Chapter 13

Software Development Method

M. A. Jackson

It is a great pleasure to be able to contribute to a Festschrift in Tony Hoare's honour. I have known him since we were both undergraduates nearly forty years ago, and have found continually renewed reason to admire his rare combination of an open mind with the highest standards of clarity and precision. Unlike many computer scientists, he is receptive to ideas painted on a broader canvas than that of a mathematical formalism. Knowing that, I have dared to take the whole of software development as my subject.

13.1 Introduction

Software development is a human activity. Like any human activity, it has many facets, and can be analysed from many points of view. Social scientists and experts in human relations examine the interactions among developers and between the development group and its customers. Lawyers and trades unionists see software systems as a means of changing working practices. Business theorists analyse development projects for their profitability, and measure return on investment. Quality control experts see the need for process optimisation based on statistical evidence. Some software developers see themselves as engineers, and want to be judged by the canons of such established disciplines as aeronautical, civil, or electrical engineering. Many say that programs are mathematical objects whose creation and study is a legitimate branch — some would say, the most challenging branch — of mathematics.

Each of these views is valid, at least for some parts of some projects. But when all the experts have claimed the substance of their different disciplines, is there any particular substance left that belongs peculiarly to software development? Or are we left only with the grin of the Cheshire Cat when its body has departed?

Software development does have a substance all of its own that gives it its special character. Software development is about the structure and technique of description. The task of the developer is to create transparently clear descriptions of complex systems in which many domains meet and interact — computing machinery and many other things too. Mathematics is an essential tool in software development, and in many parts of a large development proofs and calculi must be brought to bear. But software development is not mathematics any more than bridge building is mathematics. The central emphasis is on description more than on invention, on structure more than on calculation, on achieving self-evident clarity more than on constructing proofs of unobvious truths. Some properties of complex systems will demand and admit formal proof. But most will require a transparent clarity of description, and a separation of concerns by which clarity may be attained.

Development method, therefore, must concentrate on the separation of concerns and on their description once they have been separated: the central questions are what to describe, and how to describe it. Because the systems we develop are complex, there will be many things to describe, and their descriptions will be of many kinds. The relationships among descriptions will form non-trivial structures, and we will be concerned to keep intellectual control of these

structures and to know clearly what each description describes and how it is related to the other descriptions. These concerns are not always well served by our present culture of software development.

3.2 Our Product Is a Machine

Software development is description. Yet the goal of our description activity is the construction of a machine. At the heart of every system we build is a machine — or, in a distributed system, several machines. The machine is our central artifact, the chief domain that we must describe: a tangible machine, not a mathematical abstraction. Of course, abstraction may be a necessary tool in arriving at the machine's description; but in the final analysis we must produce a machine with a certain behaviour appropriate to supporting the administration of a library, or to switching telephone messages, manipulating documents, or controlling a motor car or washing machine. Our products will usually be far more complex than a motor car or a washing machine, but must always be just as tangible.

Although our artifact is tangible, we make it merely by describing it. A general-purpose machine — a Universal Turing Machine — accepts our description of the special-purpose machine we wish to construct — a Turing Machine — and converts itself into that machine. It does not only compute about our description; it does not only translate it or analyse it. It actually adopts the properties and behaviour of the machine we have described, and itself becomes that machine. This relationship between our descriptions and the material world allows us, if anything does, to claim that software development is engineering.

The power of the special-purpose machines we construct by our descriptions is, ultimately, limited by the power of the general-purpose machines that clothe themselves in those descriptions. We can specialise the behaviour of the general-purpose machine, and we can understand its specialised behaviour in terms of our own choosing. But we can not enlarge its behaviour — we can neither extend its state space nor add to its alphabet of events. We can only reduce it to meet our special needs, and we must take care to understand the extent of that reduction.

If, for example, our need is for a machine specially adapted to evaluating pure function applications, we may implement a pure functional language by describing the machine that interprets that language. The implementation provides us with a machine that operates in a Read-Eval-Print loop, each iteration of the loop reading a description of a function and its arguments and computing and printing the resulting value. The functional language itself rigorously eschews side-effects and abstracts from the order of function evaluation, this abstraction being a key advantage of such a language for its declared purpose. The resulting machine is ideally suited to its special purpose; but it is not at all suited to controlling a motor car, or to switching messages, or to word processing. Although the behaviours necessary to such purposes could be *described* in the functional language, and the function-evaluation machine could *compute about* them, it can not itself *adopt* such behaviours. Its behaviour has been specialised to the Read-Eval-Print loop. The fact that it can compute — without side-effects — about motor car control, message switching, and word processing does not make it useful for performing those tasks. And the goal of software development is building useful machines.

