleading. The word ‘environment’ suggests that ideally the surrounding physical and human world affects the proper functioning of the software either benignly or not at all. If the environment provides the right temperature and humidity, and no earthquakes occur, the software can get on with its business independently, without interference from the world.

This is far from the truth. Dijkstra observed [Dijkstra 68] that the true subject matter of a programmer is the computation evoked by the program and performed by the computer. In a computer-based system, in which the computer interacts with the physical and human world, we must go further. For such a system, the true subject matter of the software engineer is the behaviour evoked by the software in the world outside the computer. The system’s purpose is firmly located in its *problem world*: that is, in those parts of the physical and human world with which it interacts directly or indirectly. Software engineers must be intimately concerned with the problem world of the system whose software they are developing, because it is that problem world, enacting the behaviour evoked and controlled by the software, that is their true product. The success or failure of a radiotherapy system must be judged not by examining the software, but by observing and evaluating its effects outside the computer. Do the patients receive their prescribed doses of radiation, directed exactly at the prescribed locations? Are individual patients dependably identified or are there occasional confusions? Is the equipment efficiently used? So developers of a radiotherapy system must be concerned with the detailed properties and behaviour of every part of the therapy machine equipment, with the positioning and restraint of patients undergoing treatment, with the radiologist's procedures, the oncologist's prescriptions, the patients' behaviours, needs and vulnerabilities, and with everything else that contributes substantially to the whole system.

3. The Software and Its Problem World Are Inseparable

Twenty years ago Dijkstra [Dijkstra 89] argued that the intimate relationship between the software and its problem world could—and should—be dissolved by interposing a formal program specification between them:

> "The choice of functional specifications—and of the notation to write them down in—may be far from obvious, but their role is clear: it is to act as a logical 'firewall' between two different concerns. The one is the 'pleasantness problem,' i.e. the question of whether an engine meeting the specification is the engine we would like to have; the other one is the 'correctness problem,' ie the question of how to design an engine meeting the specification. [...] the two problems are most effectively tackled by [...] psychology and experimentation for the pleasantness problem and symbol manipulation for the correctness problem."

The argument, attractive at first sight, does not hold up under examination. Even for such problems as GCD, the sieve of Eratosthenes, and the convex hull of a set of points in three-dimensional space, the desired firewall is more apparent than real. The knowledge and skill demanded of the software developer are not restricted to the formal symbol manipulations foreseen in the programming language semantics: they must also include familiarity with the relevant mathematical problem world and with a sufficient body of theorems. A candidate developer of a program to "print the first thousand prime numbers" would be conclusively disqualified by ignorance of the notion of a prime number, of integer multiplication and division, and of the integers themselves. Far from acting as a firewall between the program and its problem world, the program specification is inescapably interwoven with the relevant part of the problem world, and tacitly relies on the programmer's presumed prior knowledge of that world.

The firewall notion fails even more obviously for computer-based systems whose chief purposes lie in their interactions with the physical world. The problem world for each such system is a specific heterogeneous assemblage of non-formal problem domains. In any feasible design for the software of such a system the behaviours of the software components and problem world domains are closely intertwined. The locus of execution control in the system moves to and fro among them all, at every level and every granularity. The software components interact with each other at software interfaces within the computer; but they also interact through the problem world. A software component must take account not only of the values of program variables, but also of the states of problem domains outside the computer. The problem domains, therefore, effectively act as shared variables, introducing additional interaction paths between software components. An intelligible formal specification of the system, cleanly separating the behaviour of the software from the behaviour of the problem domains, is simply impossible.

4. A Computer-Based System Is a Contrivance In the World

The products of the traditional branches of engineering are examples of what the physical chemist and philosopher Michael Polanyi calls [Polanyi 58] ‘contrivances’. A device or machine such as a motor car, a pendulum clock, or a suspension bridge is a particular kind of contrivance—a physical artifact designed and built to achieve a specific human purpose in a restricted context in the world. A computer-based system is a contrivance in exactly the same sense.

