The Name and Nature of Software Engineering

Conference Paper · January 2007

DOI: 10.1007/978-3-540-89762-0_1 · Source: DBLP

1 author:

Michael Jackson
The Open University (UK)
142 PUBLICATIONS **6,837** CITATIONS

THE NAME AND NATURE OF SOFTWARE ENGINEERING

DRAFT OF 29 JUNE 2008

Michael Jackson

The Open University
jacksonma@acm.org

ABSTRACT

The nature of software engineering is discussed with particular reference to software-intensive application systems—those whose fundamental purpose is to bring about desired effects in a physical and human *problem world* by interaction with a programmed *machine*. Such systems bring together a problem world—which is typically composed of heterogeneous domains, most of which are non-formal—and the formal or semi-formal domain of the machine. A clean engineering separation of the two is rarely, if ever, possible; and attempts to treat the application problem world as an extension of the formal machine are obstructed by its non-formal nature. Software engineers have much to learn from the structure and practices of the established branches of engineering. We must learn from their treatment of formal analysis and reasoning, from their practice of intense specialisation, from their attention to particular instances no less than to general concerns, and—above all—from their reliance on normal artifact design and on normal design disciplines: both are the golden fruit of specialisation.

1 INTRODUCTION

The term ‘software engineering’ came into common use as a result of the first NATO Software Engineering Conference in 1968 [Naur 69]. The NATO Science Committee had chosen the phrase because it suggested “the need for software construction to be based on the types of theoretical foundations and practical disciplines that are traditional in the established branches of engineering.” A ‘software crisis’ of failed projects and unsatisfactory software systems was already widely recognised and much talked about. By contrast, engineers who designed and built automobiles, or aeroplanes or bridges seemed to have achieved far higher levels of success and reliability. The committee’s view was clear: software developers should learn to emulate the established engineers by adopting similar or analogous theories and practices, whether already known or yet to be devised.

Surprisingly, the organisers and participants at the conference, and at the follow-up conference that took place in the following year, did not proceed to investigate explicitly what the established engineering branches did: how they carried out their work; how they developed and exploited their theoretical foundations; how they were organised; or what their practical disciplines were and how they had been developed. Beyond the proposal, in an invited talk by Doug McIlroy, that a software components industry should be encouraged, capable of offering catalogues of routines for performing such operations as input and output or calculating trigonometric functions, there is remarkably little reference in the conference transcripts to those practices of the established branches that software engineers were invited to emulate.

The purpose of this paper is to repair this omission to some small extent. The paper’s title is inspired by the 1932 Leslie Stephen lecture at Cambridge, “The name and nature of poetry”, given by the poet and classical scholar A E Housman [Housman32]. He began by pointing out that “When one begins to discuss the nature of poetry, the first impediment in the way is the inherent vagueness of the name, and the number of its legitimate senses.” He went on to characterise the nature of poetry, as opposed to mere versification or linguistic elegance or felicity, and to locate it in the landscape of literature in general. He identified the essence of poetry as its effect on the reader, even claiming that in his own case the effect was physical: “if a line of poetry strays into my memory while I am shaving, my skin bristles so that the razor ceases to act.”

My intent is to characterise software engineering in a way similar to Housman's characterisation of poetry: to define the meaning of the term; to distinguish it from mere programming and from facility with formalisms; to locate it in the landscape of engineering in general; and to identify some of the practical implications of these ideas.

The chief substance of the paper is contained in the three following sections. Section 2 defines software engineering as the development of a programmed *machine* that will bring about desired effects in the physical *problem world*. Section 3 briefly reviews some of the most significant practices in the established engineering branches—most notably, the very high degree of *specialisation* and the evolution of specialised *normal design*—and explores some of their consequences. Section 4 returns to the topic of software engineering to draw some comparisons between practices there and in the established branches. Some concluding reflections are offered in Section 5.

Because the paper draws lessons from engineering notions and practices, I must begin with a disclaimer. I have neither the education nor the practical experience of an engineer. My understanding of engineering is drawn from the everyday observations that are available to all of us, and from some of the excellent books written for lay readers by engineering practitioners and academics. Among them are deservedly well known books by Henry Petroski [Petroski 86, Petroski 94], and illuminating books by several other authors [Ferguson 92, Gordon 78, Rogers 83, Levy 92]. An outstanding book by W G Vincenti [Vincenti 93] explores five case studies in the development of aeronautical engineering in the first half of the twentieth century and reflects deeply on the nature and growth of engineering knowledge and practice. A strong argument can be made that time devoted to reading and discussing these books, especially Vincenti's, would be well spent in any software engineering course.

2 WHAT IS SOFTWARE ENGINEERING?

Just as automobile engineers develop automobiles, so software engineers develop software. In both cases, the engineer aims to solve some problem, to satisfy some human need. Surrounding each activity is a complex structure of supporting and associated activities and concerns—economic, organisational, managerial—and an equally complex structure of constraints and goals in the larger human context—political, ethical, social, legal, and others. These structures may be regarded, perfectly legitimately, as integral parts of the engineering context, and therefore integral topics in an engineering education. But for the theme of this paper they are secondary to the technical concerns: so I shall ignore them, and focus only on some aspects of the technical development of the engineered artifact itself.

In short, like Housman, I will discuss only a few chosen aspects of my topic, omitting much that is undoubtedly important. I do not purport—and would not be competent—to offer a comprehensive survey of software engineering practice and theory in all its rich variegation.

2.1 Symbolic Problems

In software development we may be concerned with *symbolic* problems. A symbolic problem is one whose subject matter can be entirely captured in mathematical symbols. The computer, executing our software, is required to compute a symbolic output related in some specified way to the symbolic input. Examples of symbolic problems are: computing the hash of a text; playing chess against an opponent, where the input is some encoding of the opponent's moves and the required output is a winning sequence of moves by the computer; and the calculation of the convex hull of a set of points in Euclidean 3-space, the set of points being given as a sequence of real number triples and the desired output being a subset of the input triples.

Symbolic problems can be characterised as being formal and mathematical. Although the computer itself is a physical machine, it has been carefully engineered by the hardware designers to perform with high reliability as a symbol processor. If the developers of the compiler or interpreter, and of the operating system, have been equally successful, the resulting *platform* will reliably execute programs written in the appropriate well-defined programming language. The software developer can then ignore the possibility of computer or system software malfunction, and focus solely on the formal, mathematical aspects of the problem. In the words of Herman Weyl [Weyl 40], this means

that operations on the symbols are carried out "without ever having to look at the things they stand for." The resulting view of software development, as clearly expressed by Dijkstra [Dijkstra 89], is that "the programmer's ... main task is to give a formal proof that the program he proposes meets the equally formal functional specification." Software development is a formal, mathematical, discipline. The essential criteria of success are *correctness* and minimal computational complexity.

Because the computation is carried out by a physical device, even symbolic problems may force a variety of non-formal concerns on the developer's attention. Inputs must somehow be made available to the program, and outputs communicated to the user: input and output operations take the program out of the ambit of the internal purely electronic parts of the machine—in the evocative phrase of [Hoare 04], outside the 'silicon package'. Real numbers can be represented only approximately in the computer: the treatment of error terms in calculations can in principle be a formal discipline; but, like the three-body problem in Newtonian mechanics, it is intractable and in practice must be less than fully formal. Some large symbolic problems, such as protein folding or searching for evidence of extraterrestrial life, demand distributed processing over a large number of processors, the connections among the processors being inevitably imperfect. In the most demanding critical applications the programmer may be required to guard as effectively as possible against malfunctions of the computer hardware, including the silicon package. Even in less demanding applications it may be necessary to guard against the far more probable failures of the operating system, or of other programs that are sharing the same platform and may cause failures—such as resource starvation or a system crash—against which the operating system offers no protection. The formal mathematical view captures a central and vital concern of programming, but far from the whole of it.

2.2 Concrete Problems

Although the computer can be regarded as a symbol processor, symbolic problems are only one very limited kind of problem to which it can be applied. In *concrete* problems, its role is to function as one part among many in a physical system, interacting with the *problem world* outside itself to achieve some purpose in that world. (We speak of the *problem world* because it is here that the problem to be solved is located. The more commonly used term *environment*, by contrast, misleadingly suggests that the real problem is located in the machine, and that the environment is at best a neutral ambience and at worst a source of irritating obstacles to its solution.) Systems of this kind, whose purposes are located in the physical world and achieved by interaction with it, are often called *software-intensive systems*. Although the computer is only one part among many, its role is intense and crucial, monitoring and constraining the behaviour of the other parts and hence of the whole system. In its interactions with the problem world it detects events and state changes caused by the other parts of the world, and causes events and state changes in the world in response. The total effect of these interactions is to achieve the system's purpose. We may illustrate this view in a generalized *problem diagram* [Jackson 01]:

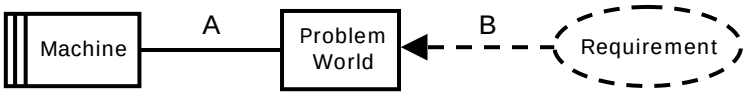

Figure 1. Generalised Problem Diagram

The *machine* is a computer executing the software. It interacts with the *problem world* at an interface *A* of shared *phenomena*—essentially, shared events and states. The *requirement* is a set of conditions on the problem world whose satisfaction the machine must ensure, expressed in terms of some phenomena *B*. The success or failure of the engineered software is judged with reference to its observable effects in the problem world. We judge a theatre booking system by asking whether booking is convenient, whether duplicate sales of the same seat at the same performance are avoided, whether the best available seats are sold in preference to inferior seats at the same price, whether correct payment is collected, and so on. For a proton therapy system we judge whether the patients receive their prescribed doses in the prescribed locations and directions, and whether the equipment is used efficiently and safely. For a lift control system we judge whether the service is efficient, whether passengers are delivered to the floors they have requested, whether acceleration of the lift car is smooth, whether the indicated information about lift position and direction of travel is reliable and

conveniently displayed, whether passenger safety is ensured in the event of equipment malfunction, and so on.

In general, the sets of phenomena *A* and *B* are distinct, though not necessarily disjoint. The machine can therefore ensure satisfaction of the requirement only by relying on some *given* properties of the problem world that relate the phenomena *A* to the phenomena *B*. These are properties that hold regardless of the behaviour of the machine. In the lift control system, they include the disposition of the lifts in the shafts, the arrangement of the floors, the causal links from the motor switch to the lift motor, from the motor to the winding gear, and from the winding gear to the lift car movement, the properties of the sensors that detect the presence of a lift car at a floor, the possible and probable behaviours of the lift users, and so on. If we represent the properties of the machine as $\mathcal{M}$, the requirement as $\mathcal{R}$, and the given properties of the problem world as $\mathcal{W}$, then for the system to meet its requirement the entailment must hold: $\mathcal{M}, \mathcal{W} \vDash \mathcal{R}$. That is: a machine with the properties $\mathcal{M}$, installed in a problem world whose given properties are $\mathcal{W}$, consistent with $\mathcal{M}$, will ensure satisfaction of the requirement $\mathcal{R}$.

2.3 The Problem World as a Given

The problem world is richly structured, and its parts and their given properties may support very complex causal and other relationships by which the machine and the world can affect and respond to each other. Investigating and analysing these properties is therefore a large part of the work of software engineering.

The view adopted here, that the problem world is given, is a simplifying assumption that allows a distinction between *software engineering* and *system engineering*. Software engineering is concerned with developing software for the given problem world; system engineering embraces also the possibility of achieving the system's purposes by changing the problem world directly. In a lift control system, for example, one possible requirement is to allow only certain privileged people to request travel to certain floors. For the given lift equipment this requirement will be impossible to satisfy if the interface *A* provides no phenomena by which the machine could detect that the requester is one of the privileged people: to satisfy the requirement it will then be necessary to add some physical device, such as a card reader, to the problem world. Here, we would regard this addition as lying outside the scope of pure software engineering *per se*. If the problem world is enlarged by the addition of the card reader, and interface *A* expanded to include communication between the reader and the machine, the problem will then fall once again within our scope.

