Problems, Descriptions, and Objects

A Keynote Address at OOIS'94: the 1994 International Conference
on Object-Oriented Information Systems • 21st December 1994

by Michael Jackson 101 Hamilton Terrace • London NW8 9QX
0171 286 1814 (voice) • 0171 266 2645 (fax)
jacksonma@attmail.com • mj@doc.ic.ac.uk

Abstract

The business of software development is solving problems. Following Polya, we can increase our ability to solve problems by focusing directly on problems themselves, on their parts and structures, and on the relationship between problem and solution method. This leads to an emphasis on describing the world outside the computer, and reasoning about it explicitly; to an approach to problem complexity and decomposition; and to a particular view of the proper role of object-orientation in software development.

1 Problems and Solutions

Software development projects, in my view, are concerned to solve engineering problems. Software developers make machines that are to be installed in the world, and are to make a difference in the world by interacting with it. For example, a word-processing system is a machine installed in an office, just like a typewriter, but much more powerful and versatile. As software developers we don't construct the physical fabric of the machine; we only describe the properties and behaviour of the machine we want. We present our descriptions to a general-purpose computer, which then magically takes on the properties and behaviour of the machine we have described. It is in this way that a software development problem is an engineering problem — the problem of creating a useful machine to fit some purpose.

People who are interested in the social, economic, ethical, and political aspects of software development — especially of the development of administrative or information systems — sometimes disagree with this view. Software systems, they say, are situated in a human context, and their specifications emerge and evolve by processes of continual

negotiation and adapation among all the interested parties. There is no succession of well-defined problems for the software developer to solve: the primary development activity is ongoing social interaction. Of course there is truth in this view for some developments in some situations. But at the end of the day, even in those situations, there are programs to be created: that is, there are machines to be built, and engineering problems to be solved.

But although we are engineers we are not, for the most part, like other engineers. Most of the established branches of engineering — civil, automobile, electrical, chemical, aeronautical — are specialisations. Automobile engineers don't turn their hands to designing bridges or chemical plants. Software engineers, in contrast, are usually generalists rather than specialists. Except for courses in compiler construction and in operating systems their education is in general principles and techniques rather than in specific problem areas. And most software development practitioners would think of themselves, with some justification, as being equally able to work in one problem area as in another.

This difference is of great importance. An automobile designer does not begin by trying to discover what the problem is. The problem is, as always, to design a particular kind of car. Both the purpose and the nature of each kind of car are very well understood. The designer of a family saloon car need not consider whether the car should be able to fly; or to carry five-ton loads; or should incorporate a crane for lifting steel girders. Nor need the designer consider whether the car should have wheels or tracks; whether the driver should sit at the front or at the back; whether it should be driven by steam or nuclear power. Both the problem and its solution are very tightly constrained, and the work of the automobile engineer — except for the most brilliantly revolutionary designers — is to make very small perturbations within the given constraints.

In bespoke software development, except for a few specialised areas, the situation is quite different. Every problem, and every solution, is new. The freedom of the software developer is several orders of magnitude greater than the freedom of the automobile engineer. This is why *analysis*, and *requirements*, and *specifications*, loom so large on the software development landscape. A large part of our effort must almost always be devoted to determining what the problem is, and to devising a new solution — because it is always a new problem.

Faced with these demands, we have traditionally paid most of our attention to solutions, and little or no attention to problems. Partly, this is because of the seductive attraction of computer programming. Many developers are programmers at heart, whatever their job titles may be; their happiest hours are those spent devising how their machines should work. Those who claim to be entirely concerned with their customer's problem often try to express that problem in a dataflow diagram, or some other representation of the internal behaviour of their planned machine. As a representation of the problem they offer a design for a solution. More damagingly, even resolute methodologists find it hard to pay serious attention to problems. Ralph Johnson, a leading proponent of the use of patterns in object-oriented development, says [Johnson 94]:

> "We have a tendency to focus on the solution, in large part because it is easier to notice a pattern in the systems that we build than it is to see the pattern in the problems we are solving that lead to the patterns in our solutions to them."

Of course it is easier to notice a pattern in the solutions. The solutions are set in the context of a programming language and environment that provide a rich structure and vocabulary for talking about solutions: procedures and functions and parameters and invocations; pipes and streams and processes; objects and methods and classes and instance variables. By contrast, a typical problem is set in a context that offers no such help. In the absence of a suitable vocabulary it is a daunting task to try to speak of problems. Solutions are easier. The sixpence is sought, as ever, under the street light.

