The World and the Machine

Michael Jackson
MAJ Consulting Ltd
101 Hamilton Terrace
London NW8 9QX England
jacksonma@attmail.att.com
mj@doc.ic.ac.uk

Abstract

As software developers we are engineers because we make useful machines. We are concerned both with the *world*, in which the machine serves a useful purpose, and with the *machine* itself. The competing demands and attractions of these two concerns must be appropriately balanced. Failure to balance them harms our work. Certainly it must take some of the blame for the gulf between researchers and practitioners in software development.

To achieve proper balance we must sometimes fight against tendencies and inclinations that are deeply ingrained in our customary practices and attitudes in software development. In this paper some aspects of the relationship between the world and the machine are explored; some sources of distortion are identified; and some suggestions are put forward for maintaining a proper balance.

1 Introduction: Engineering and the World

Because software seems to be an intangible intellectual product we can colour it to suit our interests and prejudices. For some people the central product of software development is the computation evoked. For some it is the social consensus achieved in negotiating the specifications. For some it is a mathematical edifice of axioms and theorems. Some people have been pleased to have their programs described as logical poems. Some have advocated literate programming. Some see software as an expression of business policy.

But many people here will surely want to think of software development as a kind of engineering. The most conspicuous dissenters from this view are, presumably, absent from this conference. Yet we do not speak of engineering mathematical theorems, or poems, or works of literature, or business policies. Why, then, should we speak of engineering software?

Software development is engineering because it is concerned to make useful physical devices to serve practical purposes in the world. Software is a description of a machine. We build the machine by describing it and presenting our description to a general-purpose computer that then takes on the attributes and behaviour of the machine we have described. That is why we compare ourselves to aeronautical and electrical and automotive and chemical engineers, and aspire to emulate their enviably well-established repertoires of 'theoretical foundations and practical disciplines'. They too are concerned to make useful physical devices.

The purpose of a machine, which defines its practical value, is located in the world in which the machine is to be installed and used. The value of a word-processing system is to be judged not by examining its software structure or code but by looking at the quality of the documents it produces, and at the ease and comfort and satisfaction it affords its operators. The requirements for an Air Traffic Control system are to be sought in the aeroplanes and the airspace and the runways and the control tower. The success of a theatre reservation system depends on the ease and speed of booking, the efficiency of payment collection, the convenience of dealing with cancellations.

The requirement — that is, the problem — is in the *world*; the *machine* is the solution we construct. The point is trite and obvious. But perhaps we have yet to come to terms with it, to understand it fully, and to act on that understanding.

2 Four Facets of Relationship

The relationship between the world and the machine is not simple. It has several facets, and different facets are reflected with different intensity in different systems. We can recognise at least four facets:

- the *modelling* facet, where the machine simulates the world;

ICSE'95, Seattle, Washington USA

- the *interface* facet, where the world touches the machine physically;
- the *engineering* facet, where the machine acts as an engine of control over the behaviour of the world; and
- the *problem* facet, where the shape of the world and of the problem influences the shape of the machine and of the solution.

2.1 The Modelling Facet

In many systems the machine embodies a *model* or a *simulation* of some part of the world. There are *data models, object models, process models*. The purpose of such a model is to provide efficient and convenient access to information about the world. By capturing states and events of the world and using them to build and maintain the model we provide ourselves with a stored information asset that we can exploit later when information is needed but would be harder or more expensive to acquire directly.

The model can provide the information we need because there are certain common descriptions that are true both of the model itself inside the machine and of the aspects of the world outside that it models. Of course, the descriptions must be differently interpreted when we apply them to the world and when we apply them to the model. If a common description, written using deliberately meaningless identifiers, is:

$$\forall\, x : \mathrm{B}(x) \bullet (\exists!\, y \bullet \mathrm{W}(y) \wedge \mathrm{A}(x,y))$$

then we may interpret it in the world as asserting:

> For each novel *x* there is a unique writer *y* that is the author of the novel.

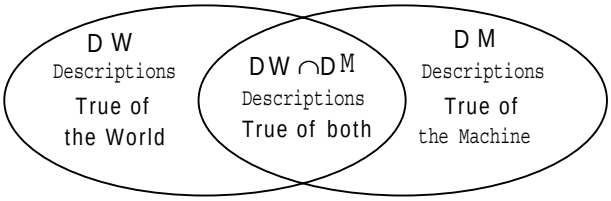
And we may interpret it in the database inside the machine as asserting:

> For each record of type B there is a unique record of type W to which there is a pointer from the B record.

If a mapping is provided between the database model and the world — for example, if each B record contains a character string that is the title of the novel and each A record contains a character string that is the name of the author — then information about the world can be obtained by inspecting the database.