13.3 The Machine and Other Domains

A fundamental rule of methodology is the rule of proportionate method: the scale of the method must be proportionate to the scale of the problem. For a trivial problem of software

development, we just write the program in an available programming language. For a slightly less trivial problem we devote some effort to structuring the program. As the problem becomes more difficult we must begin to pay serious attention to its subject matter.

Even a trivial problem is about something. It has some subject matter, some real world, some domain of discourse other than the machine itself. A payroll system is about the employees and their work and pay; a process control system is about the plant and its vessels, valves, and pipes; a system for producing a Shakespeare concordance is about the texts of Shakespeare's plays and sonnets. Even when the subject matter is abstract — a graph to be traversed or a number to be tested for primality — it still furnishes a domain distinct from the machine itself.

The subject matter domain invites description in its own right. But for a small or fairly simple problem we often neglect to make such a description. And often we do so with impunity, because the description of the subject matter can be inferred from a reading of the program texts. A Pascal program, carefully constructed, shows clearly the structures of its inputs and outputs, and, if it is conversational, how its behaviour is interleaved with that of its user. The COBOL text of a payroll program exhibits the rules for calculating gross pay, and also the format of the printed payslips. Indeed, the originators of COBOL and some other programming languages wanted them to be ‘problem-oriented languages’, in which the program text would be itself a clear description of the subject matter of the program and the problem that it solved.

This view can be vigorously justified. The machine interacts with its environment, where the problem is located. The interaction can be viewed as an interface of shared phenomena. State is shared because the environment contains sensors that the machine can interrogate, and vice versa; events are shared because both environment and machine must participate in each event occurrence. It follows that an event trace of the machine behaviour at the interface is also a trace of the environment behaviour there; and the same is true of a trace of interface states. By describing one we also necessarily describe the other. Separate descriptions would be otiose.

Another, related, justification rests on the modelling relationship between the machine and the problem domain. In many systems the machine must incorporate a model of the problem domain; in an object-oriented system, for example, the objects are models of real-world objects of interest. This means that there are certain aspects of the machine's properties and behaviour, and certain aspects of the problem domain's properties and behaviour, that are both described by the same description. We may trust that these are the only aspects of the domain that are of interest. By describing the corresponding aspects of the machine we discharge any obligation to describe the problem domain.

13.4 Requirements, Specifications and Programs

In serious development, these justifications break down. The interface between a machine and its environment is rarely transparent. In a data processing or administrative system the interface is usually an elaborate structure of hardware and software, meriting careful attention in its own right. Events and states of the domain are not shared with the machine, but are transmitted unreliably and often with considerable delay; even reordering of messages is not excluded. In a control system, or an embedded system, the interface is usually more reliable, but still often imperfect.

The modelling justification is no more robust. The domain may have important properties that the system relies on but the machine does not model explicitly. One important property of a lift mechanism is that when the motor is set to ‘up’ and turned on, the lift will start to rise: the

control system relies on this property, but it does not appear in the description of the control machine. More significantly, the machine will have many properties that are not shared with the problem domain or environment. If we are restricted to one description for both we can have no way of separating the properties they share from those that they do not.

There are many positive reasons for making separate explicit descriptions of domains other than the machine. Separation of concerns is more effective when it is accompanied by a separation of descriptions: no compiler writer would argue that a separate description of the grammar of the language to be compiled is unnecessary. Nor is the grammar thrown away or put on the shelf once it has been written down; in one way or another it provides a foundation for describing the parsing behaviour of the machine. The commonly heard plea that we should describe *what* the machine does before we describe *how* it does it is, in effect, a plea for a separate description of the problem domain. A brief thought experiment will show why. Ask yourself *what* a motor car does, and try to answer purely in terms of the car itself. The task is impossible. To say *what* a car does, we must talk of roads and passengers and drivers and fuel and baggage. They are not parts of the car.

Here we may draw a useful distinction between Requirements, Specifications, and Programs. A Requirement is located in the problem domain. Whatever the problem domain may be, it is certainly distinct from the machine domain. It is where the customer for the software will experience, interpret, and evaluate the effects brought about by the machine. A Requirement is not an informal or vague Specification. It can be as formal as we wish, even if the domain itself is informal. But it is about the problem domain, and not about the machine.

A Specification, by contrast, is a description of the machine itself. It describes those properties that the machine must have to satisfy the Requirement. Usually we expect a Specification to be a more abstract description of the machine than a Program: for example, it may describe the machine only implicitly. But it is still a description of the machine.

13.5 The Idea of a Problem Frame

The distinctions between *what* and *how*, between Requirement and Specification, and between Specification and Program, are too general to serve as the foundations of a method. It is not good enough to speak vaguely — as I have done — of the ‘problem domain’ or the ‘environment'. To get a sufficient grip on a problem we must fit it into a tighter conceptual framework. Such a framework may be called a Problem Frame. It provides a structure into which we can fit the problem so that it can be worked on, almost as a mechanical part may be fitted into a jig for machining or welding.