This assumption, that the problem world is *given*, and the associated distinction between software and system engineering, is adopted to simplify the thesis of this paper, not as a recommendation for engineering practice. In some projects the assumption will be completely realistic. In others, software development will proceed, as it should, in cooperation with engineering activity designed to change the properties of the problem world to contribute to the system's purpose.

2.4 Non-Formal Problem Domains

The problem world for a concrete problem to be solved by a software-intensive system is almost always heterogeneous and invariably non-formal. Typically, its *domains*—the parts of which it is composed—can be drawn from many sources: the natural world—for example, the earth's atmosphere for an avionics system; human participants—for example, the staff and users of a lending library, or the driver of a car; electrical and mechanical engineered devices—for example, the physical components of a lift or an ATM; other software-intensive systems—for example, those of banks and telecommunications services; constructed, largely static, parts of the world—for example, the lines of a railway network or the runways of an airport; concrete lexical components, in which information has been encoded for the machine to read—for example, credit cards and bar-coded labels; and others.

The many different parts of the problem world are proper objects of scientific theory and investigation, but they are not formal systems: any formal description useful to a software engineer is only an approximation. They exhibit many different given and potential properties, and demand many different languages to express those properties adequately. At the granularities significant for software-intensive systems, even the most reliable parts of the problem world are seldom as reliable

as the CPU of a desktop computer, partly because of their inherent nature and partly because their useful functionality is more vulnerable to the vicissitudes of the other parts of the world with which, by design or accident, they interact.

Because the problem world is central to a software-intensive system, its given properties $\mathcal{W}$ and its desired properties $\mathcal{R}$ are necessary subjects of description and reasoning. But formal description and reasoning, which are vital for a reliable process of software development, have a very different character when interpreted in non-formal domains.

2.5 Reasoning In Non-Formal Domains

Formal reasoning is necessarily dependent on abstraction: we choose some axioms and some alphabet, and use the axioms to reason over the elements of the alphabet. The useful results that we hope for from our reasoning are theorems, capturing truths that were not evident to us at the outset or of which we want stronger conviction. In a formal world this process is as reliable as our capacity to reason correctly.

In a non-formal world there are several obstacles to reliability in formal reasoning. To make our reasoning useful we must begin by establishing a correspondence between the formal terms we intend to use and the physical phenomena they denote. Here there is an immediate difficulty. In a system to control road traffic, we may decide to reason about pedestrians and their use of the controlled crossings provided for them: for example, to base some design decisions on the maximum and minimum time taken to cross the road. But what, exactly, is a 'pedestrian'? A child in a pedal car? A cyclist pushing a bicycle with an attached trailer? A user of a motorised invalid carriage? Whatever *alphabet* of formal terms—for example, of predicates, events, and entities—we choose, there will be some hard cases in the problem world for which we cannot easily decide whether or not they are properly denoted by a particular formal term. The best we can do is to choose our alphabet so that hard cases are sufficiently rare in the particular world we are reasoning about. This choice of alphabet, and its necessarily non-formal *interpretation*—that is, the mapping between the alphabet and the real phenomena of the problem world—constitute the fundamental basis of any formalisation.

Given an acceptable alphabet we can record or establish some truths—axioms, or perhaps theorems, in our formal system—about the world. In the traffic system, for example, we may need a theorem about a road segment guarded by sensors at its ends: the number of vehicles present in the segment is equal to the number that have entered minus the number that have exited. However, cars are sometimes transported on trucks. If a truck enters the segment and unloads the car it is carrying, our desired theorem is invalidated by this counterexample. Again, the best we can do is to choose our putative theorems so that they will hold well enough in most of the situations that will actually arise.

In a non-formal world the abstraction process of choosing an alphabet itself opens the door to error. We can never be certain that we have not excluded from the chosen alphabet, or from the perception that underpins a chosen axiom, some phenomenon that will eventually prove to vitiate our reasoning. This kind of error is rife in reasoning about safety or security. In a famous case [Lampson 84] the TENEX operating system *connect* call, to access a directory, checked a *password* string argument. If a left-to-right scan of the string encountered an erroneous character, it terminated and the system waited three seconds before reporting failure (to prevent an attacker from estimating where in the string the error had been found). However, if the scan encountered the boundary between an assigned and an unassigned page, the *connect* call reported a page fault to the caller, revealing that the submitted string was correct at least up to the end of the assigned page. By placing successive strings on such a boundary, an attacker could reduce the task of finding the correct password from exponential to linear complexity in the password length. Essentially, the attacker enlarges the alphabet of relevant phenomena to include the phenomenon *page-fault* that was wrongly omitted from the designer's alphabet. This attack illustrates a general obstacle to certainty that we might dignify as the *Principle of Unbounded Relevance*.

In a non-formal world, then, not only do the elements of our chosen alphabet rest on uncertain foundations. Worse, any alphabet we choose must always omit some phenomena—and we cannot be sure which ones—that may invalidate our abstractions and upset our calculations. How, then, can we reason usefully in a non-formal problem world? We must recognise that all the steps in our

descriptive and reasoning processes reflect decisions to adopt *assumptions*: assumptions that our chosen alphabet has few enough hard cases; assumptions that our chosen axioms are true often enough; and assumptions that what we would wish to regard as proven theorems are not vitiated by phenomena and considerations that we have erroneously neglected. Weaving these frail foundations of assumption into a structure strong enough to support the necessary degree of confidence in the reliability of the resulting system is a major concern in software engineering.

2.6 Relating Formal and Non-Formal

Development of a software-intensive system, then, involves a combination of formal concerns, in the construction of programs, and non-formal concerns, in the understanding, description and analysis of the requirement and given problem world properties. The question how formal approaches to program construction are to be related to the treatment of the non-formal problem world—and to the less formal aspects of the machine itself—is central to our whole understanding of software engineering.

One view of software engineering is that we should try to maintain a firm separation between its formal and non-formal aspects. This was the view of Dijkstra, who disdained the very term 'software engineering'. He explained his view, with great clarity, in responding to comments on the published text of his invited talk at the ACM Computer Science Conference of 1989 [Dijkstra 89]:

> "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 wou1d like to have; the other one is the 'correctness problem,' i.e., the question of how to design an engine meeting the specification. I firmly believe that whenever we succeed in erecting such a firewall, the effort will pay off handsomely. The reason for this belief of mine is that the two concerns deserve separation because the two problems are most effectively tackled by totally different techniques. (They are currently psychology and experimentation for the pleasantness problem and symbol manipulation for the correctness problem.)"

In the problem diagram of Figure 1, the logical firewall would be erected roughly on the line marked *A*, where the machine interacts with the problem world at its interface of shared phenomena: some adjustment of the position of the firewall may be necessary, moving it slightly closer to the internals of the machine to exclude any irreducibly non-formal aspects of the interface phenomena. In this way the task of the *programmer* would be kept free of non-formal concerns, and could be tackled by purely formal methods. On the other side of the firewall the non-formal properties would be addressed by problem world experts using non-formal techniques.

Clearly, the separation is possible only when the logical firewall—the formal specification—can be constructed. The problem world experts must be able to develop a complete specification of the machine's behaviour at its interface with the problem world. If we denote this specification by S, then the whole problem falls into two parts. The problem world experts are responsible for developing a specification S, consistent with $\mathcal{W}$, such that $S, \mathcal{W} \vDash \mathcal{R}$. The programmers are responsible for developing a machine to satisfy the formal specification while leaving the problem world properties $\mathcal{W}$ unimpaired: that is, a machine whose properties $\mathcal{M}$ ensure that $\mathcal{M} \vDash S$.

A different approach to handling the relationship between the formal and the non-formal in software-intensive systems is to treat the non-formal problem world formally, in this respect assimilating it to the machine and regarding both as parts of one formal system. The fullest expression of this approach is found in a relatively neglected paper [Marzullo 91]. Essentially, the approach reported there distinguishes the machine from the problem world, but in effect regards the two as constituting a single formal system. The fully formal requirement is assumed to be given in terms of some phenomena of the problem world, represented by *reality variables*. The values of these reality variables are functions, often functions of time, that themselves vary over time. For example, in a system to control a vehicle on a track, the position of the vehicle might be given by the variable function $p(t)$, while its acceleration is given by $a(t)$. The requirement and the problem world properties are represented by equations over the reality variables within the single formal system. The machine—that is, the program—is developed by a refinement process in the style of Dijkstra's

weakest precondition calculus. Starting from the requirement expressed in reality variables, the refinement eventually arrives at a program text by appealing to properties of the problem world.

Another formal treatment of the problem world is discussed in [Parnas 95]. The machine and problem world are called respectively the *system* and the *environment*, and the interactions between them are mediated by *sensors* and *actuators*. Four sets of variables are identified (to which the approach owes its soubriquet "Four Variable Model"). The sets of monitored and controlled variables of the environment are denoted by *m* and *c*, while *i* and *o* denote the sets of values that the machine reads from the sensors and writes to the actuators. Five relations over vectors of time functions of these variable sets are defined:

SOF: $i \leftrightarrow o$ captures the specification of the machine behaviour;

IN: $m \leftrightarrow i$ and *OUT:* $o \leftrightarrow c$ capture the properties of the sensors and actuators respectively;

NAT: $m \leftrightarrow c$ captures the given environment properties; and

REQ: $m \leftrightarrow c$ captures the requirement.

The formula $NAT \cap (IN \bullet SOF \bullet OUT) \subseteq REQ$ characterises the *acceptability* of the system. That is: the requirement *REQ* is satisfied by the behaviour of the whole system, in which the environment (whose given properties are described by *NAT*) is placed in parallel with the machine (whose behaviour, when combined with the sensors and actuators, is $IN \bullet SOF \bullet OUT$). This second approach differs in two ways from the approach of [Marzullo 91] described in the preceding paragraph. First, it proposes an explicit generalised structuring of the problem in its context, this structure being a more elaborate form of the structure shown in Figure 1. Second, it is less overtly methodological, proposing no specific calculus or refinement structure for software development. However, the Four Variable Model has formed the basis of more than one specific development technique [Faulk 95, Heitmeyer 96].

These two formal approaches, along with others not mentioned here, embody important contributions to our understanding of software engineering: some, at least, of our reasoning about the problem world must surely be formal if it is to be more than guesswork. Still, the limitations of formal reasoning applied to a non-formal problem world remain severe. To develop a successful software-intensive system we need much more than formal description and reasoning. Some of what we need can be learned from the practices of the established engineering branches, to which we now turn.

3 SOME ENGINEERING PRACTICES

This section briefly reviews some of the practices of the established engineering branches, drawing especially on the writings of Vincenti [Vincenti 93] and Petroski [Petroski 86, Petroski 94]. These practices are, of course, many and varied, and demand wider illustration and deeper explanation than the superficial account that can be given here. At root some of the most important practices stem, directly and indirectly, from the rich structure of *specialisations* that characterises engineering: we begin there.

3.1 Specialisation in Engineering

In any technical field of human endeavour, specialisation is the fundamental precondition for improvement over time. When an advance is made, successful development of the field demands the capacity to capture it, to fit it into an appropriate intellectual and cultural structure, and to retrieve and exploit it wherever the knowledge it embodies is apposite. In the most primitive stages of development, when relatively few advances have yet been made, people of extraordinary ability can master all the available knowledge of a field, or even of several fields. As time passes, and the depth and breadth of knowledge increase, the range over which one person can achieve mastery becomes relatively smaller. Knowledge gained must become the treasured possession of a community of specialists if it is to be retained for future use. Without a community of specialists to tend and nurture it, it may become effectively unavailable when needed because it has been hidden in a vast sea of other knowledge, or even entirely lost by universal neglect.

The established branches of engineering illustrate this process of increasing specialisation in a very high degree. There are specialisations by engineering artifact—automobile, aeronautical, chemical engineering; by problem world—civil and mining engineering; and even by requirement class—industrial and transportation engineering. There are also specialisations by applicable theoretical foundation—control and structural engineering; by product components—electric motors, internal combustion engines, TFT screens; by technology—welding, reinforced concrete, conductive plastics; and in other dimensions too. These specialisations have evolved in response to changing needs and opportunities: they 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 purely pragmatic to the most rigorously intellectual, from engineering that is almost craft to engineering that relies explicitly and systematically on mathematics and science.

