Proceedings Third IEEE International Conference on Engineering of Complex Systems, Como, Italy, September 8-12, 1997, pages 239-248

Problem Complexity

Michael Jackson
Independent Consultant
101 Hamilton Terrace, London NW8 9QY, England
and with AT&T Research
180 Park Avenue, Florham Park NJ 07932-0971, U S A
jacksonma@attmail.com

Abstract

An approach to problem analysis is described, based on the notion of a problem frame. Problem frames are intended to capture classes of recognisable and soluble problems. Because problems are located in the environment, not in the machine to be built, problem frames are characterised by environment properties and structures. Useful frames are necessarily very narrow: realistic problems are regarded as parallel superimpositions of subproblems. Problem complexity arises from the interactions of these subproblems and of their solutions.

1. Introduction

Software engineering traditionally focuses on solutions rather than on problems: development is faster when problem analysis is treated implicitly by folding it into the solution design. Also, problems are inconveniently informal and heterogeneous while solutions can be attractively formal and homogeneous.

The disadvantages of this focus on solutions are obvious. But shifting the focus to problems demands some kind of discipline of problem analysis. Some aspects of a problem analysis discipline are presented in this paper, with some discussion of the sources of problem complexity.

1.1 Complexity

The complexity considered here is informal and subjective. It manifests itself in the difficulty that a problem presents to human understanding, and consequently to the achievement of a satisfactory solution.

Our chief tools for mastering complexity are structuring, abstraction and formalisation. We structure a complex whole by separating it into parts that can be analysed and considered independently. For example, by separating the input phase of a compiler into a lexer (having m states) and a parser (having n states) we reduce the complexity of the input phase from $m \times n$ to $m+n$ states. This separation, of course, depends on the availability, or development, of *lexing* and *parsing* as useful and intelligible abstractions for the problem in hand. The separation would be frustrated if, perversely, the language were designed so that lexical conventions were local and depended on the syntactic context.

If the *lexing/parsing* separation works well it can serve both as a problem structure and as a solution structure. We understand lexing and parsing as subproblems, and we also understand them as distinct parts of the implemented compiler. Some form of composition of the separated parts is always necessary. Sometimes it may be a composition mediated by a simple mechanism such as procedure call, or by a more subtle mechanism such as the shared events of CSP[4]. Sometimes it may be necessary to integrate the solutions of the separated problem parts into a single solution part in which the two solutions are closely intertwined. The technical difficulty of composing the solutions is often an unrecognised force constraining the choice of problem structure: it often seems gratuitously burdensome to choose a structure in problem analysis that can not be carried forward to the solution.

Formalisation is an essential tool in mastering complexity simply because it permits more reliable reasoning. We can not reason reliably in informally understood terms, so we must formalise those parts of the informal world that we wish to reason about. This activity of formalisation is of special importance in software engineering, because software engineering problems are often — if not always — located in an informal context furnished by relevant parts of the physical world.

1.2 Focusing on the Problem

The recognition that the problem is located in the environment rather than in the machine to be built is crucial to focusing on the problem rather than the solution. The machine is the solution, not the problem. Seeking the problem in the machine has been a capital error of many methods that have claimed to address problem analysis. In Structured Analysis[1], for example, the initial Context Diagram shows the *system* (the machine) connected by data flows to the *terminals* (relevant parts of the world). Clearly, the next step should be description and analysis of the terminals; but instead it is a top-down decomposition of the system. This is structuring the solution, not analysing the problem.

Focusing on the problem and deferring consideration of putative solutions is in itself a contribution to mastering complexity in software engineering. Much of the complexity we confront results from the need for explicit composition of solutions: the solution is often more complex than the problem because the solution environment does not permit the same abstractions and separations as the problem space. Premature attention to composing the solution is likely to confuse problem analysis.

1.3 Capturing the Phenomena

To capture a problem in the physical world we must identify the phenomena we are interested in and propose to denote in our descriptions. For each class of phenomenon we must give a *designation*[6]: that is, a recognition rule (necessarily informal) by which instances of the phenomenon can be recognised in the world, and an associated formal term by which we will denote them.

