Problem Frames and Software Engineering

Michael Jackson, The Open University
{jacksonma@acm.org}

Abstract. A general account is given of the problem frames approach to the development of software-intensive systems, assuming that the reader is already familiar with its basic ideas. The approach is considered in the light of the long-standing aspiration of software developers to merit a place among practitioners of the established branches of engineering. Some of its principles are examined, and some comments offered on the range of its applicability. A view of the approach is suggested by an important account of engineering in the aeronautical industry: in particular, the problem classes captured by elementary problem frames are likened to those solved in established engineering branches by normal, rather than radical, design. The relative lack of specialisation in software development is identified as an important factor holding back the evolution of normal design practice in some areas.

1. INTRODUCTION

It is widely held that software development should aspire to deserve recognition as an engineering discipline.[1] This view motivated the two NATO Software Engineering conferences of 1968 and 1969, and is clearly stated in the report [25] of the first conference:

> "The phrase 'software engineering' was deliberately chosen as being provocative, in implying the need for software manufacture to be based on the types of theoretical foundations and practical disciplines, that are traditional in the established branches of engineering."

The report reflected what was clearly the consensus of the participants: software is, or should become, one more class of engineering product, to be set alongside such established engineering products as bridges, motor cars, chemical plants and aeroplanes. The consensus was unchanged one year later at the second conference in 1969, and has been widely accepted ever since.

The participants assumed implicitly that the phrase *software engineering* was to be narrowly interpreted, to mean the engineering of the software itself—that is, of the structure and content of program texts—and that it was primarily concerned with the processes of software design, programming and testing, and with program execution. The alternative broader interpretation of the phrase, to mean the engineering of change in the world by devising and installing software-intensive systems, was not seriously considered. Surprisingly, the conference participants devoted little of their discussion to exploring and discussing the established practices in the engineering branches they hoped to emulate. (An exception was the talk by W D McIlroy at the first conference, in which he advocated the development and use of mass-produced software components, citing as possible examples trigonometric functions and input-output routines.) Instead they focused on the processes, products and challenges they recognised in their own practice of software development, without comparing them with those of other disciplines.

A very interesting paper by Maibaum [24] discusses some aspects of the relationship between software engineering and the established branches, drawing heavily on Vincenti's illuminating book *What Engineers Know and How They Know It* [31]. Vincenti writes chiefly about aeronautical engineering in the first half of the twentieth century when the field was

[1] In this paper we will use the terms *software development* and *software engineering* interchangeably.

being established by researchers and practitioners. He gives illuminating detail of five case studies including the problems of aerofoil design, a series of wind-tunnel experiments on propellers, the invention and development of flush riveting, the concept of a control volume as a theoretical tool in fluid dynamics, and the process by which the initially vague notion of 'flying qualities' became more exact and amenable to quantitative analysis and design. He also discusses the nature and anatomy of engineering knowledge, and the processes by which it evolves and increases over time.

The present paper, too, draws on Vincenti's book as a source of well-founded observations about engineering practice. Its purpose is to discuss some principles and aspects of the problem frames approach (which we will refer to as 'PF') to software development, and to relate them to the practice that Vincenti describes. The problem frames approach is not a development method. It is, rather, a perspective and a conceptual framework, embodying a certain way of looking at an important group of problem classes and of structuring the intellectual processes of developing good solutions. It is intended to be usable in several ways: constructively, to guide development; analytically, to help understanding of completed developments; and methodologically, to help understanding of existing and proposed approaches to software engineering.

2. THE MACHINE AND THE PROBLEM WORLD

The PF view of software development takes as its starting point the alternative broader interpretation of the phrase *software engineering*, corresponding closely to Rogers's definition [30] of engineering, quoted and amplified by Vincenti:

> "Engineering refers to the practice of organising the design and construction [and, I would add, the operation] of any artifice which transforms the physical world around us to meet some recognised need."

This definition succinctly captures the PF view. The development task is to design and construct an artifice. In PF we call this artifice the *machine*, constructed by building software that is then executed on a general-purpose computer, specialising the computer to serve a particular purpose. That purpose is to meet a recognised need, which we call the *requirement*. Satisfying the requirement involves transforming the physical world around us: in PF, the part of the world to be transformed, in which the requirement is located, is called the *problem world*. Using Polya's term, we may say that the *principal parts* [27] of a software development problem are the machine, the problem world, and the requirement. Their relationships are shown in the generalised problem diagram in Figure 1:

Figure 1. Generalised Problem Diagram

The machine interacts with the problem world at an interface of shared phenomena *a*. Typically, these phenomena are events and states, controlled either by the problem world or by the machine and shared at input-output ports or registers of the machine. (It is often convenient to raise the level of abstraction here—for example, by regarding an extremely reliable input-output device, such as an attached printer, as an integral part of the machine: the phenomena *a* would then include print events.) The machine and the problem world are both *physical*, conforming to their characterisation in Rogers's definition; their interactions at *a* are unmediated and therefore also physical. The requirement is shown by a dashed oval,

indicating its intangible quality. The requirement is not a tangible part of the problem: it is a predicate or condition on the problem world that the machine is required to bring about. The link between the requirement and the problem world is represented by a dashed line. This link denotes references by the requirement to physical phenomena *b* of the problem world. These are the phenomena that the customer for the system would observe to determine whether the requirement is satisfied. In general, the phenomena *b*, which we may call the *requirement phenomena*, are distinct—and sometimes disjoint—from the phenomena *a*. We may call the latter the *specification phenomena*: they are the phenomena in terms of which the external behaviour of the machine—that is, of the software—must eventually be specified.

