Software Manufacture

Michael Jackson
Michael Jackson Systems Limited
London, United Kingdom

1 INTRODUCTION

We can think about software development in many different ways. Certainly, it is a social activity, in which the developers, the buyers, and the users of the software interact in both conflict and cooperation. It is also a branch of mathematics, in which the program texts to be produced may be regarded as mathematical objects demanding an axiomatic basis for the theorems, lemmas, and proofs required for their correct construction. It is also a branch — or perhaps several branches — of engineering, in which experienced software developers apply established practical techniques whose theoretical basis may have been forgotten or, in some cases, never fully worked out.

We can also think of software development as a manufacturing activity, in which the development department is like a jobbing manufacturer of mechanical products: not like a motor car factory, in which essentially the same product is produced over and over again in large volumes, because each software project that the department must carry out is different enough for the developers to think of their work as the production of a unique new product. It is this view of software development — the view that it is a manufacturing activity — that I would like to pursue in this article.

2 DESCRIPTIONS

When a software development has been completed, what, exactly, has been produced? There will be some executable program texts, some operating system commands, perhaps an interpretable definition of a database structure, perhaps some data files containing modifiable parameters for execution of the programs; there will be some kind of user manual and guide to the operation of the software. During the development there may have been a statement of requirements, specifications of various aspects and parts of the software and its environment, descriptions of its subject matter, and many other pieces of documentation.

I would like to call all of these 'descriptions'. A program text in C or Ada or Pascal or Cobol is a description of the computation to be performed; a data model is a description of the data on which the programs will operate, and also a description of the subject matter of the system; the user guide may contain a description of the menus offered by the system to the user and what transitions are permitted from one menu to another; the requirements statement may contain a description of database volumes and response times for various functions; the specification may contain a description of output documents and displays that the system produces.

The term 'description' is perhaps surprising. I am applying it to every item produced during the whole development, whether in text or in pictorial form, whether stored on paper or magnetic medium or merely displayed on a screen. Certainly, some of these items can sometimes be regarded as 'commands' to a computer or to the development team; some as 'questions'; some as 'warnings'; some as 'constraints' or 'rules'. But I think the general term 'description' is useful; partly because I want to be able to discuss some considerations that apply to everything produced during development, and partly because it is a valuable discipline to think of everything produced as describing some aspect or part of the software itself, or of its purpose, its behaviour, or its environment, or even of the organisation and progress of the project in which it is being produced.

The business of software development, then, is the business of manufacturing descriptions. When we look back at a completed project we can recount its history by saying what descriptions were produced, in what order they were produced, and how each one was derived from new information or from existing descriptions. The manufacturing operations create new descriptions, and this activity continues until enough descriptions have been produced, of the right quality and content, to satisfy the customer's requirement. We use the computer itself, as well as we are able, as a tool for the manufacture of descriptions: some manufacturing operations, such as compiling an object program from a source program text, are fully automated; some, such as creating a completely new description from new information, use the computer as no more than a hand tool to help in the task of entering and editing the description; some operations are carried out partly automatically and partly by hand.

3 LANGUAGES

The raw materials of descriptions are languages. Just as I am using the term ‘description’ in a broad sense, so I am using the term ‘language’ in a broad sense too. Some descriptions are made from programming language; some, such as statements of the software’s purpose, may be made from natural language; some may be made from a data modelling language, or a finite-state machine language, or a logic language such as Horn clauses, or any useful mathematical notation. Diagrams, such as the data structure diagrams used in JSP, or the action diagrams used in SADT, may also be thought of as being made from linguistic raw material — in these cases, diagrammatic languages.

We can draw an analogy between the raw materials used for parts of mechanical products such as motor cars and the raw materials, the languages, used for the descriptions produced in software development. Motor car manufacturers know that they must use the right material for each part: the windows must be made from glass, not from steel or rubber; the gear wheels in the differential must be made from steel, not from glass or plastic; the body structure must be made from sheet steel. In the same way, a software developer must use the right raw material, the right language, for each description.

The two most important criteria in choosing a language for a description are that it should be understandable and that it should be processable. A description is understandable if it conveys its meaning directly and unambiguously to a human reader. It is processable if it can be usefully processed by a computer in producing other descriptions. For some descriptions, only one of these criteria applies. For example, an initial description of the purpose of a system must certainly be understandable, but usually it cannot and need not be processable. By contrast, an intermediate representation created by the syntax analysis phase of a compiler need not be understandable, but it must certainly be processable.

In those development activities that are often called ‘requirements analysis’, ‘specification’, and ‘design’, we always want our descriptions to be understandable. Where possible, we would also like them to be processable, so that we can use them for mechanical derivation — or, at least, checking — of other descriptions. Unfortunately, because our development tools are still very primitive, the criteria of understandability and processability are often in conflict. We might caricature the formal and the informal development approaches by saying that in the formal approaches everything is processable, but nothing is understandable; while in the informal approaches, everything is understandable but nothing is processable.