3.2 Normal Design

The fundamental benefit of specialisation is what Constant [Constant 80], calls *normal* design, and all that flows from it. Following Constant, Vincenti [Vincenti 93] draws a strong distinction between normal and *radical* design. Most engineering practice is the practice of normal design, in which the task is to make an incremental improvement in a product class whose firmly established and well understood standard design already has a long record of success:

> "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-fuelled, 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."

By contrast:

> "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."

The phrase *normal design* can be understood in two senses. It denotes the structure and properties common to all instances of a particular class of artifact. It also denotes the practical discipline that designers follow in developing new instances of the class. The practical design discipline presents a repertoire of options among which the designer must choose, and parameters for which values must be set. In some extreme cases—for example, in the design of a small electrical power transformer—the parameter values are determined by a fixed procedure. The normal discipline also embodies what is known about analysing proposed designs—in civil engineering, for example, the techniques of stress calculations for load-bearing structures of specific types. It also embodies the lessons learned from past failures about the risks that demand particular and careful application of those techniques—for example, the need to analyse the aerodynamic properties and vertical oscillation modes of a suspension bridge roadway, to avoid a failure of the kind that destroyed the Tacoma Narrows bridge in 1940.

3.3 The Fruit of Specialisation

Normal design, in both senses, is the product of a long evolution. It can emerge only from specialisation, because it demands the concentrated attention of a community of specialists over a long period. From the radical design of Karl Benz's three-wheeled car of 1886 it took about thirty five years for normal automobile design to evolve. By 1920 normal design mandated four wheels, internal combustion engine, gearbox, electric starter motor, a closed cab (except for sports and some touring

cars), pneumatic tyres, and four-wheel drum brakes. At that point, automobile designers could properly be said to know what were the customary features of a car and how to configure them to good advantage, and to have a good likelihood of producing a solid, reliable vehicle. Establishment of normal design for a product class does not mark the end point of evolution and development: on the contrary, it marks the point at which a stable base has been created for reliable further advances. Continued steady progress in the nine decades following 1920 has brought cars to their present level of refinement, the result of a hundred and twenty years of concentrated specialisation in the industry.

The specialisation necessary for this kind of progress is very intensive. In one of the five case studies described in [Vincenti 93], William F Durand and Everett P Lesley, professors of mechanical engineering at Stanford University, devoted themselves over ten years, from 1916 to 1926, to experimental wind-tunnel studies of the relative performance of aircraft propellers of many different shapes; Lesley continued to work on propellers for another twelve years until 1938. In another of Vincenti's case studies, the problem to be solved was the development of a satisfactory technique for flush riveting. Riveting the metal skin of an aircraft to the frame by round-head rivets caused significant aerodynamic drag that could be eliminated by ensuring that the rivet head was flat and flush with the skin. Because the metal skin was thin—1mm was not untypical—it was far from obvious how this could be done while avoiding damage to the rivet and weakening of the skin that would lead to loosening under the stresses of flight operation. In the mid-1930s engineers at a group of aircraft manufacturers studied the problem intensively. They experimented with various configurations of rivet shape and various ways of forming a recessed 'dimple' in the frame and skin. By around 1940 the design problem was solved, though improvements in the manufacturing process continued to be made into the 1950s and beyond. Wherever a problem is recognised, or an opportunity for worthwhile improvement, specialised normal design demands intense attention to every design detail.

3.4 Component Structure

A central theme in any normal design is *component structure*: that is, how the functionality of the product in question is distributed over its component parts at many levels, and how those parts are configured so that they work together effectively to embody the *operational principle* of the design. Quoting Polanyi [Polanyi 58], Vincenti explains that the operational principle of an engineering artifact is "how its characteristic parts ... fulfil their special function in combining to an overall operation which achieves the purpose [of the artifact]." He cites the remarkable example [Cayley 09] of Sir George Cayley's statement, published in 1809, of the operational principle of modern aircraft: "to make a surface support a given weight by the application of power to the resistance of air." A century before the Wright brothers, this operational principle clearly distinguished modern aircraft from the hot air balloons that had achieved some success in the eighteenth century and from the rigid dirigible airships that would be build a hundred years later, in the first decades of the twentieth century. In a more mundane example, wooden cart wheels and wire-spoked bicycle wheels differ radically in their operational principles: in a wooden wheel the spokes are in compression, but in a wire wheel they are in tension.

The decomposition of system function is, of course, only one conceptual part of the process of component structure design. The identified components must be arranged—and, if necessary, modified—to work together to achieve the system purpose. The design of this composition is itself a major part of normal design. What a community of engineers learns in the course of a long history of design advances is not only how to design the many components that provide the product's functionality, but also how best to configure them together. At the highest level an advance in component composition may involve the merging of functionalities by combining previously distinct functions in one component. In automobile engineering, the introduction of the unitary body in 1938 and of tubeless tyres in 1954 are examples of such advances. At a lower level composition involves the development of interfaces. The most cursory inspection of a car shows that every major interface between components has itself been the object of a design evolution. Each particular interface is closely tailored to the components it connects and to their interaction and cooperation. Where possible the interface is fully integrated into the components, as, for example, in the split bell-shaped housing within which the engine is connected to the clutch and gearbox.

For obvious practical reasons, a community of specialists engaged in normal design of products of a particular class will, over time, develop a refined nomenclature to refer to the normal components at the various structural levels of the product. Nomenclature emerges to denote the different classes of artifact that depend on different operational principles: *swing bridge, bascule bridge, suspension bridge, arch bridge, inverted arch bridge, cantilever*. Each will have its component nomenclature: *piers, chains (or cables), hangers, anchorage, voussoirs*. The existence of such a component nomenclature is a precondition for convenient discussion of design alternatives, and a salient symptom of specialisation and normal design.

3.5 Formal Analysis In Normal Design

The intellectual activities of formal reasoning and calculation play an important role in normal engineering, but they are conceptually subservient to the activity of selecting a design from the corpus of normal designs known to be candidates for the purpose in hand. A normal design brings with it a repertoire of formal analysis techniques known to be applicable. By deviating too far from the normal design configuration—let alone by selecting a radical design in its place—the engineer is likely to render the current tools of analysis and calculation unreliable or even totally ineffective.

One of the criteria of design selection is therefore availability of applicable analysis techniques specifically adapted to the putative design. When Robert Stephenson was considering possible designs for a railway bridge over the Menai Strait, he considered, but eventually rejected, a suspension bridge. According to [Clark 50], quoted in [Petroski 94], Stephenson believed that the heavy loads imposed by railway traffic—which at that time had never been carried by a suspension bridge—would alter the curvature of the suspension chains so dramatically that "the direction and amount of the complicated strains throughout the trussing [would] become incalculable as far as all practical purposes are concerned." In other words, in the absence of analytical or computational tools to predict how close to failure the design would be, Stephenson did not feel able to proceed with a suspension bridge design.

The broad intellectual structure, then, is the selection of a well-understood normal design pattern, followed by a combination of instantiating the pattern by choosing values for the options and for the variable dimensions, and formally analysing the resulting design instance to determine whether the choices made enable the design to satisfy the requirements. The process is likely to be iterative, options and parameter values being adjusted in the light of analysis of an earlier choice or of a subset of the possible choices. The formal analysis is crucial here, and depends fundamentally on scientific knowledge—for example, of static or dynamic mechanics: but without the initially chosen normal design there is nothing to analyse. An arbitrarily chosen configuration, owing nothing to any previously evolved normal design, will have many disadvantages: in particular, it is likely to be intractably difficult to analyse.

Yet, because the problem world and the artifact are physical and therefore non-formal, the results of the formal analysis must still be treated with caution. Normal design practice embodies knowledge of what formalisations are likely to yield more reliable models in particular cases; but the initial chosen formalisation itself—the analytical model of the artifact or problem world—is still only an approximation, and the formal reasoning and calculation is correspondingly unreliable. Unreliability increases as the chain of reasoning lengthens, especially where reasoning about the combined properties of component assemblages is concerned. As a noted structural engineer wrote [Addis 01]:

> "It must never be forgotten, however, that the primary models of loads, materials and structure are all idealisations and simplifications of the real world, and the behavioural output of the composite model is merely an infallible consequence of the information contained in the primary models, not of their real-world counterparts. Any predictions made from the output of the composite model about the likely behaviour of the completed structure must be treated with intelligence and care."

3.6 Normal Properties and Analysis

It is not too much to say that in dealing with the physical world formal reasoning can show the presence of errors, but not their absence. The practical empirical evidence from a long evolution of successful normal design of each class of artifact retains its central role in avoiding failure.

The configuration and properties that characterise each class are not formally, or even explicitly, defined. Instead there is a broad consensus among practitioners in the particular engineering area of the bounds within which a proposed design can be considered to be normal, and outside which it should be considered to be novel and therefore potentially problematical. The case of the Tacoma Narrows Bridge, destroyed in 1940 by the effects of a wind of only 40mph, illustrates the point well [Holloway 99]. The bridge designer, Leon Moisseiff, had adopted a somewhat novel mathematical theory for analysing the performance under load of a suspension bridge. In accordance with this theory the girders stiffening the bridge roadway were eight feet deep instead of the 25 feet proposed by the Washington State Highways Department. The roadway itself was also very narrow, being required to carry only light motor traffic. The resulting slenderness of the bridge, measured as the ratio of span to roadway width, exceeded that of the Golden Gate Bridge, completed only three years earlier, by more than 50%, the Golden Gate Bridge itself having exceeded the earlier maximum ratio by 40%. Theodore L Condron, consultant engineer for the insurers, pointed out this radical aspect of the design, and proposed that the roadway be widened from 39 feet to 52 feet to increase its stiffness. The proposals both of Condron and of the Highways Department were judged excessively conservative, and the construction of the designed bridge went ahead with the well known disastrous consequences.

The eventual consensus among engineers was that Moisseiff's deflection theory had considered only lateral deflections of the roadway. It was vertical oscillation that destroyed his bridge. With the wisdom of hindsight we are surprised at his seemingly obvious error. But it was not obvious to his fellow engineers, whether researchers or practitioners. What was obvious to Condron was simply that the proposed design had strayed too far outside the bounds of the normal. For that reason alone it should be recognised as potentially dangerous.

3.7 Normal Design and Requirements

The requirements of a system are, essentially, its desired properties. In the context of a normal design discipline, the desired or expected properties of the final product are a combination of properties stemming from two sources. Some *analytical* properties correspond to conscious design goals and choices, and must be confirmed by analysis and calculation; but others are *standard* properties that inevitably result from adopting the normal design. These standard properties may be known either explicitly or tacitly. If questioned about such a tacitly known property the engineer might well reply "I don't think that has ever been a problem with this kind of design;" or, questioned about an explicitly known property, might reply "That's completely standard—look, let me see if I can reproduce the calculation that justifies it." In a highly developed normal design discipline, these standard properties, whose justification is tacitly underpinned by the fact that the design adopted is the normal design, will greatly outnumber the analytical properties—those that demand explicit analytical justification. The analytical properties are, roughly speaking, those that vary with the design options and parameter values to be chosen by the engineer within the relatively tight constraints of the normal design: the analysis validates these choices. The standard properties correspond, roughly speaking, to every design choice made by all those who have contributed successfully to the evolution of the current normal design. The richer the evolution, the larger the number of these past design choices that by now are taken for granted.

Normal design makes requirements much easier to state. The requirements themselves will often belong to the standard requirements class corresponding to the product class; also, the existence of the standard product design invites the customer, or the customer's advisers, to state the requirements in terms of the chief design parameters. The requirement statement then falls naturally into two parts: one identifying the product class, and the other stating the parameter values. So a requirement for a desktop PC, for example, may be almost as succinct as 'Desktop PC, mid-tower, 500GB HDD, dual-layer DVD±R, 4GB RAM, 3GHz, 802.11b/g wireless;" and a requirement for a family car may be almost as succinct as "5-door hatchback, 1.6l diesel, 4-speed auto gearbox, sun roof, alloy wheels, leather seats." The normal design, of course, is hierarchical, and so too are its implicit requirements: the desktop PC requirement clause "500GB HDD" states not only the "500GB" storage capacity parameter of the PC, but also mentions the name of a normal design component class, "HDD," on whose standard properties the customer is again entitled to rely.