2 Problem Frames: Polya

But a good starting point for talking about problems is readily available. Polya shows the way in his monograph *How to Solve It* [Polya 57]. There he expounds the work of the Greek mathematicians, especially Pappus, in the field of heuristics — techniques for finding solutions to problems for which no algorithmic method is known.

The Greeks classified simple mathematical problems into *problems to prove*, and *problems to find*. For example, the problem 'Show that if the sides of a quadrilateral are equal its diagonals bisect each other' is a problem to prove; while the problem 'Given three lengths a, b, and c, find a triangle whose sides are of those lengths' is a problem to find. The different kinds of problem can be recognised by their principal parts and the associated solution task. A problem to find always has:

- the *unknown*: here, a triangle;
- the *data*: here, the three lengths;
- the *condition*: here, that the triangle's sides should be equal to the given three lengths.

The *solution task* in a problem to find is always to find or construct the *unknown* so that it bears the relation to the *data* that is expressed by the *condition*.

Polya gives a number of heuristics for solving problems to find. For example:

> Check that you are using all the data.
> Check that you are using all the condition.
> Split the condition into parts.
> Think of a familiar problem having a similar unknown.

These heuristics can offer useful advice for solving all problems of the class because — and only because — they are expressed in terms of the problem alone. That is, they are expressed in terms of what Polya calls the *principal parts* of the problem, not parts or aspects of any putative solution.

The assemblage of principal parts and solution task that characterises a problem class merits a name. I call it a *problem frame*. I think of it as a kind of structure or jig into which a problem may be fitted so that it can be worked on. By fitting a problem into a problem frame we should be classifying it precisely enough to be able to select an appropriate method for its solution. The key requirement for a good problem frame is that it should be precise enough to give a really good grip on any problem that fits it. And the key requirement for a good method is that it should be associated with, and exploit, a sufficiently precise problem frame.

Polya writes of two problem frames for small mathematical problems. For software development, very many more will evidently be needed. Attempts to find a single general problem frame and an associated method that will work well for all software development problems are doomed to failure. They lead to the discredited and vapid emptiness of top-down decomposition, with its vacuous problem frame. The problem is characterised as — well, a problem. The method is to break it down to sub-problems that are themselves — well, problems; to break these sub-problems down to sub-sub-problems; and to continue until no further

decomposition is necessary. The fact that this prescription fits every imaginable problem is the clearest possible symptom of its lack of efficacy.

3 Software Development Problem Frames and Methods

A useful problem frame must fit its problems tightly. So a symptom of its utility will be that it excludes most problems and fits only relatively few. Here are three examples: the JSD Information System Frame, the Simple Control System Frame, and the Workpieces Frame.

The JSD Information System Frame may be suitable for an information system to be used in a commercial organisation. It has these principal parts:

- The *System*. This is the machine we must build.
- The *Real World*. This is the world about which information is required. It is dynamic, but autonomous. That is, events and state changes take place, but they are to be regarded as spontaneous and unexplained.
- The *Information Outputs*. These are the reports and displays containing the required information.
- The *Information Requests*. These are query transactions and requests for various kinds of information outputs.
- The *Information Function*. This is a relationship between the *Real World* and the *Information Outputs*.

The solution task is to construct the *System* so that it produces the *Information Outputs* in their correct relationship to the *Real World*, in response to the *Information Requests* and to events and states of the *Real World*.

The Simple Control System Frame might be suitable for controlling a device such as a Washing Machine. It has these principal parts:

- The *Controller*. This is the machine we must build.
- The *Controlled Domain*. This is the domain to be controlled. It is dynamic and both active and reactive. That is, some events and state changes occur spontaneously, without external stimulus; and there are also events that are externally controlled and cause the domain to respond by internal events and predictable internal state changes.

– The *Desired Behaviour*. This is the desired relationship among the various events and state changes of the *Controlled Domain*.

The solution task is to construct the *Controller* so that it brings about the *Desired Behaviour* in the *Controlled Domain*.

Finally, the Workpieces Frame might be suitable for a very small and simple CASE tool. It has these principal parts:

– The *Tool*. This is the machine we must build.

