Chapter 8 in “CASE On Trial”; edited by Kathy Spurr and Paul Layzell; Wiley, 1990

CASE Tools and Development Methods

Michael Jackson

ABSTRACT

CASE tools are the machine tools for software manufacture. Software manufacture is the activity in which we construct descriptions of the software, its purpose, function, structure, data, program texts, environment, operation, and anything else that is relevant. These descriptions are the objects to be machined by CASE tools. A development method is a prescribed manufacturing plan: it specifies what descriptions are to be made, in what notations or languages, in what order, and by what manufacturing operations.

There is an obvious symbiosis of CASE tools with methods: a tool mechanises the operations prescribed by the method, and the method takes advantage of the tools to prescribe desirable operations, notations, and description types that would not be practicable without them. In the same way, the use of helical gear wheels, pressed steel car bodies, and plastic mouldings are practicable only because the necessary machine tools are available: these parts can not be made by hand tools.

The earliest and most dramatic example of this symbiosis in software development is the method of writing programs in ‘high- level’ languages: no programmer would write COBOL or Ada or Pascal if the next step were a translation, by hand, into machine code. The availability of compilers makes the methods and their languages practicable.

Unfortunately, the success of compilers and high-level languages has created a traditional culture in software development from which we are still trying to emerge. This culture views the development process as the creation of one description, (of the computation the software must perform) in one language, (the programming language) to be machined in one operation (compilation). We are scarcely yet emerging from this culture into a new culture in which the developer uses many descriptions, expressed in many languages, machined by many different operations.

Today’s software development methods have grown up in the traditional culture, in which the only machine tool is the compiler; and it is these methods that today’s CASE tools support. Much attention has been paid to WIMP interfaces and to development databases and repositories, but relatively little to the possibilities of new and more powerful methods that a new generation of CASE tools could bring. This is the challenge of CASE.

THE NATURE OF SOFTWARE ENGINEERING

Software developers have aspired to the status of engineers at least since the NATO Conferences of 1968 and 1969. We look with admiration at the established disciplines of aeronautical, chemical, electrical and civil engineering, and we hope that one day we, too, will merit the title: we, too, will be engineers.

The aspiration is virtuous, but its meaning is not clear. Is software engineering to be one more discipline to set alongside those other established disciplines? Or is it a collection of new disciplines, specialised by product or perhaps by technology? Should we have Ada software engineers, C software engineers, and COBOL software engineers? Or should we have financial software engineers, switching software engineers, and embedded software engineers? Or perhaps database software engineers, HCI software engineers, and distributed software engineers? If there are to be specialisations, how will the specialists work together? If not, what will every qualified software engineer be expected to know?

There is a more fundamental distinction, too, between software development and traditional engineering. In traditional engineering there is a clear separation between specification and design

on the one hand and construction on the other hand. For a civil engineer there is a clearly marked point at which the calculations and the drawings cease, and the pouring of concrete and the cutting of steel begin. For a software developer there is no such clear separation. The software developer works throughout with descriptions: descriptions of computations expressed in COBOL or Ada; descriptions of problem domains expressed in temporal logic, in sequential processes, or in Entity-Relation graphs; descriptions of software structures expressed in module or procedure hierarchies or flow diagrams; descriptions of customer purposes expressed in natural language or decision tables.

Software Descriptions

In the earliest days of software it was supposed that the only explicit description needed was a description of the computation --- probably expressed in machine code. I remember, as a schoolboy, seeing Christopher Strachey's draughts-playing program for the 'Manchester machine': columns of numeric machine code written in pencil on a page of lined foolscap. No doubt Christopher had made all kinds of notes and doodles when he thought about the problem, but they formed no part of the software product as he saw it.

Over the years it has become clear that we need explicit descriptions of much more than the computation. Because the technology of the execution machine has become complex, we need job-control procedures and database schemas and screen maps. Because the software has become much larger we need all kinds of behavioural and structural descriptions, operating instructions, and maintenance manuals. Because the project is longer and more expensive we need requirements statements, problem domain descriptions, specifications, feasibility studies and software procurement contracts. All of these are descriptions that the software developers must create and manipulate in various ways. They are related to one another because, above all, they describe aspects and parts of the same software product and its purpose and environment from many different points of view, using many different abstractions.