Because the world and the machine are both physical realities and not merely abstractions, the common description captures only a part of the truth about each. For each, there are many other descriptions that might be given. Some novels have more than one author, or are anonymous; writers sometimes use pseudonyms; some novels are linked to others in a series such as Trollope's Barchester novels; some books appear in revised editions. All these aspects of the world may have been ignored in the modelling, and have no reflection in the database. In the database, similarly, records may be deleted to save space, or carefully placed in physical storage to speed access; relations may be in 3rd or 4th or 5th normal form; fields may have null values; there may be backwards pointers and indices. None of these database properties reflects any aspect of the world being modelled.

Considering the set *DW* of all descriptions that assert truths about the world, and the set *DM* of all descriptions that assert truths about the machine, the modelling relationship involves precisely those that are in their intersection:



2.2 The Interface Facet

The problem is not in the machine; and yet the machine can provide the solution to the problem. This is possible, of course, only because there is interaction at an interface between the machine and the world. By 'interaction' I mean the sharing of phenomena. This is not interaction at a distance, by message passing or remote procedure call or writing and reading on a blackboard, but direct participation in common events. The participation is not symmetrical: one party may have the power to initiate the event, and the other may or may not have the power to inhibit it. States may be similarly shared; one party may have the power to change the value of the state, and both may have the power to sense it.

Consider, for example, a system to control a lift. There are sensor switches in the lift shaft at the floors, turned on and off by the arrivals and departures of the lift car. The states of these switches are shared with the machine. When the sensor at floor 3 in the world is *on*, the bit in the machine in the array element *floor_sensors[3]* is set to *1*. This is a shared state, controlled by the world.

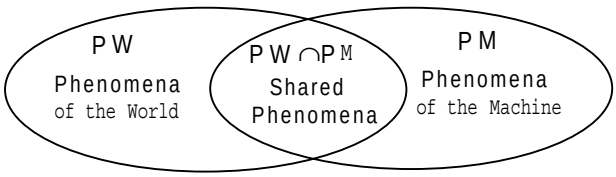
When the upwards call button is pressed at floor 3, this is an event in the world. It is also an event in the machine, where is is observable as the occurrence of an input signal on line *U3*. This is a shared event, controlled by the world.

When the machine emits a signal on its output line *MU*, the polarity of the lift motor is set to upwards; when it emits a signal on its output line *M+*, the motor is switched on. These are shared events, controlled by the machine.

These shared events and states lock the machine and the world in a partnership, sharing the traces of events and states in which they both participate. At a certain level of

abstraction, their behaviours can be described identically. But it is a very abstract level indeed, for at least two reasons.

First, because the shared phenomena are only a subset — typically, a small subset — of the phenomena of the world, and an equally small subset of the phenomena of the machine. If we call the set of phenomena of the world *PW*, and the set of phenomena of the machine *PM*, then the set of shared phenomena is the (relatively small) intersection of these two sets:



Second, because a description that describes the world and the machine identically must necessarily abstract away the control properties. An event controlled by the world has quite different significance from an event controlled by the machine. A description in which this distinction is not made is a very pallid reflection indeed of the reality with which it is concerned. As Lamport pointed out in an account of TLA [Lam89], a stack that leaves the invocation of all *new*, *push* and *pop* operations to the user is very different from — and infinitely preferable to — one that sometimes initiates a *push* or *pop* on its own initiative.

2.3 The Engineering Facet

The recognition of the two intersecting sets of phenomena suggests a systematic usage of those difficult terms: *requirements*, *specifications*, and *programs*.

Requirements are concerned solely with phenomena in the world: that is, with phenomena in the set *PW*. Our customers want us to engineer effects in the world, not in the machine. They want the seats profitably allocated, or the aeroplanes safely controlled, or the documents conveniently edited and neatly displayed and printed.

Programs, by contrast, are concerned solely with the machine phenomena in the set *PM*. Their purpose is to describe those properties and behaviours of the machine that will, ultimately, satisfy the customers.

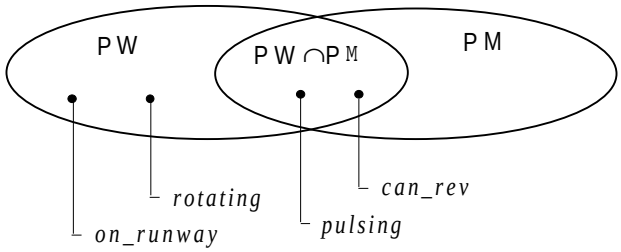
The gap between the two is bridged by specifications. Specifications are concerned solely with the shared phenomena in *PW*∩*PM*. A specification is both a requirement and program. It is a requirement because it is concerned solely with phenomena of the world; and it is a program because it is concerned solely with phenomena of the machine. Naturally, as one might expect, it is satisfactory neither as a requirement nor as a program. It is unsatisfactory as a requirement because it is too limited. The customer's purposes are not confined to the coastline where the world meets the machine, but may range freely over any part of the world that is of interest. And it is not satisfactory as a program because it may not be executable. Programs are descriptions of desired machines, but they must be descriptions of machines that our general-purpose computers can emulate and they must be cast in terms that our computers can interpret.