The generalised problem diagram of Figure 1 emphasises the physical phenomena of the requirement, the problem world and the specification: the machine and problem world, their interaction, and the requirement, are all to be understood in terms of physical phenomena rather than in terms of purely mathematical abstractions. That is not to say that PF eschews abstraction, but rather that it insists that abstractions must be firmly grounded in observable physical reality. For example, if we consider the striking of a key by a computer user to be a shared phenomenon at the specification interface, we are abstracting from the causal chain that runs from depression of the keytop to assigning the corresponding encoded value to a machine register. This causal chain may be very complex, involving interpretation of the key codes and execution of a debouncing algorithm; but we choose to regard the key depression and the assignment as a single shared event. Such abstractions are inevitable in any treatment of physical phenomena. What we insist on is that any abstraction can be unambiguously explained in terms of phenomena that we can observe in the physical world.

Clearly this stipulation excludes certain problem classes. For example, PF makes no claim to offer guidance in the problems of factorising a large integer, finding the shortest path in a graph, or beating the world champion at chess or go. Nor can such problems be brought within the scope of PF by a gratuitous reification—for example, by stipulating a physical representation of the integer or graph or chessboard in a disk file. In some problems, such as the construction of a single-person computer game, the physical problem world may contain nothing other than the operator or user, and the problem world phenomena (other than the specification phenomena) are purely imaginary. The PF approach may then need at least some modification to accommodate the absence of an objective reality to which problem world descriptions can be referred. This kind of problem is discussed in [1]. However, it is important to recognise that problem worlds involving the behaviour of people interacting with the system are fully appropriate to PF: an interesting extension [2] of the approach allows the knowledge they acquire and use in their activities to be brought within the PF scope.

3. PROBLEM REDUCTION

Because PF is concerned with engineering in the world, it demands a clear distinction among three descriptions:

- the requirement $\mathcal{R}$, describing the customer's requirement—how the world is desired to be—in terms of the requirement phenomena *b*;
- the specification *S*, describing, in terms of the specification phenomena *a*, the machine behaviour that must result from execution of the software to be developed;
- the world properties $\mathcal{W}$, describing the given properties of the problem world that hold independently of the machine's behaviour and that govern the possible relationships between the specification and requirement phenomena.

To show that the machine will indeed satisfy the requirement it is then necessary to show that:

$$S, \mathcal{W} \vdash \mathcal{R}$$

In other words, if a machine satisfying S is installed in a problem world satisfying $\mathcal{W}$, then the requirement $\mathcal{R}$ will be satisfied. (A deeper and more detailed discussion of this entailment can be found in [10] and [12].) The gap between what is desired at the interface *b* and what can be directly monitored and controlled by the machine at interface *a* is bridged by the given properties of the problem world. Showing, formally or informally, that this relationship holds among the three descriptions may be called the *basic problem concern*.

PF does not mandate a sequence of development tasks. But in concept we can imagine a development process that begins by capturing the customer's requirement $\mathcal{R}$, and proceeds, using the given properties $\mathcal{W}$, to devise a machine behaviour specification S. We may call such a process *full problem reduction*: starting from a problem involving the phenomena *a* and *b* and the problem world properties, it derives a *reduced problem* in which the phenomena *b* and the problem world do not appear at all. A problem of engineering in the world has been reduced to the problem of building a machine with a specified external behaviour. When the problem world is more complex, and is structured as an interacting set of *problem domains*, the problem can be partially reduced in each of a series of steps, successively removing domains that are farthest from the machine. (Hall and Rapanotti have pointed out the connection between problem reduction and the weakest-environment calculus [21].) Eventually, of course, once this reduction process is complete, a programming process will create a program whose execution will satisfy S.

Software development has traditionally dealt chiefly in what are, in this sense, *reduced problems*: the problem was seen solely as satisfying a given specification—often informal and sometimes unwritten—rather than including the process of deriving the specification. Early computers were connected very loosely and asynchronously, by data on punched card or paper or magnetic tape, to their problem worlds: it seemed natural to start at the point where the data was presented to the computer. Most software, apart from the compiler, the operating system kernel, and the software to manage input-output and system resources, consisted of batch programs performing mathematical computations or data processing tasks. Programs for mathematical computations were often written by people already fully familiar with the task to be performed, who saw no need for a written specification. Data processing tasks were often simply specified as reproducing, sometimes with little or no change, the behaviour of an existing manual or punch-card system. In general, the production, content and format of program specifications were not considered to be a significant part of the software development task.

This view was memorably expressed by Dijkstra [7]:

> "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,' ie the question of whether an engine meeting the specification is the engine we would like to have; the other one is the 'correctness problem,' ie the question of how to design an engine meeting the specification. ... the two problems are most effectively tackled by ... psychology and experimentation for the pleasantness problem and symbol manipulation for the correctness problem."

