Proceedings of NordData-91, pages 31-45, Oslo, 16-19 June, 1991

Technology: Master or Servant?

Michael Jackson
Software Development Consultant
101 Hamilton Terrace
London NW8 9QY
England
Tel: +44 71 286 1814
Fax: +44 71 266 2645

Abstract
Enthusiasts for technology, especially enthusiasts for so-called Artificial Intelligence, believe that execution of computer programs will be able to replace human thought, and that this is a prospect to be welcomed. They are wrong on many — but especially on intellectual — grounds: existing and foreseeable techniques of software development are fundamentally inadequate for the task. Even when great improvements have been made, technology will remain a very imperfect servant: we must certainly not allow it to become our master.

1 Formal and Informal Worlds

The impact of technology has been growing since the industrial revolution began in the early 18th century. From the beginning it has influenced how we work and how we think by changing the circumstances of our lives. The world of a pre-industrial weaver in a cottage was quite different from the world of a worker who tends a steam-driven loom. The world of a car owner living near a motorway is quite different from the world of a villager in a remote village before the age of turnpike roads.

Computers and IT have an even greater impact than earlier inventions. Almost everyone is involved, at least in the developed countries, if only because they must deal with organisations such as electricity suppliers that rely on IT to administer their dealings with the public. And for some people, interacting with computers as programmers or users makes a much more intimate relationship with the technology than flying in an aeroplane or watching a television set.

Just as we recognise — or ought to recognise — that pedestrians on roads must be isolated and protected from hurtling cars and trucks and buses, so we ought to recognise that human beings need protection from computer systems. But in the case of computers the protection must be intellectual and emotional, not physical. People need protection from computer systems because all human things are organic and informal, while all computer systems are symbolic and formal. Computers and IT systems are essentially inhuman: at bottom the computer is just a machine for manipulating symbols; but human beings are not machines and human thought is not symbol manipulation.

The difference between the formal and the informal is to do with a certain kind of finiteness. Given a Prolog program, perhaps one that provides assistance in medical diagnosis, we can reasonably ask: how many pieces of ‘knowledge’ does this program have? The answer can be obtained simply by counting the ‘facts’ and ‘goals’ embodied in the program text. Similarly we can count the nodes and arcs in a semantic net, or in a relational database schema, and use these numbers to answer questions about how much the program ‘knows’.

But we can not ask the same questions of a human being: ‘how many things do you know?’ is a silly question. It is a silly question because human knowledge can not be formalised in this kind of way. Try the following experiment. Close your eyes, then open them and look around you for ten seconds or so. Then close your eyes again. Now answer these questions: how many things did you see? how many properties of each thing did you observe? how many facts did you learn from what you saw? These questions are unanswerable because when your eyes were open you had a certain kind of looking and seeing experience — a different experience for each one of you; this experience is grounded in our senses and in the physical world, and is not reducible to any list of formal symbols representing facts or objects. Human experience can not be fully described as the ingestion of formal truth values over a set of symbols; human thought is not symbol processing.

2 What's New About IT

Interaction with formal systems is not a new experience for the human race. For thousands of years governments have imposed legal and administrative systems on their subjects, and these systems are in many respects formal. We recognise their formality by saying that 'the law is an ass', by joking and complaining about the stupidity of bureaucrats, by pointing to Catch 22 in the regulations: such formal systems can never deal adequately with the infinite variety of human experience.

But the follies of legal and administrative systems are mitigated by the fact that they are operated by human beings. Reasonable lawyers and administrators 'bend the rules': they find ways around the idiocies that must inevitably arise when a fixed set of formal rules is applied to an infinitely rich domain. Further, the inherent rigidity of the system is softened by the use of natural language in framing its rules: usually the meaning of some crucial word can be stretched to mean what it needs to mean if the result is to be reasonable. And in the last resort, if one lawyer or administrator is unreasonable we can take our case to a higher authority who is likely to be less rigid in applying the rules.