The essential idea of a problem frame — though not by that name — is explained by Polya in his monograph *How To Solve It*. A problem can be analysed into its ‘principal parts'. The idea is due to the ancient Greek mathematicians, and Polya illustrates it with problems in elementary Euclidean geometry. He distinguishes ‘problems to prove’ from ‘problems to find’. An example of a problem to find is: Construct a triangle whose sides have the lengths *a*, *b*, and *c*. The principal parts of a problem to find are the Unknown, the Data, and the Condition. The task for the solver is to find an Unknown of the required kind that satisfies the Condition. Here the Unknown is a triangle; the Data are the lengths *a*, *b*, and *c*; and the Condition is that the triangle should have sides of the lengths given in the Data.

A problem to prove has different principal parts: it has a Hypothesis and a Conclusion. The task for the solver is to deduce the Conclusion from the Hypothesis, or else to show that the deduction is impossible. We recognise the different kinds of problem because they have different principal parts. More exactly, we choose to treat a given problem as a problem of one kind or the other because we can fit it comfortably into the prescribed structure of

principal parts. But this choice is not objective, and is not always clear. A problem to prove, whose statement begins ‘prove that there is at least one integer such that ...’, may obviously be treated as a problem to find; a problem to find, whose statement begins ‘find an integer such that ...’, may sometimes be solved by guessing the integer and proving that it has the required property.

Having identified the principal parts of the problem, Polya proceeds to recommend a method. The method consists largely of heuristics: Split the Condition into parts; Check that you are using all the Data; Vary the Unknown to bring it closer to the Data; Think of a familiar problem having a similar Unknown. Only the establishment of the problem frame makes it possible to discuss method in this way. The recommendations are cast in terms of the defined problem frame, of the named principal parts and their distinctly identifiable roles, and in terms of their relationships to one another.

13.6 Problem Frames for Describing Machines

This is the essence of any problem-solving method, and methods for solving software development problems are no exception. Each method offers a particular problem frame, a characterisation of its principal parts, and a prescription for solving the problem by building a particular sequence of descriptions of its parts, culminating in a description of the desired machine. Usually certain languages are stipulated for describing particular parts, because they can capture and exploit the characteristic properties of those parts. Certain operations may also be stipulated for constructing the descriptions, exploiting the properties of their stipulated languages and thus again, indirectly, the characteristic properties of the principal parts.

Here, for example, is the traditional problem frame for Top-Down Functional Decomposition:

Machine The machine that is to be described. It autonomously executes one procedure.

Function What the Machine is to do: that is, the procedure it is to execute.

The solution task is to find a Machine that can execute the Function. A typical prescription for solution is to describe the Function in a succession of procedural descriptions. The first describes it at a single procedural level in natural language. Later descriptions introduce additional levels of invoked procedures, and rely less on natural language and more on the programming language. At each step the current set of descriptions forms a hierarchy of procedures. The final set, expressed entirely in the programming language, is interpretable by the general-purpose computer. It describes the desired Machine, and so solves the problem.

This problem frame, and hence inevitably any method that uses it, is extremely weak. The principal parts are so general that it is impossible to imagine a software development problem that they could not fit. But they fit no problem well. The only specific aspect is the characterisation of the Machine as a hierarchy of executable procedures. This would no doubt frustrate a developer resolved or required to use Prolog: but the characterisation is more one of the solution than of the problem or its parts. It is therefore impossible for Top-Down Functional Decomposition to be regarded as a serious method: it simply provides no grip on the problem.

Here is another problem frame, associated with Model-Oriented Specification methods:

Machine The machine that is to be described. It does nothing autonomously, but responds to user requests.

Operations Operations, atomic from the user's point of view, that the Machine performs at the user's request.

Model The state space of the Machine, as constrained by certain invariants and traversed by performance of the requested Operations.

The multiplicity of Operations, their atomic nature, and their relationship to the state space and its invariants, make this a stronger problem frame. The solution task is to find a Machine that can perform the requested Operations while maintaining the invariants of the Model. A number of different prescriptions are offered for solution. Various languages are prescribed for describing the Model structure and invariants, and for describing the Operations by relations on Model states. Eventually the Model is to be transformed into a data structure within the Machine, and the Operations into executable procedures. In a data-processing problem the Model data structure may be implemented in a relational database. In a different kind of problem it may be the representation of an abstract data type instance.

A related, but different, problem frame is used by Property-Oriented Specification methods. There is no Model principal part, and the Operations are described entirely in terms of their relationships to one another. For problems of a certain class this provides a simplification, and a release from suspicions that the Model is biased towards its eventual implementation.

13.7 Richer Problem Frames

These problem frames are relatively impoverished by their exclusive focus on the machine. Development of more demanding systems needs richer problem frames in which some of the principal parts are explicitly devoted to other domains.