Of course, these are extreme and exaggerated examples. Individual purchasers of PCs and cars often have many detailed preferences, and their initial desires may be very dimly perceived and far removed from any product specification. However, it remains true that for a normal design product much of the semantic weight of the requirement statement is carried by the name of the product class. That name brings with it a large set of requirements that the variants of the product class are known to be capable of satisfying, and a set of product design options and parameter values whose possible choices are mapped by experience to the possible requirements. For a radical design, of course, the requirements are harder to state. First because in the absence of the standard properties inherent in a normal design it becomes necessary to state the requirements explicitly in far more detail; and second because in the absence of the known structure of design choices and values it is much harder to find a good structure for the requirements statement. Development of explicit detailed requirements from first principles is very expensive, and unlikely to succeed.

3.8 The Role of Failure

Petroski emphasises [Petroski 94] the fundamental importance of failure in engineering practice:

> "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."

It is not only for ethical or legal reasons that engineering failures demand careful investigation and analysis. The experience and analysis of failure contributes vitally to the improvement of normal designs. Directly, it prompts improvements specifically aimed at avoiding similar failures in the future. In the early 1950s several De Havilland Comet 1 aircraft suffered catastrophic structural failure in the air. Pieces of one of the aircraft were retrieved from the sea bed and the failed structure was reassembled at an aeronautical research centre. This enormously expensive exercise showed clearly that the failures were due to metal fatigue, and that the corners of the aircraft's square passenger windows had provided sites at which the fatigue cracks started to develop. This is why aircraft windows today are always rounded, avoiding angular corners. It is important to observe that the lesson learned was not "Engineers must perform more careful analysis;" nor was it "Engineers must consider metal fatigue," or even "Aeronautical engineers must consider metal fatigue." It was "To minimise the risk of metal fatigue, aeronautical engineers must avoid local design configurations at which fatigue cracks can easily originate, ensuring, in particular, that apertures in the structure for windows and for passenger and cargo doors have rounded corners of large radius."

Less direct, but no less important, is the general role of failures in helping engineers to understand their designs more fully. Every engineering product has an envelope of possible satisfactory operation, the envelope being defined by the interactions of the many loads imposed on the product externally by the problem world and internally by its own weight and other properties. Engineers design with safety factors, whose purpose is to ensure that operation never strays over the boundary of this envelope. These safety factors are sometimes called 'factors of ignorance', because the designer rarely has exact knowledge of the boundaries of the envelope or of the conditions that can obtain in operation. Any failure is important because it identifies a specific point near but beyond the boundary, and so helps to map the boundary more exactly. Increased safety factors may then enlarge the envelope to accommodate the wider range of conditions now known to be possible. Interestingly, Petroski points out [Petroski 86] that safety concerns can generate a cyclic pattern. When safety factors have increased and failures become very rare there is a tendency to believe that the products in question are over-engineered; the safety factors are then progressively reduced until the incidence of failures eventually increases and the cycle repeats itself with a call for increased safety.

Engineers design with failure in mind. That is: they consciously consider the consequences of failures. Because their artifacts are physical, failure is eventually inevitable; even within the designed life of the artifact unexpected changes in the problem world can cause failure. A general principle in engineering loaded structures such as bridges and buildings is that the designer must consider *alternate load paths*. When one component fails the load it was carrying will be distributed to other components of the structure, which should be strong enough to carry the increased load and so avoid a cascading failure of the whole system.

3.9 Unique and Standard Problem Worlds

Many failures in the established engineering branches are of bridges and large buildings. One factor in the difficulty of designing such structures is that their problem worlds are always unique, at least in some of their given properties. The design of a large suspension bridge over a river must be closely tailored to the particular properties of the terrain in which the towers and the cable anchorages will be embedded, to the water flow around the towers, to the navigation traffic in the river, to the ambient weather and winds, and to the characteristics of the traffic to be borne by the bridge. The structural design of a building must take account of the ground on which its foundations will rest, any restrictions imposed by surrounding buildings, vibrations caused by nearby road or rail traffic, and the impact of the local weather and winds.

Because their problem worlds are unique, such artifacts themselves are also at least partly radical in design. The Tacoma Narrows bridge has already been mentioned. Another example is the roof of the Hartford Civic Center Arena, which collapsed [Levy 92] under the weight of a heavy snowfall in 1978. The two-and-a-half acre roof was supported by a novel space-frame made possible only by the adoption of recently available computer software to calculate the stresses involved. Lack of experience with the new technique led to two sources of error in the design. First, the calculations of stress at the outer boundaries of the space frame, where the failure originated in the buckling of a horizontal component, were inadequate. Second, the construction company found it impossible to fabricate the frame on site exactly as designed. Some of the components whose centre lines should have met at a point were in fact slightly offset; the consequences of this apparently small deviation from the design were not calculated until after the collapse, when they were found to be large.

Unique problem worlds and radical artifacts are two sides of the same coin: they introduce into the design problem factors of which the designers have too little experience. It is not surprising that catastrophic failures are much rarer in normally designed artifacts operating in standard problem worlds. This is true particularly of products, such as cars and aeroplanes, that are manufactured in large numbers of many different variants of many different designs. For the engineering of such products the problem world is, to a considerable extent, standardised, and its properties have been systematically codified over many years. For example, the driver and passengers are parts of the problem world for a motor car. Their physical properties—weight, strength, resilience, resistance to crushing, physical dimensions, dexterity in operating the controls—are sufficiently standard for crash testing to be carried out using standard dummies, and for the requirements for a car's seating and driving position and controls, and the sizes of doors, to be largely standardised. The design of motor cars is based also on standard assumptions about the surfacing and configuration of roads, about the available fuel, and about the earth's atmosphere close to its surface. For the working engineer, these standard assumptions become almost unconscious, demanding at most occasional reference to tables in a handbook. For the ordinary lay observer the highly evolved adaptation of the artifact to its standard problem world becomes almost invisible. Sumo wrestlers and professional basketball players see it more clearly.

4 SOME TENTATIVE COMPARISONS

The foundation of success in the established engineering branches, as it has been described in the preceding section, is normal design; and the foundation of normal design is specialisation. Specialisation stimulates the increase of knowledge in many dimensions and in many overlapping areas from the most theoretical to the most empirical. It encourages the growth of communities by whom knowledge is preserved and increased, and of intellectual and social structures within which knowledge gained is codified and becomes an immediately available resource for working engineers. Specialisation and normal design are therefore fundamental topics for comparison with the practices of software engineering.

4.1 Specialisation in Software Engineering

Several attempts have been made in recent years to establish a taxonomy of topics in software engineering expertise. For example, the IEEE Guide to the Software Engineering Body of Knowledge [SWEBOK 04] lists ten knowledge areas:

> Software requirements; Software design; Software construction; Software testing; Software maintenance; Software configuration management; Software engineering management; Software engineering process; Software engineering tools and methods; and Software quality.

Each knowledge area is broken down into several subareas. For example, the Software design are is broken down into: Software design fundamentals; Key issues in software design; Software structure and architecture; Software design quality analysis and evaluation; Software design notations; and Software design strategies and methods.

In a similar vein, Capers Jones, in an article on software specialisation [Jones 95], lists the specialisations his consulting company had found in organisations of every size. The organisations surveyed ranged from very small, with fewer than 10 software staff, to very large, with as many as 40,000 software staff. The list was:

> Architecture, Configuration control, Cost estimating, Customer support, Database administration, Education and training, Function point counting, Human factors, Information systems, Integration, Maintenance and enhancement, Measurement (productivity, quality, etc), Network (local, wide area), Package acquisition, Performance, Planning, Process improvement, Quality assurance, Requirements, Reusability, Standards, Systems software support, Technical writing, Technology (object-oriented, GUI, etc), Testing, Tool development.

The SWEBOK topics, and the specialisations listed here, lay their emphasis on the general processes of software development rather than on the specifics of its products. While these general processes are of obvious importance, they can play at most a supporting role to the kinds of specialisation that are basic to the established engineering branches and that nourish the evolution of normal design.

For successful engineering, the essential specialisations are product specialisations. They are fundamental because the end products of engineering are its specific artifacts and the artifacts that they embody as components at every level. The object of normal design is a class of engineering artifact. Aeroplanes, for example, are developed by practitioners of many interacting specialisations from the most theoretical to the most practical; but above all they are the product of engine designers, undercarriage designers, fuselage designers, wing designers, and, of course, aeroplane designers. There is no substitute for this kind of specialisation by artifact, because it brings together in one place the totality of the experience that the product will eventually provide. Of course, the end product of one engineering specialisation may be a component in the end product of another, so the notion of ‘end product’ is somewhat relative; but this relativity does not blur the edge of the specialist engineer’s responsibility for the performance and quality of the delivered artifact.

A more relevant example of specialisation in software engineering is therefore the construction of compilers, which became a commercial specialisation at the beginning of the 1960s. At that time, when computer instruction sets and architectures varied between manufacturers, and between different models from the same manufacturer, companies such as CSC and Digitek undertook to build Fortran and other compilers that manufacturers could supply without charge to customers who bought or rented their extremely expensive hardware. Specialisation in compiler construction has continued to grow, along with important advances in knowledge of grammar theory, programming languages, parsing, and optimisation, and in practical developments of tools such as parser generators. It has also branched out into the more ambitious field of integrated development environments. Other examples of successful product specialisation in software engineering include relational DBMSs, file systems, SAT solvers, web page builders, operating systems, GUI builders and web browsers.

4.2 ‘System’ and ‘Application’ Software Products

A high proportion of the most successfully specialised products of software engineering are of classes that are commonly used by software developers, or otherwise familiar or accessible to them. They include the tools of the software development trade, such as compilers, interpreters, version management systems, editors, word processors and GUI builders, and the components of the computing infrastructure, such as operating systems, file systems, DBMSs, router software and web

browsers. They also include some products, such as spreadsheets, that solve symbolic problems: their problem worlds are easily understood by a software developer, and are uncomplicated by the buzzing blooming confusion of the physical and human world. I shall characterise these as *system* software products, contrasting them with the *application* software products that compose systems in such areas as avionics, administration, business, telephony, smart home systems, banking, e-commerce, process control, medical therapy and manufacturing. Of course, this distinction between system and application artifacts is very rough and ready. Not all system software artifacts are the product of successful specialisation, and not all application artifacts exhibit the defects of a lack of normal design: ATMS, for example, can be expected to work reliably and well. But the broad distinction stands up well in the light of the evidence of system failures reported in The Risks Digest [RISKS 08]. For most application artifacts and systems, dependability has been hard to achieve. It is here that the relative lack of evolving specialisations in software-intensive systems, and hence of normal design and normal design practice, has had its harmful effect.

At first sight it might be thought that many, or even most, of these application software-intensive systems—especially in socio-technical applications where human participants form a dominant part of the problem world—should be relatively undemanding. They rarely pose design challenges as intricate, complex or critical as the generation of highly optimised code in a compiler, or the correct management of transactions in a heavily loaded DBMS. In many systems, much of the functionality is associated with interaction with human participants in their roles as operators, users or sources of information: this interaction, surely, must be simple enough for the human participant, and cannot therefore pose significant problems of understanding for the software developer. It seems reasonable to expect that an effective approach to such systems can be based on the application of universal software engineering principles. There is no need for specialisation or normal design. Sound application of a well-understood general development method, in a disciplined environment, will be enough.

