Engineering and Software

(Draft of 30th January 2009)

Michael Jackson, The Open University
jacksonma@acm.org

Abstract Software development has long aspired to merit the status of a professional engineering discipline like those of the established engineering branches. This paper discusses this aspiration with particular reference to software-intensive or *computer-based* systems. Some opportunities are pointed out for learning important lessons from the established branches. These lessons stem above all from the highly specialised nature of traditional engineering practice. They centre on the crucial distinction between *radical* and *normal* design, the content of normal design practice, and the social and cultural infrastructures that make effective specialisation possible.

1. INTRODUCTION

In the earliest years of software development some notable successes were achieved. Perhaps the most remarkable was the development of business data processing systems for J Lyons, an English company that blended its own teas and baked its own breads and cakes for sale in its restaurants and teashops. As early as 1947, the company understood the potential benefit of computerised data processing at a time when few people imagined that workable computers would eventually become commercial products available for purchase and use. The company contributed to the funding of the EDSAC computer being built at Cambridge University, and on the basis of the EDSAC design they developed and manufactured their own electronic computer, LEO 1. The name LEO was an acronym—Lyons Electronic Office—and the computer was intended to run business data processing applications that Lyons own staff would design and program. The first application, bakery valuation, computed the money value of the cakes, pies and pastries produced by the company's bakeries and distributed to their teashops, restaurants and other sales channels: it ran successfully on 16th November 1951 and every week thereafter for many years. In 1954 the J Lyons payroll application began weekly operation, and from December 1955 the payroll of the Ford Motor Company's Dagenham plant ran on LEO as a service outsourced by Ford to J Lyons. By the end of 1956 [Caminer97], "LEO was processing a representative load of office applications—payroll, distribution, sales invoicing, accounting and stock control—and, at the same time, expediting the physical operations of Lyons and providing timely information for remedial managerial action."

A remarkable system of a very different kind was SAGE, the Semi-Automatic Ground Environment system designed to defend North America against bombing attack by aircraft fleets of potentially hostile powers. Based on an earlier prototype system, and using AN/FSQ-7 and AN/FSQ-8 computers specially developed by IBM, SAGE collected and processed radar inputs for display to operators, and helped the operators to react appropriately and to communicate indirectly with interceptor aircraft. The system first became operational in 1959 and ceased operation nearly twenty five years later in 1983. The system was never put to the test by a real hostile attack, and became obsolete quite early in its life when the perceived threat from long-range missiles superseded that from bombers; but its development was judged to have been very successful, and was certainly extremely ambitious for its time. Herbert D Benington was one of the leaders of the software development for SAGE, and described the work in a 1956 paper [Benington56]. When his

1956 paper was republished [Everett83] in 1983, Benington added a foreword in which he reflected on his experiences. He was in no doubt about the foundation of the project's success:

> "It is easy for me to single out the one factor that I think led to our relative success: we were all engineers and had been trained to organize our efforts along engineering lines. We had a need to rationalize the job; to define a system of documentation so that others would know what was being done; to define interfaces and police them carefully; to recognize that things would not work well the first, second, or third time, and therefore that much independent testing was needed in successive phases; to create development tools that would help build products and test tools and to make sure they worked; to keep a record of everything that really went wrong and to see whether it really got fixed; and, most important, to have a chief engineer who was cognizant of these activities and responsible for orchestrating their interplay. In other words, as engineers, anything other than structured programming or a top-down approach would have been foreign to us."

In the later 1950s, and in the first half of the 1960s, there were still successes, but now there were very many failures too. As machines became more affordable there was a need for many more programmers, and inevitably few, if any, of the new recruits were trained engineers like Herbert Benington or deeply experienced business analysts like David Caminer. Jules Schwartz [Buxton70] colourfully described the later recruitment to the SAGE project:

> "People were recruited and trained from a variety of walks of life. Street-car conductors, undertakers (with at least one year of training in calculus), school teachers, curtain cleaners and others were hastily assembled, trained in programming for some number of weeks and assigned parts in a very complex organization."

Whatever the reasons, by the early 1960s there was widespread talk of a 'software crisis'. It was commonly said that software was full of errors; that software systems did not deliver the functionality that was needed; and that software projects too often grossly exceeded their budgets and schedules—many very expensive software projects even failing to deliver anything usable at all. Something had to be done.

In 1967 the NATO Science Committee established a Study Group on Computer Science. The Study Group recommended the holding of a working conference on 'Software Engineering', and two NATO conferences were held: one in Garmisch and one in Rome. The introduction to the report of the first conference [Naur69] states the motivation clearly:

> "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."

The motivation was clear: software developers should learn from engineers. What exactly they should learn was less clear. For some people, 'Software Engineering' meant simply an improved and more careful approach to the programming task: programmers should be more meticulous; they should pay more attention to design, and should check their programs before executing, or even before compiling, them; they should abandon the 'code-and-fix' approach that had caused so much trouble. For others, there were more specific lessons to be learned.

Some saw software development as an essentially industrial production process that could, and should, be subjected to fine-grain industrial disciplines of the kind that Frederick Taylor had devised and promoted in the early 20th century under the title "scientific management". Unsurprisingly, software workers, like factory workers, were inclined to resist the imposition of this kind of managerial rule. In a book [Kraft77] published in 1977, the sociologist Philip Kraft even argued that the introduction of structured programming was an attempt by managers to control their workers by imposing a form of Taylorism on them:

> "Until human programmers were eliminated altogether, their work would be made as machine-like—that is, as simple and limited and routine—as possible. Briefly, programmers using structured programming would belimited to a handful of logical procedures which they could use—no others were permitted. They could call only for certain kinds of information ... They could not, for example, call for information not contained in the original data set assigned to them. ... in this way, the ability to produce large and complex systems has not been impaired, only the opportunity of the average programmer to produce them."

Watts Humphrey [Humphrey00] softened the harsh wind of Taylorism by inviting programmers to be their own managers, but his view of their work was consciously based on Taylor's principles:

> "The principal difference between manual and intellectual work is that the knowledge worker is essentially autonomous. That is, in addition to deciding how to do tasks, he or she must also decide what tasks to do and the order in which to do them. The manual worker commonly follows a relatively fixed task order, essentially prescribed by the production line. So studying and improving the performance of intellectual work must not only address the most efficient way to do each task but also consider how to select and order these tasks. The is essentially the role of a defined process and a detailed plan. The process defines the tasks, task order, and task measures, while the plan sizes the tasks and defines the task schedule for the job being done."