Dijkstra was concerned to maintain the mathematical purity of computer science, and rejected the notion of software engineering in any sense. He dubbed software engineering 'the doomed discipline', and claimed that its goal was self-contradictory because computers were purely symbol manipulation machines. The 'pleasantness problem'—that is, obtaining the functional specification—was for others to address. Computer scientists and software developers should

restrict their attention to the 'correctness problem', and should address it by formal mathematical techniques.

This narrow view of the software development task has had an enduring effect. Although lip service is almost universally paid to the need to analyse and describe the problem world, the reality is that most descriptions—certainly most formal descriptions—constructed in software projects are descriptions of the machine rather than of the world. What may have begun as an effort to describe the world slips quickly into a focus on an object or database structure that is intended to represent it in the machine. The PF emphasis on the explicit investigation, description and use of the given problem world properties can be seen as an attempt to exert a countervailing intellectual influence. PF views the process of problem reduction as an integral and central part of the software engineering discipline.

4. REDUCED PROBLEMS IN ENGINEERING

It may perhaps seem that in spite of Rogers's definition the PF emphasis on the problem world is at odds with established engineering practice. At first sight, engineering practice seems to focus very directly on the product, or artifice, to be designed and constructed, leaving the world around it very much in the background. Automobile engineers appear to think much more about cars than about roads or drivers, so justifying an analogous emphasis in software engineering on the machine rather than the problem world.

A distinction should be made here between what we may call *local* and *ubiquitous* problems. In a local problem the problem world is unique to a particular place, and the engineer must examine that problem world in all its particularity. Consider, for example, the building of a road or rail bridge over a navigable river. The bridge designer must obviously carry out an explicit and detailed investigation and analysis of the given properties of the particular problem world: the character of the ground on which the bridge supports will be erected; the river currents; the sizes and speeds of vessels navigating the river; the local winds; the density, nature and flow patterns of the planned traffic; and the seasonal variations in temperature and other atmospheric conditions.

In a ubiquitous problem, by contrast, the engineering product is intended to be used anywhere that the operational environment exhibits certain broad characteristics that are essentially independent of the specific operational locality. Automobile engineering is concerned with wheeled vehicles that travel on *terra firma*; aeronautical engineering is concerned with vehicles that travel in the earth's atmosphere. The problem world properties of interest are then location-independent properties, and do not demand a fresh explicit description of roadways or of the atmosphere for each new car or aircraft design. Rather than *ad hoc* products of the development project in hand, these descriptions are standard reference material.

Vincenti writes of the indicative problem world properties to be considered by the aeronautical engineer:

> "Descriptive knowledge is knowledge of how things are. Descriptive data needed by [aeronautical] designers include physical constants (acceleration of gravity, for example) as well as properties of substances (failing strength of materials, coefficients of viscosity of fluids, etc.) and of physical processes (rate of chemical reactions and so forth). Occasionally they deal with operational conditions in the physical world (frequency and strength of atmospheric gusts for aircraft fatigue-loading calculations). As we have seen with flying qualities, they also encompass information on human beings (maximum forces exerted by pilots [on an aircraft's manual controls])."

Certainly, some of this descriptive knowledge—strength of materials and rate of chemical reactions—is clearly about the substrate of the engineering artifice itself rather than its problem world or operating environment. Frequency and strength of gusts, however, and forces exerted by pilots, are certainly properties of the problem world. But they do not require to be established afresh on each occasion. In respect of these properties the engineers need only address the reduced problem, reduced in the light of standard knowledge of the properties of the problem world or environment.

5. NORMAL AND RADICAL DESIGN

Vincenti draws on his own experience of aeronautical engineering and on the work of the philosopher Michael Polanyi [26] and the historian Edward Constant [6] to explain the central importance of *normal*—as opposed to *radical*—design for established engineers. Normal design is what allows them to succeed where they do and as often as they do. Of an engineer practising normal design he writes:

> "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."

> "Every device possesses an operational principle, and, once the device has become an object of normal, everyday design, a normal configuration. Engineers doing normal design bring these concepts to their task usually without thinking about them."

In short, it is usually otiose, or even harmful, to rework from basic principles a problem whose requirements, design and solution have been firmly established by long and successful experience. Vincenti gives the example of automobile design:

> "Automobile designers of today usually (but not invariably) assume without much thinking about it that their vehicle should have four (as against possibly three) wheels and a front-mounted, liquid-cooled engine."

Vincenti lays some stress on the hierarchical structure of engineering products in the established branches. He distinguishes between *devices* and *systems*. Devices are "single, relatively compact entities", while systems are "assemblies of devices brought together for a collective purpose." The distinction is, of course, relative. The struts of an aircraft's landing gear are a device—a part of the landing gear viewed as a system. The landing gear, viewed as a device, is a part of the aircraft regarded as a system. The aircraft itself can be viewed as a device forming a part of the airline system, or even of a national or global system of air transport.

Although the distinction between devices and systems is relative in this sense, it is also closely associated with the distinction between normal and radical design. Normal design can be applied only to a device, to a "single, relatively compact entity". Radical design is always necessary for a system, where there is no history of successful development of closely fitting precedents. Radical design may also be necessary when a new kind of device is required. Then the engineer's ambitions must be dramatically lowered:

> "... 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."