A contrivance has a configuration of characteristic components, each with its own properties and behaviour. In a computer-based system these characteristic components are the problem world domains, interacting under the control of the software. The components are configured so that their interactions fulfil the purpose of the contrivance. The way in which the purpose is fulfilled by the components is the ‘operational principle’ of the contrivance: that is, the explanation of how it works. In a pendulum clock, for example, gravity acting on the weight causes a rotational force on the barrel; this force is transmitted by the gear train to the hands, causing them to turn; it is also transmitted to the escapement wheel, which rotates by one tooth for each swing of the pendulum; the rotation of the hands is therefore proportional to the number of pendulum swings; since the pendulum swings at an almost constant rate, the changing angular position of the hands indicates how much time has elapsed since they were set to a chosen reference point.

The design and the operational principle of such a machine reflect what Polanyi calls ‘the logic of contrivance’. This is something different from, and not reducible to, physics or mathematics: it is the comprehension of a human purpose and the accompanying intuitive grasp of how that purpose can be achieved by the designed contrivance. It is, above all, the exercise of invention, supported by the human faculties of visualisation and embodied imagination. It is not in itself formal, although it can be applied to a formal subject matter—for instance in the conception of a mathematical theorem or the understanding of a physical process. Natasha Myers gives an account [Myers09] of how a brilliant teacher of chemistry explained and illustrated the folding of a protein to students by physically enacting the folding process with his arms and hands. “In this process,” she writes, “scientists’ bodies become instruments for learning and communicating their knowledge to others.” The knowledge here is, essentially the ‘feel’ for how the protein folding works.

5. General Laws and Specific Contrivances

The important distinction between engineering and natural science is reflected in the difference between the logic of contrivance and the laws of nature.

The scientist’s ambition is to discover laws of a universal—or, at least, very general—nature. Experiments are designed to test a putative law by excluding, as rigorously as possible, all complexities that are not pertinent to the law as formulated. The generality, and hence the value, of the experimental result depends on this isolation from everything regarded as irrelevant. In a chemical experiment the apparatus must be perfectly clean and the chemicals used must be as pure as possible.

In an electrical experiment extraordinary measures may be taken to exclude stray capacitance or inductance. Ideally, only the effect of the putative law that is the subject of the experiment is to be observed: the effects of all other laws must be in some sense cancelled out or held constant.

An engineered contrivance, by contrast, is designed to work only in a restricted context implied by the engineer's mandate. The pendulum clock cannot work where there is no gravitational field of appropriate strength. It cannot work on a ship at sea, because it must maintain an upright position relative to the gravitational force and must not as a whole be subject to acceleration. It cannot work submerged in oil or water, because the resistance to pendulum motion would be too great, and the effect of the weight would be too far diminished by the upward thrust of the liquid in which it is submerged. The chosen context—albeit restricted—must then be accepted for what it is. The contrivance is intended for practical use, and must work well enough in the reality of its chosen context. The design may be frustrated by the operation of some previously unknown natural law or some phenomenon whose effects have been neglected or underestimated. These inconveniences cannot be wished away or judged irrelevant: when a failure or deficiency is revealed, the engineer must find a way to resolve the problem by improving the design. Legislating a further restriction in the context is rarely possible and never desirable. For example, when it became evident that early pendulum clocks ran faster in winter and slower in summer, it was certainly not a possible option to restrict their use to one season of the year. An engineering solution was necessary, and was found in the temperature-compensated pendulum, which maintains a constant distance between the pendulum's centre of gravity and the axis on which it swings.

Of course, the contrivance and its component parts cannot flout the laws of nature; but the laws of nature provide only the outermost layer of constraint on the contrivance's behaviour. Within this outermost layer is an inner layer of constraint due to the restricted context of use. The laws of nature would perhaps allow the engineer to predict by calculation how the clock would behave on the moon, but the prediction is pointless because the clock is not intended for operation on the moon. Within the constraints of the restricted context there is the further layer of constraint due to the engineer's design of the contrivance. The parts and their interactions are shaped by the engineer, within the boundaries left undetermined by the laws of nature as they apply within the restricted context, specifically to embody the operational principle of the contrivance.

It is the specific human purpose, the restriction of the context, and the engineer's shaping of the contrivance to take advantage of the context to fulfil the purpose, that make engineering more than science. It is a mistake to characterise engineering as an application of natural science. If it were, we would look to eminent physicists to design bridges and tunnels. We do not, because engineering is not reducible to physics. In Polanyi's words [Polanyi 66, p.39]:

> "Engineering and physics are two different sciences. Engineering includes the operational principles of machines and some knowledge of physics bearing on those principles. Physics and chemistry, on the other hand, include no knowledge of the operational principles of machines. Hence a complete physical and chemical topography of an object would not tell us whether it is a machine, and if so, how it works, and for what purpose. Physical and chemical investigations

> of a machine are meaningless, unless undertaken with a bearing on the previously established operational principles of the machine.”