– The *Workpieces*. These are the objects that the users of the *Tool* create and work on, with the help of the *Tool*. They are intangible graphical or textual objects, realised entirely within the *Tool*. They are dynamic but inert: that is, their states can change, but only in response to externally controlled events.

– The *Operation Requests*. These are the users' requests for operations — such as graphic or text object creation and editing — to be performed by the *Tool*. They occur autonomously.

– The *Operation Properties*. These are desired relationships between the occurrences of the *Operation Requests* and the states of the *Workpieces*.

The solution task is to construct the *Tool* so that it responds to *Operation Requests* by operating on the *Workpieces* in accordance with the *Operation Properties*.

These three problem frames — the JSD Information System Frame, the Simple Control System Frame, and the Workpieces Frame — are significantly different. Most notably, they differ in the characteristics of those principal parts that could be said to reflect their central subject matter. In the JSD Frame, the *Real World* is active and autonomous; in the Control Frame, the *Controlled Domain* is both active and reactive; in the Workpieces Frame, the *Workpieces* are inert.

These differences, and others also, are reflected in the different methods that may be associated with each frame. The JSD method [Jackson 83, Cameron 89] may be understood as a method for solving JSD Information System problems. JSD exploits the dynamic and autonomous nature of the *Real World* by using concurrent simple sequential processes to represent the subject matter about which the *Information Outputs* must be produced. Events are regarded as primary, and states as secondary: states are defined in terms of event histories.

For Simple Control System problems a candidate method is a simplified version of the method described by Parnas and Madey [Parnas 91] and incorporated into the Core method [Faulk 92]. A central theme in this method is a distinction that is missing from JSD (and is not needed in problems that fit the JSD Information System frame). Two descriptions are made of the *Controlled Domain*: one to capture those *natural* properties that it possesses regardless of the behaviour of the Machine; and another to capture the *Desired Behaviour* — those properties with which the Machine is *required* to endow it. Parnas and Madey represent both sets of properties as relations over variables of the *Controlled Domain*. They call the first relation *NAT*, and the second *REQ*.

For a Workpieces problem it would be appropriate to use a method based on abstract data types, such as Larch [Guttag 85] or VDM [Jones 90] or Z [Wordsworth 92]. The definition of the type is, in effect, a description of the *Workpieces*. The operations to be performed by the *Tool* in response to the *Operation Requests* are the operations of the type. In a model-based method such as VDM or Z, the description of the model state is, of course, the description of the *Workpieces* viewed as a data structure.

The accounts given here of the different problem frames and the different associated methods are hugely simplified. This simplification is partly just a matter of the brevity of presentation in this paper. But it is much more fundamental than that. A problem frame must be simple, and the associated methods must be simple too. It is only by stripping away the tangles of complications that surround realistic problems that we can see a basis for classifying them and devising powerful methods. A method is powerful only to the extent that it exploits the particular properties of the class of problem being solved. The resulting simplification and lack of realism is a central theme in dealing with complexity. I shall return to this theme later in the paper.

4 Describing the World

If we mean to think seriously about software development problems we must focus our attention initially on everything except the machine we will ultimately build. The *System*, the *Controller*, and the *Tool* each constitute only one principal part among several in the problem frame, and initially they are the least interesting. Our customer's requirement lies elsewhere, in the world outside the machine: that is, in the other principal parts of the frame. The machine we build will — if we are

successful — satisfy the requirement; but it does not itself embody that requirement.

Our primary concern is with understanding and describing what we may call the *application domain* and the *requirement*: in JSD, the *Real World* and the *Information Outputs, Requests,* and *Function*; in a Simple Control problem, the *Controlled Domain* and the *Desired Behaviour*; in a Workpieces problem, the *Workpieces* themselves and the *Operation Requests* and *Properties*.

Explicit description of the application domain is unnecessary in the small mathematical problems that Polya discusses. For those problems, the application domain is formal, and is already well known to the problem solver. When the *data* in a problem to find is a triangle, we do not expect to devote serious effort to understanding and describing the mathematical notion of a triangle. We already know what it is, and we already know its essential properties: that the sum of its interior angles is equal to two right angles; that its area is one half of the height multiplied by the base; that the length of each side is less than the sum of the two other sides; and so on. When the *unknown* is a prime number, we already know what a prime number is.