This activity of designating phenomena of interest formalises an informal reality, and is therefore also an activity of approximation. The phenomena to be designated must not be chosen arbitrarily. In most real problems the obvious criterion of relevance to the problem in hand leaves many choices open. We must also consider and estimate the error involved in the formalisation, choosing the designated phenomena so that this error, and the consequential errors that will arise from formal reasoning with the designated terms, will be acceptably small. The point is familiar to anyone who has ever given directions to a passing motorist. It is very unsatisfactory to give directions in terms of ‘bends in the road’; ‘turnings’ and ‘crossroads’ may be not much better; ‘traffic lights’ are often reliably recognisable, although even there difficulties arise with lights that may be out of use in certain periods or serve only to guard a pedestrian crossing. In giving directions we seek to use the designated phenomena with the smallest errors; in describing a software engineering problem we can hardly do less.

1.4 Some Simple Structuring

The problem context presents immediate opportunities for simple structuring. It is natural to divide the world into *domains* — groupings of phenomena that are conveniently considered together because they are in some sense compact. That is, the mutual relationships and constraints among the phenomena within one domain are much richer and stronger than those — if there are any — that reach across domains. So in the Patient Monitoring problem[10] it is natural to divide the context into the Patients, the Nurses’ Station, the Analogue Devices, the Database and the Medical Staff (who specify the monitoring required for each patient). Each of these domains has its own internal properties; it is the purpose of the machine we build to constrain those properties further and to connect the domains together.

Suitably chosen domains are not only compact: they are also likely to have distinctive phenomenological characteristics. The Patients are autonomous, but the Database is inert; the Nurses’ Station is reactive — it displays messages to the nurses when stimulated to do so by the machine; but the Medical Staff are active — they act without external stimulus.

In describing the world we may structure our descriptions by domain. We must also structure them according to a fundamental distinction between what is given and what is required. This distinction was well known to traditional grammarians: what is given may be expressed by statements in the indicative mood; what is required may be expressed by statements in the optative mood. It is not enough to make this distinction by the local grammatically dubious use of the words ‘shall’ and ‘will’: the distinction must be regarded as a fundamental structuring principle of problem analysis and hence of software engineering documentation.

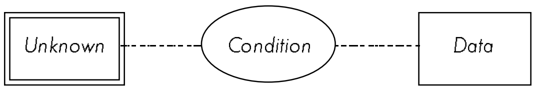
2. Problem Structuring

Structuring the world is an important aspect of problem analysis, but we also need a direct structuring of the problem itself. The key idea presented here is the idea of a *problem frame*[6], derived in concept, though not in name, from the work of the mathematician Polya[9].

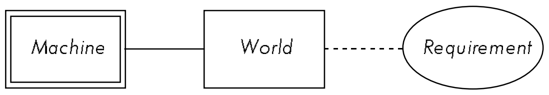
2.1 Polya

Polya, expounding the work of the ancient Greek mathematicians, distinguishes two kinds of small mathematical problem: problems to prove, and problems to find or construct. Each kind has it own characteristic *principal parts*. The parts of a problem to find or construct are the *data*, the *unknown*, and the *condition*. For example, in the problem “Given three lengths a, b, and c, construct a triangle with sides of those lengths”, the *data* is the lengths a, b and c; the *unknown* is a triangle; and the *condition* is that the sides of the triangle should be of the three given lengths. The solution task is to construct or find an *unknown* which satisfies the *condition* with respect to the *data*.

We can represent the structure of a problem to find in a problem frame diagram. The *unknown* is represented by a rectangle with a double outline (indicating that it is to be constructed); the *data* by a rectangle with a single outline (a given domain); and the *condition* by an ellipse. Dotted lines connect the ellipse to the domains over which it is required to hold.



A similar representation may be adopted for a general problem frame for software engineering problems. Here, being concerned with physical phenomena, we need to represent the interaction of two domains by shared phenomena; for this we use an unbroken line.



The principal parts of a general software engineering problem are the *machine*, the *world* and the *requirement*. The *machine* and the *world* interact by shared phenomena; the *requirement* is a condition over the phenomena of the *world*.

2.2 Methods and Problem Frames

Having identified and named the principal parts of a class of problem, we can begin to talk about methods for problem analysis and solution. For problems to find or construct Polya gives these (and other) heuristics:

- Consider other problems with similar *data* or *unknown*
- Check that you are using all the *data*
- Decompose the *condition* into parts
- Change the *data* to bring it nearer the *unknown*
- Consider how the *data* determines the *unknown*

These heuristics, useful as they are, take no account of the part characteristics of the particular problem in hand. The same problem frame and method fits both “Given three lengths, find a triangle” and the very different problem “Given a matrix, find its inverse”. As a result, Polya’s heuristics can neither focus on nor exploit any specific solution opportunities that the particular problem in hand may offer. The heuristics are very general: any specialisation is left to the insight and ingenuity of the user. The method embodied in the heuristics is therefore less effective in particular cases than it might otherwise be.