6. The Lesson Of the Established Branches

Over forty years ago there was much dissatisfaction with the progress and achievements of software development, and much talk of a ‘software crisis’. Although in those days the primary and central role of the physical problem world in a computer-based system had not yet become a direct focus of interest, many people nonetheless looked to the established engineering branches as a model to emulate. This was the explicit motivation of the famous NATO software engineering conferences [Naur 69, Buxton 70] of 1968 and 1969:

> “In late 1967 the Study Group recommended the holding of a working conference on Software Engineering. The phrase ‘Software Engineering’ was deliberately chosen as being provocative, in implying the need for software manufacture to be based on the types of theoretical foundations and practical disciplines, that are traditional in the established branches of engineering.”

Certainly, the participants were not complacent. At the 1968 meeting Dijkstra said in discussion:

> “The general admission of the existence of the software failure in this group of responsible people is the most refreshing experience I have had in a number of years, because the admission of shortcomings is the primary condition for improvement.”

Yet somehow the explicit motivation of the Study Group played little part in the presentations and discussions. An invited talk by M D McIlroy [Naur 69, pp. 138–155] was perhaps a lone exception. McIlroy argued the need for a components industry that would offer catalogues of software components for input and output, trigonometric functions, or symbol tables for use in compilers; but this suggestion stimulated no broader discussion beyond the single session in which it was made.

The conference participants did not explore the practice and theory that they were invited to emulate. Had they made even a modest start in this direction, they would surely have recognised the broadest, simplest and most conspicuous lesson to be learned from the established branches. From their mere plurality—they are ‘the established *branches*’, not ‘the established *branch*’—it can be seen at once that they are specialised. Civil engineers do not design motor cars, electrical power engineers do not design bridges, and aeronautical engineers do not design chemical plants. Our historical failure to learn and apply this lesson fully is epitomized by the persistence of the phrase ‘software engineering’. Software engineering can be a single discipline only on the assumption, by analogy, that the established branches constitute a single discipline of ‘tangible engineering’. On the contrary: they do not. The very successes that we hoped to emulate depend above all on their specialisation into distinct branches.

7. Specialisation Has Many Dimensions

Specialisation, in the sense that matters here, is not the concentrated focus of one individual on a single personal goal. It is the focus of an enduring community over an extended period, developing, preserving, exploiting and enlarging their shared knowledge of a particular field of technical or scientific endeavour.

Specialisations emerge and evolve in response to changing needs and opportunities, and focus on many different interlocking and cross-cutting aspects and dimensions of a field. The established branches of engineering illustrate this process in a very high degree. There are specialisations by engineering artifact—automobile, aeronautical, naval and chemical engineering; by problem world—civil and mining engineering; and by requirement—production engineering, industrial and transportation engineering. There are specialisations in theoretical foundations—control and structural engineering; in techniques for solving mathematical problems that arise in the analysis of engineering products—finite-element analysis and control-volume analysis; in engineered components for use in larger systems—electric motors, internal combustion engines, and TFT screens; in technology and materials—welding, reinforced concrete, conductive plastics; and in other dimensions too.

These specialisations do not fall into any simple hierarchical structure. They focus, in their many dimensions, on overlapping areas at every granularity, and on every concern from the most pragmatic to the most theoretical, from the kind of engineering practice that is almost a traditional craft to such theoretical topics as the advances in thermodynamics that provided the scientific basis for eliminating boiler explosions in the high-pressure steam engines of the first half of the nineteenth century.

8. The Key to Dependability Is Artifact Specialisation

In choosing whom among the established engineers we ought to emulate, there is no reason to exclude any of the many dimensions of specialisation that they exhibit; but there is a very practical reason to regard one dimension of engineering specialisation as fundamental and indispensable. This is what we may call ‘artifact specialisation’: that is, specialisation in the design and construction of artifacts of a particular class. Because a component in one engineer’s design may itself be the artifact in which another engineer specialises, and the same artifacts may appear as components in different containing artifact classes, there is a potentially complex structure of artifact specialisations. Within this structure an artifact specialisation can be identified wherever an adequately specialised engineering group exists for whom that artifact is their group product, designed for delivery to customers outside the group. The specialist artifact engineer is responsible for the completed artifact, and for the total of the value, experience and affordances it offers its users and everyone affected by it. There are no loopholes and no escape clause. This is the essence of engineering responsibility—a responsibility which is both the criterion and the stimulus of artifact specialisation.