The problem frame for the basic version of Structured Analysis and Specification has these parts:

System The machine that is to be described, including the computing machinery and possibly some clerical operations. It transforms input data flows to output data flows.

External Entities Entities, such as customers, suppliers, and other systems, that supply flows of input data to the System and receive its output data flows.

Function What the System must do to transform and process its inputs to produce its outputs.

Data Stores What the System must remember to perform the Function.

The solution task is to find a System that will perform the Function, interacting appropriately with the External Entities. The basis of the method is to describe the System as a process communicating by data flows with its External Entities, and maintaining and using the Data Stores. The resulting data flow diagram is then elaborated by replacing the System process symbol by a more detailed diagram showing two or more processes communicating by data flows, and so on successively until finally every process represented is simple enough to be directly described in a procedural ‘mini-specification’.

This method is far less rich than it may appear at first sight, and far less rich than its problem frame would allow. The External Entities, which may be regarded as embodying the problem domain or environment, are not directly described. The input and output data flows of each Entity are separately described (but not the relationship of an Entity’s inputs to its outputs), and a part of its state may be indirectly modelled in the Data Stores; but nothing more.

JSD is richer as a method. The principal parts of its problem frame are:

System The machine to be described. It runs a simulation and produces information about it both autonomously and on request.

Real World The problem domain about which the Machine is to compute, and of which it is to embody a model. The Real World has an autonomous behaviour over time, which is the subject matter of the simulation.

Function The production of output messages and reports by the Machine, containing information about the Real World obtained from the simulation.

The solution task is to find a System that models or simulates the Real World and performs the Function by extracting information from its Real World model. The first step of the method describes the Real World as a set of sequential processes in which events are represented abstractly. The same descriptions are then used to define the model within the System, the event representations now denoting receipt of messages about Real World events. The Function is described in terms of output operations embedded in model processes, and additional processes that communicate with the model and produce outputs reflecting its state. Finally, an executable description of the System is obtained by transforming the resulting process network so that it may be composed with the description of a scheduling scheme.

Another problem frame may be called the Workpiece problem frame. It may be suitable for such applications as word-processing. Its principal parts are:

Machine The machine to be described. It creates, manipulates, displays and exports objects at the user's request.

Workpieces The objects, often textual and graphic documents, that are to be worked on with the help of the Machine. The objects are inert: that is, they have no autonomous behaviour.

Worker The user — the person operating the Machine and working autonomously on the Workpieces.

Operations The operations that the Worker can ask the Machine to perform on the Workpieces.

The solution task is to find a Machine that allows the Worker to work on the Workpieces by performing the Operations on them. The first prescribed step may be to describe the type of the Workpiece as a data object with invariants. Then to describe the permitted sequences of Operations on a Workpiece, and the effect of each Operation. Next, to describe the behaviour of the Worker in terms of permitted sequences of Operations on a set of Workpieces. And finally to describe the Machine behaviour in response to the Worker behaviour, the Machine invoking appropriate Operations on the selected Workpiece objects or engaging in a diagnostic dialogue when the Worker has made a mistake.

Our last example may be called the Environment-Effect frame: some recent work by Parnas uses a version of this problem frame. It is a richer frame, and the associated method is more elaborate. It may be suitable for an embedded system that controls an external domain. Its principal parts are:

Machine The machine to be described. Its behaviour may be partly autonomous and partly responsive.

Environment The domain to be controlled by the Machine. It has state, and a behaviour that is partly autonomous and partly responsive.

Requirement The domain properties and behaviour — relationships among domain phenomena — that the Machine is to bring about.

Connection The connection between the Machine and the Environment by which the Machine can sense and affect states and events in the Environment.

The solution task is to construct a Machine that senses and controls the Environment through the Connection, and brings about the Requirement. In doing so, the Machine must take proper

account, and proper advantage, of the properties that the Environment possesses independently of the Machine's behaviour.

A method adopting the Environment-Effect problem frame might prescribe that the first description should be of the Environment, stating the properties and behaviour that it possesses independently of the Machine. Then the Requirement should be described, stating the additional behaviour and properties that we desire the Environment to have. In general, the Requirement will be expressed at least partly in terms of phenomena that the Machine can not control directly, and perhaps can not even sense through the Connection.

The Requirement description should therefore then be refined, using relationships stated in the Environment description, so that it becomes feasible. That is, it must require only such Environment states to hold, and only such Environment events to occur, as the Machine can cause through the Connection; it must forbid only such Environment states and events as the Machine can inhibit through the Connection; and the conditions for causation and inhibition must be expressed only in terms of Environment states and events that the Machine can sense through the Connection.