The specification link is necessary because a specification is a staging post on the hazardous journey from a requirement to a program. Our engineering of the world is completely captured in our refinement of the requirement to a specification. The transition from specification to program concerns only the machine. The first part of the journey, from requirement to specification, can be illustrated by a little example. A requirement in a certain avionics system is to ensure that reverse thrust can be engaged if, and only if, the plane is landing and already on the runway. The requirement is:

REQ: $can\_rev \leftrightarrow on\_runway$

but only the *can_rev* phenomenon is shared with the machine. It is necessary to find a way of connecting *on_runway* with the machine.

Sensors fitted to the landing wheels generate pulses when the wheels are rotating. The state *pulsing* is shared with the machine, although the state *rotating* is not. So these phenomena are available to the engineer:



The developers decide that the following properties hold in the world:

WORLD1: $pulsing \leftrightarrow rotating$

WORLD2: $rotating \leftrightarrow on\_runway$

That is, that the pulses are generated if, and only if, the wheels are rotating; and that the wheels are rotating if, and only if, the plane is landing and on the runway. Relying on these properties they derive the specification:

SPEC: $can\_rev \leftrightarrow pulsing$

That is, reverse thrust can be engaged if, and only if, wheel pulses are being generated. They prove their specification correct by showing that

WORLD1, WORLD2, SPEC $\vdash$ REQ

Unfortunately, property WORLD2 does not in fact hold in the world. On one occasion a plane landed in heavy rain on a runway covered with water. The wheels were aquaplaning, not turning. The pilot was prevented from engag-

ing reverse thrust, and the plane ran off the end of the runway.

The solutions to many development problems — notably, but not only, embedded systems — involve not just engineering *in* the world, but also engineering *of* the world. In this way, software development is like building bridges. The builder must study the geology and soil mechanics of the site, and the traffic both over and under the bridge. The engineering of the bridge is also engineering of its environment. The engineer must understand the properties of the world and manipulate and exploit those properties to achieve the purposes of the system. A computer system, like a bridge, can not be designed in isolation from the world into which it fits and in which it provides the solution to a problem.

2.4 The Problem Facet

The problems we aim to solve by introducing and using computer systems are often complex, and demand careful structuring and decomposition. Because we hope to recognise some rationality in the problem our customer asks us to solve, we expect to be able to structure the problem convincingly and then to base the structure of the solution on the structure of the problem it solves.

But problem structures have proved elusive. It is distressingly difficult to separate discourse about problems from discourse about solutions. The distinction is somehow related to the mysterious distinction between *what* and *how*, or the distinction between the *specification* and the *implementation*.

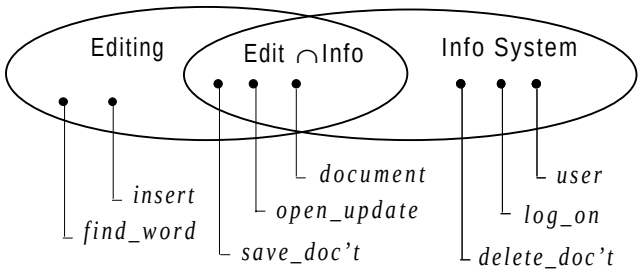
The difficulty arises from the relationship between the machine and the world. The machine will furnish the solution, but the problem is in the world. Discourse about the problem must therefore be discourse about the world and about the requirement that our customer has in the world. Since the world is very multifarious we should expect to find that there are many different kinds of problem. Controlling an elevator is not at all like compiling source programs, which in turn is not at all like switching telephone calls; and none of them is like processing texts in a word processor.

Problems as varied as this can not be effectively structured by naïve approaches that rely on those two broken reeds: hierarchical structure and homogeneous decomposition. As a general rule, neither the world nor the problems it offers exhibit hierarchical structure. Problems usually exhibit a parallel structure, in which the key connective is the logical connective *and*. Nor do they allow homogeneous decomposition. It is certainly possible to devise effective programming environments in which homogeneity reigns: everything is a procedure, or everything is an object, or everything is a sequential process or a recursive function or a list. But the world, and its problems, are infinitely more varied and will fit no such Procrustean bed.