Effective artifact specialisation is not an easy option. In its most successful forms it demands intensive and sustained research efforts by individuals and groups within the specialist community. Walter Vincenti, in his book *What Engineers Know and How they Know It* [Vincenti 93], describes three remarkable examples in aeronautical engineering of the years from 1915 to 1945.

The first example addressed the problem of propeller design. Most early aircraft were driven by two-bladed propellers. The design of such a propeller is aerodynamically more complex than the design of an aircraft wing, because the propeller blades are not only moving in the direction in which the aircraft is flying but are also rotating about the propeller's axis. To address this problem, W F Durand and E P Lesley carried out detailed analytical studies and wind-tunnel tests on 150 propeller designs, working together over the years 1915–1926.

The second example concerned the problem of flush-riveting the metal skin to the frame of a fuselage or tailplane or fin. When metal skins first became practicable for aircraft they were fixed to the frames by rivets with domed heads. Soon it became apparent that the domed heads were reducing performance by causing significant aerodynamic drag: flush riveting was therefore desirable, in which the rivet head would have a countersunk profile and would not protrude above the skin surface. The skin was thin—typically, no more than 1mm thick; the necessary countersinking had to be achieved without weakening the skin or causing it to loosen under the stresses of operation. A satisfactory solution to this problem took twenty years' cooperation among several aircraft manufacturers from 1930 to 1950.

The third example has perhaps more immediately obvious counterparts in software engineering. By about 1918 pilots were beginning to articulate a strong desire for certain non-functional requirements in aircraft: they wanted aircraft with 'good flying qualities—stable, responsive, unsurprising and satisfactory to fly'. The questions arose: What did the pilots really mean by these quality requirements? What behavioural properties of the designed artifacts—the aircraft—would ensure these qualities? How could designers achieve these behavioural properties? Specialist research into these questions was conducted over the following twenty years and more. By about 1940 the main elements of the answers had been established for aircraft of the normal design of the time—that is, aircraft with lateral symmetry, a single wing with a straight leading edge, and a horizontal tailplane and vertical fin at the rear. The answers were inevitably very technical. They applied to the established standard design; they were expressed in terms of that design; and finding them took the dedicated specialised effort of designers, pilots and instrument engineers over a quarter of a century.

9. Normal and Radical Design

The fruit and direct expression of each artifact specialisation is the associated 'normal design': that is, the engineering design practice that is standard in the specialisation, and the standard design products that it creates. Vincenti characterises normal design like this [Vincenti 93]:

> "[...] the engineer knows at the outset how the device in question works, what are its customary features, and that, if properly designed along such lines, it has a good likelihood of accomplishing the desired task."

He illustrates normal design by the practice of aero engine design in the period before 1945:

> "A designer of a normal aircraft engine prior to the turbojet, for example, took it for granted that the engine should be piston-driven by a gasoline-fueled, four-stroke, internal-combustion cycle. The arrangement of cylinders for a high-powered engine would also be taken as given (radial if air-cooled and in linear banks if liquid-cooled). So also would other, less obvious, features (e.g., tappet as against, say, sleeve valves). The designer was familiar with engines of this sort and knew they had a long tradition of success. The design problem—often highly demanding within its limits—was one of improvement in the direction of decreased weight and fuel consumption or increased power output or both."

In *radical design*, by contrast,

> "[…] how the device should be arranged or even how it works is largely unknown. The designer has never seen such a device before and has no presumption of success. The problem is to design something that will function well enough to warrant further development."