Then the Connection should be described, to make explicit the association between phenomena it shares with the Environment at one end, and phenomena it shares with the Machine at the other end. Finally, the refined Requirement should be further modified, using this Connection description, into a Specification: that is, a description of the desired Machine expressed entirely in terms of Machine phenomena.

A simpler version of the Environment-Effects frame omits the Connection, assuming it to be reliable and effectively transparent.

13.8 Fitting the Frame to the Problem

Even from these cursory, disputable, and greatly simplified accounts of some methods and their associated problem frames, it is apparent that many different problem frames are possible, and that the choice of an appropriate problem frame is a matter of serious concern. An ill-fitting problem frame will be as irksome as an ill-fitting pair of shoes. Progress is not impossible, but it is painful; and as the journey proceeds the pain may eventually become crippling. Developers who, for whatever reason, find themselves using an unsuitable problem frame can resort to various unsatisfactory expedients. Like Procrustes they can force the problem into the frame whether it fits or not. More intelligently, they can bend the frame to accommodate the problem, or add parts to the frame structure to accommodate problem parts that would otherwise be neglected. Of course, there is a price to be paid. The accompanying method becomes progressively less useful as the problem frame becomes more distorted.

The fitness of a frame to a problem may be checked by some simple informal tests. They should be applied consciously at the outset of a development, while the shoe has not yet started to pinch, and a reasonably unconstrained choice may still be made.

The *separability test* requires that the problem can indeed be teased apart so that the principal parts are properly separated. The Model-Oriented Specification frame would fail this test for the problem of specifying a program that conducts a typical interactive dialogue with the user. The dialogue would have to be viewed as a series of user inputs, alternating with machine responses. Each user input would be regarded as an atomic Operation request, and the program's response as the Machine's performance of the requested Operation. But a dialogue can not usually be separated into request-response pairs in this way. The interaction is more complex, the initiative passing to and fro between the user and the program. Separating the dialogue into a succession of pairs, each consisting of one user request and one corresponding program response, would be very difficult and entirely inappropriate.

The *completeness test* requires that every part of the problem should be accommodated in a natural way in some principal part of the frame. The JSD frame would fail this test for certain kinds of control problem. The Real World model part of the JSD frame accommodates a description of the real world as it actually is at any point in the history of the system. But it provides no accommodation for a description of what in the Environment-Effect frame is called the Requirement — that is, of the real world as it is supposed to be but will not be unless the system ensures that it is so. To take a trivial illustration, consider the requirement in a library system that no borrower may have more than 6 books out on loan at any time. Clearly, the limit of 6 books is not a part of the Real World model: in the absence of a properly functioning system borrowers will no doubt borrow as many books as they wish. Nor can it be regarded as a part of the Function: the System may produce a suitable warning output when a borrower threatens to exceed the limit, but the description of such a warning is not itself a statement of the Requirement. There is simply nowhere in the JSD frame to state the Requirement.

The *proportionality test* requires that the parts of the problem frame should be filled approximately equally: no principal part should be filled to overflowing while another is almost empty. The Structured Analysis and Specification frame would fail this test for the problem of printing the Nth prime number. The External Entities are the anonymous source of the input number N and the anonymous recipient of the computed prime. The sole input data flow is just one integer, and so is the sole output data flow. The Function accommodates almost the whole of the problem, while every other principal part is empty or nearly so.

The *part-characteristics test* requires the identified principal parts to exhibit the characteristics expected of them. The Workpiece frame would fail this test for a process control problem. If the plant to be controlled were identified as the Workpiece, it would lack an essential characteristic expected of that principal part. The Workpiece is expected to be inert: it does nothing of its own initiative, but only waits to be operated upon at the Worker's request. But the plant in a process control system, if left to its own devices, is not inert; it is replete with vessels overflowing and emptying, with liquids reaching critical temperatures and gases reaching critical pressures.

The tests are, unsurprisingly, far from orthogonal. A frame that fails to fit on one test can usually be judged to fail on another too. And if we stretch a point to pass one test we will usually aggravate the misfit elsewhere.

13.9 Problem Frame Complexity

In applying these tests, we must always bear in mind the rule of proportionate method. Even with a large repertoire of problem frames we should not expect to fit every problem perfectly. Where the misfit is not too serious, and the project admits of some compromise, we must be willing to bend the frame a little, or perhaps to defer a small and non-critical part of the problem until a later stage when it can be dealt with ad hoc. The pass mark in the frame fitness tests is less than 100%. But even when we lower the pass mark we will often find problems that simply do not fit any frame.

Such problems may need more than one frame. This is a form of problem complexity, to be measured, not by an absolute measure, but by the power of the problem frames and methods we have available. We tackle it by analysing a problem into subproblems, each with its own appropriately chosen problem frame.