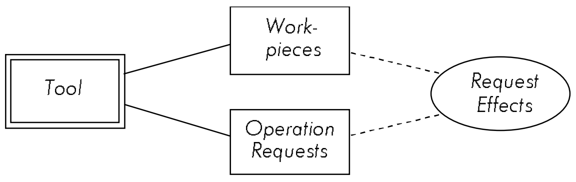
3. Close-Fitting Frames

Effective methods need close-fitting problem frames. The weakness of the problem frame shown above for software engineering problems is its generality: it fits all problems, and so can fit none of them very well.

A close-fitting problem frame fits a narrow class of problem very well, but fits most problems not at all. The frame may specify a particular decomposition of the world into domains; the phenomenological characteristics that a domain must have; the mathematical structures — for example, sequence, group, connected graph, tree — of relationships among phenomena of the domains; and the structure of the condition. In this section some examples of such frames are given and briefly discussed.

3.1 Workpieces Frame

The Workpieces frame may be thought of as capturing a class of problem in which the machine is used as a simple machine tool to create and edit textual or graphic objects. The frame can be represented as shown below.



The principal parts are the *tool*, the *workpieces*, the *operation requests* and the *request effects*. The *tool* is the machine to be built. The *workpieces* are the textual or graphic objects to be worked on; they are intangible, but dynamic; they are inert — left to their own devices they will not undergo spontaneous state changes; they are also mutually independent, there being no connection between one *workpiece* and another. The dynamic nature of the workpieces rests on the elementary editing events, such as ‘insert the character *a* at position 2134’. These are events in the *workpieces* domain, although they can occur only when externally initiated. A proper description of the domain must include a description of these events and the state changes they cause. Because the *workpieces* domain is intangible, we will look to the machine to give it a physical realisation.

The *operation requests* are the requests made by users of the *tool* for operations on the *workpieces*. They form an active dynamic domain: the request events occur spontaneously within the domain. The domain structure is that of a stream of request events over time whose only grammar is request*: there are no relationships among requests other than their raw temporal order.

A request event may not make sense with respect to the current state of the current workpiece; even if it does make sense, it will in general be expressed in terms of phenomena — for example, the current cursor position — that are not phenomena of the workpiece. The *request effects* part in the frame, which is the condition to be satisfied, stipulates whether each request is to give rise to an operation on the workpieces, and, if so, to what operation and with what operands. It is, of course, the developer's task to construct the machine so that it connects the *requests* domain to the *workpieces* domain in a way that satisfies this condition.

3.2 Indicative and Optative

In describing a software engineering problem we will always need to make both indicative and optative descriptions. At any point in a development we have, or must make, indicative descriptions of what is given, and optative descriptions of what is required. As development progresses, some or all of what was required in an earlier stage becomes what is given in a later stage. Old descriptions must be fitted into new places in the indicative/optative structure. (This is one reason why the *shall/will* convention is unsatisfactory.)

For the workpieces frame we must give indicative descriptions of the *operation requests* domain and of the *workpieces* domain. For the first, we need only enumerate and describe the request types: there is, by virtue of the applicability of the frame, no structure over these requests to be described. For the *workpieces* we must describe the state space, perhaps in terms of character strings, the types of event that change the state, and the specified changes for each event type.

The description of the *request effects* condition is optative: we describe the behaviour that we require of the *tool*. Eventually, when we come to describe the *tool* itself in an appropriate programming language, we will be making another description. It is the special power of a general-purpose computer that it can accept such an description and grant the programmer's wishes by becoming the machine described.

It is worth noting that in the workpieces frame we are demanding that the machine play two distinct roles. First, it must provide the physical realisation of the intangible *workpieces* domain. It may seem surprising that we regard the description of this domain as indicative. We do so because we take the view that a problem frame has only one part expressing a required condition: all other parts are domains, and are treated as given. The *workpieces* domain is treated as a given because we are describing the workpieces' properties, not designing them; from a descriptive point of view it is not crucial that they will be realised in the machine.

Second, the machine must ensure satisfaction of the *request effects* requirement. Effectively this will mean receiving operation requests and either initiating or refusing to initiate operations on the *workpieces*.

We will return later to this topic of multiple machine roles.