No design is ever completely and utterly radical in every respect, because the designer's previous knowledge and experience inevitably bring something potentially relevant and useful to the task. Karl Benz, for example, had worked successfully on the design of a gasoline-fuelled engine before he applied for a patent [Benz Patent 86] for his first complete automobile design. Details of Benz's design and photographs of exact replicas (two series of replicas were manufactured in the late twentieth century) are easily available on the web [Benz 86]. The car had three wheels, each with radial metal spokes and solid rubber tyres. The driver steered by a crank handle that turned the vertically pivoted front wheel through a rack-and-pinion mechanism. The car was powered by a single-cylinder four-stroke engine with an open crankcase, mounted behind the bench seat between the rear wheels, and positioned with the crankshaft vertical. A large open flywheel was mounted on the crankshaft, and to start the engine the flywheel was rotated by hand. Power was transmitted to the wheels through bevel gears, to a belt drive that acted also as a clutch, a differential, and finally by a chain drive to each rear wheel. Braking was provided by a handbrake acting on an intermediate shaft.

This design incorporated many innovations and was undoubtedly a work of genius. As a practicable engineering artifact it satisfied Vincenti's criterion—it worked well enough to warrant further development—but it had many defects. The most notable failure was in the design of the steering. The single front wheel made the ride unstable on any but the smoothest surface, and the very small radius of the crank by which the vehicle was steered gave the driver too little control. Control could be improved by a longer crank handle, but the single front wheel was a major design defect. It is thought that Benz adopted the single front wheel because he was unable to solve the problem of steering two front wheels. To ensure that the trajectory of each front wheel is tangential to the path of travel, the inner wheel's turning angle must be greater than the outer's. The arrangement needed to achieve this—now commonly known as 'Ackermann steering geometry'—had been patented some seventy years earlier by a builder of horse-drawn carriages and also was used ten years earlier in a steam-powered vehicle designed in France, but neither Benz nor his contemporary Gottlieb

Daimler was aware of it. Five years later Benz rediscovered the principle, and even succeeded in obtaining a patent in 1891. His 1893 car, the Victoria, had four wheels and incorporated Ackermann steering.

10. Artifact Specialisations In Software Engineering

Software engineering certainly has many visible specialisations. There are many active specialisations in theoretical areas that may be regarded as belonging to computer science but have direct application to the engineering of software-based systems: for example, concurrency and complexity theory. There is a profusion of specialisations in relevant tools and technologies: for example, software verification, model-checking, formal specification languages, and programming languages for developing web applications.

There are artifact specialisations too, but chiefly for artifacts whose problem world, while not being purely abstract, remains comfortingly remote from the complications and uncertainties of the physical and human world. Fred Brooks characterised this class of artifact neatly in a comment [Brooks 10, pp. 56–57] on the open source movement:

> "The conspicuous success of the bazaar process in the Linux community seems to me to derive directly from the fact that the builders are also the users. Their requirements flow from themselves and their work. Their desiderata, criteria and taste come unbidden from their own experience. The whole requirements determination is implicit, hence finessed. I strongly doubt if Open source works as well when the builders are not themselves users and have only secondhand knowledge of the users' needs."

The admirable principle "eat your own dogfood" applies well when the developers are also the customers, or can easily play the customer's role with full conviction. So there are effective specialisations working on compilers, file systems, relational database management systems, SAT solvers and even spellcheckers. Use of these artifacts falls well within the imaginative purview of their developers, and the buzzing blooming confusion of the physical world is safely out of sight.

The same can not be said of radiotherapy or automotive systems, or of systems to control a nuclear power station or an electricity grid. It is not yet clear to what extent these systems, and many other computer-based systems that are less critical but still important, have begin to benefit from the intensive specialisation that has marked the established engineering branches over their long histories. Even when a specialisation itself has become established the evolution of satisfactory normal design is likely to take many years. Karl Benz's invention had given rise to an international industry by about 1905; but it was not until the 1920s that automobiles could be said to have become the object of normal design.

11. Artifact Specialisation Needs Visible Exemplars

There are many cultural, social and organisational obstacles to the emergence and development of artifact specialisations in software engineering. One obstacle is what may be called a preference for the general and a corresponding impatience with the particular. In journals and conferences descriptions of specific real systems are rarely found. At most a confected case study may be presented as a sketched example to support a general thesis. We seem to assume that there is little worth learning that can be learned from a detailed study of one system.

