SEARCC8 Productivity Tools and Development Methodology

Keynote Address: Software Manufacture

M A Jackson

1 Introduction

Our business is software development. We can look at software and its development from many different points of view. For example, we can regard a program as a mathematical object, and its development as a mathematical activity having much in common with the conception and proof of a mathematical theorem. Or we can regard software as an engineering product, and look to traditional branches of engineering, such as civil and chemical and aeronautical engineering, for indications of how we may advance our understanding and practice of software engineering. Some people point to the increasing power and scope of expert systems, and invite us to think of software development as a special case of what they call knowledge engineering. Others point out that software development is a human social activity, and focus their attention on the personal and group relationships among the people involved in buying, building, and using software systems.

There is truth in all of these views, and we can learn from them all provided that we recognise that no one view has a monopoly of the truth. Each view present a different aspect of the truth, and because software and its development are very complex the truth about them has many aspects. In this talk I would like to present another view, hoping that it will be illuminating but recognising that it can offer at most a part of the truth. I would like to view software development as a manufacturing activity, in which a product is created by creating, modifying, and combining various parts made from various raw materials. For example, we may view the Pascal text of a procedure as a part that is processed by the compiler to give another part, which is the relocatable machine-language text; and we may view that machine-language text as one of several parts that are combined by the link-editor to give an executable machine-language program. The work of software development is the work of carrying out such manufacturing operations using whatever machine tools are available.

I do not mean to limit this view to programming, or to what might be called the implementation stage of software development. On the contrary, I wish to apply it to the whole of software development from the earliest stages of requirements analysis and specification through to delivery of the product to the customer and its installation and maintenance. Throughout the development stages, the developers are concerned with textual and diagrammatic documents, and it is all of these that I wish to regard as parts in the manufacturing activity. Because the word 'part' is likely to suggest a module of the finished product, I prefer to use the more neutral word 'description': software developers are concerned with descriptions of the customer's requirements, of the problem domain, of the context in which the software will be embedded, of the system's behaviour, of the rules for its use, of the

system's operational characteristics, and with descriptions of the software itself and of its components at every level of abstraction and detail. A Pascal program text is a description, in terms of Pascal semantics, of a computation to be performed; a machine-language text is another description, in terms of the semantics of the execution machine, of the same computation; a procedure hierarchy diagram, showing the procedure invocation structure, is yet another description of the program, and a proof of correctness, showing pre-conditions and post-conditions and loop invariants, is yet another again. A statement of the behaviour of a company's employees - their arrival at work and their leaving after work, their promotion from grade to grade, their holidays and illnesses - is a description of an aspect of the problem domain for a payroll system, just as a statement of the grammar of Pascal is a description of an aspect of the problem domain for a Pascal compiler.

For any non-trivial piece of software there is an infinity of descriptions that may be made and used, manipulated and composed. Software development methodology is concerned with the creation of effective manufacturing schemes that apply to usefully large classes of software product. A manufacturing scheme for software defines the set of descriptions that should be produced, the order - usually a partial order - in which they should be produced, and the operations to be used in their production. We can not, of course, hope to devise a single manufacturing scheme that will be applicable to all classes of software and will also be effective, but must aim instead to devise several schemes, each applicable to a particular class. A method appropriate for developing life-critical avionics software will not be appropriate for developing personal computer applications on DBase III, alhtough they may have much in common. However, in this talk I do not wish to concentrate on methodological issues, but rather to explore other implications of the manufacturing view of software that I have briefly outlined above.

2 Raw Materials for Descriptions

A software description, like any part made in an ordinary manufacturing activity, is made of some raw material; I take the raw material of a description to be the language in which it is expressed. Here I am using the word 'language' in a very broad sense, to include any textual or graphical notation that may be used in software development. Thus Pascal and COBOL are languages, and so too are directed graphs and lambda notation and Hoare's CSP and BNF notation for grammars and decision tables and OS/360 JCL and Horn clause logic. Nor do I exclude natural language, which is certainly a necessary raw material for many descriptions that we must create.

One of the most crucial choices in software development is the choice of a language for each description that is to be manufactured, just as a crucial choice in conventional manufacturing is the choice of raw material for each part. The software developer may delegate this choice by default, perhaps to the inventor of the chosen development method, or to the provider of a complete development environment, but the choice must still be made, and must be made for each individual description.