Effective problem decomposition means decomposing into problems that are *recognised* and *known to be soluble*. For example, the problem of constructing a simple CASE tool might be decomposed into these problems, each of a recognisable type:–

- A simple *editing* problem. In an editing problem there is an inert and intangible object — such as a text — belonging to the world but realised within the machine. The operator may request the machine to perform operations on the object, rather as a piece of metal might be worked on a machine tool such as a lathe.
- A *GUI* problem. In a GUI problem there is a user who engages in an assisted dialogue with the machine. The assistance is provided by displays of options currently available and of information that the user will find helpful when choosing an option.
- A simple *information system* problem. In an information system problem there is some reality about which information is desired. For the CASE tool, this may be information about the progress of the work. Information requests are presented to the machine, which embodies a model of the reality of interest and answers them from the model.

The decomposition is parallel, not hierarchical. The different subproblems are concerned with different — but overlapping — subsets of the phenomena of the world. For example, here are some phenomena relevant to the editing problem and the information system problem:



The *insert* operation is relevant only to the editing problem; information about progress is of a coarser grain. Similarly, the *log_on* operation, the individual *users*, and the *delete_doc't* operation are of interest only to the information system problem. The editing ignores any distinction between one user and another, and is not involved in *log_on* or *delete_doc't* operations. However, the *save_doc't* and *open_update* operations and the individual *documents*, are relevant to both subproblems.

3 Four Kinds of Denial

If indeed software development is concerned both with the machine and with the world, as I have suggested, we might still ask whether the world outside the machine is

really our proper concern as software engineers. Let an aeronautical engineer study the relationship between the landing gear and the runway. Automobile engineers are not expected to be expert in route planning, or in human physiology. Electrical power distribution engineers are not expected to know about demographic shifts and the TV schedules that affect patterns of consumption. Why, then, should we as software engineers not confine our attention similarly to the machines that are our artifacts, and leave other people to analyse the world outside the machine and the problem, and to establish the customer's requirements?

Rightly or wrongly, wisely or foolishly, we answered this question long ago. We said that we would do it. With help from domain experts, perhaps; and with qualifications and disclaimers to cover ourselves in case of disaster. But as a community or as a profession we never said 'Problem analysis not our responsibility. We are mere builders of machines to given specifications. We do not judge their fitness for any purpose. Other people must supply the specification, and we will build the machine to meet it'.

In this way we took on the responsibility of dealing with whatever part of the world furnished the context for each particular software development problem. We undertook to concern ourselves not only with the machine, but also with the purposes it is intended to serve. That is why an important department of software engineering is *requirements engineering*: the elicitation, description, and analysis of the requirements that must be satisfied by the system being built. What, exactly, does the customer demand? What, exactly, is the problem? What purposes must the system serve? What functions must it provide?

But we are not at ease with our responsibility. The relationship between the machine and the world creates a conflict. Most developers, for various reasons, are inclined to pay more attention to the machine than to the world. And they have found many ways to manifest and to justify their inclination, and to evade the responsibility that, as a community, we have implicitly undertaken.

3.1 Denial by Prior Knowledge

There is a legitimate kind of denial of the world. In some engineering problems a domain analysis and requirements study is essential. The bridge builder who neglected to survey the terrain and the local geology would be grossly negligent. But for some problems a detailed and careful survey of the problem context, and even of the problem itself, is less important or even quite inappropriate. A group of automobile engineers setting out to develop a five-door hatchback does not begin by asking questions about the purposes to which cars are put, or the physical constitution of the drivers and passengers, or the nature of road surfaces. They do not ask themselves whether the car should be amphibious, or capable of carrying twenty-ton loads, or of travelling at supersonic speeds.

The phrase 'five-door hatchback car' answers those questions implicitly: the requirements are already well-understood and standardised. The design of the product to satisfy those requirements is also well-understood and standardised. The designers need not consider such options as steam power, articulated legs or tracks instead of wheels, or having the driver sitting at the back facing sideways and steering by a tiller. Over a period of a hundred years the customer's needs and the automobile engineer's products have grown into a symbiotic harmony. There is no need for the machine's designers to consider the world and the problem explicitly. The world and the problem will be much as expected, and if the machine is also much as expected it will serve its purpose. For a car designer, explicit attention to the world and the requirement would be 'rethinking the motor car'.

This parallel standardisation of requirements and products is proceeding in many areas of software development too. Magazine reviews of shrink-wrapped word-processors and spreadsheets and databases and graphics packages and compilers and accounting systems read more and more like magazine reviews of cars. They apply standard criteria to measure the products against well-understood needs and against the competition. If you can drive one word-processor you expect to be able to jump into any other word-processor and drive it the same way. This is not just standardisation of user interfaces: it is standardisation of problems and solutions.

As a problem class progresses towards this standardised state, it becomes increasingly legitimate for the developers of solutions to ignore the world and concentrate on the details of their machine. The world and the problem are already well understood, and the knowledge and understanding are embodied in the standard design from which they will derive their solution by an almost imperceptibly small perturbation.