This disdain for the specific militates strongly against the growth of effective artifact specialisations. In the established engineering branches practising engineers learn from specific real artifacts about the established normal design, their properties and their failures. For example, the Earthquake Engineering Research Center of UC Berkeley maintains a large library [Godden 10] of slides showing exemplars of real civil engineering structures such as bridges and large buildings of many kinds. The collection is arranged primarily by artifact class, and serves as a teaching resource for undergraduate and graduate courses. Students learn not only by acquiring knowledge of theory, but also by informed examination of specific real engineering examples: each example has its place in a rich taxonomy, and its own particular lessons to teach.

Failures are no less—perhaps even more—important than successes. Every engineering undergraduate is shown the famous amateur movie that captured the collapse [Holloway 99] of the Tacoma Narrows Bridge in 1940. The designer, Leonid Moisseiff, had given too little consideration to the aerodynamic effects of wind on the bridge's narrow and shallow roadway, and especially to vertical deflections. In a wind of only 40 mph, vertical oscillations of the roadway built up to a magnitude that completely destroyed the bridge. Designers of suspension bridges learned the lesson, and have recognised their obligation to preserve it and hand it on to their successors. It is a very specific lesson. Moisseiff's mistake will not be repeated.

In software engineering we can be less confident that we will not repeat the mistakes we discover in our designs. Without artifact specialisation there may be no structure within which a sufficiently specific lesson can be learned. Some years after the Therac-25 experiences an excellent investigative paper [Leveson 93] was published in IEEE Computer and appeared in an improved form as an appendix to [Leveson 95]. The researchers studied all the available evidence, and their paper identified certain specific software and system errors. But the absence of a normal design for the software of a radiotherapy machine was clearly evidenced by the lack of nomenclature for the software components. The electro-mechanical equipment parts were given their standard names in the diagrams that appeared in the paper: the turntable, the flattener, the mirror, the electron scan magnet, and so on. But the software parts had no standard names, and were referred to, for example, as "tasks and subroutines in the code blamed for the Tyler accidents", individual subroutines being given names taken from the program code. Inevitably, the recommendations offered at the end of the paper concentrated chiefly on general points about the development process that had been used: they focused on documentation, software specifications, avoidance of dangerous coding practices, audit trails, testing, formal analysis and similar concerns. No specific recommendations could be made about the software artifact itself, because there was no standard normal design to which such recommendations could refer. The errors of the Therac-25 software engineers were, regrettably, repeated twenty five years later.

12. The Challenge Of Software System Structures

Almost every realistic computer-based system is complex in many planes. This complexity is a major obstacle to the design of a system, and to communicating and explaining its structure. It is also, therefore, a major obstacle to the evolution of normal designs.

More properly, we should speak of many structures, not just of one. The system provides many features, each one an identifiable unit of functionality that can be called into play according to circumstances and user demands. Each feature can be regarded as realised by an associated contrivance. The components of the contrivance are problem domains and parts or projections of the system's software, arranged in a suitably designed configuration within which they interact to fulfil the functionality of the feature.

The features will inevitably interact: they may share problem domains and software resources; several may be active in combination at any time; and their requirements may conflict. These interactions and combinations will itself induce a distinct structure, demanding careful study and analysis both in terms of the requirements they are expected to satisfy and in terms of the implementation. This *feature interaction problem* [Zave 04] came to prominence in the telephone systems of the late twentieth century, where interactions of call processing features such as call forwarding and call barring were seen to produce surprising and sometimes potentially harmful results. The structure of these interactions might be captured, for example, in graphs of the kind used in Distributed Feature Composition [Jackson 98]. The feature interaction problem hugely increased the costs and difficulties of building telephone system software. It soon became apparent that the same problem was inherent in complex systems of every kind.

The system as a whole, regarded as a single contrivance, will have several contexts of operation. For example, a system to control lifts in a building containing offices, shops and residential apartments must take account of the different demand patterns imposed from hour to hour and from day to day by this usage. There will also be extraordinary operational contexts such as emergency operation under control of the fire department, operation under test control of a representative of the licensing authority, and operation during periodic maintenance of the equipment. If an equipment fault is detected the system must take action to mitigate the fault if possible—for example, by isolating the faulty subset of the equipment or by executing emergency procedures to ensure the safety of any users currently travelling in the lift cars. These contexts are not necessarily disjoint: for example, a fault may be detected during testing, and the safety of the test engineer must be ensured.