In effect, only normal design can realistically hope to produce a reliable product with no major defects or failures. Though less conspicuous than radical design, normal design makes up by far the bulk of day-to-day engineering enterprise. Unfortunately, this is not true of software engineering.

6. ELEMENTARY PROBLEM FRAMES

Elementary problem frames can be regarded as defining problem classes in software engineering that are currently the object of normal design. Their machines, set firmly in the context of the problem worlds that provide their purpose, play the role that Vincenti's well-understood devices play in the established engineering branches. Particular elementary *problem frames*, specialised from the universal problem diagram of Figure 1, capture classes of software development problems whose software solutions take the form of individual programs or, at most, very small systems of programs. For example:

- In a *WorkPieces* problem a *User* edits a *WorkPiece* such as a text or graphic document. The requirement is that the edit commands issued by the *User* should effect appropriate corresponding changes in the *WorkPiece*.
- In a *Required Behaviour* problem the *Control Machine* is required to impose a particular behaviour on a *Controlled Domain*.
- In an *Information Display* problem the *Information Machine* is required to monitor the state and behaviour of a *Real World* and to display information about it on a *Display*.

Problem frames do not aim to capture classes of problems of realistic size and complexity. Most of them capture simple subproblems that can appear only as ingredients or aspects of realistic problems. For example, the WorkPieces frame captures problems of simple document editing. The subsequent use of the document, and even the user's ability to inspect different parts of the document while it is being edited, are ignored: the solution to a WorkPieces problem is not useful in isolation.

Each problem frame constitutes what we might call a *problem pattern*. The very well-known work on object-oriented design patterns [9, 3] has some obvious similarities, and is clearly concerned with the identification of devices that, once identified, can become the objects of normal design. But design patterns are firmly located within the machine: all parts of each pattern are represented in programming terms. The work on analysis patterns [8] is an interesting treatment of some problem fragments in an object-oriented setting.

A problem pattern stipulates:

- a decomposition of the problem world into a particular set of physical *problem domains* interacting with each other and with the machine in a particular topology;
- a characterisation of each problem domain according to its physical properties—causal, lexical or biddable—as they affect the problem (the Controlled Domain is causal, the WorkPiece is lexical, and the User is biddable);
- a characterisation of each interface according to the types of the subsets of the phenomena—events, states, symbols—that are shared at the interface and to the assignment of control of each subset to one of the sharing domains;
- the nature of the requirement and of its link—none, reference only, or constraining—to each problem domain.

Problems that fit different elementary frames specialise the basic problem concern in different ways, giving rise to *frame concerns* of different forms. The specialisation arises

because the given properties $\mathcal{W}$ of each problem domain d must be separately described, and these descriptions must be combined with each other and with the specification S and requirement $\mathcal{R}$ in a way that respects the connectivity of the frame diagram and the control of shared phenomena.

The frame concern of a problem class can be regarded as loosely analogous to the *operational principle* of a device class in normal design—how the characteristic parts of the device fulfil their special function in combining to an overall operation which achieves the device's purpose. Vincenti cites Sir George Cayley's famous statement of 1809 [5] in which he characterised the principle of the fixed-wing aircraft: "to make a surface support a given weight by the application of power to the resistance of air." The surface provides the lift because the applied power drives the surface against the resistance of the air.

In addition to the frame concern each problem frame raises a number of *specific concerns* that must be addressed if the solution is to be acceptable. For example:

- Initialisation. The machine necessarily begins operation in its initial state, and at that moment the problem world is also in one of some set of states. If the developer has not considered the problem world state that in fact holds initially in a particular case, the specified machine behaviour is unlikely to satisfy the requirement, at least for some initial period of operation and possibly for the whole lifetime of the system.
- Breakage. A causal domain may be damaged if certain sequences of operations are performed on it. For example, some component of a Controlled Domain may be damaged if certain operation sequences are performed by the machine. The developer must identify such sequences, and ensure that the machine always avoids them.
- Reliability. A causal problem domain typically satisfies its properties description with high—but never with perfect—reliability. If the reliability is too low in relation to the criticality of the system it is necessary to detect, or even to anticipate, failures and to prevent, mitigate or compensate their effects.
- Information deficit. In an Information Display problem the information directly and currently available to the machine at the time it must produce each display output may be inadequate. It may have been available at an earlier time; or it may be information that must be accumulated from the beginning of execution; or it may be deducible only by elaborate calculation or estimation from the information available earlier. The developer must then find some effective way of overcoming this information deficit.

In general, different problem frames raise different specific concerns. Information deficit does not arise in WorkPiece problems. Information Display problems do not raise reliability concerns. Breakage is not a concern in WorkPieces problems. To a significant extent the set of concerns that must be addressed, and the detail of each concern, depend on the characteristics of the frame's problem domains. Breakage and reliability do not arise for lexical domains; information deficit does not arise for biddable domains. However, the importance of a particular concern can not be finally determined by considering abstract characteristics either of domains or of problem frames. It must be ultimately determined by the knowledge, experience and judgement embodied in a particular class of normal design task, applied to the particular problem world of the case in hand. The engineer practising normal design knows in advance what concerns will merit most of his attention.