The criteria for choice are quite simple. First and foremost, the language must allow a direct and clear expression of the meaning of the description; in particular, the description must be immediately understandable to any person who will need to create or read it. We are not talking here about the expressive power of languages in the formal sense in which regular languages have the same expressive power as finite-state machines, but about immediacy of human understanding. We would not choose to describe a project plan in SQL or a mathematical computation in the form of a decision table: they are inappropriate raw materials for those descriptions. Second, we should prefer a formal to an informal language wherever possible. There is no point in creating gratuitous opportunities for ambiguity and misunderstanding. We would not choose to describe the operations on an abstract data type in natural language when we might give an unambiguous description in an algebraic or state-based formalism instead. Third, we should prefer a language that allows us to use mechanical manipulations over one for which we have no useful tools. We want to mechanise as much as possible of our work in software development, and the availability of tools is a valid criterion in the choice of language for an individual description.

These criteria lead us to see at once that in any non-trivial software development we will need to use many languages, in the broad sense in which I am using the word 'language'. Just as the automobile engineer sees that certain parts must be made of steel and others of glass or aluminium or plastic, so the software engineer must see that some of his descriptions must be made of graphs and others of predicate logic and others again of recursively defined functions. It is a mark of the extreme immaturity of the computing discipline that so much of the serious academic work, and so much too of the practical work, is confined to consideration of one language in virtual isolation from all others. There is great intellectual and aesthetic beauty to be found by exploring the limits of what can be done with a single material: the silversmith and the sculptor who works in stone may produce great works of art. But you can not make a motor car in that way. A motor car must have windows made of glass and pistons made of steel and tyres made of rubber, or it will be of little value. A software development must similarly use many materials, each one where it is appropriate.

3 Informal Languages

I have suggested above that the languages to be used in software development must include informal languages, and, in particular, natural language. We are, of course, reluctant to use informal languages where we could use a formal language, because of their potential for ambiguity and our inability to manipulate them mechanically; but sometimes we have no good alternative. One reason may be that we simply know too little about some aspect of what we wish to describe to give a formal description: for example, we want input formats to be convenient, or output formats to be easy to read and appealing to the eye, but we know too little about the ergonomics of the human-computer interface to be able to express such requirements

in any formal way, and must rely, at the requirements stage, on such vague statements expressed in natural language.

A more significant reason appears in descriptions of the problem domain, where the problem domain is in the natural world. This will be true of most data processing systems, such as administrative systems, payroll systems, and sales order processing systems. Here the problem domain is inherently informal, and we wish to compute about it in a system that is itself necessarily formal: we are therefore required to build a bridge between the informal natural world and our formal system, and this bridge can be made only in an informal language. It is an important methodological principle that this bridge between the problem domain and the system should be as narrow and as localised as possible, to ensure that the unavoidable informality does not spread to descriptions that have no need of it.

4 Simple Manufacturing Operations

In conventional manufacture of things like motor cars and washing machines, there is a repertoire of relatively simple operations such as drilling, turning, milling, pressing, and grinding. These simple operations are applied to a single part, and they have the effect of modifying it in some way: for example, by drilling a part we make a hole in it, by pressing we convert a flat part into a three-dimensional part. The original form of the part is, of course, lost in the operation, and only the modified result is available once the operation is complete.

In software manufacture there is, or there ought to be, a similar repertoire of simple operations on descriptions, with the important and useful difference that the original form of the description remains available after the operation, along with the modified form. The possible operations will, of course, depend on the raw material - the language - of the description. Obvious examples of such operations may be drawn from the standard literature: for example, a non-deterministic finite-state machine may be converted into a deterministic finite-state machine; a deterministic finite-state machine may be converted into a regular grammar, and vice versa; a finite-state machine may be reduced to a minimum form; a grammar with left recursion may be converted to an equivalent grammar without left recursion; some grammars may be converted directly into recursive descent parsers. It is notable that the most fruitful source for such examples is to be found in compiler technology, where historically there has been both a trend to specialisation and a broad cooperation between the theoreticians and the practitioners of software manufacture.

We may characterise these simple operations as being non-interactive and as being functions of one argument. They can be executed without human intervention, and they simply take one description and modify it to give another. There are many trivial additions we might make to the list of such operations: for example, we might derive a list of terminals from a grammar, or a list of non-terminals, or a list of nodes from a finite-state machine or indeed from any graph. But it is not yet clear that such

operations would be useful, because we do not yet have a context in which we could put them to good use.

5 Composing Descriptions

The simple operations mentioned above are certainly not enough for the manufacture of motor cars or washing machines: they suffice only for the manufacture of products that can be machined from a single block of raw material. To make a motor car or a washing machine it is necessary to be able to put parts together, and in the same way the manufacture of non-trivial software requires the ability to put descriptions together. That is, we need operations that compose two or more descriptions to give a combined result: our repertoire of manufacturing functions must include functions of two and more arguments.