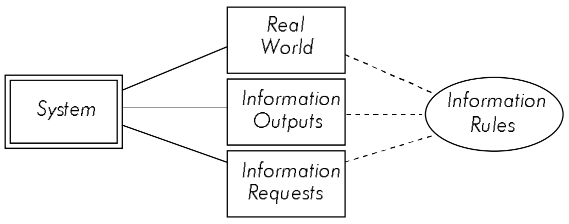
3.3 Workpieces Method

An appropriate description analysis and solution method for a workpieces problem would be one, such as Z[11], based on model-oriented abstract data types. The workpieces are instances of the type. The operations on the workpieces — not the *operation requests* — are the operations of the type. A model-oriented method is appropriate because it allows the workpieces state, which is expected to be relatively complex, to be dealt with in a direct and natural way.

Notice that the problem frame is so simple that it does not provide for error diagnostic messages. A request to delete selected text when no text has been selected will be simply ignored, in accordance with the *request effects* condition. The description of the *workpieces* domain does not, of course, mention text selection: selection is not a property of that domain but only of the required behaviour of the *tool*. It is the *tool* that maintains the relationship between the workpiece contents and current cursor position.

3.4 Dynamic Information Frame

The Dynamic Information frame is shown below.



It captures a class of problem in which the machine is used to provide information about a dynamic domain. For example, a very simple financial information system might provide information about the repayment of loans by bank customers. The principal parts in the frame are the *system*, the *real world*, the *information outputs*, the *information requests*, and the *information rules*.

The *system* is the machine to be built. The *real world* is the domain about which information is required. It is an autonomous dynamic domain, entirely unaffected by the behaviour of the system. The converse, of course, is not true: the *system* is affected by the *real world* because it must maintain some kind of internal model or simulation of that domain in order to be able to generate the *information outputs*.

The *information outputs* domain is dynamic: the outputs are produced over time in response to events and state changes in the *real world* and in response to the *information requests*.

The *information requests* domain is similar to the *operation requests* domain in the workpieces frame. It is an active dynamic domain, consisting of a stream of independent requests over which the only structure is their ordering in time.

The *information rules* part is a condition relating the *real world*, *information requests* and *information outputs* domains. It stipulates the output to be produced in all circumstances.

3.5 Dynamic Information Method

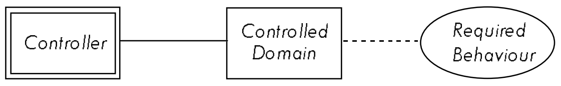
An appropriate method for describing, analysing and solving a dynamic information problem would be one, such as JSD[5], which stipulates the construction of an explicit model, or simulation, of the *real world*. The JSD model is a process model, which makes it more suitable to the dynamic character of the *real world* domain than, for example, a model founded in a database schema.

It is now commonly observed that every software system should embody a model or simulation of its environment. But this observation is not correct. A model in this sense is required only when the system must make use of information about the relevant domain that is not accessible at the time when it is needed. Not all systems are of this nature. For example, a system to control a set of traffic lights can consist merely of the algorithm for stepping the lights through their defined sequence, perhaps modifying the sequence in response to buttons pushed by pedestrians who wish to cross the road. A full description of the environment, including the layout and dimensions of the intersection to be controlled by the lights, the speeds of all vehicles and pedestrians, and the desired traffic flows, is surely necessary in the problem analysis. But it is entirely unnecessary to model these domains in the implemented system. If they appear at all in the software texts it should be only as explanatory comment.

The autonomous nature of the *real world* domain is an important simplifying restriction. In the JSD method there is no provision for describing a reactive *real world* — that is, one that can be controlled by the system. The problem frame does not provide for such control, and when the problem in hand requires the machine to control the environment, then a method such as JSD is found to be much less effective than it is for a pure information system. Essentially, the frame contains no condition over the phenomena of the *real world* alone: an associated method need therefore provide no place for any optative description of the *real world* domain.

3.6 Control Frame

Problems in which the machine must control a part of the world demand a different problem frame. The Control frame is shown below.



The machine here is called the *controller*. It controls the *controlled domain*, ensuring that it satisfies the *required behaviour*.

The *controlled domain* is both active and reactive. That is, it initiates some events and state changes spontaneously, and initiates others in response to stimuli from the *controller*.

A suitable method for a control problem might be a simplified version of the Parnas/Madey method[8], or of the related SCR method[3], or, perhaps, a method based on finite-state machines.