Software Development Method

A software development project can be regarded as a manufacturing project. A large set of descriptions is to be manufactured. Some will be delivered to the customer in the form of executable code, control and definition statements for system software, operating manuals, and other descriptions that the customer may demand. Others will be used to help the developers in their work: design documents, intermediate descriptions such as input to parser generators, test rigs and sets of test data and results.

In this context, a method is a manufacturing plan. Whether adopted ready-made, tailored for the project, or simply left unspoken and implicit in the successive decisions of the developers, the chosen method answers the questions:

1. What descriptions must be made?

2. In what order must the descriptions be made?

3. In what language must each description be expressed?

4. By what manufacturing operations and using which previously made descriptions must each description be made?

Existing Methods

Existing methods have grown up within the existing software culture. Essentially, this culture has been based on the success of compiler writers and programming language designers. It may be characterised, with only a little exaggeration, as a culture in which we expect to develop software by making only one description, using only one language, and applying only one manufacturing operation. The one description is a description of the computation that the software must perform. The one language is the programming language. And the one manufacturing operation is compiling.

Of course, this is an exaggeration. But the effects of the compiler culture are very evident. A motor car is a far less complex artifact than a significant software system, but its manufacture involves many more distinct parts, distinct materials, and distinct manufacturing operations than are found in a software development project. Only a very large software project would create a thousand distinct descriptions; not even the simplest motor car could be built from so few parts. A typical software development team might use a dozen different languages, but motor car manufacturers use many more materials than that. Automobile engineers use many kinds of steel and plastic, glass and rubber, aluminium and iron, nylon and brass. They understand that the windows must not only be made of glass, but of toughened or laminated glass; that the crankshaft must not only be made of steel, but of forged steel.

Most dramatically, and most relevantly when we consider CASE tools, motor car manufacture involves many kinds of operation: casting and forging, milling and grinding, moulding, welding, drilling, reaming, turning, pressing, extruding, gluing. All of these are needed to work the different materials and to put the resulting parts together. The software developer seems to have a far smaller repertoire of available mechanised operations.

By these measures, software development is a primitive technology. Lacking highly developed tools, primitive craftsmen restrict themselves to making objects that can be made of very few parts using very few materials. In the same way, because our tools --- other than compilers --- are undeveloped, we instinctively minimise the number of descriptions to be manufactured. We do not want to find ourselves making many descriptions if each description is very expensive in time and effort and we have little or no technology for putting descriptions together. We therefore select for manufacture a small set of descriptions intended to serve as many purposes as possible: typically, a description will be intended to serve as an intermediate product in the development process, as a maintenance document, and as a medium of communication within the development team and between the team and the customer. Because there will be only a few descriptions, we must put as much as possible into each description, emulating the young man of Japan:

There was a young man of Japan Whose limericks never would scan. When asked why it was, He replied ""It's because I always try to get as much into the last line as ever I possibly can"

So we are not content to show only module hierarchy in a module hierachy diagram: we try also to show the parameters and results, and even a curiously stunted hint of control flow by marking module connections with diamonds or grouping them with 'looping arrows'. In the same way, we are not content to show only entity relationships on an entity relationship diagram: we try also to show aggregation and classification, and mutual exclusion among relationships by marking mutually exclusive connecting lines with arcs.

CASE TOOLS AND METHODS

The relationship between CASE tools and software development method is symbiotic: a tool mechanises the operations prescribed by the method, and the method takes advantage of the tools to prescribe desirable operations, notations, and description types that would not be practicable without them. In the development of CASE tools until now, the emphasis has been almost entirely on the first half of this symbiosis: we have many tools to mechanise manual methods, but few

advances in method based on the availability of powerful tools. Ironically, the most notable example of the latter is in the very basis of the compiler culture. Today, our standard method of describing a computation is to write a description in a 'high-level' programming language, relying on the compiler to translate it into executable code: if we had to carry out this translation by hand we would do it rarely, if ever, and would usually prefer to write directly in machine code. Just as the language makes the compiler necessary, so the compiler makes the language usable.