3.2 Denial by Hacking

Sometimes the reasons for denying the importance of the world are more personal. The phenomenon of *hacking* — not in the sense of breaking into other people's systems but in the sense of obsessive devotion to interacting with computers — is well known. It is not surprising, because computers are obsessively interesting things. There are few other things in human life that put so much power into your hands, the opportunity to create a Golem and to enjoy immediately the pleasure of admiring the elaborate and intricate functioning of your creation. Who, faced with such opportunities, would want to waste time on problem statements and domain descriptions and analyses?

The fascination is not confined to computer hackers. All people who work at creating physical artifacts do so

because they are in love with those artifacts. Their creations give them a huge satisfaction with which little else can compare. When Isambard Kingdom Brunel, the great builder of railways and steamships, was dying, he begged to be taken to see the new Royal Albert Bridge at Saltash, one of his most brilliant and noble creations. He did not ask to be taken into his office to see the drawings. He did not ask his assistant to remind him of the stress calculations, or to bring him his slide rule. He wanted to see the bridge.

This fascination with the machine has a long history in software development. We software developers have always offered our customers representations of the machine in place of the descriptions and analyses of their worlds and problems that they really needed. We did so when we offered flowcharts and tape record layouts;we did so when we offered structured pseudocode; and we still do it today when we offer object models and data flow diagrams and Z schemas.

3.3 Denial by Abstraction

In the dimension we are considering here, formalists are closer to hackers than they may care to admit. The machine can be seen as a Protean symbol-processing device, taking as many forms as we care to invent formalisms: the Universal Turing Machine can mimic any Turing Machine, including another Universal Turing Machine. So the machine, viewed abstractly and mathematically, is an inexhaustible source of mathematical delight, and understandably so. There is no need to be concerned with the world. Our product is computations, and computations are mathematical objects.

But mathematics is no more the essence of software development than it is of bridge building. Hermann Weyl, quoted by Abelson and Sussman [Abel85], wrote:

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

As an expression of one important intellectual strategy this is admirable. As a rule of life for a software developer it is catastrophic. The software developer should sometimes forget what the symbols stand for, but only occasionally, and then only briefly. The world and its problems are rich and informal, and large mathematical abstractions rarely capture the important concerns.

Unfortunately, much writing and teaching on the subject of software development inculcates a disdain for the inconveniently messy real world. Courses and books need small problems that provide neatly circumscribed class exercises. If you see software development as a discipline of mathematical calculi and symbol manipulation, you will naturally seek out problems with a clean and easy formulation, purged of inconvenient informality.

So students of software development learn implicitly that typical programming problems are GCD, Eight Queens, Towers of Hanoi, and other traditional pearls. It's impossible not to be reminded of the tale of the prison visitor. The visitor, taking lunch with the prisoners, was surprised to hear one of them shout out 'Joke Number 43'. Everyone laughed. A little later another called out 'Joke Number 16' and everyone laughed again. They had reduced their jokes, by long repetition, to a standard repertoire that could be evoked by merely mentioning their numbers. GCD is simply Joke Number 1.

The implicit lesson is powerful, and harmful. It is that software development *problems* can be captured in a few words, and that all the difficulty lies in devising a *solution*. The problem itself, and therefore its context, merit no serious attention. The student learns to be impatient of the world in which the problem is found, hurrying through the tedious business of eliciting the problem from those stupid and mathematically uneducated people known as users and customers, so that the real work, the enjoyable work, of software design and programming can begin.

Martin Gardner, in one of his books of puzzles, gives an example of a kind of puzzle everyone knows well. A traveller is in a distant land where there are two kinds of people: one always lies and the other always tells the truth. The traveller meets two people, and asks one 'Are you a truthteller?' The reply is 'goom'. The other person says 'He said Yes, but he is lying'. The traveller must decide: Is the first person a truthteller?

Gardner reports that one reader produced an unusual solution. The first person clearly does not speak English, and must have said something like 'Sorry, I don't understand'. Therefore the second person is a liar. Therefore the first person is a truthteller. The reader who produced this solution had clearly not taken a course in logic. She failed to make the standard abstraction. She failed to recognise that this is Problem Number 87. But if I did go to that distant land I would feel safer with her as a companion.

3.4 Denial by Vagueness

There is another subtle, and widely practised, way of avoiding the task of describing and understanding the world. Write descriptions of the machine but imply vaguely that they are actually descriptions of the world. Readers may be sufficiently confused not to notice. This technique is practised both developers of every stamp. Almost every book about a structured or object-oriented development method promises to 'analyse the problem' or to 'describe the real world', and immediately offers a description of the

internal workings of the machine. Formalists do the same. Look at this extract from the preamble to a Z specification example:

> "… the Z approach is to construct a specification document which consists of a judicious mix of informal prose with precise mathematical statements. … the in formal text … can be consulted to find out what aspects of the real world are being described … . The formal text on the other hand provides the precise definition of the system and hence can be used to resolve any ambiguities present in the informal text."