For reasons of space we will not discuss the Control frame further here.

4. Domain Characteristics

The differences between problem frames are partly in their topology. Different frames decompose the world into different numbers of domains: one for Control, two for Workpieces, three for Dynamic Information. In all of these frames every domain interacts directly with the machine, but this is not true of all frames. For example, a frame suitable for the Patient Monitoring problem mentioned in Section 1.4 above would show the Patients domain interacting only with the Analogue Devices, not with the machine: the Patients are an essential part of the problem space, but they are not directly connected to the machine.

But these topological differences are relatively small: many different frames may be expected to have the same topology. More significant are the differences among the phenomenological and structural characteristics of the domains in different frames.

4.1 Domains and Phenomena

The structural characteristics of domains can be captured in familiar mathematical structures, and are not discussed here. The phenomenological characteristics are perhaps less familiar. We can approach them by postulating concepts and notations to capture the crucial differences between temporal and timeless phenomena and among phenomena controlled or causally constrained by different domains.

Phenomena are declared within individual domain descriptions. Where phenomena are shared by two or more domains, the sharing is separately declared; it is not discussed here. However, we observe that an intangible domain such as the *workpieces* domain must share all of its phenomena with another, machine, domain.

The distinction between temporal and timeless phenomena is fundamental. We recognise the following base types of phenomenon, of which the declared phenomena of each domain are subtypes:

- VALUEs and TRUTHs. These are timeless phenomena: a VALUE is an entity whose properties are constant over time; a TRUTH is a fact whose truth value is constant over time. The integers 3 and 5 are VALUEs, and "3<5" is a TRUTH.
- THINGs. THINGs are temporal entities whose properties can change over time. A company is a THING, and so is a valve in a chemical plant.
- STATEs. A STATE is a temporal fact whose truth value is not constant over time. The STATEs "IsOpen(valve1)" and "LivesAt(address1,customer1)" are true at one time but not at another.
- EVENTs. EVENTs are temporal individuals.

Evidently, a static domain is one in which there are only VALUEs and TRUTHs, while a dynamic domain has also THINGs, STATEs and EVENTs.

4.2 Control and Constraint

The control and causal constraint characteristics of declared phenomena are to be understood with respect to the domain in whose description they are declared.

Declarations of EVENTs and STATEs use a convention similar to the CSP and Z convention for distinguishing input from output, but for a different purpose. Our distinction is between external and internal control (in the sense of initiation) of an event or a state change. For example, in the description of a Keyboard domain we may write

```
?KeyStroke = EVENT;
```

while in the description of the User domain we may write

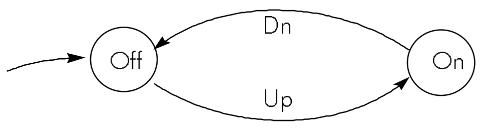
```
!KeyStroke = EVENT;
```

declaring that KeyStroke events are initiated by the User domain. States are treated similarly, their declarations showing whether the domain in whose description they appear initiates the state changes.

We may also declare explicitly the causal constraints which a domain enforces over events and state changes, both internally and externally initiated. The detail of a constraint must be given in a formal description, but it is useful to summarise constraints in more abstract declarations. Consider, for example, a simple switch controlling an electric light. We declare the phenomena:

```
?Up = EVENT;
?Dn = EVENT;
!On = STATE;
!Off = STATE;
```

Up and Dn events are externally controlled; On and Off states are internally controlled. A simple FSM description shows the time-ordering of these events and states:



The constraints involved may be represented directly, distinguishing between safety (-) and liveness (+) constraints:

```
{?Up,?Dn} ±> {!On,!Off};
{!On,!Off} –> {?Up,?Dn};
```

The externally controlled Up and Dn events exercise both safety and liveness constraints on the internally controlled On and Off states; the states exercise safety constraints on the externally controlled events. (It is, of course, impossible for a domain to exercise a liveness constraint on externally controlled phenomena.)

4.3 Dynamic Domain Characteristics

The characteristics of a dynamic domain reflect the phenomena declared in the domain, their control properties, and the constraints exercised within the domain. For example:

- An inert dynamic domain has internally controlled states, but no internally controlled events. All its internally controlled states are subject to both safety and liveness constraints exercised by externally controlled events.
- An fully autonomous dynamic domain has no externally controlled events or states. In any domain, those internally controlled events and states are autonomous that are not subject, directly or indirectly, to constraint by externally controlled phenomena.
- A purely reactive dynamic domain has internally controlled events that are subject to both liveness and safety constraints exercised by externally controlled phenomena.
- An active dynamic domain has internally controlled events that are not subject to liveness constraints exercised by externally controlled phenomena.