Consider this small mathematical problem: Given a point *p* and three lengths *a*, *b*, and *c*, construct the circle centred at *p* and circumscribing a square whose area is equal to that of a triangle whose sides are of lengths *a*, *b*, and *c*. Polya's simple Unknown-Data-Condition

frame does not fit this problem: it fails the proportionality test, because the Condition that relates the Data (*p*, *a*, *b*, and *c*) to the Unknown (a circle) is disproportionately elaborate. The problem may be structured into three subproblems, each fitting Polya's frame:

Subproblem-1

Unknown-1 A triangle.
Data-1 The lengths *a*, *b*, and *c*.
Condition-1 The triangle sides are equal to the lengths.

Subproblem-2

Unknown-2 A square.

Data-2 A triangle.

Condition-2 The square and triangle have equal areas.

Subproblem-3

Unknown-3 A circle.

Data-3 A point *p* and a square.

Condition-3 The circle is centred at *p* and circumscribes the square.

This little complexity is of an obvious kind, and its solution is obvious. There is only one problem frame to be applied, and the problem is solved by applying it three times. The subproblems are linked together in a chain, the Unknown of one subproblem serving as the Data of the next. This chain structuring of subproblems occurs also in the design of sequential processes. Starting with a problem that we expect will fit a frame whose parts are Input, Process, and Output, we find that the problem does not fit the frame. Perhaps it fails the proportionality test, or perhaps the method insists on some mapping between Input and Output that can not be achieved within one simple sequential process. We then form a structure of subproblems, the Output of one furnishing the Input of the next.

Top-Down Functional Decomposition can be viewed in a similar light. Instead of a chain of subproblems there is a tree or hierarchy. Criteria for decomposition may be mere rules of thumb: no procedure text may be more than 50 lines long. Or they may attempt to capture some required characteristic of procedures, which are the chief constituents of the principal parts in the Top-Down problem frame: for example, by defining measures such as those of coupling and cohesion.

13.10 Heterogeneous Complexity

But the weakness of the Top-Down problem frame allows only very weak criteria for problem decomposition. Nor is it usual to find that one problem frame will suffice for a problem, even if it is applied more than once. Just as a richer problem frame allows a more helpful method at the level of solving individual problems — the problem can be fitted with more confidence into a frame whose parts have stronger characteristics, and the prescriptions of the method can be more specific and more detailed — so a repertoire of several different richer and stronger problem frames makes it easier to structure a complex problem into subproblems. The aspects or parts of the problem that should be identified as individual subproblems and fitted into separate frames are more easily identified because the frames themselves are more tightly constraining.

Consider, for example, the problem of constructing a simple CASE tool. The Workpiece problem frame would be obviously appropriate, the Workpieces here being the software descriptions whose construction and manipulation the tool is to support. But if the tool is also

to provide some information for the management of the development project, this frame is clearly not enough. For the problem of providing the managment information the JSD problem frame may be suitable. The principal parts of the two subproblems are then:

Workpiece Frame-1

Machine-1 The machine supporting substantive development work.

Workpieces-1 The software descriptions being developed.

Worker-1 The software developer.

Operations-1 The development operations on the software descriptions.

JSD Frame-2

System-2 The machine providing management information about the development.

Real-World-2 The development products**(Workpieces-1)** and the work done on them **(Operations-1)**.

Function-2 The reports and messages containing management information about the development.

The problem decomposition is uncontentious, given the small repertoire of problem frames we have made available to ourselves in our discussion, and the relative strength of those frames. The two problem frames are linked together by their common parts: the Workpieces-1 and Operations-1 parts of the first frame furnish the Real-World-2 part of the second. Machine-1 and System-2 need not be realised by the same general-purpose computer, although that will often be a convenient choice.

Here are some other examples of such complexities and possible solutions:–

- In a problem for which the Environment-Effect frame has been chosen, the connection is not provided ready-made, but must be built by the developers.

  Environment-Effect Frame-1

  Machine-1 The machine of the substantive system.

  Environment-1 The environment of the substantive system.

  Requirement-1 The substantive requirement, not including the behaviour of the connection.

  Connection-1 The connection to be constructed.

  Environment-Effect Frame-2

  Machine-2 This is **Connection-1**.

  Environment-2 This is **Machine-1** and **Environment-1**.

  Requirement-2 This is the behaviour required of **Connection-1**, connecting phenomena of **Machine-1** to phenomena of **Environment-1**.

  The second subproblem also uses the Environment-Effect frame, but this time without the Connection part. In the second subproblem the connection is presumed to be reliable and transparent, comprising just the interfaces between Connection-1 and Machine-1 and between Connection-1 and Environment-1. These, of course, we must indeed regard as interfaces of shared phenomena: if we did not, we would have an infinite regress of connections.

- A substantive system is to be developed with significant security and access control properties.

  Environment-Effect Frame-1

  Machine-1 The machine of the substantive system.

  Environment-1 The environment of the substantive system.