In thinking about CASE tools we should pay much more attention to understanding the limitations of our present methods, and how they might be loosened or even removed by the availability of appropriate tools.

Managing Many Descriptions

One aim in CASE tool development should be to make the use of large numbers of descriptions thoroughly practical. Our chief tool for mastering complexity is the separation of concerns, and this separation is reflected in the separation between the description of one concern and the description of another. This means that we will need to manage large numbers of descriptions, controlling and recording the relationships among them.

These relationships will be of many kinds. One description may use terms that are defined in another; one description may rely on the truth of another; one description may add to the constraints contained in another; one description may define a domain that another description must cover completely; one description may define a domain that must enclose another description.

When the number of descriptions is small, it is possible to make serious errors by misunderstanding relationships among descriptions. When the number is large, managing these relationships becomes an important CASE tool function.

Derived Descriptions

Many of the descriptions used in software development are secondary, in the sense that they are mechanically derivable from others. The situation is analogous to the derivation of many different reports from a database, and requires analogous tool support. There are already some tools that can produce stereotype natural language descriptions from formal requirement and specification descriptions.

Many Simpler Languages

A direct consequence of being able to make many more descriptions and to use our tools to put them together is that we will be able to use languages that are better because they are simpler. Each primary description that contains new information should be in a well-understood simple language, at least up to the final manufacturing stages in which we must produce executable code in the languages provided by the execution environment.

A complex language exacts many penalties from its users. It is harder to write correctly, because the meaning of what is written is relatively obscure. For the same reason it is harder to read and understand. And, very importantly, descriptions written in a complex language are harder to manipulate. They are more fragile because there are more interconnections, explicit and implicit, among the parts of the description, and any manipulation of the description runs the risk of damaging these interconnections.

For this reason the computer science textbooks are full of well- understood operations that can be performed on graphs, on regular expressions, on formal grammars, on formulae of propositional and predicate calculus, on finite-state machines. But there are few that can be performed on Ada or

COBOL program texts, or on module hierarchy diagrams decorated with parameters and results and control flow symbols. To manipulate these more complex descriptions we must resort to manual operations, which are slower, more expensive, more cumbersome, and even more error-prone than the task itself implies.

Many Manufacturing Operations

Software development needs a large repertoire of manufacturing operations. This repertoire should be built up on a base of well-understood elementary operations: it is a mistake, born of impatience, to devise elaborate operations directly instead of constructing them from existing elementary operations. A few simple elementary operations are outlined in this section; the next section discusses operations that compose descriptions.

Editing Operations

Clearly, editing operations are needed to allow entry and modification of descriptions in every simple language that is in use. We are already reasonably well-served in this respect, since these operations provide a machine-based implementation of what was previously done with pencil and paper.

Manipulation Within One Language

Simple languages provide repertoires of operations by which a description expressed in the language can be derived from another description expressed in the language that is equivalent or is related in some other way. Here are some illustrations:

From a general Boolean expression we can derive an equivalent Boolean expression in minimal disjunctive normal form — that is, in the form x1 or x2 .. or xn, where each xi may contain nots and ors but no ands, and the number of xi is as small as possible.

From a finite-state machine F that accepts any sequence in the set A we can derive another finite-state machine F' that accepts any sequence that is not in the set A.

From a graph G we can delete edges to derive a 'spanning tree' T (that contains exactly one path from any node to any other).

From a non-deterministic finite-state machine F, that has some states in which there are two or more different outgoing transitions on the same input, we can derive an equivalent deterministic finite-state machine F' in which no state has different outgoing transitions on the same input.

Translations Between Languages

Sometimes it is desirable to use different languages for different descriptions although they have the same theoretical expressive power. For example, JSP trees are equivalent to regular grammars, which are equivalent to finite-state machines. It may then be useful to translate from one language to another.