Composition, I suggest, is the fundamental type of activity in software manufacture. Some traditional techniques of software development, such as top-down functional decomposition, and stepwise refinement, have embodied the opposite view, that the fundamental activity is decomposition: we begin with a notion of the complete software product, and we decompose this notion, level by level, until we reach a level at which the decomposed parts are already available in the programming language. I believe that this view is seriously misleading. I prefer to think of software development as an activity in which we create descriptions - among others - of the desired product from a number of different points of view, and successively manipulate and compose those descriptions until we have succeeded in describing a product of which all of our original descriptions are true.

I am not here arguing for bottom-up development in place of top-down development. The descriptions to be composed are not, in general, descriptions of software parts to be fitted together in a hierarchical structure. Rather, they are descriptions of different aspects of the software, different abstractions of the same thing, descriptions of the same thing viewed from different angles. There is an illuminating analogy with architectural descriptions of a building. The architect may produce a perspective drawing of the building, and also floor plans and several elevations at different cross-sections; eventually the architect and builder must produce a building of which all those descriptions are true, but there will be no part of the building that can be identified with a particular plan or elevation.

Software manufacture abounds in simple examples of this kind of composition. For instance, consider the operation of composing two finite-state machines to give their union, the machine that accepts any sequence that is accepted by either of the two machines. If we interpret the description of each of the two original machines as meaning 'this machine accepts at least this set of sequences', then we may interpret the description of their union as meaning 'this machine accepts at least this set of sequences and at least that set also'. Another example is the composition of data structures to give a program structure in the JSP design method. Given descriptions of a program's input and output streams as regular grammars represented as trees, the

program structure is formed as a superset tree, of which each of the stream structure trees is a pruning and simplification.

We may also consider a much larger example of such composition. Suppose that we are developing a data processing system, perhaps a payroll system, that has a significant database. Then we may give several different descriptions of the system: we may describe the structure of the database by some kind of decorated graph; we may describe the way in which the system models the problem domain by a set of cooperating sequential processes, one process instance for each employee; we may describe the semantics of each transaction in terms of a partitioned global state with pre-conditions and post-conditions for the transaction. Each of these descriptions is appropriate for certain aspects of the system structure and behaviour. A full description of the finished system is a composition of such descriptions: the finished system is an object of which each of the individual descriptions is true.

6 The Importance of Parallel Composition Operations

This kind of composition is parallel rather than hierarchical. It is important because of what Dijkstra has aptly called 'the separation of concerns': we need to be able, in software development, to pay attention now to one aspect of our task and now to another. But the separation of concerns is of little value unless we are able at a later stage to merge the results of what was previously separated. Sometimes we will be able to avoid this merging, allowing ourselves to map the individual descriptions we have made on to individual components of the finished software. But more often we will be compelled to compose the descriptions so that we obtain a product of which they are all true.

One compelling reason for composing different descriptions is that they may be describing different structures over the same elements. The records of the employee database are the activation records of the processes that model the employees, and the components of the global state are the components of those activation records. We can not separate these elements in the finished system, so we must compose the different descriptions.

Another reason is the need for efficiency. Pierre de Marneffe called attention long ago to what is known as the Shanley principle in mechanical engineering, the principle that one component should satisfy more than one part of the specification. In the rocket technology of the 1940s it was recognised that a rocket required an aerodynamic outer skin, a vessel to hold the rocket fuel, and considerable structural strength; a major breakthrough came when the engineers followed the Shanley principle and designed the body as a tube that had the required aerodynamic properties, was capable of containing the fuel directly, and provided the necessary structural strength: the one component satisfied all three parts of the specification. Another example may be taken from motor car design. Until the 1950s, motor cars were built with the chassis separate from the body. The chassis provided the frame on which the engine, the wheels, and the power transmission were mounted; the

body, carrying the seats and providing protection for the passengers, was a separate component. Then it became evident that the body and chassis could be designed as one component made from welded pressed steel panels, with considerable savings in weight and in manufacturing costs.

The Shanley principle applies equally to software manufacture. Even where it is possible to realise different descriptions by different components of the finished system, it will often be inefficient and cumbersome. If we need to compute two functions of a graph, we will want to describe each function separately but compute both functions together in one traversal. If we have a large master file from which we require two summaries, we will want to give separate descriptions for the two summaries but obtain them both from one pass of the file.

7 Composition of Dissimilar Materials

Motor car manufacturers have techniques that allow them to compose different parts made from the same material: a car body is made by welding together several pressed steel panels. But they also have a much wider range of techniques for composing parts made from different materials: they know how to bolt aluminium to cast iron, how to mould rubber round nylon cords, how to glue glass to mild steel channel, how to shrink a steel component around a brass bearing.