Many of the failures documented in P G Neumann's Risks forum [29] can be attributed to the absence of established normal design practice for at least the failing part of the system. With hindsight the mistake or neglect that led to the failure is conspicuous, and we may often wonder why the developers did not see it. The reason is that they were doing radical design,

not normal design, and their knowledge, experience and judgement were not specifically honed for the particular design task they were doing. In Vincenti's words, they "had no presumption of success." Depending on the circumstances, blame for the failure may be laid at the developers' door for ignoring existing normal design practice; or at the customer's door for stipulating requirements that went too far beyond what can be reliably delivered by current practice; or, more generally, at the software development community's door for failing to evolve a normal design practice when enough knowledge and experience existed to do so.

7. INSTANTIATING ELEMENTARY FRAMES

In an established branch such as aeronautical engineering [6], the task of normal design is "the improvement of the accepted tradition or its application under new or more stringent conditions". Relatively small design changes give quantitative increments in cost, performance or economy, while retaining the customary configuration of parts and the accepted operating principle. Examples in software engineering are most readily found in the improvement of what we may call internal software components—components all of whose interactions are with other formally specified software components, within the machine. The classic handbook for a large variety of such software devices is Knuth's magnum opus *The Art of Computer Programming* [17,18,19], where detailed discussions are given of the construction and performance of algorithms for sorting, tree traversal, garbage collection and many other purposes. Some examples can be cited closer to the problem world. Spell checkers, for instance, appear to have become the object of normal design, to judge by their greatly improved speed and reliability.

In the design of solutions for problems fitting elementary problem frames the need for incremental quantitative improvement may be present, but is usually dominated by the need to instantiate known solutions for the detailed properties of specific problem domains. Instances of the WorkPieces frame, for example, will differ not only in response time and speed of saving the document being edited, but more importantly in the different lexical structures and operations of the WorkPieces they are required to handle. Similarly, instances of the Required Behaviour frame will differ in the structural and dynamic properties of the Controlled Domain. These differences provide the core subject matter for software development methods. For example, a model-based method such as Z [13] or VDM [15] is very appropriate to instantiating the common data structures that characterise a set of WorkPieces together with the repertoire of editing operations which can be applied to them. Similarly, problems fitting the *Transformation* frame, in which the task is to produce sequential output streams from sequential input streams, can often be appropriately solved by the JSP design method [14]: the sequential structures of the streams are described in regular expressions which are then combined to give the required program structure.

8. COMPOSITE ELEMENTARY PROBLEMS

A problem fitting an elementary frame will sometimes be soluble by a simple machine, whose specification can be a single text. But sometimes a specific concern will demand a decomposition into two or more subproblems. For well-understood elementary problem frames this decomposition will be highly standardised. This standardised decomposition is another aspect of what Vincenti calls "a given operational principle and normal configuration": it is understood not only how the device should work but also what arrangement of parts will best allow it to work in that way.

Consider, for example, an Information Display problem in which the requirement is to maintain a Display showing information about a Real World. Figure 2 shows the problem frame diagram:

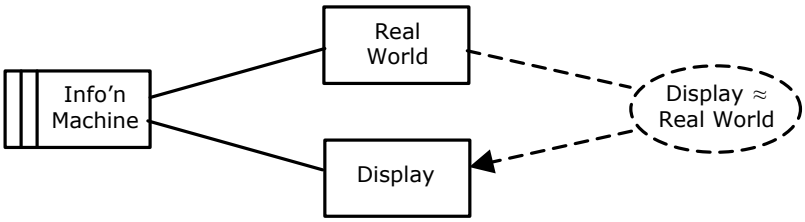

Figure 2. Information Display Problem Diagram

The requirement is that the visible phenomena of the Display should correspond in a certain way to the state of the Real World. If the particular problem in hand presents a significant information deficit problem (as most Information Display problems do), the standard solution is to decompose the problem into two subproblems communicating by a Model domain, as shown in Figure 3:

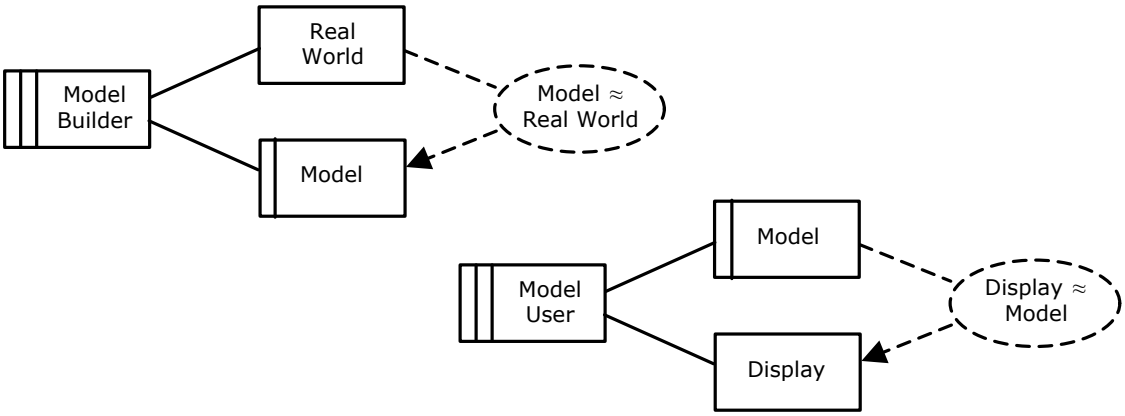

Figure 3. A Standard Decomposition

The Model domain is an invented lexical domain, often a database on disk. One subproblem constructs and maintains the Model, satisfying the requirement that the Model should correspond in a certain way to the Real World; the other subproblem maintains the Display, using the Model and satisfying the requirement that the display should correspond in a certain way to the Model. If we can express the requirements Display ≈ Model and Model ≈ Real World as relations, it seems clearly desirable that their relational composition should be equal to the original requirement Display ≈ Real World.

Introducing the Model domain is an unavoidable, and necessarily universal, practice in software development. Decomposing the problem into the two subproblems shown, however, is not standard practice: essentially it is a separation into two concurrent processes. They will subsequently need to be composed. In normal design of a composite device, the technique and mechanism of composition is standardised. Here, for example, a standard technique of mutual exclusion, applied to critical regions of appropriately chosen granularity, may be sufficient.

9. DECOMPOSING REALISTIC PROBLEMS

If problems fitting elementary problem frames are devices in Vincenti's terms, then realistic problems are usually systems—'assemblies of devices brought together for a collective purpose'—and consequently the object of radical design. To repeat Vincenti's words, "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 need for radical design springs from either or both of two sources. Some of the devices making up the system may be currently unfamiliar, themselves demanding radical design; and the particular combination of devices may be novel, posing unfamiliar problems of interaction.

The PF approach aims to contain and minimise the radical aspects of development in any particular case by application of two interlocking principles. The first principle is that a realistic problem should be decomposed, so far as possible, into subproblems that fit known elementary problem frames. That is: all those parts or aspects of the system that can be properly regarded as familiar devices should be identified, and developed by the applicable techniques of normal design. What makes PF distinctive here is the granularity and compactness of the parts into which the problem is decomposed. Elementary problem frames capture closed systems, which, even if not realistically useful in isolation, are complete in themselves. Each subproblem has its own machine, its own problem world, and its own requirement, and has no external connections.

This decomposition rule has important consequences for the granularity of the decomposition. Some commonly used development techniques, by contrast, decompose problems into open parts, usually of smaller granularity. This is true, for example, of decomposition into programming objects (which offer and invoke methods); of dataflow decomposition into sequential processes (which assume stream or database communication with other processes); and of requirements decomposition into use cases (each detailing one episode of user interaction within a larger, usually unspecified, context). Each of these techniques has its place; but from a PF point of view none is satisfactory as the chief form of problem decomposition.

The second principle, flowing from the first, is that subproblem composition is a major development topic in its own right, demanding explicit and separate consideration. Because the devices identified in the decomposition are closed systems, and to a large extent standardised, their normal development pays—and should pay—no attention to the need to combine them with other devices into a realistic system. But eventually, of course, the decomposed subproblems must be recombined or composed. Composition brings its own concerns: the composition of the subproblem requirements must be shown to approximate closely enough to the original problem requirement; the subproblem machines must be configured for some kind of parallel execution, cooperating to whatever extent is necessary; conflicts between subproblems must be resolved.

In the PF approach, the central principle to be applied here is that treatment of the composition concerns should be deferred until each subproblem has been identified and analysed to the point of obtaining a satisfactory machine behaviour specification and problem world description. Until this point, the identification and treatment of the decomposed subproblems should—in principle—take no account of the eventual need to make them work together.

This principle stands in opposition to the most commonly used techniques, in which consideration of the composition concerns is integrated at the outset into the treatment of the subproblems. A seeming advantage of these traditional techniques is that composition can become a relatively trivial mechanical matter of matching actual to formal parameters or output stream to input stream definitions. In practice, however, the widespread emphasis on integration testing indicates that the common technique is hard to carry through successfully. An important advantage of the PF technique of deferring the composition concerns is that the subproblems are then seen in their purest forms, in which they correspond exactly to known

devices, not yet complicated by their eventual interactions. Further, the composition concerns can be dealt with later as an explicit task of composing known devices into a system. If the deferred composition concerns then prove to demand substantial reworking of the subproblems to be composed, this is not a disadvantage of the separation: it is rather an indication that dealing simultaneously with the subproblems and their composition concerns would have been very difficult.