  Requirement-1 The substantive requirement, not including access restrictions.

  Connection-1 The connection of the substantive system, perhaps including terminals at which users can interact with the system, and at which access is to be restricted.

  Environment-Effect Frame-2

  Machine-2 This is the machine that implements the access control rules.

  Environment-2 This is **Connection-1**, the terminals and interactions whose use is to be restricted, together with other phenomena (such as passwords) associated with access control.

  Requirement-2 This is the constraint on **Environment-2** (that is, on **Connection-1**), by which access to interaction with the substantive system is restricted.

  Again, the Environment-Effect frame is used for each of two subproblems. The terminals and interaction procedures for the substantive system appear as part of Connection-1 in the first subproblem. In the second subproblem they appear as the part of the Environment that is to be controlled.

- A CASE tool is to support a software development activity, and also to impose constraints reflecting management or methodological considerations.

  Workpiece Frame-1

  Machine-1 The machine supporting substantive development work.

  Workpieces-1 The software descriptions being developed.

  Worker-1 The software developer.

  Operations-1 The development operations on the software descriptions.

  Environment-Effect Frame-2

  Machine-2 The machine providing management and methodological control of the development activity.

  Environment-2 The products of development **(Workpieces-1)**, the work done on them **(Operations-1)**, and possibly also the software developer **Worker-1**.

  Requirement-2 The constraints to be imposed on the software development activity.

  The development activity domains of the first subproblem furnish the environment to be controlled in the second subproblem. The behaviour of Worker-1, regarded as essentially autonomous in the first subproblem, is to be brought under control of the machine in the second subproblem.

Decomposition into subproblems often suggests a natural order of proceeding with the development. For example, where a domain common to two subproblems appears as the Machine in one of them, it may well be desirable to solve that subproblem first, at least to the point at which the domain has been described well enough for its use in the other subproblem. Such ordering constraints can sometimes be circumvented by suitable abstractions: the

outcome of the first subproblem can be prejudged well enough for the purposes of the second. Sometimes the constraints can not be overcome in this way. Sometimes, what is even worse, there is a circularity in the natural ordering — in essence, a high-level failure of separability in the problem.

13.11 Languages for Descriptions

The use of richer problem frames, and of more than one for a single problem, demands descriptive techniques outside the main stream of the traditional culture of software development.

The traditional culture has grown from its origins in programming. Programming languages — even those of the rococo school — aim at relatively simple computational models: that is, they adopt simple problem frames. Elegance is achieved by economy and simplicity of principal parts. This elegance is so highly valued that some language designers even banished the vital but inconvenient complexities of input and output from the scope of the programming language itself, so that what remains could be captured in a few simple concepts, or — even better — in just one. If the frame can not be reduced to a single principal part, it may still be possible to give all the principal parts the same characteristics. Everything is an S-expression; or everything is an object; or everything is a goal that can succeed or fail. We expect to write each program in a single language, in which such qualities as uniformity and referential transparency are highly prized.

This approach led to notable success in the design and definition of programming languages, and in the construction of compilers. So it is natural to persist in the same tradition in the larger work of systems requirements, specification, analysis, and design. We still hope to express our descriptions in a single language that shares the prized qualities of traditional programming languages.

But the tradition, for all its virtues, constrains us too tightly when we want to step outside the bounds of programming. The world is not homogeneous like the computational model of an elegant programming language. We need to describe many diverse domains, possessing many different characteristics. Some domains are inert, changing only by force of externally applied operations. Some have autonomous behaviour, with events and state changes occurring spontaneously — that is, for reasons that we choose to leave unexplained and unexplored. In some domains nothing ever changes: there is no time dimension at all. We need to describe diverse relationships — especially causal and control relationships — among the parts of one problem frame. And we must also describe causality within a domain, distinguishing events and state changes we regard as spontaneous from those we regard as being caused by mechanisms within the domain.

These descriptive needs cannot be met by a single language. Just as engineers choose the most appropriate material for each physical part of their products, so we must choose the most appropriate language for each description. The idea of a universal formal language, a raw material suitable for every description, is a chimera; and the use of an informal or unsuitable language is a serious obstacle to success.

13.12 Designating Domain Phenomena

The essence of the mathematical approach to description is to forget what our symbols mean in the physical world and consider only their formal significance. But the forgetfulness must be very temporary, lasting only for individual bouts of concentrated reasoning. Both before and after each bout we must be very clear indeed what reality we are talking about. Where

there is a modelling relationship between a machine and a problem domain, we will apply some of our descriptions to both, some to one only, and some to the other. Where there is a non-transparent connection between the machine and its environment, we will need to describe event occurrences in each domain separately. Where a domain phenomenon appears in more than one principal part, in the same or in different problem frames, we will refer to it in more than one description: an action of the CASE tool user on a Workpiece will need to be described in Workpieces-1, in Operations-1, in Worker-1, in Environment-2, and in Requirement-2. In each description we must always be able to say very exactly what we are talking about: as John von Neumann pointed out, there is no sense in being precise if you don't know what you are talking about.