Less harshly yet, some people saw the software problem as an interplay of technical and managerial aspects, still with a strong emphasis on the definition and management of the development process. There was general agreement that software development should become like engineering, but little agreement about what that would mean. The present author has suggested [Jackson82] that specialisation is a basic characteristic of successful engineering, but that suggestion was not related in any detailed way to the practices of the established engineering branches. The present paper aims to repair that omission to some extent.

In an insightful paper [Shaw90] published in 1990, Mary Shaw described the evolution of the established branches of engineering from their beginnings in crafts and cottage industries. Chemical engineering—which she took as her primary example—evolved in three stages. An industrial process emerged in the late 18th century, under the commercial pressure for more efficient production of the alkali needed for the manufacture of glass, soap and textiles. Early in the 19th century Dalton's atomic theory provided a scientific foundation by explaining the underlying chemistry. In the mid-19th century G E Davis recognised that chemical manufacturing depended on a core set of basic operations, later called *unit operations*, of which every manufacturing process in use was composed.

Shaw also points out the distinction between routine and innovative design, and the crucial value of a handbook in which known good designs and their applicable parameters are recorded and codified. Finally, among her recommendations for the steps necessary for

software development to become a true engineering discipline, she includes the development of specialisation: *internal specialisation* in the technical content of program design "as the core of software grows deeper"; and *external specialisation* in "applications that require both substantive application knowledge and substantive computing knowledge."

The intent of this paper is to build on some of these insights, especially Shaw's, and to draw some further lessons from a consideration of the practice of the established engineering branches. The specific lessons drawn are applicable chiefly to a particular, but very broad, class of system—what are often called *computer-based systems*. The central role of specialisation, and its essential preconditions, are discussed, and a particular dimension of specialisation—specialisation by *artifact*—is identified that has played a vital role in achieving dependably successful engineering products.

2. COMPUTER-BASED SYSTEMS

It is in computer-based systems, or CBSs, that software development can learn the deepest lessons from the traditional branches of engineering, and can gain most from learning them. In a CBS, sometimes called a *software-intensive system*, the computer's role is to interact with the physical world—that is, with the natural world of the universe and the physical products of human engineering, with human beings themselves, and with other CBSs. The *machine* that we produce as software developers is a computer executing the software we develop. It plays its role in its *problem world*—the relevant parts of the physical world—functioning as one part among several, monitoring and controlling the behaviours of the other parts and establishing and maintaining relations among them.

For an avionics system the earth's atmosphere is a part of its problem world, along with the aircraft itself, the pilot, the airport runways, the passengers, the air traffic control system, and so on. For a heart pacemaker system the problem world contains the human patient, regarded from both behavioural and physiological points of view, the external devices by which the pacemaker's behaviour can be monitored and adjusted, and the operators of those devices. For a theatre booking system the problem world contains the theatres, the potential audiences, credit cards, the physical tickets to be issued, and so on. For a medical radiation therapy system the problem world includes the patients, the radiation equipment, the equipment operators, the medical staff, and the movable bed on which patients lie and are precisely positioned for treatment. For a system to control the lifts in an office building the problem world contains the electrical and mechanical lift equipment, the arrangement of the building's floors, the behaviour of individual users, and their group behaviour evidenced in patterns of traffic between the building's floors.

These parts, or *domains*, of the physical world are heterogeneous, varying greatly in the inherent properties and behaviours they exhibit, both in general and in their participation in different systems. The relevant capabilities and propensities of an airline pilot flying an aircraft are different from those of the same person engaged in booking a theatre seat. The properties of the earth's atmosphere that are important in an avionics system are different from those that matter in a system to control fuel injection in a motor car. For each system the developers must investigate and analyse the properties of the problem world domains and of their interactions with each other and with the machine to be built: they must devise a machine whose interactions with the domains to which it is directly connected will ensure that the system requirements—the purposes of the system—are satisfied. If they misunderstand the requirements, or misunderstand the behaviours and properties of the problem world domains, they will fail as surely as if they produce erroneous programs. This possibility of failure is not confined to control systems: an information system, too, will fail if its developers have misunderstood how the phenomena about which information is to be

produced are related by the domain properties to the phenomena directly accessible to the machine.

The success or failure of the developed software in a CBS, then, is not to be judged by its satisfaction of a formal specification of machine behaviour, but by its observable effects in the problem world. The theatre booking system is successful if people can book seats conveniently, if duplicate tickets for the same seat at the same performance are never issued, if better seats at each price are sold first, if credit cards are correctly charged, and so on. The radiation therapy system is successful if the patients receive their doses of radiation exactly as prescribed, if the equipment is efficiently utilised, and if safe operation of the equipment is ensured.

This character of the development of a CBS is shared by the work of the established branches of engineering. G F C Rogers defined [Rogers83] engineering as

> "the practice of organising the design and construction of any artifice which transforms the physical world around us to meet some recognised need,"

The artifice, or artifact, constructed by CBS software developers is the *machine*—the computer executing the software; the physical world around us is the *problem world*; and the system *requirements* are the recognised need. In this fundamental sense, software development of a CBS is indeed engineering, and should be able to profit from what engineers have learned over their long history.

3. SPECIALISATION BY ARTIFACT

An obvious aspiration has been to enrol software engineering as one new member of the established college: automotive engineers develop motor cars, and naval engineers develop ships: clearly, people who develop software should be enrolled as software engineers. This aspiration is based on the identification of the software itself, considered in its narrow confines within the computer, as the artifact produced by CBS software development: the product of software development is identified with the program text.

Certainly, from a pure programming point of view this aspiration to a single engineering discipline seems to make good sense: the software of practically all computer-based systems has much in common. The program text describes the computer's internal behaviour and states by which it can be brought to exhibit the desired behaviour at its interface with the problem world. From a pure programming point of view, this internal behaviour, and the technical challenge of designing it and describing it in a program text, are of direct and intense interest. The programmer must take proper account of the relevant algorithms and data structures, the practicalities of the operating system and programming language, the allocation of the computer's resources, the possible failures of the hardware and software infrastructure on which the program is to be executed, and many other matters that engage the attention of software engineers.