4.4 The Nature of Problem Frames

Software engineering is vitally concerned with making and manipulating descriptions. We progress — though not necessarily monotonically — from problem description and analysis to solution design and construction, all the time embodying the work in relevant descriptions.

A close-fitting problem frame provides the necessary basis for method. It identifies and names the principal parts of the problem: these are the subject matter of description throughout the problem-oriented phase of the development.

More importantly, the frame restricts the problem in ways that an effective method can exploit. Restricted domain characteristics and structures relieve the developer from the obligation to consider the full range of imaginable possibilities. In a workpieces problem it is not necessary to consider relationships between one workpiece and another: there can be no significant relationship because the domain structure stipulates that the workpieces are mutually independent. Nor is it necessary to consider, or even explicitly exclude, the possibility mentioned by Lamport[11]:

> "Consider a Modula-2 package that implements a queue by providing *get* and *put* procedures. If we failed to specify that only the environment can call these procedures, then the specification would be satisfied by an implementation that calls the *put* procedure itself to cause random elements to appear in the queue."

By characterising the workpieces domain as inert we tersely exclude the whole range of such possiblities.

Restricted domain structure allows a method to stipulate the use of less powerful but still adequate descriptive languages, thus simplifying both reasoning and manipulation. For example, a restricted time-ordering structure may be described by a regular language rather than by a more powerful but less convenient language.

5. The Structures of Realistic Problems

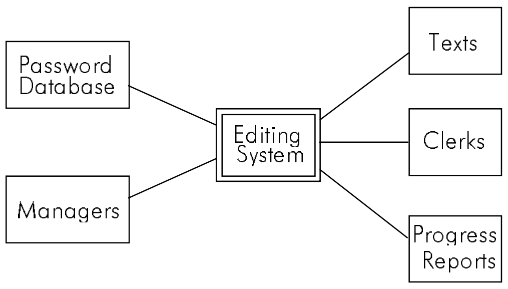
The methodological simplifications made possible by restricted and close-fitting problem frames are bought at a high price. A close-fitting frame is surpassingly unlikely to fit any realistic problem: too much has been excluded, and too many restrictions imposed.

But close-fitting frames can still be applied to realistic problems. A realistic problem can be regarded as a superimposition of subproblems, each fitting a recognised problem frame. Problem frames then play an additional role: they guide the decomposition of a realistic problem into subproblems for which solution methods are known. By taking care to check the fit between a putative subproblem and its close-fitting frame, the conscientious engineer benefits from the frame restrictions by a high degree of confidence that the identified subproblem will be soluble and that the proposed decomposition is, to that extent at least, sound.

5.1 Example: An Editing System

Consider a problem of constructing an editing system to support the work of clerks in a confidential environment. The clerks work on editing texts; there are managers who demand information on the progress of the work; and there are access rules that stipulate that certain operations may be performed only by staff with certain levels of security clearance reflected in their passwords.

The context of the problem can be decomposed as follows:



The problem of the editing system can be regarded as the superimposition of three subproblems:

- A workpieces subproblem, concerned with the task of text editing. In the Workpieces frame the *operation requests* part is the Clerks domain (ignoring the identity of the individual clerks); the *tool* is the Editing System; the *workpieces* part is the Texts; and the *request effects* condition is provided by the specific editing requirements.
- A dynamic information subproblem, concerned with the provision of mangagement information. In the Dynamic Information frame the *system* is the Editing System; the *real world* part is the Clerks and Texts domains taken together; the *information requests* part is the Managers domain; the *information outputs* part is the management reports; and the *information rules* condition is the specification of the make-up of those reports.
- A control subproblem, concerned with enforcing the access rules. The *controlled domain* is the Clerks, Texts, and Password Database taken together; the Controller is the Editing System; and the *required behaviour* condition is provided by the access rules themselves.

5.2 Problem Structure

Of course, the example is contrived and somewhat obvious, and the identified subproblems match just the frames previously discussed. But the example is enough to illustrate some important points about the structure in which subproblems fit together to give realistic problems.

First, the structure is a parallel, not a hierarchical, structure. The subproblems fit together by superimposition.