An analogous range of techniques is needed in software manufacture. If one aspect of a program is best described as a recursive function definition, and another aspect is best described as a sequential process, we need to be able to compose these descriptions although they are made of different materials. This requirement of software manufacture has been sadly neglected by computer scientists, partly, I suspect, because it presents great difficulties, and partly because so much effort has been invested in techniques limited to the handling of a single material. Many computer scientists are like the small boy with a hammer, to whom everything in the world looks like a nail. To a computer scientist with a logic programming system, everything in the world looks like a Horn clause. Nor is it only academic and theoretical people who too often think this way. Practitioners too are inclined to seek panaceas, single languages or techniques that will cure all of the world's ills: one need only think of the exaggerated claims made for the universality of data modelling, or functional programming, or relational databases, or expert systems.

But it would be wrong to suggest that no work at all is being done in this important area. Boyle has shown how functional programs written in Lisp may be translated into Fortran, thus providing a part of a possible solution of the problem for one pair of languages - albeit programming languages; Zave has shown how a small system may be built using a combination of her own functional language, Paisley, and logic programming and a version of CSP; some work has been done at Imperial College on the addition of sequential constraints to a functional specification language. It is my hope that much more work of this kind will be done in the future, especially concentrating on what might be called specification languages rather than

programming languages, and on the static composition of descriptions in different languages rather than on the dynamic interleaving of their interpretation in execution.

8 Wide-Spectrum Languages

It might be suggested that the solution to the problems I have mentioned lies in the creation of a truly wide-spectrum language, in fact a universal language for software development. But I think this would be a mistake, analogous to the mistake of seeking a universal material from which every part of a motor car could be fashioned. Even if such a language were possible, it would certainly be horrendously large and complex, much worse than PL/I or Ada. It would not only provide the possibility of declaring and using every conceivable data type and operation with every conceivable property, but it would also provide every conceivable mechanism for combining such types and operations both statically and dynamically.

And there are even more serious objections to the idea. First, the accepted aims in programming language design of orthogonality and referential transparency would be completely unattainable: as a result, the language would be bedevilled everywhere by anomalies and inconsistencies, and its semantics would be quite impossible to specify even approximately. Second, the idea of a single language, presumably accompanied by a single processor that processes texts written in that language, is tied to what we might call the 'big bang' approach to development. Software manufacture demands the opposite, an incremental approach; not in the sense that a system should be delivered incrementally to the customer, although that is no doubt true for many systems, but in the sense that the developer must be continually involved in the manufacturing process, sometimes to choose what operation should be performed next and sometimes to introduce new information in the form of new descriptions or decorations of existing descriptions. Third, many descriptions are best made in a pictorial language, which would be hard to accommodate in the notion of a single wide-spectrum language: a graph, for example, may be much more easily created and viewed in pictorial form than in the form of lists of nodes and arcs.

9 Tools and Operations

There are already very many software tools available. There are editors and interpreters, compilers and linkers, theorem provers and data dictionaries, CASE tools and complete development environments. But it seems to me that these tools, excellent as they may be for their stated purposes, are not enough. With the exception of the editors and perhaps of some data dictionaries, they are too large and elaborate to serve the general purposes of software manufacture. We have special purpose tools to perform highly complex and elaborate operations, but too few tools to perform simpler and more fundamental operations. We have the machine to bore out complete cylinder blocks for motor cars, but we lack the general purpose drill to bore one hole at a time in any component we choose.

One reason for this lack is the absence of a context into which such small and simple tools could be fitted. To some extent this lack is being repaired by the CAIS (Common Apse Interface System) and PCTE (Portable Common Tools Environment) standardisation efforts in the USA and Europe. But even if these projects are moving in the right direction, and many people have doubts about their underlying technical assumptions, it will be a long time before tangible results emerge. Meanwhile tool vendors are compelled for good commercial reasons to provide more or less complete environments instead of individual tools, and such environments lock their users into a closed world into which no foreign tools can be introduced. At the programming stage, if you are programming in Smalltalk 80 you can not switch briefly into Prolog or Lisp. At the specification stage, if you are using a graphic-based data modelling tool you can not switch into a Larch-style algebraic specification of an abstract data type, or into CSP or the lambda calculus. At each step, the software developer is locked into one fixed environment and limited to the use of one fixed set of tools designed for the handling of descriptions made from one material only.