In computer systems, of course, there is no such escape clause and there is no human agency: the computer grinds away at its symbol manipulation to produce an inexorable result unaffected by common sense, compassion, or intelligence. This complete automation of formality is a new phenomenon made possible by IT. Its ill effects can be spread very far and wide by the ability of the machines to multiply their work almost without limit. The individual administrator can browbeat and frustrate only a dozen citizens in each hour, and his colleague may be more decent or sensible: but the same program can run on a thousand machines, and each machine can perform a thousand transactions an hour.

Another important new factor is the ability of computer systems to deceive. Nearly thirty years ago Weizenbaum created his Eliza program. The program carries on a dialogue with a human being in what appears to be natural language, playing a part determined by the choice of one of its 'scripts'. The most famous script is the Rogerian psychotherapist, who engages tirelessly in apparently soothing exchanges like:

"What is your name?"
"My name is John."
"Hallo John. What would you like to talk to me about?"
"I would like to talk about my father."
"Why would you like to talk about your father?"

Behind the apparently human intelligence is a relatively simple text manipulation algorithm that can play back what the 'patient' types into the computer having made a few simple but convincing syntactic changes from first to second person, and from statement to question.

The sad lesson of Eliza was that most of the 'patients' were deceived. They wanted to be deceived; even after the program had been explained to them, they acted as if they believed that a human psychotherapist was behind a screen, typing back to them the answers that in fact were concocted by the program's simple algorithm. once communication has been reduced to the passing of symbols, and this reduction has been accepted, then in a carefully defined context it becomes possible for the programmer to succeed in deceiving the computer user, and for the user to cooperate in the deception.

This reduction to purely symbolic communication can take place in many fields. The texture, smell, oiliness, consistency, and shading of real oil paints can be reduced to 'choosing from a palette of 2**24 colours'. The subtleties of literary criticism can be reduced to the banalities of style checking programs. A consultation with your doctor can be reduced to an interactive session with an expert system based on 11,185 facts and 6,934 diagnostic rules.

Of course, this is not to say that paint programs, word processors, and expert systems to aid medical diagnosis are not often valuable; sometimes they outperform the 'real thing' in some particular task. But we should always be aware of how little truly lies behind the VDU screen, and of the dehumanising effect of accustoming ourselves to interact with a computer as if it were a person.

3 System Development: The Past

To create a computer-based system we have to bridge the gap between the infinitely varied informal human world and the narrow, limited, world of some symbolic formalism. We have to reduce our description of the world to recursive function definitions, or to a set of Horn clauses, or to a set of relations in Boyce-Codd Normal Form, or to a set of objects arranged in a classification tree and communicating by message passing, or to a set of algebraically defined abstract data types.

This reduction is limiting for the reasons I discussed briefly above: the informality of the real world can not be adequately captured, except for very simple purposes, by such formalisms. Even the attempt to define a term such as 'customer' or 'employee' in a data processing system is likely to prove too difficult. But after we have understood and accepted this limitation arising from the difference between the formal and the informal, there is still another level of difficulty that must concern us very directly if we want to understand software and its development.

As human beings looking at the world we adopt many different points of view, simultaneously and at different times. One decision is likely to have many different dimensions — political, social, personal, financial, emotional — and we must see it from all these points of view using appropriate language for each. This simultaneous adoption of many languages and many models is quite beyond our present technology of software development, and intensifies our inability to do our job properly.

Broadly speaking, we are confined to development techniques in which we are expected to describe only one thing: usually the system itself; and to use only one language the specification language favoured by our chosen development method. One method sees the world as a network of communicating sequential processes; another as a universe of true statements in the predicate calculus; one as a state machine with operations; another as a structure of entities and relations.

The weakness here is not that these views are totally unsuitable (although, of course, they are all pure formalisms and therefore limited by that characteristic). On the contrary, each formal view has its proponents, and each has its entirely convincing examples of appropriate use. Are you dealing with a queue? Then you will certainly wish to adopt the abstract data type view.