Abstracting a Description

Abstraction is the process of taking away: to make an abstraction of a description is to make another description in which something has been left out. Suppose, for example, we have a description of a finite-state machine F in which the inputs are a, b, c, and d. And now suppose that we want to describe the same set of input sequences as before, except that we leave out the transitions on b: we are not changing the set of sequences by stipulating that b events never happen, but simply giving a

more abstract description in which we do not describe occurrences of b events because we do not see them. Then we can make a new finite-state machine G which is the abstraction of F by leaving out those transitions.

We can perform similar abstraction operations on descriptions expressed in almost any well-defined language. For example, we can leave some entity sets (and their relations) out of an ER diagram; we can leave the truth value of some propositions out of a Boolean expression; we can leave some data flows out of a data flow diagram.

Composition Operations

In a software development utopia we would be able to separate concerns as fully as we wished, and would never need to put them together again explicitly. Each fragment of requirement or specification or design description would be translated into a fragment of executable code, and the execution environment would put them together for us. In a limited way, in certain contexts, this is possible. For example, if we define a set of concurrent processes that communicate by message passing, then in some environments the communication is handled automatically at run time, and we need never explicitly compose the concurrent processes into a single program.

But more often separation must be followed by composition. Having divided to conquer, we must reunite if we wish to rule. Having separated one problem into many subproblems, we must compose their solutions to give one solution.

There are several factors that force composition on us. One is the limited nature of the composition available in many execution environments. Another is the need for efficiency. Suppose, for example, that we need to evaluate two functions defined over a large data structure --- perhaps a database, or a large structure linked by pointers in a Pascal program. Then separation of concerns leads us to describe two traversals, one for the evaluation of each function; but efficiency demands that we compose these two traversals if we possibly can, to exploit the single traversal for both evaluations.

This kind of composition arises more often than we may be inclined to recognise. Because of our upbringing in the compiler culture, we expect that the design and construction of software that does several things at once will be an activity in which we ourselves will be forced to think about several things at once, without resorting to explicit separation. The expert programmer is one who can hold many requirements simultaneously in mind while writing the program that satisfies them all. That programmer is performing mentally the composition that might be done more reliably if it were explicit and mechanised.

One simple example of such composition occurs in JSP. The JSP method for designing a sequential process is to describe the regular grammar --- or structure --- of each input and output stream separately and explicitly, and then compose these structures to give a program structure. This composition can be mechanised, and has been mechanised. We need many more composition operations of this general kind in our repertoire.

MOUNTAINS AND MOLEHILLS

For some people, the views I have put forward here may seem to be making mountains out of molehills. Do we really need to use many languages, many descriptions, and many operations? Have we not already devised reasonable methods for software development, and now need only to provide good supporting tools for those methods with well-engineered human interfaces?

Certainly there is a sound methodological principle that the method and the task should be in proportion. Faced with the task of writing the ‘Hallo World’ program, we should not be separating concerns and manufacturing many descriptions in many languages: we should simply write:

```
Program P; begin writeln('Hallo World') end.
```

But this is not usually the task that confronts us. Software development tasks are complex, and our record of achievement is not one that should lead to complacency.

Do We Really Need Computer Science?

Many years ago two prominent American software methodologists gave a course which they advertised in the following terms:

'Building workable sotware systems is not a "science". So- called computer scientists try to convince us that our systems are really directed graphs or n-tuples of normalized forms or finite-state automata ... their pronouncements are more relevant to Zen than to the no-nonsense business of building useful, maintainable programs and systems. ...

'In place of academic approaches, we require a set of hard-nosed practical methods that deal squarely with the fallible nature of project personnel, users, machines, maintainers and the development process. Such a set of methods is called Software Engineering. ...'

This view seems to me to be very wrong-headed indeed. Certainly software development is a human activity and has many aspects --- social, managerial, psychological and economic --- that lie entirely outside the realms in which computer scientists work. But it is also a technical and even mathematical activity that can and should exploit what has been learned about computer science. A program in a procedural language can be developed and understood much better by someone who knows about finite- state automata, and the traversal of a database structure by someone who knows about directed graphs. These and many other languages should be familiar to all developers, and their manipulation supported by the tools that developers use. These are necessary, but certainly not sufficient, conditions for advancing the state of software development and the quality and usefulness of CASE tools.