Nonetheless, in the case of computer-based systems, and perhaps of some other software systems too, this identification of the product with the program text is misplaced: software engineering for computer-based systems is not one aspiring engineering discipline, but many. The real artifact produced by the software developers is the combined behaviour of the machine and of the physical problem world: not only at the interface where they meet and interact, but also in their respective hinterlands remote from that interface. The pure programming point of view does not capture the essential purpose of the work: the internal computations signify nothing except as an instrumental means to achieve the machine's external behaviour; and the external behaviour signifies nothing except as an instrumental

means to achieve the purposes of the system in its problem world. The meaningful artifact of a CBS is the whole system, considered with a primary focus on the problem world. The huge variety in the physical problem worlds of CBSs, together with the variety of their required functions in those worlds, is then seen to constitute a huge variety in the artifacts of CBS software development. From this point of view, the most conspicuous practical characteristic of the established branches is their very plurality. There is not just one established branch of physical engineering. We should not expect, then, that there should be just one branch of CBS software development. We should expect a broad structure of specialisation according to the different classes of system—or subsystem—to be developed.

Certainly, there are common intellectual principles shared by all engineers, and both the 'hard' science of physics and chemistry and the 'soft' behavioural sciences are of shared relevance because of engineers' intense concern with the physical and human world. However, most of this common ground lies far below the working practice of engineering, which is concerned with particular outcomes in particular situations. As more scientific knowledge becomes available it informs the possibilities of innovation; but engineering practice is concerned with the design and analysis of particular artifacts. At the level of particularities, the artifacts and the associated problem worlds of the different engineering branches are very different. This is why they specialise. Civil engineers do not design chemical plants, and automobile engineers do not design ships or networks for the distribution of electrical power. If we hope to emulate their successes, and achieve the levels of quality and dependability that we have come to expect in their products, we must study and emulate their degree and manner of specialisation.

Specialisation in the established branches has many dimensions, and software development can legitimately claim to exhibit parallel or analogous specialisations in some of those dimensions. They have specialisations by theory, such as control and structural engineering, and fluid dynamics; software has concurrency, type theory and complexity theory. They have specialisations by technology, such as micro-electronics and welding; software has functional and object-oriented programming. They have specialisations by materials, such as pre-stressed concrete and electrically conducting plastics; software has Java and PHP and SQL. All of these specialisations are important, and all feed into the overall success of the established branches, and into the successes of software developers.

Where software development falls short in specialisation is in the most fundamental dimension of all: specialisation by artifact. The other dimensions are important, but the crucial dimension is specialisation by artifact. Only the specialist in a particular class of artifact—motor cars, or dams, or electric motors, or disk drives—can bring together and understand all the particular factors that determine the quality of the artifact and its value for its designed purpose, and make judicious choices about the interactions of those factors. The extent to which engineering rests chiefly on a foundation of science is debatable; but science, quite certainly, is not enough, even when expanded into its own branching specialisations. The full effects of applicable scientific laws on a particular artifact in the particular situations it will encounter are in principle incalculable: the phenomena that might affect the outcomes cannot be exhaustively enumerated; nor can their effects be quantified with enough precision for the engineer to know with certainty which laws will have the largest effects and will thus combine to dominate the outcome. This is why one must not expect a group of physicists, however brilliant and however perfect their understanding of the laws of physics, to be able to design and build a good motor car or aeroplane or bridge. In this difficulty, only the engineer specialised by artifact can address the totality of what the 'end-user' of the artifact (who may, of course, be another engineer for whose artifact the first engineer's artifact is one of several components) can expect to experience.

Software development does show some specialisations by artifact, but too few of them are found in the development of CBSs. For example, one very successful specialisation is in SAT solvers, which solve the completely abstract problem of finding an assignment of values to variables that satisfies a given predicate; another is in model checking, which is again an entirely abstract problem. Others are in compilers, file systems, relational database systems, and networking, in all of which the problem world is approximately bounded by the world of other software systems and of hardware devices—for example, disk drives—specifically designed for high reliability and for interacting conveniently with software. In the development of CBSs, there is clear evidence of specialisation in some kinds of computer-controlled system or subsystem that work very well and very reliably: these probably include modern lift-control systems provided by the major manufacturers, ATMs, credit-card charging software used by major e-commerce websites, ABS braking systems in cars, and others. However, artifact specialisation is more than the production of successful examples: it is essentially the product of an evolved culture.

4. THE GROWTH OF SPECIALISATIONS

Specialisation is fundamental to intellectual progress. In the earliest stages outstandingly able people can master all the existing knowledge of a field. As knowledge increases, acquired from experience or from a deepening understanding of an underlying science, the sum of available knowledge in the field becomes more than any one person, however able, can master. Eventually, each able individual must choose between becoming a generalist, who knows less and less about more and more, or a specialist, who knows more and more about less and less. In the first half of the 19th century, the great engineer Isambard Kingdom Brunel pursued a masterful career as an engineer of railways, bridges and tunnels, large ocean liners, artillery pieces, and modular, transportable, military hospitals. Today, with the exception of bridges and tunnels, both of which fall within the competence of some practising civil engineers, knowledge in each one of these artifact categories has developed to a point at which only a specialist can be fully competent: Brunel himself would be unable to master them all.

The pressure to specialise is not felt only by individuals. In fact, is it only in the earliest stages that specialisation is the possession of individuals. The touchstone of this kind of specialisation is that the artifact knowledge becomes the valued possession of a community rather than of individual people. Individual people pursue specialised careers; companies specialise their products; there are research journals and educational curricula; there are careful descriptions, models, and sometimes even repositories or museums of notable exemplars—all of these focused on the design on the specialised artifact classes in question. The community works to increase its knowledge and improve the quality of its products, often under pressure of competition among individuals or companies within the community.

For the most part, specialisations emerge in response to commercial opportunities, technical opportunities and challenges, and sometimes legal and social pressure arising from a high incidence of failures in a particular class of artifacts. The specialisation in compilers arose in response to a commercial opportunity of the early 1960s when computer hardware architecture was still hugely varied. Some computers used a machine order code in which each instruction specified one address; some used two; some used three addresses. Word length could be 12, 16, 24, 36, 48 or almost any other even number of bits. There were different schemes for structuring and addressing primary and secondary storage; some machines had a built-in hardware stack. The manufacturers needed to be able to supply a free Fortran compiler to each customer who bought or rented one of their very expensive machines. Companies like Digitek and Computer Sciences Corporation saw the commercial

opportunity and rushed to exploit it. Over the next twenty years or more, compiler construction became a notable specialisation. Applied theory of grammars, parsing, code generation and optimisation developed along with a recognition of the accepted decomposition of a compiler: lexical analyser, symbol table, syntax analyser, semantic routines to be associated with nodes of the syntax tree, global and peephole optimiser. The field has continued to develop: faster machines allow just-in-time compilation and compilation to a bytecode for interpretation by a virtual machine; integrated development environments integrate compilation into program design, editing, and debugging, along with the use of comprehensive module libraries. The result was that compilers eventually exhibited high quality and reliability, and were easily capable of compiling programs in languages that in earlier years would have been thought impossibly difficult to compile.