10. COMPOSITION CONCERNS

Composition concerns arise when subproblems have parts of their problem worlds in common. Examples of composition concerns are:

- Interference. Two subproblems interfere if one causes change in a problem domain and the other inspects the same domain: for example, the Model Builder machine changes the Model domain while the Model User machine inspects it. The Model User subproblem has a domain properties description of the Model, and it is necessary to ensure that the Model conforms to this description during each inspection. A suitable granularity must be chosen for mutual exclusion of atomic changes and inspections.
- Scheduling. A scheduling concern arises when interference can not be dealt with by atomicity of changes and inspections. Suppose, for example, that in a library system one subproblem deals with membership and another deals with borrowing and returning books. In the books subproblem membership is treated as static, although in reality it is constantly changing as people join and leave the library. Questions then arise about book loans towards the end of a membership period. Can a member whose membership has only one week to run borrow a book for two weeks? What if a borrower resigns from membership before returning the book?
- Conflict. The requirements of two subproblems that cause change in the same domain may sometimes conflict. For example, in a lift control problem the subproblem requirement of providing lift service may conflict with the subproblem of ensuring safe operation in the presence of mechanical and electrical faults: lift service may demand movement of the lift car to service a request, while safety demands that it be locked by the emergency brake at its current position in the shaft. Clearly two conflicting requirements can not both be satisfied: one must be given precedence.

The interference concern can be regarded as an implementation matter. The properties description of the Model domain as seen by the Model User machine has already been given. To address the concern it is necessary only to ensure that this description holds at the relevant times. But the scheduling and conflict concerns, by contrast, raise requirements issues that were not visible when the subproblems were considered in isolation.

11. IMPLEMENTING COMPOSITION

In the PF approach architecture can be seen as the composition and implementation of the subproblem machines in a realistic problem. Because a full-scale realistic problem is likely to be the object of radical design, there are several approaches that can be valuable in different cases.

Two reported approaches aim to bring more realistic problem classes within the ambit of normal design. The first is reported in [11], where the machine of a subproblem is assumed to be specialised in a standard way that implements a particular scheduling and prioritisation scheme. The problem is to satisfy customer orders in a warehouse. Investigation of the requirement shows the potential for interference between orders for the same product, leading

to duplicate allocation of the same stock. It also shows the need for a fair scheduling, ensuring that an order placed earlier will be allocated scarce stock in preference to a later competing order. The solution proposed is to use a specialised machine rather than the generic general-purpose computer usually assumed in an elementary problem frame. The specialised machine has built-in behavioural properties *one-at-a-time* and *first-come-first-served*, each capable of being instantiated for the problem in hand. In effect, aspects of the requirement are being delegated to the implementation. Some part of the requirement need not be stated in explicit detail because it is implicit in the choice of the specialised machine.

The second of these approaches is reported in [28], where the idea of an architectural frame or *AFrame* is introduced as an elaboration of the problem frame diagram to accommodate an early architectural decision. One of their examples is a Transformation problem frame in which a standard decomposition into sequential processes is expected, and the architectural decision has been made to implement composition of the resulting machines by the pipe-and-filter architecture. Another example draws on the well-known MVC pattern [20].

Both of these approaches can be regarded as enlarging the set of devices that can be the object of normal design by defining specialised composite elementary problem frames. A different approach is reported in [22], where the task of composing conflicting requirements is addressed. In this approach an additional machine—which may be viewed as an architectural connector—is introduced between the conflicting machines and their common problem domains, and arbitrates in cases of conflict. In [22] the approach is examined in several different cases of real and potential conflict, and offers, at least in part, the possibility of application in the composition of many different realistic problems.

12. SPECIALISATION AND THE GROWTH OF KNOWLEDGE

It is apparent from Vincenti's account that an important part of the success of the established branches of engineering can be attributed to specialisation. Vincenti takes it entirely for granted that aeronautical engineers do not work on problems in other branches: they do not design bridges or chemical plants or even motor cars. The specialisation goes deeper, to much lower levels. For example, W F Durand and E P Lesley devoted their work from 1916 to 1926 on the design of propellers [31]. One whole chapter of Vincenti's book is devoted to the development by the aircraft companies of flush riveting, in which the aerodynamic properties of wings and fuselages are improved by the use of rivets whose head do not protrude above the surface of the skin.

Even a cursory inspection of the literature of an established branch of engineering reveals a similar level of specialisation. Whereas much software engineering literature (like the present paper, it might be said) tends to focus on general concerns or on methods or approaches proposed for universal or nearly universal application, the literature of established engineering tends rather to focus on highly specialised applications and problems. For example, one issue of *The Journal of Structural Engineering* [16] contains papers on such topics as "Performance Evaluation of Controlled Steel Frames under Multilevel Seismic Loads", "Stress Concentration Factors of Doubler Plate Reinforced Tubular T Joints", "Reliability Assessment of Highway Truss Sign Supports", and "Wind Sensitivity of Recycled Plastic Soundwalls". This kind of specialisation seems an essential foundation for the incremental growth of knowledge in the established branches of engineering.

Specialisation is essential to the growth of knowledge because it allows experience to be accumulated effectively and systematically. The accumulation of experience and knowledge demands a conceptual structure within which it can be evaluated and stored, and from which it can be reliably and easily retrieved when it is relevant. Excessively narrow specialisation

results, perhaps, in a smaller contribution to knowledge than would otherwise have been possible, but in a contribution nonetheless. Absence of specialisation often results in no contribution at all: the knowledge gained is too vaguely expressed, and is added to a largely unstructured corpus of software engineering knowledge in general rather than to a specific place or places in a detailed structure. The existence of a structured corpus of knowledge offers no guarantee that the knowledge will be used—that is, retrieved and applied when it should. The notorious failure of the Tacoma Narrows Bridge in 1940 was due to wind-induced oscillation of a kind that had been observed repeatedly in suspension bridges and recorded from as early as 1836 [23]. But the absence of such a knowledge structure virtually guarantees that it will not be used.