3.1 An illustration: Using the wrong language

One formalism on which much work has been done is the entity-relation model and the associated relational data description technique. If we have this formalism available, with developers skilled in its use and tools suitable for its manipulation, we are naturally inclined to suppose that it is the only, or the best, way of describing the subject matter of an information system. Suppose, for example, that our subject matter concerns construction projects in which each project buys certain items from certain suppliers for use in the project. Then we might make an entity-relation model in which we define a relation SPI, as shown in Figure 1. An instance of the relation SPI in which Supplier = S123, Item = 1456, and Project = P789 means that ‘Supplier S123 is a supplier of Item 1456 to Project P789’ the instance of the relation states a particular fact about the real world, and the relational model describes the real world by stating that some set of such particular facts is true.

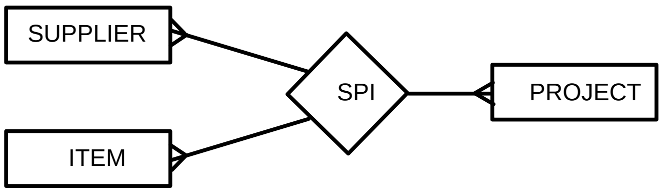

Figure 1

But there is a difficulty in understanding this description. The difficulty arises from the static, timeless, nature of the language chosen for the description, and the dynamic, time-ordered, nature of the reality to be described. Readers may like to experiment for themselves with the understandability of the description. Show the description, as given in Figure 1, to a number of people, and ask each of them: what does 'is a supplier of' mean? It will be found, first, that many different answers are given to this question. And, second, that answers will be given mostly in terms of events happening in some order. For example, one might explain the meaning as 'the supplier has been put on the approved list for this item for this project, and has not been removed from the list'; or as 'the supplier has supplied this item to this project on at least one occasion'; or as 'this item has been ordered by this project from this supplier and the order has been accepted'; or in any one of a hundred such ways.

The central point is that the most understandable description for the user of the system is a description in terms of events, but the software developers offer a description in terms of relations because that fits in with their database technology and with the single language they have decided to use to describe the real world. This, unfortunately, is a common situation in software development. Developers who use Prolog offer a rule-based description of the world; developers who use Lisp see the world in terms of recursively-defined functions; developers who use Smalltalk see the world in terms of objects responding to messages from other objects. To a small boy with a hammer, everything in the world looks like a nail.

4 THE NEED FOR MANY DESCRIPTIONS

For any non-trivial software development, we need to produce many different descriptions. We need to describe the real world which is the subject matter of the system; we need to describe how the system transforms its inputs to its outputs; we need to describe the modular structure of the software itself and the properties of each module; we need to describe the allowable sequences of operations that can be invoked by its users. And within each of these general categories we need to give more than one description.

In some cases, the need for more than one description is well recognised and understood. For example, data modellers recognise that different users of a database system will need different user views of the same data; and specifiers of printed reports recognise that it is necessary to define both the logical structure of a report (what information is contained in the report and how it is organised) and the physical structure (the arrangement of this information in lines and pages). But in both of these cases, the two descriptions are of the same kind: they can both be made from the same language.

The harder cases are those in which we need, for understandability, to produce descriptions made from different languages describing different aspects of the same subject matter. For example, to describe the real world of most information systems, we need to give both static and dynamic descriptions. So we will need to produce descriptions in the language of entity-relation modelling and descriptions in the language of time-ordered events. The problem we face is that we do not have good enough tools and techniques for handling such multilingual sets of descriptions; that is why we often resort to using only one language where we should be using many more. Within each general category of description, we behave like manufacturers who have no techniques for combining different materials and are therefore compelled to make each product from one material only. We like to think of ourselves as engineers, but in this respect we are more like potters or stone masons.

5 MANUFACTURING OPERATIONS AND TOOLS

A notable feature of established manufacturing practice is its repertoire of well understood operations, and of tools with whose assistance such operations may be performed. Parts can be cast, forged, extruded, and pressed; they can be turned on lathes, cut, ground, drilled, formed. planed, and milled; they can be joined by gluing, riveting, and welding. Different operations are required for different materials: plastics and soft metals can be extruded, but steel must be rolled.

We might seek an analogous repertoire of operations on the descriptions that are the parts of software manufacture, and an analogous set of tools to help us to carry out those operations. it seems that an important principle here is to limit our initial ambitions to seeking operations that are simple but of great general utility and, only later, when simple and generally useful operations have been identified, to consider how they may be combined to give the large and complex operations of which program compilation is the most obvious example.

Existing complex tools, such as compilers, tend to present the software developer with a single, indivisible operation. The source language program text is presented to the compiler which produces the required object program in what, from the developer's point of view, is a single uninterruptible step. This approach gives fast and efficient compilation, which is an important benefit. But it also gives an inflexibility that limits the usefulness of the tool in some contexts. It is true that the Ada language and its compilers allow package specifications to be compiled separately from package implementations, and that some compilers allow the developer to request the suppression of code generation, so that syntax errors may be quickly found; but we need much more than this.