The book is a fine book; the example specification is a fine specification. But evidently the writer is quite unsure whether the document describes the 'real world' or the 'system' — that is, the machine.

This vagueness is possible for a number of reasons. The most cogent is the modelling facet of the relationship between the machine and the world. If the machine embodies a model of the world, then surely one description will do for both. But of course it won't, as we have already seen. There is plenty to say about the world that can not be said of the machine, and plenty to say about the machine that can not be said of the world.

(Sadly, there was a moment when an understanding of modelling was nearly captured and disseminated to the software development community, but the opportunity was missed and the butterfly escaped the net. The Codasyl committee on database systems recognised thirty years ago that the implementation details of a database did not reflect anything in the world being modelled. Two descriptions, at least, were necessary. The committee could have called them the *machine schema* and the *world schema*, and so written their names in the golden book of those who benefited the human race. Instead, alas, they called them the *physical schema* and the *conceptual schema*. What a mistake. What a shame.)

Further, modelling is a less ubiquitous facet of the relationship than many developers seem to suppose. It is only in information systems that modelling — in the sense I am using it — is a central concern. Much of what we do does not involve modelling at all. For example, the avionics system needs no *model* of the world properties that connect the wheel pulses, the wheel rotation, and the plane's position in relation to the runway. It needs a careful examination and description of those properties; and their description may play a role in the reasoning that justifies the eventual specification and implementation of the software. But there will be no part of the machine that simulates those properties. Not everything in software development is modelling in this sense.

4 Four Principles for Description

Traditionally, I am claiming, we pay too little attention to the world in which our problems are found. In software development, paying attention to a subject must mean describing it carefully and precisely, for description is the medium in which software developers fashion their work.

But describing the world is difficult, and our reluctance to pay it due attention is easy to understand. Four principles are suggested here that can help us to avoid some of the difficulties I have mentioned. They are:

- von Neumann's principle;
- the principle of reductionism;
- the Shanley principle; and
- Montaigne's principle.

4.1 von Neumann's Principle

In The Theory of Games [vonN44], John von Neumann and Oskar Morgenstern wrote:

> "There is no point in using exact methods where there is no clarity in the concepts and issues to which they are to be applied."

Our very first obligation is to clarify the concepts and issues with which a system is concerned.

This means that we must begin by establishing the vocabulary of ground terms that we will use in talking about the world and the machine. We must identify the phenomena of interest, give a rule by which each kind of phenomenon can be reliably recognised, and give the formal term by which we will refer to it in our descriptions. If we want to assert that:

> For each novel *x* there is a unique writer *y* that is the author of the novel.

then we had better say, and say precisely, what we mean by '*x* is a novel', what we mean by '*y* is a writer', and what we mean by '*y* is the author of *x*'. This is not an easy task, because in essence it is the task of formalising a part of the intransigently informal world. For each term we must give a — necessarily informal — *recognition rule* by which the phenomenon we are referring to can be recognised in the world. And we must also give the *formal term* — for example, a predicate symbol and formal argument list — by which we will refer to it in our descriptions. The *recognition rule* and *formal term* together constitute a *designation*.

The task is possible only because it is bounded in two ways. The first bound limits our subject area: we are not obliged to formalise the whole world, even the whole of those parts of the world in which books and writers are to be found. We are concerned, perhaps, only with English novels of the nineteenth century, and only with those that were published and offered for sale to the public.

The second bound limits the requirements of the system we are building, and so limits the aspects of our already bounded subject area that will concern us. One system will be concerned with the literary aspects; another

with the commercial relationships between authors and publishers; another with the social effects of the novels and the distribution of their readers among social classes and geographic areas; another with the technology of book production. There is no bound to the number of such aspects that might be of interest, but only one or two can be of interest in a particular system. It is not possible to have a system about '*absolutely everything* to do with English novels of the nineteenth century'. That is why the efforts to create enterprise models have failed. They were systems about '*absolutely everything* to do with the ABC Company'.

By writing explicit *designations* and defining our ground terms precisely we give meaning to our descriptions in the most important sense. Formal semantics gives meaning only in a formal sense: the abstract formal text is explained in terms of the abstract semantic domain. We give practical meaning to our descriptions by grounding the formal text also in terms of the informal reality in the world that it describes. Explicit designations make our descriptions *refutable*, and deprive us of the evasion of saying 'Well, it all depends on what you mean by *novel* and what you mean by *writer*'. Without this grounding, formal precision has no place to stand and can not move the world.

In a crisper conversational version of his principle, von Neumann said more simply:

> "There is no sense in being precise when you don't know what you are talking about."