This convenient, but atypical, property is shared by the integer and integer array problems that provide so much material for the exposition of certain formal styles of program development. Those styles, and their expositors, are sometimes criticised on the ground that their techniques do not scale up: they deal in small and simple problems when real problems are large and complex. But a far more serious criticism is that they have taught generations of software developers that problem capture is a trivial task: one need scarcely do more than mention the problem before setting about its solution. Rather like the prison inmate who need only shout out 'joke number 43' to make his fellow inmates laugh.

On the contrary. In most real problems description of the application domain should consume a very large part of the total effort. The context of the problem is in the application domain, not in the machine. But immediately, this raises a severe difficulty. The application domain is almost always informal. This has several consequences.

The first consequence is that generalisation and classification is always imperfect, and always vulnerable to the production of new evidence, new counterexamples, and new objections. Everyone knows that the meaning of terms such as 'customer' and 'employee' varies from department to

department within the same organisation. The meaning of 'motor vehicle' depends on whether you are talking to a lawyer, a manufacturer, a licensing authority, or the AA. There is no simple general definition under which all the meanings can be precisely subsumed.

Another consequence is that reasoning in the application domain is inherently unreliable. We may formalise our premises, and reason with correct logic to a conclusion. Yet we may still find that our conclusion is false when translated back into statements about the domain. The inevitable imprecision of our original formalisation introduces an error term — analogous, if you like, to the arithmetic error introduced when reals are represented by integers — that can vitiate the reasoning.

Another consequence is that we need a much richer and more varied repertoire of languages and notations than is needed for describing machines. The task of describing a machine — that is, of writing a program — is greatly helped by the freedom to decide that the machine should embody a simple and consistent phenomenology. So good programming languages are concisely definable, and exhibit such properties as syntactic and semantic consistency, or referential transparency. But in describing the world outside the machine we have no such freedom. We must describe the world — at least approximately — as it really is. We will certainly impose some degree of systematic consistency on our descriptions: even natural languages do that. But we are not free to sweep away the variety of the world by treating everything as a sequential process, or as a relation, or even as an object.

5 Description Types and Structures

To deal with this richness we must pay a lot of attention to the technology of description. That means that we must be aware of how descriptions are related to the subject matter they describe, of the different kinds of description that may be needed, of the properties and dimensions of each description, and of the many different structures by which descriptions may be usefully related to one another.

The basis of the relationship between a description and what it describes is the *designation*. A designation singles out certain phenomena of interest. It gives an (ineluctably) informal explanation of how the phenomena may be recognised in the application domain, and it gives a formal term — such as a predicate — by which the phenomena will be referred. For example, in the designation:

"The human genetic mother of x is m $\approx$ Mother(x,m)"

the designated phenomenon is the relationship of genetic motherhood between two people; it will be denoted in descriptions by using the predicate Mother (x,m). Designations form the bridge between the formal descriptions we produce in software development and the informal real world. Without explicit designations it is impossible to determine with any confidence whether a description of the world is true or false. To quote John von Neumann: There is no point in being precise if you don't know what you are talking about.

A *definition* gives a formal definition of a term that may be used by other descriptions. For example, the definition:

"Child(x,y) $\triangleq$ Mother(y,x) $\vee$ Father(y,x)"

defines the term "Child(x,y)" to mean exactly the same as "Mother(y,x) or Father(y,x)". A definition can not be true or false: it can only be well-formed or not well-formed, and useful or not useful. It conveys no information whatsoever about the application domain or the machine.

A *refutable description* describes some part of the world, saying something about it that could — in principle — be refuted or disproved. Whether it could in practice be disproved is another matter: the important thing is that it could make sense to disprove it, as it can't make sense to disprove a definition. For example, here's a tiny refutable description:

"$\forall$ m,x • Mother(x,m) $\rightarrow$ $\neg$Mother(m,x)"

Whatever m and x you choose, if m is the human genetic mother of x, then x is not the human genetic mother of m. To refute it you would have to find a pair of mutual genetic mothers. Inconceivable. But not nonsensical.