ACKNOWLEDGEMENTS

Tom Maibaum brought Vincenti's book to my attention (he acknowledges his debt to William Newman, who brought it to his). Work with colleagues at the Open University, at the University of Newcastle, at the University of Leicester and at MIT has provided much enjoyment, illuminated many aspects of PF, and revealed many needs and opportunities for its further development and application.

REFERENCES

[1] Ian K Bray and Karl Cox; *The Simulator; Another, Elementary Problem Frame?* in Proceedings of the 9th International Workshop on Requirements Engineering: Foundation For Software Quality (REFSQ03), 2003.

[2] John Brier, Lucia Rapanotti and Jon G Hall; *Problem Frames for Socio-technical Systems: predictability and change*; in Proceedings of the 1st International Workshop on Advances and Applications of Problem Frames, 2004.

[3] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad and Michael Stahl; *Pattern-Oriented Software Architecture: A System of Patterns*; John Wiley 1996.

[4] J R Cameron; JSP & JSD: *The Jackson Approach to Software Development*; IEEE Computer Society Press, 2nd Edition 1989.

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

[6] E W Constant; *The Origins of the Turbojet Revolution*; John Hopkins University Press, Baltimore 1980.

[7] E W Dijkstra; *On the Cruelty of Really Teaching Computer Science*; Communications of the ACM Volume 32 Number 12, December 1989.

[8] Martin Fowler; *Analysis Patterns: Reusable Object Models*; Addison-Wesley 1996.

[9] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides; *Design Patterns: Elements of Object-Oriented Software*; Addison-Wesley, 1994.

[10] Carl A Gunter, Elsa L Gunter, Michael Jackson and Pamela Zave; *A Reference Model for Requirements and Specifications*; Proceedings of ICRE 2000, Chicago Ill, USA; reprinted in IEEE Software Volume 17 Number 3, pages 37-43, May/June 2000.

[11] Jon G Hall, Michael Jackson, Robin Laney, Bashar Nuseibeh and Lucia Rapanotti; *Relating Software Requirements and Architectures Using Problem Frames*; Proceedings of the 2002 International Conference on Requirements Engineering (RE'02), Essen, 2002.

[12] Jon G Hall and Lucia Rapanotti; *A Reference Model for Requirements Engineering*; in Proceedings of the 11th Joint International Conference of Requirements Engineering (RE'03), 2003.

[13] Ian Hayes (ed); *Specification Case Studies*; Prentice-Hall International, 1987.

[14] M A Jackson; *Constructive Methods of Program Design*; in Proceedings of the 1st Conference of the European Cooperation in Informatics, pages 236-262; G Goos & J Hartmanis eds; Springer-Verlag LNCS 44, 1976; reprinted in [4] pages 46-62.

[15] Cliff Jones; *Systematic Software Development Using VDM*; Prentice-Hall International, 2nd Edition 1990.

[16] The Journal of Structural Engineering November 2002; Volume 128, Issue 11.

[17] Donald E Knuth; *The Art of Computer Programming; Volume 1: Fundamental Algorithms*; Addison-Wesley, 1968.

[18] Donald E Knuth; *The Art of Computer Programming; Volume 2: Seminumerical Algorithms*; Addison-Wesley, 1969.

[19] Donald E Knuth; *The Art of Computer Programming; Volume 3: Sorting and Searching*; Addison-Wesley, 1972.

[20] G E Krasner and S T Pope; *A cookbook for using the Model-View-Controller user interface paradigm in Smalltalk-80*; Journal of Object-Oriented Programming Volume 1 Number 3, pages 26-49, August/September 1988.

[21] Luming Lai and J W Sanders; *A Weakest-Environment Calculus for Communicating Processes*; Technical Report PRG-TR-12-95, Oxford University Computing Laboratory, 1995.

[22] Robin Laney, Leonor Barroca, Michael Jackson and Bashar Nuseibeh; *Composing Requirements Using Problem Frames*; in Proceedings of the 2004 International Conference on Requirements Engineering RE'04, IEEE CS Press, 2004.

[23] Matthys Levy and Mario Salvadori; *Why Buildings Fall Down: How Structures Fail*; W W Norton, 1994.

[24] 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.

[25] 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.

[26] Michael Polanyi; *Personal Knowledge: Towards a Post-Critical Philosophy*; Routledge and Kegan Paul, London, 1958.

[27] G Polya; *How To Solve It*; Princeton University Press, 2nd Edition 1957.

[28] Lucia Rapanotti, Jon G. Hall, Michael Jackson and Bashar Nuseibeh; *Architecture-driven Problem Decomposition*; in Proceedings of the 2004 International Conference on Requirements Engineering RE'04, Kyoto, IEEE CS Press, 2004.

[29] http://www.catless.ncl.ac.uk/Risks

[30] G F C Rogers; *The Nature of Engineering: A Philosophy of Technology*; Palgrave Macmillan, 1983.

[31] 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.