In considering and analysing each subproblem we ignore the existence of the other subproblems. They are supposed solved: that is, their optative descriptions, wherever they are relevant to the subproblem in hand, are taken as indicative.

Each subproblem is concerned with a subset of the domains of the complete context, and with a subset of the phenomena of those domains. The subproblems are pinned together by phenomena with which they are concerned in common.

Consider, for example, an event in which some text is opened for updating. In the Workpieces subproblem only the operation type (open_for_update) and the text (text T) are relevant; in the Dynamic Information subproblem the operation type, the text and the clerk (clerk C) are relevant; in the Control subproblem the operation type, the text, the clerk and the clerk's password (password P) are all relevant. This individual event is one of the many single points in the problem world at which the three subproblems come together, much like a common event in a complex of CSP processes. The subproblems are not pinned together by large abstractions, nor by the expansion of what is elementary in one subproblem — such as a procedure call — into what is composite in another — such as the procedure body.

An appropriate metaphor for such a problem structure is the CMYK separation familiar from colour printing: each subproblem has its own particular projection of the problem world, and the complete problem is composed by superimposing these projections.

5.3 Composing Subproblems

Where the whole problem requirement is decomposed into the condition parts of the subproblem frames, there is in principle the possibility of conflict. Such a conflict is present in the editing system example.

In the workpieces subproblem the *tool* is required to initiate operations on *workpieces* in response to *operation requests*. Essentially, it is absolved from this obligation only for an ill-defined request, such as a request to delete the current selection when there is no current selection. In the eventual description of the *tool* — that is, the machine to be built — there will be a liveness constraint, applicable to a subset of requests:

{?request} +> {!operation}

However, in the control subproblem there is a safety constraint, applicable to another subset of requests:

{?request} –> {!operation}

Here the relevant requests are those emanating from clerks whose passwords indicate that they are not permitted to access the requested operation. There is a conflict because the two subsets of requests are not disjoint.

It is necessary to detect and resolve any such conflicts. Ideally, the resolution would fully respect the separation of subproblems, but it is not clear whether this should be done. One possibility approach is to compose the descriptions of the conflicting constraints, resolving the conflict in the composition. That is, the description of the liveness constraint would contain embedded conditions reflecting the access rule restrictions. Another, slightly more attractive, possibility is to embed in the liveness constraint only a communication with the safety constraint, abstracting in the workpieces subproblem from the details of the access rules but not from the fact that they must be applied.

Another, very different, possibility is to impose a precedence ordering either on the constraints themselves or on the whole of the containing subproblems. For example, we might specify that the safety constraint in the control subproblem must take precedence over the liveness constraint in the workpieces subproblem. The attraction of such a possibility is that it respects the subproblem separation perfectly: the interaction between the subproblems is confined to a description that is itself a part of neither subproblem.

5.4 Multiple Machine Roles

We observed in discussing the Workpieces frame that the machine has two roles to fulfil. It must act as the *tool*, initiating workpiece operations in response to the *operation requests*; and it must also provide a realisation of the intangible *workpieces* themselves.

Similarly, we could regard the Dynamic Information frame as imposing two obligations on the machine: to realise a suitable model of the *real world*, and to initiate and implement the production of *information outputs* in response to *information requests* and to *real world* events and state changes.

Further, we have perhaps implicitly assumed in the Editing System problem that the machines to be built as solutions to the three subproblems are all to be implemented in one machine.

In considering these multiple machine roles we are beginning to enter the space of solutions rather than problems. One general-purpose computer, by suitable software structuring, can be treated as a complex of machines that may be connected in many different ways; this machine structuring is certainly solution design rather than problem analysis.

The problem frame notation can be slightly extended, if we wish, to allow us to represent machine roles quite directly. In particular, we can indicate on a domain-to-domain connection that one domain is entirely contained in the other (as the *workpieces* domain is entirely contained in the machine that realises it); and we can also elaborate problem frames whose methods demand explicit models to represent those models as separate domains (as we might represent the model of the *real world* domain in the Dynamic Information frame as a separate domain).

5.5 Composing Solutions

In general, even in the absence of conflict, it will be necessary to achieve some composition of the subproblem solutions. Pieces and aspects of machines that implement subproblem requirements must be made to work together, at least to the extent of co-ordinating their execution; and the same is true of pieces of machines that realise intangible domains or model tangible domains. This need brings its own complexities in the solution space.