Consider, for example, the handling of string variables in Pascal. If we want to maintain compatibility with the standard Pascal that can be compiled by any standard compiler, we must declare string variables as packed arrays and provide our own procedures for the necessary operations of string concatenation, insertion, deletion, substring searching, conversion between integers and strings, and so on. But we might also want to take advantage of a particular compiler's extensions to the standard language in which, perhaps, we can write

```
type
  string12 = string[12];
  string24 = string[24];
var
  s1, s2 : string12;
  s3 : string24;
begin
  s3 := concat('ABC',s2,'XYZ');
   ...
end...
```

and a variety of other similarly convenient statements. Because we can not intervene in the internal operations of the compiler, we are forced to tackle this problem entirely outside the compiler, using a preprocessor of some kind, in which a large part of the preprocessing consists of the same syntactic analysis that the compiler will perform on the preprocessor's output text. The lack of flexibility in configuring the manufacturing operations is a severe disadvantage, even in the handling of this small and simple problem.

6 COMPOSING DESCRIPTIONS

Traditionally, software development has often been regarded as an activity of decomposing descriptions. For example, if we wish to create a program that will print the first thousand prime numbers, we might start by writing the description:

```
begin
  build table of 1000 primes; print table of 1000 primes;
end;
```

and follow this by expanding the statements 'build table' and 'print table'. This is traditional functional decomposition or stepwise refinement. It induces a hierarchical structure, in which earlier descriptions are at a higher level of the hierarchy than later descriptions. Altematively, the same hierarchical structure may be obtained by a bottom-up rather than a top-down approach, in which the earlier descriptions are at a lower level of the hierarchy and the later descriptions are at a higher level.

Whether we proceed top-down or bottom-up, or by some mixture of these, we are presupposing a hierarchical structure in which the form of composition is what we might call 'whole and part' composition. The whole of the description of 'build table' forms a part of the description of the program, and the two descriptions are composed by embedding one within the other, or by procedure call or macro expansion or some similar technique.

But there are other forms of composition which support a more general and powerful approach to software manufacture. Notably, descriptions may be composed in parallel rather than in hierarchical structure. Parallel composition of control structures is provided in many programming languages in the form of concurrent processes and process communication by such mechanisms as message passing, coroutine calls or the Ada rendezvous. Parallel composition of data structures is found in the JSP technique of designing a program structure by composing the structures of its input and output data streams, and in the database technique of composing a global view from many individual user views of the data. Undoubtedly, many other forms of parallel composition of descriptions await discovery and development into standardised techniques available to all software developers and supported by appropriate tools.

7 COMPOSITION AND ABSTRACTION

An abstraction of a description is constructed by simply omitting some parts or aspects of the description. For example, if we have a program text in a procedural language, we can form an abstraction by omitting all the declarations, operations, and expressions except those that refer to a chosen variable. The result will be a description of how the program behaves with respect to that variable, from which we can, for example, determine whether the program can ever attempt to use the value of the variable without having previously initialised it. If the chosen variable is a sequential data stream together with its records, then the abstraction will be a description of the structure of the data stream as imposed or assumed by the program

Abstraction, in this sense, is the complement of composition. By making enough abstractions of a description, we can obtain a set of descriptions from which the original description may be considered to have been manufactured by composition. Abstraction is the analysis that is complementary to the composition synthesis.

Sometimes we will need to use abstraction in this way because the task of composition is simply too difficult to do in a completely systematic and rigorous way. For example, the composition of data stream structures to give a program structure may sometimes demand a degree of invention and creativity; but once the program structure has been described, it is a straightforward and easy task to form one abstraction for each data stream, and so to check that the program structure does indeed constitute a correct composition of the data stream structures. A more extreme example, in which the composition is too difficult to do at all, is seen in the task of satisfying response time requirements in an interactive or real time program. Having constructed what we hope will be a suitable program text, we may form an abstraction for each critical input-output pair and decorate each abstraction with the execution time of each instruction it contains, in order to analyse the time that will elapse between the input and the corresponding output. Obviously, abstraction, like all manufacturing operations, needs support by appropriate tools.

8 SUMMARY

We can regard software development as a manufacturing activity in which we can learn something useful from established manufacturing technique in other fields. In particular, we can see that we need to use many different materials to form many different descriptions of one software product, and many different manufacturing operations on these descriptions. But, of course, this is only one view of the software development activity. Just as we need many different descriptions of the software product, so we need many different descriptions of the software development activity itself; and we need the ability to compose those descriptions into a many-faceted approach to our work.

Computing: The Next Generation
Peter Salenieks ed
pages 203-212
Ellis Horwood Ltd, 1988