This optimistic approach may be effective for some software-intensive systems; but it pays too little attention to the sources of difficulties and obstacles that application systems often present. One source of difficulty has already been discussed in an earlier section. Because of the non-formal nature of the problem world, including its human parts, any formalisation is at best imperfectly reliable. The task of choosing a good enough formalisation, and designing the treatment of any residual deviations from it, demands specific experience-based knowledge of the system class and of its problem world: it cannot be adequately addressed on the basis of general principles alone. Another source of difficulty is the need for a highly evolved design, which cannot be achieved by sporadic attention from generalists: it demands specialised effort over a considerable time. And yet another source of difficulty is the increasingly multifarious nature of software-intensive systems: a typical system comprises many heterogeneous functions and features whose interactions can potentially give rise to complexity that grows exponentially with the number of functions.

Two of these sources of difficulty—non-formality and feature richness—need impose no great design burden in a system amply equipped with human overrides. Wherever the problem world behaviour goes outside the bounds of what the developers have anticipated, a human override—for example, a corrective credit or debit applied to a bank account at the manager's discretion, or an operator's overriding intervention in a process control system—can avoid failure and restore an appropriate system state and behaviour. Unfortunately, the availability of such human overrides militates against the economically attractive goal of increasing automation, and for that reason they become less attractive in the more ambitious systems where the need for them is greater. To make such systems dependable lays a heavy responsibility on the investigation and analysis of the assumed problem world properties on which the machine will rely to satisfy the requirement. Wherever the world fails to conform to the assumptions, the machine's behaviour will be defective, admitting no possibility of human intervention to rescue the system from failure. The most dramatic illustration of the point is an old one. On 5th October 1960 the US Ballistic Missile Early Warning System indicated that a major missile attack by the Soviet Union was in progress. No counter-attack was launched, however, because the system was not fully automated: an urgently convened meeting of senior military experts judged for quite extraneous reasons—Khruschev was in New York at the time, and US-Soviet relations were not unusually tense—that the indication was faulty. In fact, the rising moon

had caused the system's radar signals to be reflected in a way that the designers had not anticipated [Cantwell Smith 85]. The human override avoided a disastrous war.

4.3 Radical Design in Software-Intensive Systems

The relative lack of product-oriented and component-oriented specialisations in software-intensive systems has had its inevitable result: too much development is essentially radical design. Certainly many individuals, and some organisations too, have accumulated substantial experience in particular areas, but this distributed experience is not an adequate infrastructure for the evolution of normal design. In the absence of specialised communities and their mechanisms for collating, recording and distributing design knowledge, the expertise of individuals is likely to be dissipated and lost. The clearest evidence of lack of specialisation can be seen in the literature on almost any aspect of software development in application software-intensive systems: discussion is conducted on a general level, seemingly on the assumption that the differences between one class of system and another are unimportant for the purpose in hand.

Radical design, of course, is not absent in the established engineering branches. Vincenti points out [Vincenti 93] that:

> "Design, apart from being normal or radical, is also multilevel and hierarchical. Interacting levels of design exist, depending on the nature of the immediate design task, the identity of some component of the device, or the engineering discipline required. ... Whether design at a given location in the 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."

In software engineering, too, there is an intermixture of normal and radical design, but unfortunately it cannot be said that normal design usually prevails throughout. The engineering of a software-intensive system too often has an excessive ingredient of radical design at every level. Much of this is not recognised to be radical design: the design task seems too simple and straightforward to demand the codified knowledge and constrained development of a normal design discipline. But just as it is easy to write an incorrect small program for an operation on a linked list, so it is also easy to fail by making inappropriate assumptions about the problem world of a small component in a software-intensive system, or about the behaviours and interactions of the many parts of an apparently simple software-intensive system.

At the level of the complete system, the lack of normal design can make it hard to know what is possible, or to know what technical and other resources will be needed. There is a long catalogue of failed projects, often projects proposed by governments, whose overall requirements eventually proved to be far beyond the achievable range of existing design practice. A notorious example is the US Government's proposals for the Strategic Defense Initiative, put forward in the early 1980s. David Parnas resigned from the Panel on Computing in Support of Battle Management, convened by the SDI Organization, and published his reasons in [Parnas 85], where he explained "the properties of the proposed SDI software that make it unattainable." A major part of his argument rested on the complex unpredictability of the non-formal problem world of missiles, decoys, sensors, weapons and targets; another part rested on the sheer unprecedented scale of the proposed system.

The technical difficulty of finding good enough formalisations for the problem world is found also in apparently simpler environments with comparatively low levels of technology. The currently planned system for the UK National Health Service, intended to computerise records and care administration for 50 million patients at a cost currently estimated at £12.4 billion, seems unlikely to deliver its intended benefits. According to Brian Randell and his colleagues [Randell 07], the apparent mistakes include excessive centralisation of system function and the letting of huge procurement contracts for inadequately specified deliverables. These mistakes are recognisable from general principles alone; but there is also a large failure to address the socio-technical issues. These are essentially concerns about the problem world, including the possible and expected behaviour of people interacting with the system as patients, doctors and administrators. The design of the system with respect to these interactions has evidently been radical: the designer had no presumption of success. The result is that many interactions are troublesome, leading to users' efforts to circumvent designed system constraints, with the consequence that design assumptions about system data are

invalidated. For example, staff in some hospital departments circumvent security controls because they are too cumbersome and obstruct timely response to medical emergencies: the design goal of traceable responsibility for patient treatment is entirely frustrated. General medical practitioners circumvent the procedure for referring patients to specialised consultants because the procedure assumes that a specific diagnosis is already known, which is often not the case. In a highly connected system, local failures like these are likely to combine to produce failures on a larger scale and even to bring the whole system close to inoperability.

4.4 Mitigations for Radical Design

Much of the substance of software engineering discourse, and of the approaches and methods proposed for development, can be regarded as efforts, in more than one dimension, to mitigate the effects of radical design, working towards a reasonable product in spite of the absence of directly relevant normal design.

One dimension of effort, focusing on the form of the development *process*, has classic representatives at its two opposite poles. The *waterfall* process is implicitly based on the belief that careful and systematic thought can compensate for lack of experience. Development proceeds, phase by phase, from establishing system requirements to software design, coding, testing and deployment. In a pure waterfall process there is some iteration, in which the output of one phase can be revisited and modified when a defect is revealed in a later phase; but the broad scheme is to work towards a single delivery of a complete product. At the opposite pole are the *agile* processes. Development starts with a very vague and brief statement of some part of the system requirement; a very partial product is built and put it into operation, and the results are evaluated. The next increment of functionality is then selected, and the process repeated iteratively, the developers being willing at each iteration to modify and even restructure what has already been built and installed. Agile processes are explicitly based on the belief that lack of experience should be compensated for by experiment and feedback rather than by deeper investigation and greater care and precision in initial design. It could be said that such agile approaches echo the travails of pioneering inventors.

Another dimension of effort is *structural*, focusing on the structure of the system and of the problem it is intended to solve. One approach, much used in object-oriented software development [Meyer 88], is to identify classes of entities in the problem world and to associate a software object class with each one, the object instances providing a kind of simulation, or surrogate, for the individual problem world entities. Further behaviour and further object classes can then be added to provide additional system functionality. Conceptually, this approach has much in common with JSD [Jackson 83] in which problem world entities in an information system are associated with software processes, and further behaviour and further processes are added to provide the system's desired information outputs. Another architectural approach, *KAOS* [Darimont 96], is very different. It relates system functionality to requirements or *goals* that are formally decomposed. Responsibility for achieving each goal is eventually assigned to one or more *agents*, the agents being either parts of the software or parts of the problem world.

A central question in structural approaches to development is the relationship between the structure of the problem—that is, of the requirement and problem world—and the structure of the software. Sometimes the gross software structure is inescapably determined by the machine environment. The most obvious example of structural determination is the *Three-Layer Architecture*, which structures a client-server application into a *Presentation Layer* running in the client computer, an *Application Layer* running in the server, and a *Data Layer* provided by the server's installed DBMS. More often there is a freer choice of software structure, and the choice may be made at a gross level by selecting an *architectural style* [Perry 92, Shaw 96]. The designer selects a style, and must then allocate the functionality of the system to software components of that style. When a uniform style is chosen in this way, in which each component conforms to the same general type constraints—for example, each component is a *passive object*, a *filter*, or a *procedure*—the technical task of fitting the components together is greatly simplified; but the task of fitting the functionality into the Procrustean bed of the component type is likely to be difficult. An interesting discussion of these difficulties in the design of the software for an oscilloscope is given in [Delisle 90], and further general discussion in [Shaw 96].

4.5 A Problem-Oriented Approach to Structure

Decomposition and structuring of system functionality is an essential tool in mitigating the effects of radical design. Functional decomposition means decomposition of the problem to be solved by the system as a whole, and this problem is firmly located in the problem world. It is therefore appropriate to regard components of system functionality as subsystems—or *subproblems*, in the sense that each component conforms to the problem diagram of Figure 1.

This approach is based on the notion of *problem frames* [Jackson 01] that has already been mentioned; it is illustrated by fragments of a simple example discussed in the following sections. The purpose of the discussion is not to propose or advocate a design method, nor to illustrate novel or recommended solutions to significant design problems. It is to reveal some of the concerns that arise in understanding the overall design problem, in identifying an appropriate set of system components, in designing the functionality of each component, and in configuring and connecting the components to achieve the overall functionality of the system. The example is intended only as a stimulus to thought, not as a serious depiction of a realistic system.

The characteristic of this approach that makes it suitable to the central theme of the paper is that it makes the relationship between the problem, the problem world, and the machine functionality fully explicit in a very direct way. The parts, or *domains*, of the problem world with which the component machines interact in providing each part or aspect of its functionality are explicitly shown in their problem diagrams. Such components can perhaps offer a basis for specialised normal design focused on the design artifact. The possibility of implementing each subproblem machine as a software module of the whole machine is not excluded, but neither is it assumed: a subproblem machine may be only a projection of the implemented whole machine. The primary purpose of the approach, as its name suggests, is to permit analysis and understanding of the development *problem*: the eventually implemented machine will be the *solution*: this perspective is adopted at every level of the design hierarchy.

A component or subproblem is regarded as having its own machine, its own problem world, with which it interacts, and its own requirement. The component machine may be software executed by the same computer as other, perhaps all other, component machines, and it may be distributed among several software modules. The component problem world may contain problem world parts, or *domains*, that appear also in other components' problem worlds; and some of the shared phenomena by which it interacts with its problem world may be common to other components.

Each component machine in a software-intensive system interacts with the relevant parts of the system's problem world, and is responsible for satisfying its own part of the system's requirements. It will also interact with other component machines, both directly by issuing and responding to control instructions and indirectly by accessing shared data structures in primary or other storage and by interacting with additional components introduced specifically for purposes of composing the component machines. These additional components and shared data structures may be partially or entirely internal to the undecomposed machine, just as the electrical wiring harness and the cardan shaft are internal to a car. At the level at which they are the objects of design, their problem worlds contain the subproblem components whose interactions they serve. The design of the whole system, then, comprises not only the identification and design of the components among which the system functionality is distributed, but also the explicit design of their interactions as a distinct development concern.

4.6 Problem-Oriented Components: an Example

In a very small system to impose one-way vehicle traffic over a segment of road under repair, the segment is guarded by traffic lights at each end. Sensor tubes fixed to the road surface at the segment boundaries detect the passage of a vehicle as its wheels compress each tube in turn. A control computer, connected to the light units and sensors, is equipped with a small keyboard and a simple character display. These are used from time to time by the site manager to specify the lengths of the traffic phases, thus allowing different absolute and relative traffic densities in the two directions to be accommodated with minimum inconvenience to the road users. Figure 2 shows a tentative problem diagram:

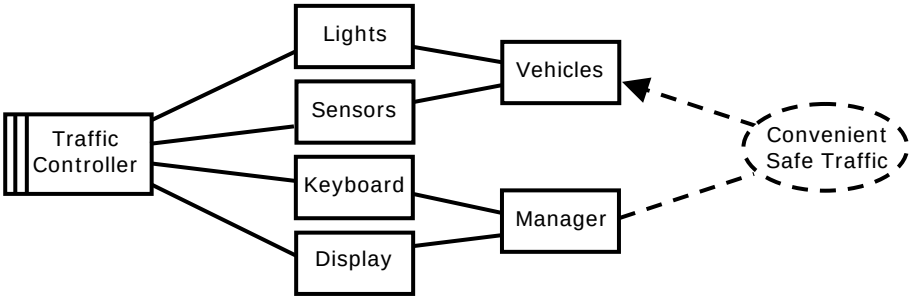

Figure 2. Problem Diagram: One-Way Traffic Control

The requirement stipulates that the machine, the Traffic Controller, must constrain the Vehicles to achieve *Convenient Safe Traffic* in accordance with the Manager's most recent specification of phasing. The Vehicles, like aircraft in an air traffic control system, can be constrained only under the broad assumption that their drivers obey the light signals. By contrast, the Manager need not and can not be constrained: the machine can reject or ignore inappropriate keyboard inputs. As in Figure 1, the solid lines connecting the problem world domains to each other and to the machine represent interfaces of shared phenomena.