The composition of conflicting requirements has already been briefly discussed. Composition is also needed in the absence of conflict. For example, an individual occurrence of an open_to_update operation event must be initiated by the workpieces subproblem *tool*, and it must also be recorded — more properly, modelled or simulated — by the *system* in the dynamic information frame. Indeed, the system in the dynamic information frame may need to model a considerable part of the history of each clerk and each text: some at least of the text history will replicate a part of what is realised for the text as a *workpiece*.

This necessary composition can be achieved by a variety of strategies located at or between two extremes. At one extreme is what we may call the 'strategy of tightest composition'. In this strategy each object implemented in the machine combines the properties it needs for all subproblems to which it is relevant. This is the traditional approach. Its advantage is that it leads to solutions that are simple in implementation terms, even if they are complex in problem terms. Its crucial disadvantage, long ago implicitly recognised by researchers in object technology, is that it virtually guarantees that the resulting implementation objects will have no value for re-use: the particular combination of subproblems that gave rise to them will not recur.

At the other extreme is the 'strategy of loosest composition'. In this strategy each machine object implements only one subproblem projection, whether of a requirement or of a domain model or realisation. The advantage of this strategy is clear: the separation and structure achieved in the problem decomposition is maintained in the solution. The disadvantage is that it demands implementation support that is scarcely yet available. The inadequacy of a hierarchical scheme of object class inheritance — especially in the single-inheritance form — is well understood. But it is not yet clear what must take its place. Some of the patterns now well known in the object-oriented world (for example, the *Decorator* pattern[6]) aim to provide ways of combining different subproblem projections. But combination is not enough. It is also necessary to implement identity: to be able to specify simply that the event x in A is the same event as the event y in B. This is a much harder need to satisfy.

6. Some Observations

The problems we set ourselves in software engineering, and our consequent expectations, are perhaps unreasonably demanding. Traditional engineers, who design and build aeroplanes or TV sets, chemical plants or bridges, motor cars or electricity generating plants, certainly deal with realistic problems. In spite of what we may like to think, their problems are not always dramatically simpler than ours. But when we castigate ourselves for our failure to achieve the levels of reliability and confidence of traditional engineers, we tend to forget an important difference.

Traditional engineers work mostly on problems that are only small perturbations of problems that have been solved many times before. They do not have, and do not need, constructive methods for tackling large and complex problems de novo: instead they apply their collegiate experience to construct solutions that are as far as possible just like the known solutions to previously solved problems. Constructive methods in traditional engineering are focused on very tightly constrained solutions to very tightly constrained problems.

The use of problem frames, and the decomposition of realistic problems into parallel structures of subproblems that fit close-fitting frames, can be seen as an attempt to emulate traditional engineering in this respect. The goal is to absorb significant parts of software engineering into specific well-recognised problem classes with well-recognised solutions. The major difficulties arise in the composition of solutions to the subproblems.

Problem frames can also be seen as embodying an attempt to capture and systematise some of the heuristics and expedients that are well-known to experienced software engineers: find a related but simpler problem; think of a problem that you have previously solved that partially solves your present problem; recognise recurring patterns; think of solutions you have previously used and check them for fit to your problem. The similarity to work in object-oriented patterns is strong, but there is an important difference. Most of the patterns work focuses on solutions, but problem frames aim to focus on problems. Problem analysis should be problem-oriented.

References

[1] Tom DeMarco. Structured Analysis and System Specification. Yourdon Press, 1978.

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

[3] 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 (July 1996), pp231-261.

[4] C A R Hoare. Communicating Sequential Processes. Prentice-Hall International, 1985.

[5] M A Jackson. System Development. Prentice-Hall International, 1983.

[6] Michael Jackson. Software Requirements & Specifications: A Lexicon of Practice, Principles, and Prejudices. Addison-Wesley, 1995.

[7] Leslie Lamport. A Simple Approach to Specfiying Concurrent Systems. Comm ACM, Volume 32 Number 1 (January 1989), pp32-45.

[8] D L Parnas and J Madey. Functional Documentation for Computer Systems Engineering (Version 2). CRL Report 237, McMaster University, Hamilton Ontario, Canada, 1991.

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

[10] W P Stevens, G J Myers, and L L Constantine. Structured Design. IBM Systems Journal Volume 13 Number 2 pages 115-139, 1974; reprinted in [Freeman 83] pages 328-352.

[11] Jim Woodcock and Jim Davies. Using Z: Specification, Refinement, and Proof. Prentice-Hall, 1996.