The properties and behaviours of the individual problem world domains—the lift shafts, the floors, the doors, the lift cars, the users, the hoist motors—must be analysed in the development process. The design of the contrivance realising each feature must take account of the relevant properties and behaviours of those problem domains that figure as components in its design. It may be necessary for the executed software to embody software objects that model the problem domains. A domain will then have a model for each feature in which it participates, and these models must be reconciled and in some way combined.

The implemented software itself will have some kind of modular structure, its parts interacting by the mechanisms provided by the execution infrastructure and the features of the programming languages used. The extreme malleability of software, which is not shared by physical artifacts, allows the implementation structure to differ radically from the other structures of the system. The relationships among these structures may then be mediated by transformations.

All of these structures must be somehow captured, designed, analysed, reconciled and combined. The task can certainly not be performed successfully by simplistic generalised techniques such as top-down decomposition or refinement. Nor can it be evaded by the kind of reductionist approach that fragments the system into a large collection of elements—for example, events or objects or one-sentence requirements. In general, this is an unsolved problem. In a normal design task a satisfactory solution has been evolved over a long period and is adopted by the specialist practitioners. Where no normal design is available, and major parts and aspects of the design task are unavoidably radical, the full weight of the difficulty bears down on the designer. This difficulty is a salient feature of the design of many computer-based systems.

13. Forgetting the Importance Of Structure

In an earlier era, when computers were slower, programs were smaller and computer-based systems had not yet begun to occupy the centre of attention in the software world, program structure was recognised as an important topic and object of study. The discipline of structured programming, as advocated [Dijkstra 68] by Dijkstra in the famous letter, was explicitly motivated by the belief that software developers should be able to understand the programs they were creating. The key benefit of the structure lay in a much clearer relationship between the static program text and the dynamic computation. A structured program provides a useful coordinate system for understanding the progress of the computation: the coordinates are the text pointer and the execution counters for any loops within which each point in the text is nested. Dijkstra wrote:

> "Why do we need such independent coordinates? The reason is—and this seems to be inherent to sequential processes—that we can interpret the value of a variable only with respect to the progress of the process."

A structured program places the successive values of each state component in a clearly defined context that maps in a simple way to the evolving state of the whole computation. The program structure tree shows at each level how each lexical component—elementary statement, loop, if-else, or concatenation—is positioned within the text. If the lexical component can be executed more than once, the execution counter for each enclosing loop shows how its executions are positioned within the whole computation. The programmer's understanding of each part of the program is placed in a structure of nested contexts that reaches all the way to the root of the structure tree.

Other structural notions beyond the control structure discipline advocated for structured programming—for example, layered abstractions and Naur's discipline [Naur 69] of programming by action clusters—were explored, always with the intention of simplifying and improving program development by supporting human understanding. The primary need was for intelligible structure. Rigorous or formal proof of correctness would then be a feasible by-product, the structure of the proof being guided by the structure of the designed program.

Work on the formal aspects of this plan proved brilliantly successful during the following years. Unfortunately, however, it was so successful that the emphasis on human understanding gradually faded, and many of the most creative researchers simply lost interest in the human aspects. It began to seem more attractive to allow the formal proof techniques to drive the program structuring. By 1982 Naur was complaining [Naur 82]:

> "Emphasis on formalization is shown to have harmful effects on program development, such as neglect of informal precision and simple formalizations. A style of specification using formalizations only to enhance intuitive understandability is recommended."

For small and simple programs it was feasible to allow a stepwise proof development to dictate the program structure. Typically, the purpose of the program was to establish a result that could easily be specified tersely and formally: the proof was then required to establish this result from the given precondition. But Naur's complaint was justified. It is surely a sound principle that human understanding and intuition must be the foundation of software engineering, not a primitive scaffolding that can be discarded once some formal apparatus has been put in place.

The success of the formalist enterprise has led to a seriously diminished interest in structural questions and concerns, and in the role of human understanding, just when they are becoming more important. The constantly increasing complexity of computer-based systems cannot be properly addressed by purely formal means. Forty five years ago, the apparently intractable problem of designing and understanding program flowcharts yielded to the introduction of structure in the service of human understanding. Managing and mastering the complexity of the systems being built now and in the future demands human understanding: formal techniques are essential, but they must be deployed within humanly intelligible structures.

References

[Benz 86] http://www.youtube.com/watch?v=9MjvAqF9Tgo (last accessed 3rd July 2010).