10 Re-usability

At the NATO Software Engineering Conference of 1968, Doug McIlroy of Bell Laboratories complained that re-use of standard components, which is a hallmark of mature engineering, was sadly rare in software engineering. Today, twenty years later, the picture is not much better. Certainly there are widely used libraries of Fortran mathematical routines, and libraries of graphic interface routines for mouse-driven personal computer systems, and other similar examples can be quoted. But the world is still full of programmers programming linear searches and tree traversals, implementing sets and bags and queues and stacks, and translating into COBOL the same tax regulations as their colleagues in the company across the road.

Some people seek the solution to this problem in object-oriented design and programming, or in some discipline based on abstract data types. The generics feature of Ada is intended to allow general purpose software components to be created that can then be specialised or instantiated for particular uses. In his delightful book on Ada, John Barnes paints a fanciful picture of the software component shop of the future. The customer comes into the shop with a component specification; the shopkeeper offers to instantiate the component while the customer waits, having first determined that it is the de luxe version - fully validated and guaranteed to raise no exceptions - that the customer requires rather than the standard version. The happy customer leaves with the desired component under his arm. Similar aspirations can be seen in more sophisticated approaches, such as that of the Larch specification language developed by Guttag, Horning and Wing.

Certainly component re-use has been achieved to some extent in such environments as those of Ada and Smalltalk. But there is something deeply unconvincing about the fanciful pictures painted of a future in which most software will be built of standard components. For components to be re-usable, there must be a certain relationship

between the complexity of the specification and the difficulty of implementation. An electrical engineer has no difficulty in re-using such components as resistors and capacitors, because the required component can be specified in a very few standardised names and integers. The required resistor is 2 watt wirewound 1 Kilo-ohm: the specification contains no more than 3 integers of information, and can be written down in a second or two; the component can be bought for perhaps 10 pence, and would cost perhaps 100 times as much to make by hand. The specification of a transistor is more complex, but the cost to make is correspondingly greater in proportion to the cost to buy, and the same is even truer of integrated circuits which are prohibitively expensive to make in small quantities.

The best example of software components that exhibit a similar relationship between the complexity of specification and the difficulty of implementation is found in mathematical subroutines. Given the body of existing mathematical conventions, a routine for inverting a matrix or for solving a system of differential equations is relatively simple to specify and relatively difficult to implement, so it is a natural candidate for re-use. Furthermore, such a routine, like an integrated circuit, is of a sufficiently high value for the engineer who uses it to be willing to accommodate his design to its interface demands. None of this is true of a routine to perform a linear search, or to traverse a tree in preorder.

And there is another obstacle to re-use to consider. The Shanley principle, which I discussed briefy a little earlier, will usually lead us to want to combine different descriptions into one component: we want the component to exhibit more behaviour than is described in its catalogue specification, so we are led to modify it and thus lose almost all the benefit of our original purchase: the game is no longer worth the candle.

I believe that re-use in software manufacture can make sense only if we can re-use descriptions in quite general ways, and that this will be possible only if we are equipped with the tools for performing powerful operations to manipulate and compose descriptions. There is not much more that can be done in the way of re-using completed software components when our basic toolset contains little more than a compiler.

11 Some Concluding Remarks

In the very earliest days of software development for electronic computers it seemed that one description of a program was enough: the developer's task was to describe the program in machine code, and to have that description punched into paper tape. Then it became clear that it would be very useful to give a description of the computation in a language that abstracted to some extent from the execution characteristics of the particular machine, and simple symbolic assembly languages and autocodes were introduced along with their associated compilers. Later, the need for some kind of functional specification became apparent: in the time-honoured cliche, it was necessary to describe not only how the program worked but

also what it did. Then it became apparent that a more general statement of the customer's requirements was needed, and also an explicit description of the problem domain - the subject matter about which the software computed - and that the specification should not be limited to functional aspects but should include also information about performance and other behavioural characteristics. Meanwhile, the combination of hardware and software on which the developed system was to be executed became much richer and more complex: data structures on backing store, protocols for communication, schemes for managing the use of virtual and cache storage, all these and much else demanded careful description and design. The number and complexity of the descriptions that may be given of a non-trivial piece of software have grown almost without limit.

Some of these descriptions are inherently informal, and can be used only for human communication. But the others can be used in the software manufacturing activity, either constructively, to derive other descriptions and, eventually, to derive the finished product, or analytically, as gauges and templates to measure the conformity of the product to its specification. No compiler writer doubts that a description of the problem domain, which for a compiler is a description of the language to be compiled, can and must be used constructively in the manufacture of the finished product; or that suitable tools, such as a lexical or syntactical compiler compiler, can and should be used to carry out manufacturing operations on these descriptions. I am doing no more than suggesting that we should take the same attitude to most of the many descriptions that can be produced of our other software products, and that we should pay much more attention that we do now to making software manufacture in this sense into a reality.