5. THE BENEFITS OF ARTIFACT SPECIALISATION

In engineering, the primary benefit of artifact specialisation is the emergence, adoption and evolution of *normal design* for the artifacts. Following Constant [Constant80], Vincenti describes [Vincenti93] normal design:

> "[Normal design is] the improvement of the accepted tradition, or its application under new or more stringent conditions. ... The engineer engaged in such design 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.
>
> "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 (eg, 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."

Normal design in this sense is conspicuous in modern cars, in large passenger aircraft, in mobile phones, in television sets, and in many other well designed and reliable artifacts with which we are familiar. Certainly, there are differences between one manufacturer's products and another's, and between this year's models and last year's. There are also differences between subclasses within one class—in cars, for example, between people carriers and five-door hatchbacks. But these differences are less significant than the similarities, which have emerged from the gradual evolution of normal design and its adoption by the specialised engineering communities.

Vincenti contrasts normal design with *radical design*:

> "In radical design, 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."

A clear example of radical design is Karl Benz's Patent Motorwagen of 1886, arguably the first successful motor car. (A very careful replica of the car has been built, and a good selection of photographs is available at [Benz86]). The car was completely open to the elements; there were three wire-spoked wheels with solid tyres, and an unsprung single front

wheel; the driver sat in the centre and steered with a small tiller. The engine was started by manually turning the large horizontally mounted flywheel; it was lubricated by a drip feed; the single crank was unenclosed; the drive to the rear wheels was by belt and pulley, and there was no gearbox to vary the ratio of engine speed to road speed. It was a remarkable achievement by Benz's wife, Berta, to drive this car 65 miles from Mannheim to Pforzheim in the course of a single day. Benz had succeeded in solving the radical design problem exactly as Vincenti characterised it: he had 'designed something that functioned well enough to warrant further development'.

In the following years, the growing community of specialised automobile engineers developed their products to the point at which by about 1920 they could be said to embody a normal design: an electric starter for the engine; four sprung wheels, all with pneumatic tyres and brakes; a closed cab with the driver sitting on the offside and controlling front-wheel Ackerman steering geometry by a raked steering wheel; a standard layout of the drive train, including a friction clutch, three-speed or four-speed gearbox, longitudinal propeller shaft, and a rear axle casing enclosing a differential driving the rear wheels through half-shafts.

The successful evolution of a normal design does not mark the end of innovation. On the contrary: it provides a stable and dependable foundation on which further innovations can be developed. In automobile engineering the last eighty or ninety years have seen continual incremental innovation within the established but still evolving normal design. The overall vehicle structure has been improved: for example, the separate body and chassis frame have been replaced by a unitary pressed steel body that combines the functions of both, and the front beam axle has been replaced by independent front suspension. Individual components and subsystems have been improved by the introduction of tubeless tyres, automatic gearbox, fuel injection and many other new features.

6. THE CONTENT OF NORMAL DESIGN

'Design', of course, is both a noun and a verb. The phrase 'normal design' denotes both the standard configuration and component structure of the designed artifact in its particular class, and also the practical disciplines that its designers are expected to follow in developing each new instance of the class. These practical disciplines do conform to some very general notions of engineers' responsibilities that are common to all or most engineering branches, and they rest on a common basis of scientific knowledge of the physical world; but their most significant practical content is special to each product class and is largely focused on its component structure at all levels.

This is what makes possible the reliable division of a design project among a group of several designers. Vincenti outlines parts of the typical project structure in aeroplane design as "Major-component design—division of project into wing design, fuselage design, landing-gear design, electrical-system design, etc" and "Subdivision of areas of component design ... according to engineering discipline required (eg aerodynamic wing design, structural wing design, mechanical wing design)." This project structuring is, of course, closely tied to the normal product structure. While some details of the decomposition into work assignments may be open to doubt, the general shape is clearly mandated by the normal design.

The central point here, which deserves repetition, is this: the decomposition of artifact functionality in a normal design is not *ad hoc*. Parnas rightly identified [Parnas78] the importance of "the decomposition of programming projects into work assignments (modules)." In a normal design discipline this decomposition is already broadly known, and the assignment of the components to individuals or groups is likely to be determined by their known specialisations at the component level. Any significant departure from the established

decomposition into components arranged in the standard configuration, is to be recognised as an innovation, and as the introduction of an element of radical design that inevitably brings with it an increased risk of failure.

The development work within each part of the project is quite tightly constrained by the standard, normal, design. In extreme cases the designer is choosing from a small set of design options and fixing parameters from a well-defined range. The chosen design must be validated, eventually by testing, but in the earlier design stages by analysing the properties of each successively proposed design version to determine whether the choices it embodies would enable the design to satisfy its requirements. This analysis is typically mathematical, and rests on two foundations: one is scientific knowledge of the physical phenomena involved; the other is a set of known procedures for sufficiently accurate analysis of significant properties of versions that fall within the bounds of the established normal design. Without the scientific knowledge, reliable analysis is impossible; it may also be impossible if the design to be analysed is unprecedented and arbitrarily chosen. Theoretical scientific knowledge is concerned with physical phenomena such as mechanical forces and chemical processes acting in isolation. Engineering requires a good enough understanding of specific situations and artifacts in which, inescapably, many different forces and processes are at work. This good enough understanding can be achieved only by analytical models which are simultaneously good enough approximations to the enormously complex reality and also tractable by the available mathematics and science. It is a crucial characteristic of normal designs that they are susceptible of adequate standard analysis in this way.