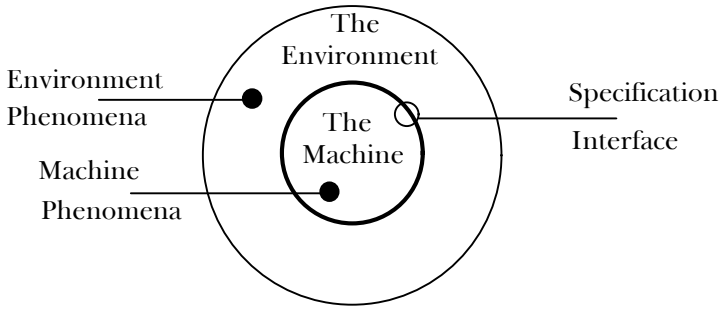
Because there are many parts and aspects of the world to be described, it will always be necessary to build quite elaborate structures of designations, definitions, and refutable descriptions. Determining how to separate the whole description into a number of partial descriptions — that is, how to separate concerns — is a central development activity. At a certain scale, separation is prescribed by the chosen method: JSD enjoins the developer to separate the description of the *Real World* from the description of the *Information Function*; and Parnas and Madey enjoin a separation of the *NAT* and *REQ* relations. At a larger scale, separation

will be guided by the treatment of complexity, in which several methods and frames are deployed on one problem.

6 The Machine and the World

The requirement is in the application domain, but its satisfaction is to be achieved by the machine. The transition from one to the other depends on a relationship between the two. Two aspects of the relationship are of special importance for descriptive technique: interaction, and modelling.

The machine interacts with its physical environment. This means that some phenomena — events and states — are common to the machine and the environment. These common phenomena constitute what we may call the specification interface. I picture it like this:



Problems and requirements are to be expressed solely in terms of the environment phenomena. Programs are to be expressed solely in terms of the machine phenomena. The bridge between them is formed by specifications, which are expressed in terms of the phenomena common to the environment and the machine.

A specification is both a restricted kind of requirement, and a restricted kind of program. Programs in general may be expressed partly in terms of phenomena that are private to the machine and not visible at the interface. Programs that refer to private machine phenomena are not specifications. (Or they are very bad specifications that may be said to exhibit implementation bias [Jones 90].) Similarly, requirements in general may be expressed partly in terms of phenomena that are private to the environment and are not shared with the machine. They, too, are not specifications: they do not give enough information for the construction of a program.

This distinction among requirements, specifications, and programs is of great significance. In a trivial problem we may write the program text

directly: no sane software developer would progress from requirement through specification to program to construct the 'Hallo World' program. For a slightly less trivial problem we may content ourselves with developing a specification and progressing from there to a program. This is the danger zone for more demanding problems. The specification is a requirement — that is, it is expressed in terms of environment phenomena. So it is easy to suppose that it is the customer's requirement, ignoring the virtual certainty that the customer's requirement is not confined to the part of the environment that interfaces directly with the machine.

This is the source of some of the failures in which a system fails to achieve the required effects in the application domain. Avionics system failures furnish some notable and tragic examples. In one system a plane overshot the runway on landing in rain because the thrust reversers could not be turned on. The wheels were aquaplaning on the wet runway, and an interlock prevented engagement of the thrust reversers when the landing wheels were not turning.

Even in a context in which it is proper to confine attention to the shared phenomena there is a further danger. One may easily suppose that an interface can be equally well described from either side. But it is not so. On the machine side we can manage very well with descriptive techniques that abstract from causality and from the distinction between what is true and what we would like to be true. The causal properties of the machine are well understood in terms of the programming language, especially if it is an imperative programming language. Also, we know that we are always constructing the machine *de novo*: a general-purpose computer left alone will do nothing, it will exhibit only a null behaviour. It is therefore fully adequate to describe behaviour from the point of view of what we may call the idiot observer: in CSP terms [Hoare 85], the trace semantics is enough. The effect of behaviour outside the bounds of the specification is, naturally enough, undefined.

But as a description of the environment this conceptual parsimony is harmful. We need to know whether the precondition on an operation specification means that the environment will never, because of its internal properties, try to execute the operation when the precondition is false; or that the environment must be externally constrained to ensure the same restriction; or that the machine itself will frustrate any attempt to flout the restriction.