[Benz Patent 86] http://www.unesco-ci.org/photos/showphoto.php/photo/5993/title/benz-patent-of-1886/cat/1009 (last accessed 4th July 2010).

[Bogdanich 10] Walt Bogdanich; The Radiation Boom; New York Times January 23 and January 26 2010.

[Brooks 10] Frederick P Brooks, Jr; *The Design of Design: Essays from a Computer Scientist*; Addison-Wesley, 2010.

[Buxton 70] J N Buxton and B Randell eds; Software Engineering Techniques; Report on a conference sponsored by the NATO SCIENCE COMMITTEE, Rome, Italy, 27th to 31st October 1969; NATO, April 1970.

[Dijkstra 68] E W Dijkstra; *A Case Against the Go To Statement*, EWD 215, published as a letter to the Editor (Go To Statement Considered Harmful): Communications of the ACM Volume 11 Number 3, pages 147-148, March 1968.

[Dijkstra 89] E W Dijkstra; Reply to comments on *On the Cruelty of Really Teaching Computer Science*; Communications of the ACM Volume 32 Number 12, December 1989, pages 1398-1404.

[Godden 10] http://nisee.berkeley.edu/godden/godden_intro.html (last accessed 25 June 2010).

[Holloway 99] C Michael Holloway; *From Bridges and Rockets, Lessons for Software Systems*; Proceedings of the 17th International System Safety Conference, Orlando, Florida, pages 598-607, August 1999.

[JacksonD 07] Daniel Jackson, Martyn Thomas and Lynette I Millett eds; *Software for Dependable Systems: Sufficient Evidence?* Report of a Committee on Certifiably Dependable Software Systems; The National Academies Press, Washington DC, USA, 2007, page 40.

[JacksonM 98] Michael Jackson and Pamela Zave; *Distributed Feature Composition: A Virtual Architecture For Telecommunications Services*; IEEE Transactions on Software Engineering Volume 24 Number 10, pages 831-847, Special Issue on Feature Interaction, October 1998.

[Krebs 08] Brian Krebs; *Cyber Incident Blamed for Nuclear Power Plant Shutdown*; Washington Post, June 5, 2008.

[Leveson 93] Nancy G Leveson and Clark S Turner; *An Investigation of the Therac-25 Accidents*; IEEE Computer Volume 26 Number 7, pages 18-41, July 1993.

[Leveson 95] Nancy G Leveson; *Safeware: System Safety and Computers*; Addison-Wesley, 1995.

[Myers 09] Natasha Myers; *Performing the Protein Fold;* in Sherry Turkle; Simulation and Its Discontents, MIT Press 2009, pages 173-4.

[Naur 69a] Peter Naur and Brian Randell eds; Software Engineering: Report on a conference sponsored by the NATO SCIENCE COMMITTEE, Garmisch, Germany, 7th to 11th October 1968; NATO, January 1969.

[Naur 69b] Peter Naur; *Programming by Action Clusters*; BIT Volume 9 Number 3, pages 245-258, September 1969; republished in: Peter Naur; Computing, A Human Activity; ACM Press, 1992, pages 335-342.

[Naur 82] Peter Naur; *Formalization in Program Development*; BIT Volume 22 Number 4, pages 437-452, December 1982; republished in: Peter Naur; Computing, A Human Activity; ACM Press, 1992, pages 433-449.

[Polanyi 58] Michael Polanyi; *Personal Knowledge: Towards a Post-Critical Philosophy*; Routledge and Kegan Paul, London, 1958, and University of Chicago Press, 1974.

[Polanyi 66] Michael Polanyi; *The Tacit Dimension*; University of Chicago Press, 1966; ; with foreword by Amartya Sen, 2009.

[Risks Digest] catless.ncl.ac.uk/risks/ (last accessed 1st July 2010).

[Vincenti 93] Walter G Vincenti; What Engineers Know and How They Know It: Analytical Studies from Aeronautical History; The Johns Hopkins University Press, Baltimore, paperback edition, 1993.

[Zave 04] P Zave; http://www2.research.att.com/~pamela/fi.html; *FAQ Sheet on Feature Interaction*; AT&T, 2004.

Paper for the symposium in honour of Bertrand Meyer, Zurich, November 2010
Associated book to be published by Springer verlag, 2011