7. CBS COMPONENT STRUCTURE

In developing, understanding, or analysing a CBS, it is necessary to consider the *machine*—the computer or computers executing the developed software—not alone, but in conjunction with the *problem world domains*—the other parts of the whole system. The end product of the software development is not the software alone: it is the machine and the problem world, and the behaviour that is the product of their interactions. From the point of view of the whole system, the machine makes sense only in its role of monitoring and controlling the problem world. Without knowledge of the problem world and the requirement, the machine must appear as an entirely arbitrary device, imposing an inexplicably obscure regime on the electrical and magnetic phenomena at its ports.

The same perspective is necessary when we consider the decomposition of a CBS into its constituent components. A component of a CBS is not a software module, a fragment of the machine in the system: it is an assemblage whose parts are a fragment of the machine, a fragment of the problem world, and a fragment of the system requirement. Conceptually, each component has its own machine, interacting with some subset of the problem domains in the physical world and responsible for satisfying some part of the requirement of the whole system. The problem domains of the components are not, in general, disjoint: each domain can play different roles in different components, each of the components depending on different properties of all or part of the same domain. (This multiplicity of roles of a problem world domains is no different in a car. Different properties of the earth's atmosphere play their parts in the functioning of the tyres, the engine cooling system, the air conditioning, the fuel injection and combustion, and the aerodynamic performance of the body.)

Nor are the component machines themselves likely to remain disjoint in the software as it is finally implemented: in addition to communicating by shared problem domains, they will need to communicate within the machine, and they will certainly contend for shared computer resources such as RAM and disk access. Furthermore, software is intangible and

malleable, and component machines can be dismembered, recombined and reconstituted almost at will. The system requirement, also, is decomposed into requirements of the individual components. This decomposition, too, is not disjoint: the component requirements can interfere with each other in many ways, including cooperation, but not excluding outright conflict.

The general conception of such components can be illustrated, superficially, in the context of a system to control the lifts in an office building. The overall purpose of the system is to provide convenient, efficient and safe lift service in response to users' requests. The requirement of convenience and efficiency can be separated from the requirement of safety. The *lift_service* component operates the electrical and mechanical equipment to provide service in response to users' requests. The *lift_safety* component continually monitors the behaviour of the equipment, including its reactions to the commands issued by *lift_service*, to detect any evidence that the equipment has developed a fault, whereupon the *lift_safety* component ensures user safety, for example, by applying the emergency brake to lock the lift in the shaft to prevent it from falling freely. Another component, *lobby_display*, may be responsible for maintaining the display in the ground floor lobby that shows the current positions of the lifts.

We may also suppose that for convenience and efficiency it is necessary to apply different priorities at different times—for example, distinguishing the traffic demand patterns of weekends from weekdays, and weekday morning, evening and lunchtime rush hours from other times. The priorities must be specified by the building manager, changed when necessary, and appropriately applied by the machine in scheduling responses to competing requests. In a decomposition of our original *lift_service* component, we may now recognise two components: *edit_priority*, which is the specification and editing of priority schemes by the manager, and *priority_lift_service*, which is the provision of lift service in accordance with the currently chosen scheme. These two components share a newly introduced domain: the data structure *schemes*, whose values represent the edited priority schemes.

For the whole system, the *schemes* data structure is a local variable of the undecomposed machine; but for each of the two components, it is a part of the component's problem world. The problem domains for *edit_priority* are *manager* and *schemes*; for *priority_lift_service* they are *schemes* together with the electrical and mechanical lift equipment, the request buttons, the floors and the users. Introducing the *schemes* data structure has allowed the *priority_lift_service* function to be decoupled from the *edit_priority* function. The introduction of the data structure exploits the characteristic power of computers to store and manipulate data. Its introduction is analogous to the introduction of the propeller shaft in a car with front engine and rear wheel drive. The propeller shaft both connects and decouples the engine and rear axle components. It separates the function of converting fuel energy into rotary motion from the function of applying rotary motion to the wheels; it also forms a common subcomponent, conveying the rotary motion from one to the other.

This briefly sketched decomposition also illustrates how the same problem domain plays different roles in different components, the components relying on different—and even mutually contradictory—domain properties. For example, for the *priority_lift_service* component the lift equipment must be assumed to be functioning correctly: when the machine sets the direction *up* and turns the motor *on*, the lift rises in the shaft; when the lift car reaches a floor the corresponding floor sensor is set *on*; and so on. These are the properties necessary for provision of the service to users. For the *lift_safety* component, however, the lift equipment may possibly be faulty, and its various failure modes are associated with phenomena that the machine can monitor: if the motor is not functioning, the lift car does not rise when expected, and the floor sensor at the departure floor remains set *on*; if the hoist cable has broken the lift moves downwards at increasing speed; and so on.

The *lift_safety* component is concerned with the whole range of equipment faults, all of which the *priority_lift_service* component properly ignores. The decomposition allows the different properties of each problem domain to be considered in the design of the component to which they are relevant.

This approach to CBS decomposition is the central theme of the problem frames technique [Jackson00]. It is radical in two ways. First, it is radical because it aims to address, explicitly, questions that lie at the root of the system and its requirement. What is the system's purpose? How can this purpose be structured for clearer understanding? How can different purposes be composed and, if necessary, reconciled? What monitoring and control behaviour must the machine exhibit for each individual purpose? How can these behaviours be composed into a coherent overall behaviour? The approach aims to allow the developer to address these questions without premature concern for the eventual programming and implementation of the component machines.

Second, it is radical because it is geared to the needs of radical, rather than normal, design. That is, its primary use is as a tool for the developer who does not already know how to solve the problem. If the problem is already the object of a normal, accepted, artifact design, the questions asked by the problem frames approach should already been addressed, and satisfactory answers embodied, by the normal design: they should not be asked again for the design task in hand. The developer of a system embodying a subproblem that perfectly fits the assumptions of the normal Model-View-Controller pattern should not reconsider the design from first principles: a satisfactory design is already available. The designers of the earliest motor cars tried many different positions for the driver: sitting sideways; sitting on the near side; sitting in the cente; even perched high at the back of the car, like the driver of a horse-drawn hansom cab. A designer today who considers anything other than the standard position—in the front on the off-side, facing forwards—would be engaging in gratuitously radical design, 'rethinking the motor car', or designing a 'concept car', aiming to question the established standard design rather than to accept and exploit it. The consequences of such an innovation cannot be dependably predicted.

The radical character of the approach does not disqualify it completely, even if the design task is substantially normal. As Vincenti says [Vincenti93] of aircraft design:

> "Whether design at a given location in the [component] hierarchy is normal or radical is a separate matter—normal design can (and usually does) prevail throughout, though radical design can be encountered at any level."