Another source of much confusion is the use of modelling. Modelling is

an important technique in a number of methods. For example, the JSD method prescribes that the foundation of the *System* should be a model of the *Real World*: that is, that the concurrent sequential processes observable in the *Real World* should be simulated by processes within the *System*. This notion of modelling gives rise to a kind of reuse of descriptions. The same refutable description can be truthfully applied both to the *Real World* and to the *System*, by using two different sets of designations.

If there is no explicit recognition of the role of designations in a development that uses modelling in this kind of way, there will always be uncertainty about whether a particular description is intended to describe the machine or the application domain. This uncertainty is particularly noticeable in data modelling methods, and also in model-oriented methods such as VDM and Z. The reader of an Entity-Relation diagram is never sure whether the diagram describes the database or the real world. The reader of a Z operation schema is never sure whether the schema is describing the changes of state within the machine, or the effects of the operation in the world outside the machine.

One might ask: If the machine is a simulation of the world, then surely the same description applies to both. Does it matter which is being described? Of course it does matter. The simulation is only a partial simulation, and the refutable description that applies to both is only a partial description of each. Both the machine and the real world have properties that do not enter into the modelling. A database, for example, is far from a complete model of the application domain: the domain has many features that are not modelled in the database, including a range of possible causal relations among events. Symmetrically, the database has features, such as record deletion, and null values in records, that are peculiar to the machine and are not models of anything in the domain. There are therefore at least three descriptions of interest: the shared description; a description peculiar to the machine; and a description peculiar to the application domain. We must always know which of the three we are writing or reading.

7 Complexity and Problem Decomposition

The problem frames and associated methods briefly discussed earlier were all simple; certainly too simple for most realistic problems. This simplicity is an essential characteristic, because the central purpose of a problem frame is to define a class of problem for which a reliable solution method is known. The problem frame is purged of the realistic

complexities that may — almost certainly will — make an otherwise simple problem difficult.

Given a repertoire of such problem frames, the complexity of realistic problems may be handled by decomposing them into simple problems. This is not the unguided decomposition of top-down functional decomposition. It differs in three important ways:

- The choice of 'subproblems' is restricted to those for which a problem frame and associated method are known. This means that the decomposition is a decomposition of a problem that you *don't* know how to solve into problems that you *do* know how to solve. (A decomposition into problems that you still don't know how to solve may make matters worse rather than better.)
- The decomposition is into *heterogeneous* subproblems. There is no *a priori* reason to decompose into several instances of one problem frame in the style of a homogeneous top-down decomposition into several procedures or into several processes. On the contrary, different aspects of a realistic problem are likely to conform to several different problem frames.
- The resulting structure of subproblems is essentially a parallel, not a hierarchical structure. The principal parts of the different problem frames overlap. In particular, they will be concerned with different views and groupings of overlapping phenomena of the problem domain. Although hierarchical relationships may sometimes be found, they will be the exception rather than the rule.

8 Objects and Problems

Object-oriented methods, especially object-oriented analysis, are an important field of application of these ideas about problems and descriptions.

The distinction between problems and solutions has not always been fully recognised in object-oriented methods. It is always tempting to fall into the traditional software trap of treating every new technique as a panacea, capable of curing all ills, and into the related trap of treating each new programming language as a language for describing problems and problem domains. In their time Fortran and COBOL were touted as problem description languages. We should avoid making the same mistake with object-oriented languages.

A salutary paper by Høydalsvik and Sindre [Høydalsvik 93] makes a number of important points. It should be compulsory reading for all students of object-orientation. The authors say:

> "An object-oriented representation might be good for some kinds of [domain] knowledge, but less suitable for other kinds of domain knowledge. Since an analysis specification has to contain many different kinds of knowledge, OOA [Object-Oriented Analysis] will only present a partial solution. …
> "The main motivation for choosing OOA … is clearly target-orientation — the analysis technique is chosen to fit in with the following design technique rather than the problem at hand."

They give two examples of domain knowledge that is hard to capture in terms of objects. Global rules, such as 'Product A should never be cheaper than product B'; and the dynamics of high-level tasks involving operations from several objects. In both cases, the granularity of an object-oriented description makes it difficult to reference all the relevant phenomena in a single description of sufficient span.