4.2 The Principle of Reductionism

Because we are talking about *phenomena*, about what *appears* to us to exist or to be the case, we often have considerable freedom to choose our ground terms. We should always choose those phenomena for which we can give the most exact and reliable recognition rules. Often this will involve applying a reductionist principle, choosing the simplest possible phenomena and — where appropriate — defining more elaborate constructs in terms of them.

One error to be avoided at all costs is the unthinking adoption of English language nouns as denoting entity classes or set-membership predicates. In a library administration system it seems obviously right to choose *member* as a ground term; in a telephone switching system to choose *call*; in a meeting scheduling system to choose *meeting*; in an airline reservation system to choose *flight*. But almost certainly these choices are serious errors.

In the library system, being a *member* is a state of an individual who has *enrolled* in the library and has not yet *resigned* or *lapsed* or been *expelled*. The events *enrol*, *resign*, *lapse*, and *expel* are probably appropriate ground terms — that is, designated phenomena. By contrast, *member* is not a ground term: it should be defined in terms of the events. The definition of *member* is not, of course, a *refutable description*. It says nothing about the world of the library, but is merely a statement about how the term will be used in descriptions. Any assertion containing the term *member* can be translated into an equivalent assertion about the defining events.

Telephone *calls*, scheduled *meetings*, and airline *flights* are even less appropriate as ground terms than library *members*. It is impossible to write reliable recognition rules for them. What, exactly, is a telephone call in a world where there are chat lines, conference calls, and call forwarding? Suppose that *A* calls *B*, and the call is forwarded to *C*. *C* then links in *D* using the conference feature, and, after a while, *C* drops out, leaving *A* talking to *D*. How many calls is that? Airline flights may be amalgamated, so that the person sitting in the seat next to you is on a different flight; and they may be split, so that one flight involves changing planes and sometimes even airlines. Whatever recognition rule you write will be inadequate to your purpose.

The ground terms you should be concerned with are, as often happens, events. Picking up a telephone handset, replacing it, dialling a digit, starting to receive a busy tone — all these are readily recognisable phenomena. So too are take-off and landing events in the life of an aeroplane, and boarding and disembarking events in the life of a passenger. These should be your ground terms. If you can reconstruct *calls* and *flights* by definitions using these ground terms, well and good. If not, you would only have been deceiving and confusing yourself and your customer by trying to treat them as ground terms directly.

4.3 Shanley's Principle

Twenty years ago, in a famous paper on Structured Programming with **go to** statements [Knuth74], Knuth quoted Pierre Arnoul de Marneffe [deMar73]:

> "... If you make a cross-section of, for instance, the German V-2 [rocket], you find external skin, structural rods, tank wall, etc. If you cut across the Saturn-B moon rocket, you find only an external skin which is at the same time a structural component and the tank wall. Rocketry engineers have used the 'Shanley Principle' thoroughly when they use the fuel pressure inside the tank to improve the rigidity of the external skin!"

de Marneffe cited the Shanley principle as a rule for efficient design. Barry Boehm has pointed out that it has disadvantages: it leads to designs with a single point of failure. And one could argue that the Shanley Principle is the direct negation of the separation of concerns. The thrust of much of the advance in programming languages and operating systems, certainly, has been towards separation of functions rather than their amalgamation. An op-

erating system that can execute and synchronise many concurrent processes relieves the programmer of the task of composing them into a single sequential process.

But we are concerned here not with the design of our machines but with the design of the world. We must recognise that the architecture of the world has been designed with the fullest possible application of the Shanley Principle. A new book [Gam94] on patterns in object-orientation ends with a provocative quotation from the software developer's favourite architect, Christopher Alexander [Alex79]:

> "It is possible to make buildings by stringing together patterns, in a rather loose way. A building made like this is an assembly of patterns. It is not dense. It is not profound. But it is also possible to put patterns together in such a way that many patterns overlap in the same physical space: the building is very dense; it has many meanings captured in a small space; and through this density it becomes profound."

The world is profound, in Alexander's sense. And the profundity reaches down to the elementary individuals. Every part of the world may play many roles, and perform many functions. Every individual may be an individual of many distinct domains; every node may be a node in many graphs; every element may be an element in many sets; every event an event in many traces.

This versatility and many-sidedness, both of larger structures and of elementary individuals, must be reflected in an appropriate approach to describing and understanding the world. At the level of domain description and problem analysis it demands parallel structuring of views and problems. The parallel decomposition of a colour picture into Cyan, Magenta, Yellow and Black colour separations is a far better metaphor for structuring than the hierarchical bill of materials assembly structures that have been the staple fare of elementary problem solving for far too long. The invention of the subroutine was not an unmixed blessing.