One place in software development where radical design tasks are commonplace is in the composition of components. Even if the components themselves are objects of normal design, their composition may pose an entirely new problem.

8. COMPOSITION OF THE COMPONENTS

Karl Benz's radical design of 1886 was a remarkable achievement. He had solved many difficult problems in the development of the components, especially in refining the design of the petrol engine that drove the car, with its inlet and exhaust valves, carburettor and high-voltage ignition, and of the differential gear by which power was distributed to the two driven wheels. He was also remarkably successful in arranging all the components together in the space behind the bench seat, finding room for the large horizontal flywheel in a position that allowed the driver to grip the rim and pull it round to start the engine. The result of his work [Benz86] conveys a striking sense of improvisation; but it is brilliantly inventive and successful improvisation.

Designing the composition is not, of course, independent of the design of the individual components to be combined, but it can be considered a distinct, though related, problem; in a more complex component hierarchy, it is not one but several design problems. An important question for any design procedure is: Which of the two tasks—component and composition design—should be carried out first? Or should they proceed wholly or partly in parallel?

In a fully evolved normal design the standardised content of the design embraces both the individual components and their compositions. Further, the interfaces that implement the composition designs have been integrated into the design of the components to be composed. In a modern car, for example, almost every interface between two directly interacting components consists of a unique design in two mating parts, one on each of the interacting components: for example, the exhaust and inlet manifolds fit exactly to the corresponding locations on the engine block, sharing a flat surface, and the engine crankshaft is connected to the transmission by matching internal and external splines. In the evolution of the normal design the components have evolved, not only to fulfil their individual functions, but also to fit more efficiently and exactly with each other. The designer working on a car of normal design has no more need to ponder how the exhaust manifold should be connected to the engine than to consider whether five wheels might be better than four.

In radical design, by contrast, the individual components are not well understood at the outset, and the problems that will be posed by the design of their composition cannot be reliably anticipated. It makes sense, therefore, to postpone consideration of the compositions until more is known about the components themselves. The danger that some rework of components design will be needed in the light of the composition design may be judged less than the danger that a top-down design approach will set inappropriate or even impossible contexts for the component designs. Richard Feynman [Ferguson92] contrasted top-down with bottom-up design:

> "In bottom-up design, the components of a system are designed, tested, and if necessary modified before the design of the entire system has been set in concrete. In the top-down mode (invented by the military), the whole system is designed at once, but without resolving the many questions and conflicts that are normally ironed out in a bottom-up design. The whole system is then built before there is time for testing of components. The deficient and incompatible components must then be located (often a difficult problem in itself), redesigned, and rebuilt—an expensive and uncertain procedure."

Premature composition may carry another, larger, penalty than the danger that it will distort the design of the components. The chosen nature of the composition itself may be heavily dependent on the exact functions of the components. In the lift control system, for example, it is necessary to compose the *edit_priority* and the *priority_lift_service* components. Either on explicit command of the building manager, or at a point specified by the new priority *scheme* itself, lift service must switch from the current scheme to the new one. The design of this switching composition will depend on the nature of the permissible schemes. For example, consider two possible forms of priority scheme.

The first scheme specifies relative weights to be assigned to each request as a function of the request time, the direction and the floor at which it was issued. Switching between two schemes of this form is relatively straightforward: the old weights already assigned to unsatisfied requests are retained; new assignments will use the new weights. The chief composition concern is then to respect the necessary mutual exclusion between reading the old and writing the new scheme into the local store of the *priority_lift_service* machine. It will be relatively easy to show that the new priorities will take effect incrementally as the outstanding requests carrying the old weights are satisfied.

The second scheme specifies what is in effect an iterative algorithm of the scheduling procedure. Each lift or bank of lifts is assigned to a subset of the floors—for example, to provide express service for heavily used upper floors. Switching between two schemes of this second form is probably more complex. The composition must not only respect the necessary mutual exclusion; it must also such concerns as the treatment of a case in which a lift is currently serving a floor under the old schedule to which it is not assigned under the new schedule. There is much more here than mutual exclusion. If the switchover is not properly designed, there may be such troublesome results as ignored requests issued at a floor, failure to deliver a passenger already in the lift to the floor requested, or even, conceivably, deadlock of the scheduling system when the scheduling system encounters a state in which its behaviour is unspecified. Essentially, the switchover must take place at a point in the execution of the old scheme at which the invariant of the new scheme is also satisfied.

It is also necessary to design the composition of the *priority_lift_service* with the *lift_safety* component. Here there is a clear conflict between the two components' requirements. When a fault is detected while there are outstanding requests, *priority_lift_service* is required to service the requests, but *lift_safety* may be required to lock the lift car in the shaft, preventing further movement, or to take some other, less dramatic, action such as forcing the return of the car to the next floor, opening the doors, and preventing further movement after that. When *lift_safety* detects a fault requiring action, the function of *priority_lift_service* is no longer required, and it is necessary to consider how the system can switch from service mode to safety-action mode.

In summary, the design of a composition will often be most effective in the light of a substantial degree of understanding of the components in their initial, uncomposed, forms.

9. LEARNING FROM FAILURE

The capacity to learn from failures is a hallmark of the established engineering branches. As Henry Petroski writes [Petroski94]:

> "Engineering advances by proactive and reactive failure analysis, and at the heart of the engineering method is an understanding of failure in all its real and imagined manifestations."

Since failures in the products of the established engineering branches are usually extremely expensive, and often involve actual or potential loss of life, they are frequently examined with the greatest care, and major efforts made to identify the lessons to be learned. Well known examples of such disasters include the 1940 collapse of the Tacoma Narrows Bridge [Holloway99], the mid-air break-up of several Comet 1 aircraft in the early 1950s [Levy92, RAE54], the failure of the Ariane-5 launch in 1996 [Lions96], and the severe injury or death of several patients treated with the Therac-25 radiation therapy system [Leveson93].

The lessons that can be learned from engineering failures are most effective when they are highly specific: lessons that can be associated with a specific design fault in a normally designed artifact are arguably the most effective of all. At first sight, it may seem that more general lessons, being more widely applicable, can spread their benefit more widely. However, generality carries a serious disadvantage. A lesson that applies to everything applies to nothing: even when explicitly articulated in a disaster enquiry report it is likely to add only a grain of additional emphasis to what was already known to be good general practice. If the lesson is "documentation should not be an afterthought," few software developers (except perhaps adherents of some of the various schools of agility) would deny

its truth; but a lesson like this is blunted by endless repetition until it ceases to be heard at all.