Developers responsible for designing this small system should, to minimise development risk, start by looking for an established normal design discipline for the whole system: not in the expectation of finding a ready-made solution that can be used directly, avoiding all development cost and greatly reducing uncertainty, but rather of identifying an established body of specialised design expertise in systems of this kind. (The question "What do we mean by 'systems of this kind'?" is, of course, just one instance of the overall question "What specialisations should exist in software engineering?") However, the present discussion will proceed on the assumption—possibly false—that no such overall established normal design is available. The difficulties of radical design are to be mitigated by identifying some normal design components that must then be combined within a radical structure.

Since our purpose here is to discuss the nature of components, we will take only two examples, leaving aside most of the design task. Two candidate components are immediately obvious. One is a component to handle the Manager's input of phasing specifications: this input process must be decoupled from the process of controlling the lights according to one of the previously completed specifications. This decoupling, in the usual way, requires the introduction of a data structure. Figure 3 shows the resulting subproblem diagram for the component whose function is to support editing of the phase specification:

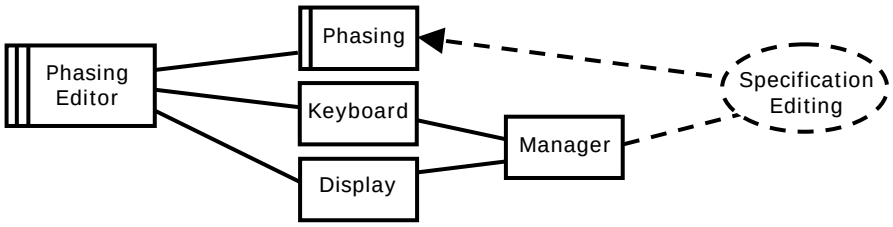

Figure 3. Problem Diagram: Editing a Phase Specification

The Phasing problem domain in this component is a part of the Traffic Controller machine. It is a *lexical domain*: that is, a data structure made concrete in primary or other storage of the computer. The stripe on the box indicates that although it appears as a problem domain, it was not given in the original problem. It is ultimately a part of the complete machine, and must be designed by the developers.

A second obvious candidate is a component to collect and interpret the information from the sensors, allowing the Traffic Controller machine to detect whether any vehicle has entered the controlled segment but not yet left it, and thus whether it is safe to allow traffic flow in the contrary direction. This function too can profitably be decoupled from the function of controlling the traffic in

accordance with the phases specified by the Manager and by the Vehicle positions. Once again the natural mechanism for decoupling these two functions, one producing information and the other consuming it at a different time or place in the system structure, is a lexical domain. Figure 4 shows the subproblem diagram:

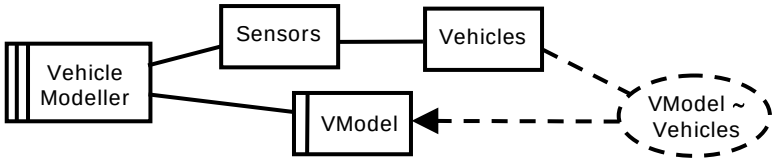

Figure 4. Problem Diagram: Building a Vehicles Model

The VModel domain functions as a surrogate or *model* of the Vehicles domain. This is not an *analytical* model expressing general properties of the Vehicles domain, but an *analogue model* in the sense that its state at any time is required to satisfy a correspondence with the state of the Vehicles domain. This correspondence allows it to be used by the Traffic Controller as a surrogate for the Vehicles: inspecting the VModel state will provide necessary information about the state of the Vehicles. In this subproblem only the VModel domain is constrained by the requirement: the behaviour of the Vehicles is regarded as autonomous. Again, the model is a designed domain: it was not given in the original problem, and must be designed by the developers.

4.7 The Content of Normal Design

It was suggested in the preceding section that the Editing Phase Specification and Building Vehicles Model subproblems are two obvious candidates for components of the system. They seem to represent 'natural' components of the system functionality; they are simpler than the whole system because their problem worlds are smaller and their requirements more limited; and they seem to be candidates for normal design, in the sense that they are examples of two recognisable problem types that are often found in software-intensive systems. In one, information, provided by human input to one part of a system, is to be saved in an internal document for later use elsewhere; in the other the changing state of some dynamic problem domain is to be detected, interpreted, and captured in a convenient representation to be used to guide system behaviour.

Merely pointing to a roughly recognisable problem type in this way is far from enough to delimit a class of normal design objects (Vincenti would call them 'devices'), let alone to provide the substantive content by which 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." That can be achieved only by the long evolution that characterises normal design in the established engineering branches. However, we can say something here about the natures of the two devices we have identified and about the concerns that will arise in their design.

At a superficial level, a more abstract view might suggest that the two subproblems we have identified belong to the same class because they exhibit the same structure. Each has a lexical domain (the Phasing or the VModel), to be created or maintained by the machine; an autonomous domain (the Manager or the Vehicles), that is the source of change; some parts (Keyboard and Display, or Sensors) connecting the autonomous domain to the machine; and a requirement that some specified relationship should hold between the lexical and autonomous domains. However, such abstraction and assimilation can seem persuasive only for the earliest incunabula of an engineering branch, whose designers must rely heavily on native wit informed by general principles. Assimilating the two problems misses the central point of normal design evolution, which lies in refining, mastering and exploiting the given and potential particularities of the device class in question, its desired function, and its problem world. Both early railway locomotives and early motor cars were sometimes conceived as a new kind of horse-drawn carriage; but as soon as normal design began to evolve from the initial radical attempts it became apparent that the problems and the desirable solutions were very different.

The two components we have identified differ greatly in the fundamental natures of their positive requirements. The purpose of the editing component is to provide a tool for human use. It must support the Manager's need to capture, and express in the designed representation, the intended pattern of traffic phases: the result must conform to certain syntactic and semantic constraints, and must accurately represent the Manager's intention. The purpose of the model-building component is to maintain a specified correspondence between the given autonomous domain of the Vehicles and the designed model domain: the resulting model must be continually updated to reflect the changing states of the Vehicles domain. These different purposes might suggest different principles of operation for the devices to achieve their positive requirements. For example, the display might be exploited to support an interactive style of question-and-answer input for the editing component: there is no equivalent for the model-building component.

4.8 Failures in Component Design

The negative requirements, to avoid predictable failures, also differ fundamentally between the two components. Design of the editing component must avoid failures in ease and convenience of the editing activity. This falls in the area of human-computer interaction (HCI) and has been extensively studied in general, and in some particular contexts for particular tasks and classes of user. Vincenti describes [Vincenti 93] an illuminating example in aeronautical engineering. Early pilots often described particular aircraft as 'stiff' or 'responsive', or 'easy to fly', or 'hard to fly'. In the twenty five years from 1918 to 1943 these ill-defined notions were studied by aeronautical engineers with increasing intensity. Eventually the chief characteristics determining 'flying quality' were identified and quantified. For example, a crucial parameter in longitudinal control was shown to be 'stick force per *g*': that is, how much force the pilot must exert on the stick to produce a given longitudinal acceleration by moving the elevators to raise or lower the nose of the aircraft. More recently the vital importance of human factors has been recognised in the design of avionics and process control software: 'operator error' and 'pilot error' are more often an indictment of the system designers than of the unfortunate operator or pilot. The human factor concerns of editing programs, such as our little example, have also received some attention, although perhaps the results here are comparatively meagre.

Another negative requirement for the editing component, again associated with human factors, is to avoid misleading the user: it is a major failure if the phasing specification captured in the lexical domain is not exactly what the Manager believes has been specified. This apparently obvious class of failure was dramatically illustrated in a major contributory cause of the patient deaths and injuries caused by the Therac-25 radiotherapy system [Leveson 93] in the period from 1985 to 1987. The parameters of the radiation dose to be delivered were entered by the operator on a keyboard and displayed on a character-based screen under the control of a data entry and editing routine. The software design and implementation made it possible for the operator to exit from the routine and activate delivery of the dose while under a misapprehension about the parameter values that had been set: the screen display did not correspond to the values set in the physical equipment.

The chief negative requirement for the model-building component is avoiding failure to satisfy the positive requirement to a sufficient degree: that is, to reduce to an acceptable level the probability of system states in which the VModel fails to reflect the reality of the Vehicles and their positions. Depending on the kinds of road in which the system may be installed and used, the real vehicles may be of many shapes and sizes, with different numbers of axles. Sensors of the kind used can be activated by many other causes than the passage of a vehicle; also, the pattern of sensor state changes caused by a vehicle may depend on factors that are hard to predict, such as the exact position and orientation of the vehicle in the road. For a sufficiently dependable system it is necessary to achieve good enough reliability in the interpretation of sensor changes, and also to understand the limits of that reliability in designing the use to be made of the resulting model domain.

This challenge in a model-building component reflects a general difficulty, in matching the engineering artifact to the problem world, that is not typical in the established branches. The behaviour of a programmed machine is determined—up to hardware and infrastructure malfunction—by the application software. The software is designed, formally or informally, on the basis of some analytical model of the problem world devised or adopted by the developers. Here, for example,

interpretation of sensor state changes is based on some analytical model of the Vehicles and their possible properties and behaviours. If this analytical model is inadequate, the machine, limited by its program and by the alphabet of its interface to the problem world, cannot compensate for the inadequacy. By contrast, an inadequate analytical model of the problem world in the design of a physical structure can be compensated by the standard engineering practice of applying safety factors in the design. In software engineering, failures in analytical modelling of the problem world are more likely to result in system failures. Avoiding system failure by adopting a good enough analytical model is an essential part of normal design.

4.9 Composition in Software Engineering

It is a commonplace of software development, both in rhetoric and practice, that complexity must be mastered by separation of concerns. However, since the separated concerns belong to a single project, they must somehow be composed and brought together again to constitute the desired whole. In general, composition is a substantial design challenge in itself. Obviously, there is the need to connect the components so that they can work together. For example, the part of the Traffic Controller that controls the lights must be given access to the information about Vehicle states captured in the VModel domain, and also to the Phasing specified by the Manager. Making the model domain available is a classic composition of access to a shared data structure, and demands use of a standard software mechanism for guaranteeing mutual exclusion between writer and reader at an appropriate granularity. The use of a standard mechanism is an element of thoroughly normal design practice.

Making the Phasing available to the Traffic Controller is a more substantial composition problem. First, it seems clear that use of a single shared data structure, even with mutually exclusive access, will not be satisfactory. The controller must always refer to a fully coherent phasing specification, in which the different phases are in the correct relationships of ordering and duration, when altering the light settings. Allowing a finer granularity—for example, allowing read access whenever the Phasing is syntactically correct—would gratuitously introduce a class of potential failures whose avoidance would be complicated and difficult. The finest practicable granularity for mutual exclusion is therefore a complete phasing specification. Since the editing process cannot proceed faster than the Manager allows it to, the time between one fully coherent specification and the next, during the whole of which the light settings must remain unchanged, is potentially unbounded. So the editing process must operate on a different instance of the Phasing data structure from the instance currently being used for control. The design questions then arise: How and when will the changeover be made? Will there be two instances or more—or perhaps a database of instances?