But even where there are no problems of granularity, the phenomenological assumptions built into the most commonly used object-oriented approaches may be very ill-suited to the realities of an application domain. The inadequacy of single inheritance is well-known: that particular shoe pinches almost every wearer. In principle, if we regard a class as a collection of properties, we should be prepared to contemplate any combination whatsoever of classes whatsoever. McAllester and Zabih's notion of Boolean classes [McAllester 86] is very attractive: discrete properties are defined in base classes, and the classes actually used in a description are drawn from the unrestricted powerset of those base classes. There is a connection here with the unrestricted composition of problem frames to capture a particular complex problem.

The presumption that classification is static is at least as arbitrarily restrictive as the presumption of single inheritance. There are remarkably few examples of static classification in realistic domains. As Hendler [Hendler 86] pointed out in motivating his work on Enhancements, undergraduates become graduates, and graduates become PhDs. We may also observe that pupils become teachers, warehouses become apartment buildings, caterpillars become butterflies, cotton mills become offices, and employees become pensioners. Real-world individuals not only partake freely of the properties of many classes; they also take on and shed those properties over time with at least equal freedom.

It seems to me that there are two important — but so far somewhat neglected — challenges now facing those who advocate object-oriented approaches to software development. First, to understand and exploit the true strengths of object orientation while recognising its limitations. It may, perhaps, be said that those aspects of object-orientation — classification and inheritance — that now receive most attention in tutorials and in method advocacy are its weakest aspects for describing the real worlds in which problems and requirements are found.

The second challenge, I believe, is to develop the work on patterns, and the closely related work on frameworks, that is now beginning to emerge. This work, developed in the right direction, promises to explore object-orientation in a very fruitful way, by focusing — as I believe we must — on the nature and structure of complexity, analysing some of the problems — at least in the programming context — that it is our business to solve. A new book [Gamma 94] on patterns in object-orientation ends with a provocative quotation from the software developer's favourite architect, Christopher Alexander [Alexander 79]:

> "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."

References

[Alexander 79] Christopher Alexander; The Timeless Way of Building; Oxford University Press, 1979.

[Cameron 89] J R Cameron; JSP & JSD: The Jackson Approach to Software Development; IEEE Computer Society Press, 2nd Edition 1989.

[Faulk 92] Stuart Faulk, John Brackett, Paul Ward, and James Kirby, Jr; The Core Method for Real-Time Requirements; IEEE Software Volume 9 Number 5 pages 22-33, September 1992.

[Gamma 94] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides; Design Patterns: Elements of Reusable Object-Oriented Software; Addison-Wesley 1994.

[Guttag 85] John V Guttag, James J Horning, and Jeannette M Wing; The Larch Family of Specification Languages; IEEE Software Volume 2 Number 5 pages 24-36, September 1985.

[Hendler 86] James Hendler; Enhancement for Multiple Inheritance; SIGPLAN Notices Volume 21 Number 10 pages 98-106, October 1986.

[Hoare 85] C A R Hoare; Communicating Sequential Processes; Prentice-Hall International, 1985.

[Høydalsvik 93] Geir Magne Høydalsvik and Guttorm Sindre; On the Purpose of Object-Oriented Analysis; with a discussion by Dave Thomas, Adele Goldberg, James Coplien, Peter Coad and Geir Magne Høydalsvik; in Proceedings of OOPSLA '93, ACM Sigplan Notices Volume 28 Number 10 pages 240-258, October 1993.

[Jackson 83] M A Jackson; System Development; Prentice-Hall International, 1983.

[Johnson 94] Ralph E Johnson; Why a Conference on Pattern Languages? ACM SE Notes, Volume 19 Number 1, pages 50-52, January 1994.

[Jones 90] Cliff Jones; Systematic Software Development Using VDM; Prentice-Hall International, 2nd Edition 1990.

[McAllester 86] David McAllester and Ramin Zabih; Boolean Classes; in OOPSLA '86 Conference Proceedings: SIGPLAN Notices Volume 21 Number 11 pages 417-423, November 1986.

[Parnas 91] D L Parnas and J Madey; Functional Documentation for Computer Systems Engineering (Version 2); CRL Report 237, McMaster University, Hamilton Ontario, Canada, 1991.

[Polya 57] G Polya; How To Solve It; Princeton University Press, 2nd Edition 1957.

[Wordsworth 92] J B Wordsworth; Software Development with Z: A Practical Approach to Formal Methods in Software Engineering; Addison-Wesley, 1992.