At the elementary level we must recognise similarly that the world is not strongly typed. It is always possible to devise a very restricted view of the world in which elementary individuals can be classified into disjoint sets and strongly typed. But such a view is always far too restricted to capture a problem of serious interest, and many such views must be adopted simultaneously. The elementary individuals in one view then appear, differently classified and differently typed, in other views. The need for multiple viewpoints is felt at the elementary level too.

4.4 Montaigne's Principle

The great sixteenth-century French essayist Montaigne wrote: 'The greater part of this world's troubles are due to questions of grammar.' Perhaps there is a degree of exaggeration here, but there is at least one question of grammar that is of the greatest importance for software developers. That is the distinction between the *indicative mood* and the *optative mood*. The indicative mood expresses what we *assert to be true*; the optative mood expresses what we *desire to be true*.

For the developers of the system to control reverse thrust the statement:

REQ: $can\_rev \leftrightarrow on\_runway$

is in the optative mood. It expresses what they desire to be true, the effect that their system is to bring about in the world. But the statement:

WORLD1: $pulsing \leftrightarrow rotating$

is in the indicative mood. It expresses what they assert to be true of the world, regardless of the behaviour of the system they are building.

The distinction is important and must be clearly made in the descriptions we write in a development. In recognition of this need, a rather confused formulation is sometimes demanded of US Government contractors:

> "Absolute tense 'shall': a binding, measurable requirement … observable when a system is delivered … in terms of an … output.
> "Future tense 'will': a reference to the future, … describing something … not under control of the system being specified.
> "Present tense: for all other verbs … in all other cases."

Part of the confusion is in the grammar. Tenses are not moods. And I remember learning at school that:

> "I shall drown. No-one will save me"

is a desperate cry for help, while:

> "I will drown. No-one shall save me"

is the proud proclamation of a determined suicide.

Natural English usage is not easily tamed, and it is a bad idea to rely on the vagaries of English verb forms to capture crucial distinctions in technical documents.

A better approach is to avoid grammatical distinctions of mood within a single description, and to indicate the mood of a description by its place in the whole development structure. One virtue of this approach is that the mood of a description is, in fact, relative. The requirement, the properties with which we want our system to endue the world, is in the optative mood. But when the system is successfully installed and operating, the requirement becomes a reality; what we desired to be true becomes true; and the optative becomes indicative. So mood is relative to the progress of the development.

It is also relative to problem decomposition. In the editing subproblem in the CASE tool, correct performance of the requested operations on the edited texts is a requirement, in the optative mood. In the information system subproblem, we assume that the editing subproblem has been solved. Correct performance of the requested operations then becomes indicative: it is regarded as a given

truth about the world.

One penalty for ignoring the distinction between indicative and optative is the confusion often felt by readers of formal specifications. Some formal specifications abstract from the distinction between the different moods. The formal specification is a homogeneous description of the behaviour of the correctly implemented machine interacting with the world. Behavioural properties attributable to indicative truths about the world are not distinguished from properties attributable to the satisfaction by the machine of the customer's optative requirements. A predicate *P* is given as the precondition on an operation *O*, but we are not told whether:

- the machine will inhibit *O* if *P* does not hold; or
- the world ought to refrain from invoking *O* when *P* does not hold; or
- the world is known never to invoke *O* when *P* does not hold.

Abstraction from the indicative/optative distinction may be useful for a number of purposes. But it is painfully confusing and frustrating to the reader who wants to draw any practical inference whatsoever from the specification.

5 Envoi

Some of what I have said may imply a hostility to formalism and to mathematical approaches. Nothing could be further from the truth. We need to make descriptions that are clear and precise, and we need to reason about them. And clarity, precision and reasoning are the business of mathematics.

Mathematics is the Queen, and the Servant, of the sciences. It has served physics and engineering well. It can serve software development well, too, if we make sure that we know what we are talking about.

References

[Abel85] Harold Abelson and Gerald Jay Sussman with Julie Sussman; Structure and Interpretation of Computer Programs; MIT Press, 1985.

[Alex79] Christopher Alexander; The Timeless Way of Building; Oxford University Press, 1979.

[deMar73] Pierre-Arnoul de Marneffe; Holon programming: A survey; Université de Liège, Service Informatique, 1973. Quoted in [Knuth74].

[Gam94] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides; Design Patterns: Elements of Reusable Object-Oriented Software; Addison-Wesley 1994.

[Knuth74] Donald E Knuth; Structured Programming with **go to** Statements; ACM Computing Surveys Volume 6 Number 4 pages 261-301, December 1974.

[Lam89] Leslie Lamport; A Simple Approach to Specifying Concurrent Systems; Comm ACM Volume 32 Number 1 pages 32-45, January 1989.

[vonN44] John von Neumann and Oskar Morgenstern; Theory of Games and Economic Behaviour; Princeton University Press, 1944.

[Weyl] Hermann Weyl; The Mathematical Way of Thinking. Princeton University Press.