The design of the changeover from one Phasing instance to another raises an aspect of composition design which appears in many guises in different contexts: we may call it the *switching concern*. In the present problem, control of the traffic lights must be switched from one specified phasing to another: the design problem is to arrange that the concatenation of the two phasings does not infringe some—possibly implicit—global requirement. Here, one such requirement is that any phase in which traffic has been allowed to flow in one direction is followed by a phase which prevents further vehicles from entering the controlled segment for long enough to allow the segment to become clear of traffic. Another global requirement may be that the two directions of traffic flow must alternate. Another example of a switching concern in a very different problem is the treatment of customers in a financial system. Accounts are normally managed according to some set of rules, but a different set of rules is applied to delinquent customers who default on loan repayments. Transferring a customer between the normal and the delinquent rules is a switching concern.

4.10 Designing for Failure

In general, composition design can raise many concerns. The composition of two simple components may be far from simple, and may offer many opportunities for failure. One lesson from the established branches is that component failure must be anticipated, and the designer must consider consciously how the system will behave when it happens. In particular, composition must respect the relative criticality of components, in the sense that the functionality of a more critical component must not be vulnerable to failure of a less critical component.

One example will suffice. A system to control a proton therapy machine may have components to set dosage, contour and direction according to the patient's prescription, to control the proton beam, to move the patient support bed, to rotate the gantry that positions the proton beam, to maintain an audit trail of commands sent to the equipment, to respond to the emergency button by switching off the beam, and so on. The relative importance of these functions demands careful consideration, and must be a major determining factor in implementation design. In one version of the software design for a certain system, dataflow among the corresponding software modules was arranged as shown in Figure 5:

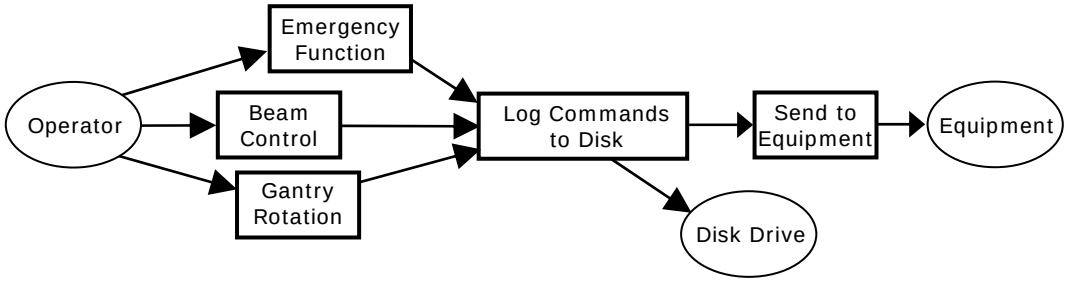

Figure 5. Dataflow in a Proton Therapy Machine's Software

The dataflow shown was chosen because the Log Commands module must log all commands sent to the equipment, including commands to switch off the beam in emergency. Unfortunately, the Log Commands module could fail if the disk were full: if it failed it neither sent the command to the Equipment nor responded to the component that had sent the command. The consequence was that running out of disk space made it impossible to switch off the beam in an emergency. The design error that concerns us here is not the inadequate design for the Log Commands module. It is the composition of the Emergency Function and Log Commands modules in an arrangement that allowed the Log Commands module to put the emergency button at risk.

4.11 Normal and Radical Composition

Normal and radical design differ fundamentally in the designer's approach to composition. In a normal design task the engineer knows at the outset not only what the components should be and how each should be designed, but also how they will work together, and how their composition should be implemented. To a large extent, the component interfaces necessary for satisfactory composition are already built into the standard component designs.

At the outset of a radical design, by contrast, the engineer does not know what the components will be, nor how they should be designed. At that early stage, when relatively little is known about the components, it is too risky to fix the design of the compositions. This is the fundamental difficulty in trying to apply top-down or refinement techniques in the context of radical design: the designer's task is to design the composition while simultaneously deciding what the components should be that are to be composed. A separation of concerns is needed, between the choice and design of the components, and the choice and design of their composition. The extent to which the component requirements and designs should be worked out in detail before their composition is considered will depend on many factors. The properties of a component that is itself an object of normal design can be anticipated with some confidence, so its design need not be carried very far before its composition with other components can be considered. A novel component should be considered in some depth and detail before its composition is considered. The penalty of component design rework to fit the designed composition is a price worth paying: it is likely to be smaller than the penalty of prematurely fixing the composition of components whose requirements and properties are at best only dimly perceived.

Richard Feynman, at the Challenger disaster review panel [Ferguson 92], made these observations about top-down and bottom-up design in the context of large NASA projects, in which the dominant design mode was inevitably radical:

> "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. ... Until the foolishness of top-down design has been dropped in a fit of common sense, the harrowing succession of flawed designs will continue to appear in high-tech, high-cost public projects."

The practice of designing the composition in terms of a chosen software architectural style does not overcome this difficulty: rather, it aggravates it in two ways. First, it focuses the designer's attention on the mechanics of the composition instead of on its content; second, it forces premature design decisions in such matters as the locus of control in component communication. As Shaw and Garlan showed in an account, cited previously, of software design for an oscilloscope [Shaw 96], early choice of an architectural style in a radical design context is likely to be little more than a barely supported conjecture.

4.12 Radical Requirements and Specifications

A significant benefit of normal design is that the tacit assumptions built into the standard artifact carry much of the burden of requirements and specifications: in an extreme case the specification may consist of no more than a few chosen options and parameter values. In radical design, evidently, this is not possible.

A natural reaction to this difficulty is to insist that the more radical the design task the more completely detailed should be the requirements and specifications. Unfortunately, the considerations that make pure top-down design impractical in a radical context conspire also to make the notion of complete requirements or specifications equally impractical. Of course, it is often necessary to pay careful attention to some particular properties of the eventual product, and it may be possible to state these properties precisely at the outset of the development project. For example, in a telecommunications system to be used by paying subscribers one such property is that subscribers should never be charged for a service that they have not themselves requested either in their subscription choices or during an episode of using the system. In an electronic purse system an essential property is that money must be conserved: the sum of the contents of two electronic purses engaged in a transfer must remain constant over the whole interaction even if the transfer is aborted. However, stating a set of necessary properties—even a large set—is a far cry from stating a necessary and sufficient requirement which ensures the adequacy of any system that satisfies the requirement and the inadequacy of any that does not.

In a realistic radically designed system there will usually be a number of components that are objects of normal design. Their requirements may be specifiable by appealing silently to the tacit part embodied in the normal design; and if the engineering of a new instance of a normal component involves relatively few design choices, it may be possible to give a formal and complete specification of the component machine's behaviour at its interface with the problem world. However, the whole system will still admit only a partial, incomplete specification. Satisfaction of this specification may be necessary for acceptability, but will never be sufficient.

4.13 Empirical Studies

Empirical methods, in the sense of systematic experiment or systematic examination of a population of existing cases, have played a fundamental role in the development of the established engineering branches. The experiments of engineers are different from those of natural scientists. Natural scientists seek truths that hold for the whole of nature. Inevitably these truths, and the search for them, must abstract from the accidental characteristics of particular situations and particular physical arrangements. Engineers, by contrast, seek to judge between different particular situations and particular physical arrangements, in order to learn how to devise the most effective designs for particular purposes. In these engineering studies, science plays an important role. The long process of overcoming steam boiler explosions in the nineteenth century [Leveson 94] depended crucially on the availability and development of scientific knowledge about the phenomenon of heat; but the goal of the engineers was to discover how to design reliable high-pressure steam boilers. (Ironically, as Leveson points out, in the United States the knowledge gained was for some time applied only to

boilers in steamboats: stationary and locomotive engines, which had not yet attracted the attention of legislators, continued to suffer explosions for several years. Specialisation can be excessive as well as insufficient, both in legislation and in engineering.)

Normal design is a precondition of effective empirical studies in engineering. The ideal context of experiment is the existence of a normal design discipline within which the optimal value sets for some particular choices and parameters are not yet adequately understood. The experiments, mentioned earlier in this paper, on flush riveting and on the efficiency of different propeller designs, were aimed at advancing two clearly defined areas of normal design by varying the values of a very small number of design parameters. The experimental results applied, and were intended to apply, to the normal design context in which they are obtained. Any wider applicability would be an unsought benefit, and would certainly demand separate confirmation in additional experiments.

Even within a normal design discipline, empirical methods have dangers when current theoretical understanding is insufficient to explain the results obtained. Vincenti, in one of his case studies [Vincenti 93], discusses the "Davis Wing", a novel design that provoked some controversy among aeronautical engineers in the late 1930s. The design seemed, on the basis of wind-tunnel experiments, to offer improved performance; but this improvement was unexplained by aerodynamic theory or orthodox design practice of the time. The Davis design was successfully adopted in the prototype Consolidated Model 31 and in the same company's B-24, which was one of the most important aircraft of the Second World War, but in no other major aircraft. Vincenti's summary comments on the whole episode are revealing. He acknowledges that the Model 31 and B-24 performed excellently for their day, but "no one can say for certain how the airplanes would have performed with a different airfoil. ... Perhaps we could call this decade the adolescence of airfoil technology."

Applied outside a normal design context, empirical methods are more vulnerable to two dangers. First, the range of applicability of their results may be ill defined. Aeronautical engineers could use the results of Durand and Lesley's propeller experiments with high confidence only because they knew they were developing propeller designs in the same normal design class that the experimenters had assumed. Second, a substitute for a missing scientific theory can, to some degree and for some engineering purposes, be provided by the constrained and conscious variation of parameters within a successful normal design discipline. If both scientific theory and normal design discipline are missing, empirical investigators can have no plausible basis for identifying the parameters to be varied, and for interpreting the empirical data that eventually emerge. In software engineering, regrettably, the tightly constrained environment of normally designed artifacts and normal design practice is seldom available. Empirical investigations must often suffer accordingly.

5 CONCLUDING REFLECTIONS

The phrase *software engineering* was originally coined with provocative intent, and in that respect it has certainly succeeded. A number of eminent computer scientists have responded to the provocation by refining, expounding and teaching their ideas about the relationship between the established branches of engineering and the discipline of software development as it is, or as it should be. The ideas of Parnas are to be found in the many papers he has written over a long career, a notable brief summary being [Parnas 97]. Maibaum has approached the matter from a more formal point of view [Maibaum 93, Maibaum 97, Haeberer 01], stressing particularly the dependence of software on logical and scientific foundations.

In this paper I have adopted a different approach, focusing rather on the structure of engineering practice, stressing similarities and analogies between software engineering and the established engineering branches. I have tried to identify critical respects in which I believe we have much to learn from their long history of specialisation and from the normal design artifacts and practices that are its fruit. *System* software, belonging to the toolset or the infrastructure of software development itself, exhibits many examples of successful specialisation. There is evidence of specialisation also in the work on object-oriented patterns [Gamma 94]. The deficit of software specialisation and of normal design is found chiefly in what I have called *application* systems—software-intensive systems whose purposes are firmly located in their physical and human problem worlds.

The relatively high incidence of failure is one very direct symptom of this lack of specialisation. Another important symptom, visible almost everywhere in the software engineering landscape, is a reluctance in the education, research and social contexts to engage deeply with particular concrete instances. We seem to prefer to occupy ourselves with concerns at a more general or abstract level. This preference militates strongly against the development of specialisations and against the increase of the knowledge of specifics and particulars that characterises the established engineering branches. The most cursory inspection of educational syllabuses and journals of research and development in the established branches shows a heavy emphasis on particular examples of engineering artifacts and their properties. For example, the Earthquake Engineering Research Center of UC Berkeley maintains a library [Godden 08] of nearly 1000 slides showing examples of real structural systems such as bridges and large buildings of many kinds. The purpose of the collection, originally made by Professor William G Godden between 1950 and 1980, and since then enlarged and enhanced, was to serve as a teaching resource for undergraduate and graduate courses. Students would 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. Similarly, a very high proportion of the articles in engineering research journals report investigations of narrowly defined specialised product classes at every level: for example “Calculation of Wind Drift in Staggered-Truss Buildings” or “Seismic Response Evaluation of Post-tensioned Precast Concrete Frames with Friction Dampers.”