The lessons from the Tacoma Narrows and Comet disasters, by contrast, were very specific and affected engineering design quite specifically. The Tacoma Narrows Bridge collapsed because a moderate wind, of about 40mph, provoked vertical oscillation in the bridge's very slender roadway. The oscillations built up quickly, and in a very short time destroyed the bridge completely. The lesson learned was specifically about suspension bridge roadways and the aerodynamic effects to which they are subject: the roadway was not stiff enough to resist the wind-induced oscillation. The associated lesson about the normal design procedure was clear: the designer, Leon Moisseiff, had made the mistake of taking account of horizontal, but not of vertical, oscillation. After the disaster, its lessons were taken to heart. Steps were taken to strengthen other bridges whose roadways were thought to be too slender or too shallow. The normal design discipline for suspension bridges changed: designers were subsequently expected to check explicitly for vertical roadway oscillation.

In the case of the Comet 1, the aircraft broke up because of metal fatigue. In an enormously expensive investigation, pieces of one of the destroyed aircraft were recovered from the sea bed and reassembled in a hangar in a research establishment. The tentative results of that part of the investigation were then confirmed by destructive tests on another sample of the same design. The investigation showed that the cause was metal fatigue in the fuselage, and that the fatigue cracks had started at the corners of the square passenger windows. Again, the normal design, in this case, of pressurised jet aircraft, was specifically modified: no such aircraft today has square corners in the passenger windows or cargo apertures. The normal design discipline, too, was affected. It was recognised that fuselage tests for torsional rigidity and for resistance to pressurisation and depressurisation must be carried out in combination: separate testing had been proved insufficient.

The Tacoma and Comet investigations were concerned with failures of the physical fabric of the engineered products. When the failure is attributable to software, it is harder for the investigation to reach very specific conclusions and to offer very specific lessons. Partly, this is because software failures, unlike physical failures, often leave no trace in the resulting wreckage. As Donald MacKenzie writes [MacKenzie94]:

> "A more particular problem concerns what this data set suggests are the two most important 'technical' causes of computer-related accidental death: electromagnetic interference and software error. A broken part will often survive even a catastrophic accident, such as an air crash, sufficiently well for investigators to be able to determine its causal role in the sequence of events. Typically, neither electromagnetic interference nor software error leave physical traces of this kind. Their role can often only be inferred from experiments seeking to reproduce the conditions leading to an accident."

Of equal, or perhaps greater, importance, is the fact that in most cases the software under investigation was not the object of normal design. The investigators, even when they can identify defects in the software as a contributory cause of failure, cannot express their conclusions in sufficiently specific terms to be confident of affecting either the usual design practices or an identifiable, currently accepted, standard design that has proved dangerous. Instead, they must content themselves with generalities that, if the truth is told, are unlikely to have a substantial effect. The case of the Ariane-5 launch, whose failure was entirely attributed to software error, is to some extent an exception. The investigation conclusions contained some general recommendations such as "Include external participants in reviews" and "Pay the same attention to justification documents as to code;" but they also included some quite specific recommendations for designers of software of the Ariane-5 class:

"• Failing sensors should not cease to transmit, but should send best-effort data.
• Alignment functions should be switched off immediately after lift-off.
• Design of the switchover between on-board computers needed more care.
• More data should be sent to telemetry on any component failure.
• Trajectory data should be included in specifications and in test requirements."

These specific recommendations are possible only because the terms used—'failing sensors', 'alignment functions', 'computer switchover', 'data sent to telemetry', and 'trajectory data'—have specific meanings, referring to specific parts of their designs, that designers working on this class of software will certainly understand.

The Therac-25 disasters also were essentially attributable to software failures. In their excellent unofficial investigation [Leveson93] of these failures, Nancy Leveson and Clark Turner identified specific software errors that had certainly played a major, or even decisive, role in the failures. Yet, in the absence of even a vestigial normal design for software to control radiation therapy equipment, they were compelled to content themselves with identifying 'basic software engineering principles that apparently were violated with the Therac-25':

"• Documentation should not be an afterthought.
• Software quality assurances practices and standards should be established.
• Designs should be kept simple.
• Ways to get information about errors—for example, software audit trails—should be designed into the software from the beginning.
• The software should be subjected to extensive testing and formal analysis at the module and software level: system testing alone is not adequate."

A reiterated general principle has value, especially when its application is in the hands of managers who can mandate improvements in development practice; but its effect for a practising design engineer is dissipated by its very generality. It calls for a degree of culture change across the whole range—or, at least, across an unspecified segment—of software development, and a quality change across the whole range of designed software. For a pratcising engineer, a lesson directly associated with a specific design artifact has a more restricted, but far stronger and more certain effect. That can be seen today by every airline passenger, in the frustrating rounded shape of the window that obscures the view of the terrain over which they are flying. Unlike too many software design mistakes, the design mistake of square corners for aeroplane windows has not been repeated.

10. HARDER AND SOFTER CBSS

What may be called the 'softer' CBSs are concerned with business and administration—which are primarily human activities—rather than with lift equipment and radiotherapy devices and chemical plants. For these systems the emphasis on the physical world, as the basis for suggesting analogies with the established branches of engineering, may seem misplaced.

Not so. It is true that such systems are rarely safety-critical; and in most cases some part of the required functionality may be very loosely textured and imprecise, admitting wide variations in quality and exactness without causing failure or even significant difficulty. For example, some of the functionality in an e-commerce system may be of this kind. If the collaborative filtering doesn't work well, or the purchase recommendations offered to regular customers are poorly calculated, the company will make less profit than it could, but otherwise no great harm is done. But this kind of looseness is exceptional. Some softer CBSs

are definitely safety-critical: for example, a system to maintain criminal records and make them available to the appropriate authorities in the appropriate circumstances; or a system to manage the prescription and delivery of patients' drugs in a large hospital. The core of the functionality in a softer CBS is no less demanding for the software developer than the core functionality in a 'harder' system: the system that the hospital pharmacist finds difficult, inconvenient and confusing to use is exhibiting exactly the same kind of defect as the avionics system that is ill-adapted to the needs of the pilot.