Saying what we are talking about means explicitly *designating* the *phenomena* of interest in the domain, explaining how they may be recognised, and associating them with terms that we can then use in precise descriptions.

Here is an example of a designation:

"At time *t* the count of parts contained in the warehouse bin *b* is *q*" ∇ NetContent(*b,q,t*)

The LHS of the designation is an informal narrative from which the designated phenomenon can be recognised reliably enough for the purposes of the system. The RHS is a formal term — here it is a predicate — that can be used in a description to refer to the designated phenomenon.

A designation is created for some phenomenon of a particular domain, which may be anything in the world that we wish to describe. We apply a description to a domain by associating it explicitly with a set of designations. The designated terms appearing in the description then have the meanings given in the LHSs of their designations. The meaning of the whole description, as associated with the designation set, is bounded by the designated phenomena: it can refer to nothing else.

As well as describing an actual or desired state of affairs in a domain, a description can also define new terms. These new terms do not refer to new phenomena: a description can not create phenomena, but can only describe their relationships. The new terms merely provide convenient ways of referring to expressions involving already designated phenomena of the described domain.

Explicit designation of phenomena has several advantages. Most notably, it allows clear criteria of correctness for descriptions of non-abstract domains. A designation set tells us how to identify the domain phenomena about whose relationships the formal descriptions make their assertions, and hence how to check whether those assertions are true or false. The designated phenomena relate descriptions to domains, just as triangulation points relate Ordnance Survey maps to the terrain they describe. When the designations of terms are merely implicit in their names, or diffused in an informal narrative accompanying the formal descriptions, it is impossible to check the truth of a domain description. If the description appears false, that may be due to errors in our assumed designations: perhaps we mistook one bridge for another and misaligned the map and the terrain. And if the description appears true, we are in no better case.

Another advantage of this phenomenological approach is that it can help to resolve, or at least to mitigate, the difficulty of using many languages in one development. Formal semantics may provide us with radically different models for different languages, and hence lead us to treat those languages as if they were completely incommensurable. The semantics of Z is based on sets; the semantics of JSD is based on regular languages over events: therefore, we may think, there is no possibility of combining the two in a formal or rigorous development. But clearly there is something seriously wrong here. Evidently many of the same real phenomena might be referred to, and the same real relationships among phenomena might be

asserted, in both languages. Any apparent incommensurability reflects only the limitations of the formal semantics, and its inability to capture meaning in the phenomenological sense.

13.13 Structures of Descriptions

Richer problem frames, and the richer descriptive techniques we need to deal with them, lead inevitably to richer structures of descriptions. There will be many more descriptions, related in many ways. In the Environment-Effect frame the Environment description is in the indicative mood, while the Requirement is in the optative mood, although it is about the same phenomena. We can rely on the truth of the Environment description in refining the Requirement, but not vice versa. In modelling we apply one description to different domains by using it with different designation sets. In incremental and partial description we apply different descriptions to one domain by using them with the same designation set. Different designation sets may have designations in common: descriptions used with those designation sets then have intersecting subject matter and their conjunction is potentially non-trivial. A description may be applied directly to a domain by a designation set, or indirectly by referring to terms defined in another description, or by a combination of the two. The definition of terms may itself rely on terms defined in yet another description.

In these and other ways description set structures become elaborate enough to demand systematic control within a development. It is not enough to formalise syntax and semantics of individual descriptions; we must also formalise — or at least systematise — the relationships among descriptions and between descriptions and the domains they describe. These relationships are the links in a multi-dimensional structure of descriptions. They can not be captured in a single linguistic framework. Evidently, richer problem frames, richer descriptive techniques, and richer descriptive structures will need mechanised support. There are simply too many descriptions, and too many relationships, to be managed by hand. Today, most development support tools are still designed to mechanise old and inadequate manual methods. This will not continue long. New tools will not only support ideas already waiting for a practical vehicle; they will also stimulate new ideas about method that today we can not readily imagine.

13.14 Acknowledgements

Many of the ideas put forward here have been explored and put to the test during several years' cooperation with Pamela Zave of AT&T Bell Laboratories, Murray Hill, New Jersey. They have also been discussed on many occasions with Daniel Jackson of Carnegie-Mellon University. Both of them kindly read earlier drafts of this paper and made many helpful comments. Cliff Jones and Jeff Kramer have also helped me. If this paper has virtues some of the credit is theirs; its defects are all my own work.

[This paper appeared as Chapter 13 of *A Classical Mind: Essays in Honour of C. A. R. Hoare*; A. W. Roscoe ed; Prentice-Hall 1994]