To some extent this focus on the particular in the established branches is a natural consequence of their already highly evolved specialisation. But it is also a precondition and a cause of specialisation. By recording very specific studies, or carefully documenting specific designs, researchers and teachers offer practitioners a continually updated corpus of detailed knowledge that they must not ignore. If only because each practitioner can master and exploit only a small part of this corpus, specialisation is an inevitable outcome. In software engineering, by contrast, educational syllabuses most often concentrate on topics that can be—and are clearly expected to be—treated at a general or abstract level. Neither of the two industry-standard compilations of software engineering knowledge—[SWEBOK 04] and [Software 04]—refers to a single actual example of a software engineering artifact. Similarly, conference and journal papers in the field refer to actual examples of artifacts either not at all or only by way of a *case study*. The purpose of a case study is rarely to provide material for learning from experience. Its more usual purpose, on the dubious principle that one swallow does make a summer, is to cite application of a proposed tool or technique to at least one plausible instance, offering this weak evidence to support a claim that the research described is of general—or, at least, wide—applicability.

There are many reasons for this lack of interest in real software engineering examples. One is the extreme difficulty that any realistic example places in the path of a would-be student. The sheer volume of the program text, and, usually, the absence of useful documentation of the problem world, the software structure, and the development decisions and their implementation, contrast with the relative ease with which the structure of a bridge, a building, a ship or an aeroplane reveals itself, at least in outline. For some *system* software artifacts—belonging to the toolset and infrastructure of software development—this deficit is now being partially repaired by the availability of open-source program code on the internet; many researchers are engaged in trying to analyse both the program code and the evidence of its design structure and of the development stages by which it has evolved. However, for software-intensive systems generally, the hopes of Stoy and Strachey [Stoy 72] that software publication would become the norm, and that actual examples, good and bad, would provide lessons for students, are as far from fulfilment as they were thirty six years ago. Commercial secrecy is not the only barrier to publication of a program of 20 million lines of code.

Another reason is a widespread uncertainty about the nature of computer science and software engineering and their respective roles. If computer science is regarded as a branch of pure mathematics, the computer scientist should not be expected to show interest in actual examples. An example can be interesting to a pure mathematician only to the extent that it is a counterexample to a theoretical conjecture, or that it stimulates the formulation of a new conjecture. This lack of interest, then, is not surprising. The practicalities of balancing complicated commercial accounts, for example, may be hard to master, but they reveal no new mathematical truths about arithmetic, and are therefore

of no interest to a number theorist. Study of the problem world of a software-intensive system, and its interactions with the machine, offers little or nothing either to the pure mathematician or to the natural scientist.

Software engineering practitioners, on their side, are notoriously, and astoundingly, sceptical about the relevance of computer science to their work. They, too, neglect the problem world and its specific manifestations: not because their minds are occupied with mathematical theorems, but because they are occupied with the ever more demanding technicalities of the complex development and execution milieux that they use.

The central thesis of this paper, that software engineering needs specialisations focused on system and component artifacts, seen as synergetic combinations of machine and problem world, has yet another hurdle to surmount. In their different ways, both the computer scientist and the software technologist focus their attention on concerns that abstract entirely, or almost entirely, from any particular problem or particular problem world. Each, then, has implicitly adopted a view of software engineering that aspires to be universal, or almost universal, across all problems and problem worlds. We want to join the established engineering branches by adding *software engineering* as one new member of the established engineering set {*aeronautical, automotive, chemical, electrical, ...*}. We do not want to hear that this long-standing ambition must be fundamentally modified, and that we must develop product specialisations within software engineering that are no less rich, no less highly evolved, and no less focused on particulars, than those already existing among the established branches. But it may be the truth.

ACKNOWLEDGEMENTS

Daniel Jackson very kindly read a very early draft of this paper and made many excellent suggestions for improving it. Valuable suggestions and comments have also been made by Daniel Berry, Manfred Broy, John Cameron, Peter Freeman, Anthony Hall, Jon Hall, Tony Hoare, Butler Lampson, Robin Laney, Ashley McNeile, Tom Maibaum, Jonathan Moffett, David Notkin, Dewayne Perry, Lutz Prechelt, Brian Randell, Kevin Ryan, Fred Schneider, Michel Sintzoff, Thein Than Tun, Shmuel Ur and Roel Wieringa. I have learned much from their comments, and I am grateful to them all.

REFERENCES

[Addis 01] Bill Addis; *Creativity and Innovation: The Structural Engineer's Contribution to Design*; Architectural Press, 2001.

[Cantwell Smith 85] Brian Cantwell Smith; *The Limits of Correctness*; Prepared for the Symposium on Unintentional Nuclear War; Fifth Congress of the International Physicians for the Prevention of Nuclear War; Budapest, Hungary, June 28 - July 1, 1985.

[Cayley 09] Sir George Cayley; On Aerial Navigation; Nicholson's Journal, issues of November 1809, February 1810, March 1810.

[Clark 50] Edwin Clark; *The Britannia and Conway Tubular Bridges: With General Inquiries on Beams and on the Properties of Materials Used in Construction*; Day and Sons, London 1850.

[Constant 80] Edward W Constant; *The Origins of the Turbojet Revolution*; The Johns Hopkins University Press, 1980.

[Darimont 96] R Darimont and A van Lamsweerde; *Formal Refinement Patterns for Goal-Driven Requirements Elaboration*; Proceedings of the Fourth ACM SIGSOFT Symposium on the Foundations of Software Engineering, San Francisco, pages 179-190, October 1996.

[DeLisle 90] Norman Delisle and David Garlan; *Applying formal specification to industrial problems: A specification of an oscilloscope*; IEEE Software, Volume 7 Number 5, pages 29-37, September 1990.

[Dijkstra 89] Edsger W Dijkstra; *On the Cruelty of Really Teaching Computing Science*; Communications of the ACM Volume 32 Number 12, pages 1398-1404, December 1989.

[Faulk 95] Stuart R Faulk; *Software requirements: A tutorial*; NRL report 7775, Naval Research Laboratory, Washington DC, 1995.

[Ferguson 92] Eugene S Ferguson; *Engineering and the Mind's Eye*; MIT Press, 1992.

[Gamma 94] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides; *Design Patterns: Elements of Object-Oriented Software*; Addison-Wesley, 1994.
[Godden 08] http://nisee.berkeley.edu/godden/godden_intro.html (last accessed 13 March 2008)
[Gordon 78] J E Gordon; *Structure, Or Why Things Don't Fall Down*; Pelican Books, 1978.
[Haeberer 01] Haeberer A M and Maibaum T S E; *Scientific Rigour, an Answer to a Pragmatic Question: A Linguistic Framework for Software Engineering*; in Proceedings of the 21st International Conference on Software Engineering, IEEE CS Press 2001, pages 463-472.
[Heitmeyer 96] Constance L Heitmeyer, Ralph D Jeffords and Bruce G Labaw; *Automated Consistency Checking of Requirements Specifications*; ACM Transactions on Software Engineering and Methodology Volume 5 Number 3, pages 231-261, July 1996.
[Hoare 04] Tony Hoare, Cliff Jones and Brian Randell; *Extending the Horizons of DSE (GC6)*; University of Newcastle upon Tyne, Technical Report CS-TR-853, July 2004.
[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.
[Housman 32] A E Housman; *The Name and Nature of Poetry*; The Leslie Stephen Lecture 1932; Cambridge University Press, 1933.
[Jackson 83] M A Jackson; *System Development*; Prentice-Hall International, 1983.
[Jackson 01] Michael Jackson; *Problem Frames: Analysing and Structuring Software Development Problems*; Addison-Wesley, 2001.
[Jones 95] Capers Jones; *Software Specialization*; IEEE Computer pages 81-82, July 1995.
[Lampson 84] Butler W Lampson; *Hints for Computer System Design*; IEEE Computer, Volume 1 Number 1, pages 11-28, January 1984.
[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 94] Nancy G Leveson; *High-Pressure Steam Engines and Computer Software*; IEEE Computer Volume 27 Number 10, pages 65-73, October 1994.
[Levy 92] Matthys Levy and Mario Salvadori; *Why Buildings Fall Down: How Structures Fail*; W W Norton and Co, 1992.
[McMenamin 84] Stephen M McMenamin and John F Palmer; *Essential Systems Analysis*; Prentice-Hall, 1984.
[Maibaum 93] T S E Maibaum; *Taking More of the Soft out of Software Engineering*; in Proceedings of the 7th International Workshop on Software Specification and Design, IEEE CS Press, 1993, pages 2-7.
[Maibaum 97] Tom S E Maibaum; *What we teach software engineers in the university: do we take engineering seriously?* Proceedings of the 6th European conference held jointly with the 5th ACM SIGSOFT international symposium on Foundations of Software Engineering, Zurich, Switzerland, September 22-25, 1997.
[Marzullo 91] Keith Marzullo, Fred B Schneider and Navin Budhiraja; *Derivation of Sequential, Real-Time Process-Control Programs.* In Foundations of Real-Time Computing: Formal Specifications and Methods, A M van Tilborg and G Koob, eds, Kluwer Academic Publishers, 1991, pages 39-54.
[Meyer 88] Bertrand Meyer; *Object-oriented Software Construction*; Prentice-Hall, 1988.
[Naur 69] 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.
[Parnas 85] David Lorge Parnas; *Software Aspects of Strategic Defense Systems*; Communications of the ACM, Volume 28 Number 12, pages 1326-1335, December 1985.
[Parnas 95] David Lorge Parnas and Jan Madey; *Functional Documents for Computer Systems*; Science of Computer Programming Volume 25 Number 1, pages 41-61, October 1995.
[Parnas 97] D L Parnas; *Software Engineering: An Unconsummated Marriage*; Communications of the ACM Volume 40 Number 9, page 128, September 1997.
[Perry 92] D E Perry and A L Wolf; *Foundations for the Study of Software Architecture*; ACM SE Notes, pages 40-52, October 1992.
[Petroski 86] Henry Petroski; *To Engineer is Human: The Role of Failure in Successful Design;* St. Martin's Press, New York, 1985; Macmillan, London, 1986.

[Petroski 94] Henry Petroski; *Design Paradigms: Case Histories of Error and Judgement in Engineering*; Cambridge University Press, 1994.
[Polanyi 58] Michael Polanyi; *Personal Knowledge: Towards a Post-Critical Philosophy*; Routledge and Kegan Paul, London, 1958.
[Randell 07] Brian Randell; *A computer scientist's reactions to NPfIT*; Journal of Information Technology (2007) **22,** 222–234. doi:10.1057/palgrave.jit.2000106 Published online 24 July 2007.
[RISKS 08] The Risks Digest: Forum on Risks to the Public in Computers and Related Systems; http://catless.ncl.ac.uk/Risks/ (last accessed 24/06/08).
[Rogers 83] G F C Rogers; *The Nature of Engineering: A Philosophy of Technology*; Palgrave Macmillan, 1983.
[Shaw 96] Mary Shaw and David Garlan; *Software Architecture: Perspectives on an Emerging Discipline*; Prentice-Hall 1996.
[Software 04] *Software Engineering 2004: Curriculum Guidelines for Undergraduate Degree Programs in Software Engineering*; IEEE Computer Society and Association for Computing Machinery Joint Task Force on Computing Curricula, August 23, 2004.
[Stoy 72] Joseph E Stoy and C Strachey; *OS6—an experimental operating system for a small computer. Part 1: general principles and structure; Part 2: input-output and filing system*; Computer Journal Volume 15 Numbers 2 and 3, pages 117-124 and 195-203.
[SWEBOK 04] Guide to the Software Engineering Body of Knowledge, 2004 Version; IEEE Computer Society Professional Practices Committee, 2004.
[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.
[Weyl 40] Herman Weyl; *The Mathematical Way of Thinking*; address given at the Bicentennial Conference at the University of Pennsylvania, 1940.