The characteristic difficulty in the design of any CBS arises because the problem world—whether a hospital pharmacy or an aeroplane—is not a formal system. This, above all, distinguishes a CBS from a program whose function is to compute about a formal abstraction, untainted by a real semantics in the non-formal, physical and human world. An innovative program to factorise very large integers, for example, is not concerned with the purpose of the factorisation, or with what the integers may denote. It is concerned with pure mathematics, in the sense explained [Weyl40] by Hermann Weyl:

> "We now come to the decisive step of mathematical abstraction: we forget about what the symbols stand for. [The mathematician] need not be idle; there are many operations he may carry out with these symbols, without ever having to look at the things they stand for."

In dealing with the integers, which are a mathematical domain, we can state exact theorems to which there are no exceptions. For example: "If the sum of the digits of an integer's decimal expression is evenly divisible by 3, then the integer itself is evenly divisible by 3." There is no approximation here, and no exceptions. By contrast, if we try to state a theorem about the behaviour of the lift equipment we will find that it must always be hedged around with caveats: the power supply may be interrupted; a sensor may stick on; the drive gears may be worn; the hoist cable may break. The same is true, for example, of the problem world of a system to administer social benefits: a recipient may change gender; a benefit payment sent by post may be delayed or lost; an immigrant recipient may have no birth certificate and be unable to state their date of birth; there may be family relationships between recipients that do not conform to the assumptions on which the entitlement regulations were framed. It is never possible to exhaust the problem world's capacity to produce new counterexamples to the assumptions on which the system design is based.

Dealing adequately with the problem world of any CBS, then, is not a matter of discovering and proving formal theorems. Rather, it is a practical matter of constructing descriptions that are formal enough to allow tentative formal reasoning and analysis, and close enough to the problem world reality to accommodate all eventualities except those that have a tolerable combination of low occurrence probability and limited consequential damage. This practical necessity is close—though not identical—to the necessity that bears on engineers in the established branches. In both softer and harder CBSs, the development of artifact specialisation, with the evolving normal design that each artifact specialisation supports, must play the central role in addressing this practical engineering necessity. As a normal artifact design evolves, it comes to accommodate more and more of the important eventualities in its problem world as lessons are learned from failures.

11. A Concluding Personal Remark

In this paper much has been said about the practice of engineering in the traditional branches. The reader should not infer that the author has either the education or the practical experience of an engineer. What is asserted here about engineering is drawn, often very directly, from the statements of those engineers who have written about their profession for a

non-professional readership. Foremost among them is Walter Vincenti. His deep and brilliant book, *What Engineers Know and How They Know It*, should be required reading for all thoughtful software engineers. Its lessons, drawn from aeronautical engineering in the period from the earliest years to the arrival of the turbojet revolution in the later 1940s, are not exhausted after many readings. I remain very grateful to Tom Maibaum for introducing me to this book several years ago.

Acknowledgements

Daniel Jackson and Mary Shaw kindly read an earlier draft of this paper and made several very helpful suggestions for improvements.

References

[Benington56] H D Benington; *Production of Large Computer Programs*; in Proc ONR Symposium on Advanced Program Methods for Digital Computers, June 1956, pages 15-27. Reprinted with additional comment in IEEE Annals of the History of Computing, Volume 5 Number 4, October 1983, pages 299-310.

[Benz86] http://www.conceptcarz.com/vehicle/z10986/Benz-Motorwagen-Replica.aspx (accessed 23 October 2008).

[Buxton70] 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.

[Caminer97] David Tresman Caminer OBE; *LEO and its Applications: The Beginning of Business Computing*; Computer Journal Volume 40 Number 10, pages 585-597, 1997.

[Constant80] Edward W Constant; *The Origins of the Turbojet Revolution*; The Johns Hopkins University Press, Baltimore 1980.

[Everett83] Robert R Everett, Charles A Zraket, Herbert D Benington; *SAGE—A Data Processing System for Air Defense*; IEEE Annals of the History of Computing, Volume 5 Number 4, pages 330-339, Oct-Dec, 1983.

[Ferguson92] Eugene S Ferguson; *Engineering and the Mind's Eye*; MIT Press, 1992.

[Holloway99] 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.

[Humphrey00] W S Humphrey; *The Personal Software Process: Status and Trends*; IEEE Software Volume 17 Number 6, November/December 2000, page72.

[Jackson82] M A Jackson; *Software Development as an Engineering Problem*; in Angewandte Informatik 2/82, pages 96-103; Vieweg & Sohn, February 1982.

[Jackson00] Michael Jackson; *Problem Frames: Analysing and Structuring Software Development Problems;* Addison-Wesley, 2001.

[Kraft77] Philip Kraft; *Programmers and Managers: The Routinization of Computer Programming in the United States*; Springer-Verlag, 1977, pages 57-58.

[Leveson93] 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.

[Levy92] Matthys Levy and Mario Salvadori; *Why Buildings Fall Down: How Structures Fail*; W W Norton and Co, 1992.

[Lions96] *ARIANE 5 Flight 501 Failure; Report by the Inquiry Board*; *The Chairman of the Board: Prof. J L Lions*; Paris 19 July 1996; available at: http://sunnyday.mit.edu/accidents/Ariane5accidentreport.html (accessed 23 October 2008).

[MacKenzie94] Donald MacKenzie; *Computer-Related Accidental Death: An Empirical Exploration*; Science and Public Policy Volume 21 Number 4, pages 233-248, 1994.

[Naur69] 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.

[Parnas78] D L Parnas; *Some Software Engineering Principles*; in Structured Analysis and Design, Infotech, 1978.

[Petroski94] Henry Petroski; *Design Paradigms: Case Histories of Error and Judgment in Engineering*; Cambridge University Press, 1994.

[RAE54] Royal Aircraft Establishment; *Report on Comet Accident Investigation*; Ministry of Supply, 1954.

[Rogers83] G F C Rogers; *The Nature of Engineering: A Philosophy of Technology*; Palgrave Macmillan, 1983; (ISBN: 0333347412).

[Shaw90] Mary Shaw;. Prospects for an Engineering Discipline of Software; IEEE Software, November 1990, pages 15-24.

[Vincenti93] 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.

[Weyl40] Herman Weyl; *The Mathematical Way of Thinking*; address given at the Bicentennial Conference at the University of Pennsylvania, 1940.