Are you dealing with a chocolate vending machine? Then you will certainly need sequential processes to describe it. Are you dealing with suppliers who supply parts to projects? Then a relational data model is what you need. Do you have to determine whether Mary loves John and whether Lucy and Mary are sisters? Then you will not succeed without logic programming. But unfortunately real problems tend to be about queues of suppliers who supply vending machines to be used by young lovers: you will need all the formalisms, but today's development methods are likely to force you to choose only one.

It is exactly as if you were setting out to build a motor car, and were forced to choose your material at the outset. If you make the car of rubber, the tyres and carpets, and perhaps the seats too, will be successful, but what about the engine? Make it from cast iron or aluminium and your engine block will be good but you will have difficulty with the windows. There is no way out.

4 System Development: The Future

To overcome these difficulties we need to be able to describe the domains of our systems, the problems our systems solve, and our systems themselves, in many different languages and formalisms: we need to be able to adopt different formalisms for different purposes, different parts, and different aspects of a development.

To some extent, there has already been some movement in this direction. Its most primitive form is the ability to mix programming languages, starting with the facility for calling Fortran subroutines from COBOL programs and progressing to efforts to combine the virtues of Lisp and Prolog, or of

Smalltalk and CSP, or of Smalltalk and C. But we need something that cuts much deeper, something that addresses the problem more generally.

We need to be able to understand and carry out our software developments as projects in which we construct, manipulate, and compose many many different descriptions of many many different things. These descriptions must not be constrained to be expressed in a single formalism any more than the parts of which a motor car is built are constrained to be made from a single material. The key to this possibility is, above all, in the word ‘compose’. We have to build software by putting together or composing descriptions whose variety reflects the variety of the real world they deal with, and we must therefore devote much more effort to understanding how such compositions can be achieved.

The oldest form of composition in software is hierarchical composition, especially of procedures. The invention of the subroutine in 1949 gave birth to a whole culture of procedural abstraction, program structure hierarchies, and top-down and stepwise refinement methods in requirements, analysis, specification, design, and programming. Such hierarchical composition can be very powerful in some cases, but it is very limited: although it may help us to build or to understand one large description, it can do nothing to help us when more than one description is needed. Anyone who has struggled to describe a data processing domain in terms of IMS structures will know that hierarchical structure alone is not enough: in IMS you must also have pointers, which are the IMS expedient for introducing parallel composition in addition to hierarchical.

Parallel composition is the essence of adopting different views of the same thing simultaneously. When a printer prepares colour separations of a picture, the separations are different views that will be composed by superimposition when the picture is printed: the printed picture is the green view superimposed on the red view superimposed on the blue view. To do this successfully in software we need to understand much more than we do now about the relationships between descriptions and what they describe, between languages and what they can say, between the subject matter of software and the software itself.

Getting this understanding and putting it to work in system development is a central task facing us today. So far little or nothing has been achieved, but the scene is being set for the problem to become more widely recognised as relevant and important. Two elements in this scene setting are the technology of CASE tools and repositories, and the problems of reuse.

Until now, CASE technology has concentrated on mechanising the methods of the past. There is ported by CASE tools, but there ought to be. The value of mechanising an activity should not lie merely in making it faster or cheaper: it should lie in introducing new and better ways of doing new and better things. The progress of engineering in many fields has been distinguished by the introduction of new materials and new processes that were not possible without sophisticated tools. You can not cut helical gear wheels by hand, or make plastic extrusions, or form sintered metal parts, or — closest to our own concerns — fabricate VLSI chips: all of these products and techniques came into being because people thought seriously about the new possibilities that mechanisation offered. In the context of CASE tools, the motivation will be supplied by repository technology. There are many vendors of repository products, but very little idea of what might be put into them. The CAIS and PCTE projects considered relationships that might exist in development databases among objects stored there, but gave only the most perfunctory consideration to the question of what those objects might be.

CASE tools and repositories make it possible to develop software by creating and using very large numbers of simple descriptions, just as washing machines and motor cars are ultimately constructed of very large numbers of simple parts. This is exactly what is needed if we are to escape from the straitjacket of the single description, single language, assumptions of our past.

A similar impetus may come from the desire to reuse the products of software development. Certainly mathematical subroutine libraries are widely used; certainly every developer in Smalltalk uses the existing classes for collections and windows and menus. But there are still thousands of programmers programming linear search and sequential file update and depth-first search and multi-level reporting and totalling. Worse still, existing COBOL systems embody the detail of business procedures that are only dimly understood and can be neither discarded nor reused in a new system. The pressure is constantly growing as the old systems become more and more fragile and less and less able to support the needs of the business. Eventually we will be forced to make serious efforts to understand how software combines the many different descriptions it embodies, and how those descriptions can be extracted from old COBOL code or reused in the development of new software.

5 The Limitations of Technology

Even when software development technique has advanced far beyond what it is today our systems will still suffer in principle from today's disadvantages and limitations. They will still be symbol manipulation systems, whose manipulations are not grounded in any experience of the physical and the human worlds; they will still be constrained by their formalisms, even if they can embody several formalisms at once.

I believe that our most important responsibility in this area is to ensure that we do not allow ourselves, or other workers in software, to claim too much for our systems, or to present them as if they were something they are not. Everyone is annoyed — although some are also amused — by the absurdities of computer-produced personalised letters. "We are so pleased, Mr Jackson, that you and your family at 101 Hamilton Terrace have been personally selected to receive this exclusive free copy of Great Books of the World, and we are sure that you will be delighted by this opportunity." Well, no, actually. We were not personally selected: our name and address were extracted from a list by some randomising algorithm, or, worse, by some mechanised rules for choosing the most suggestible victims. There is no-one who is 'so pleased': probably no-one except us yet knows that it has happened. This opportunity is anything but exclusive. And we certainly do not want to receive letters from laser printers purporting to be typed by hand.

I believe that we would do well to legislate against this kind of masquerade. I am not sure what form the laws might take, but one element might be a requirement that letters produced by computer must carry a clear statement of the number of letters produced in the same run. An exclusive opportunity of a personal selection is less deceptive when the letter states clearly that is one of 6000 similar letters.

Some years ago there was considerable debate in Britain on the role and prospects for Artificial Intelligence. One influential view was that there were essentially three approaches to Artificial Intelligence.

The first approach was to take well-defined tasks that could usefully be automated: spot-welding car bodies, or using large amounts of geological information to estimate where oil might most likely be found. In this approach, no claim is made that the computer is 'thinking': the criterion of success is simply how well the task is performed. It seems to me that the proponents of this approach had a very sound understanding of the role of computers and of computer-based systems.

The second approach was to take some activity, hitherto recognised as human, and to simulate human activity as accurately as possible. Preferably, the activity should be generally accepted to be a form of 'thinking', so that success would lend weight to the view that machines can think. Efforts at machine translation and real-time speech recognition fall into this category: so too, perhaps, do the far more successful efforts at playing chess. The proponents of this approach are to be treated with suspicion: their hidden agenda is to convince us that computers can think, and — the other side of the same coin — that we are only machines made of meat.

The third approach, the most radical, is to formulate a definite theory of some part or aspect of human activity, and to embody that theory in a machine realised as a computer program. If the machine can perform the activity convincingly, then that success is considered to lend weight to the theory. For this school it might be appropriate to claim with hindsight that the activity of psychotherapy can be appropriately described by a listing of the Eliza program. It would be appropriate also, presumably, to claim similarly that analysis of the Airbus will reveal how an eagle flies.

This third school is no danger to humanity. Their ideas are too patently ill-founded for us to worry that they might be accepted. The first school are simply practical people solving problems in the established engineering tradition. It is the second school that is most dangerous, that influences the attitudes of many people building administrative and data processing systems, and threatens to make technology our master. We can resist them by insisting always on recognising the inevitable limitations of our techniques, and on remembering the difference between organisms and machines, between the informal and the formal, and between people and software.