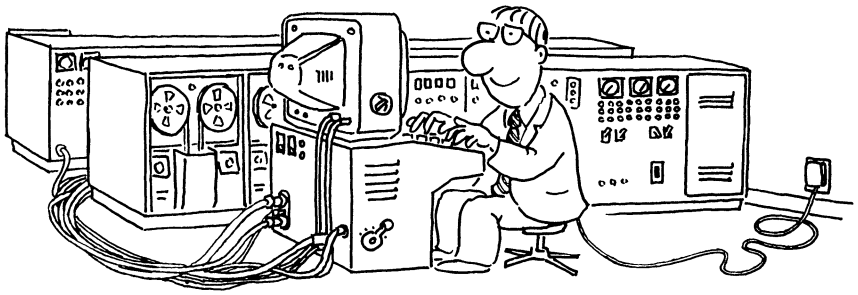


7 System development environments

PROFESSOR ERIK SANDEWALL

7.1 Introduction

There is a need to bring about a synthesis of knowledge engineering techniques or expert system techniques on the one hand, and conventional software engineering on the other. This arises from the limited market and limited range of applications for the separate expert system tools that we see today. There would be much more applicability if we could embed intelligent facilities of the type that expert systems provide into conventional software.

My recommendation for how to bring that about is that we should try to *unpackage* expert systems. Instead of buying or building fixed packages which promise to serve as a shell for an expert system, we should look inside them to see what software engineering techniques are being used there. We should then try to apply the same software engineering techniques for more mundane purposes.

In this essay I want to show how that recommendation can be carried out, in the particular context of office systems. I will do this not only as a pedagogical exercise, but also as a report from actual research projects. I have outlined the steps which we have gone through in our own research. We started as an artificial intelligence group quite a number of years ago and in the middle 70's a large part of our group switched its focus to study office systems. So we have

been working for a number of years on office systems but from the background and the perspective of artificial intelligence.

7.2 An AI perspective on office systems

The standard computer-based office services are certainly familiar from the literature, and many people today use them in their daily business. There is a need for text editing and handling structured data such as forms, and for various services such as computer mail. The computer calendar is often described and often implemented but apparently not so often used. It is of course an attractive concept in principle. You will also find a need for several more specialised services in each office application of computers, such as mailing lists and various aspects of accounting.

During the first phase of our research, we allowed our software to grow and built up the software that provided these various services in our own working environment. While we did so, we also gradually tried to generalise on the software that was being used. One of the steps that turned out useful was to build a general purpose editor. We called it ED3 because it operated on *tree* structures. (The Swedish language doesn't have the 'th' sound, so 3 and tree sounded similar to us) (Strömfors and Jonesjö 1981).

The idea in ED3 is that, instead of having a text editor, you have an editor which operates on a tree. The user decides how he wants to organise his data in terms of this tree. For example, if you have a large document which is conveniently organised as chapters, sections, and so on, then you would let each of those subdivisions be one part of the tree. If you have a program which has a block structure, then you have a similar natural tree structure in your data.

So one of the two basic parts of the ED3 editor allowed the user to change the structure of a tree: to add, delete, or move whole branches of a tree. The other basic part allowed him to do text editing in the leaves of the trees. Each leaf, or terminal node, of the tree was supposed to be a piece of text. Gradually, as we used it, we recognised that there was a need for other kinds of leaves. Sometimes it was convenient to let one leaf be a figure, a graph, a picture, a table or some other collection of structured data. The number of variants and extensions of this tree editor grew. There was EDG for graphics, EDF for forms and so on, and ED* for the most general system.

Similar ideas have emerged in other places, and are presently becoming popular under the catchword of 'outline processors'.

These general purpose editors capture one key idea in what we think is the key software engineering principle used in AI – always try to write general purpose software that applies to tagged data. Instead of having a number of separate programs, each of which applies to one specific kind of data, we attempt to write one program which is able to cover all those needs, at least when those programs have a similar structure. We organise the data so that local tags specify which specific kind of data occurs in each position.

This might seem like a very simple observation, and it is a very simple observation, but at the same time it does run counter to some of the underlying practices and underlying principles of conventional programming languages. When you start out to write a program in COBOL, PASCAL, or any other conventional programming language, you first write declarations of the data you are going to use. After that, you write procedures which operate relative to the declarations. Your program is therefore tied down to those particular data structures that are expressed in the declarations. If you have a number of different applications which require essentially the same procedures, but different data structures, this programming discipline forces you, or at least strongly encourages you, to write several programs with a similar procedure content.

The alternative is, not to begin by writing the declarations. Instead write the program first, and organise it so that the declarations are variable. This is often an issue which is brought up in the debate about conventional programming languages versus languages such as LISP or PROLOG. The proponents of the classical languages complain about the lack of declarations in, for example, LISP. The answer is that people in the LISP culture certainly see the need for descriptions of the structure of data. That facility is so important that it should not be restricted to be just a constant for a program. It is important to be able to write programs which can accept data descriptions, i.e. the kind of information that you put in declarations, and which can decode those data descriptions. Instead of having fixed declarations *in* the program, we should be able to treat the declarations *as data structures* and let the program decode them.

We then proceeded to examine the specialised services in our office environment. We found some services which could be satisfied in the context of that tree structure editor or outline processor. We took the opportunity to implement various services as far as possible in the uniform environment rather than build special software for it. There were also some applications in the office environment that required

more than the general purpose editor could provide. This led us to implement a number of tools for these particular classes of applications (Sandewall 1982).

One very obvious tool was a forms handler, or forms management system. It was of course an example of the principle of writing general software rather than using declarations. A second tool which built on the forms management tool was an information flow handler. The need for this tool was first seen in an application in the university hospital, namely the information traffic between the patient ward and the laboratory for chemical analysis of samples from patients.

Essentially, the patient ward issues purchase requests, internal requests for analysis for each sample that they wanted to have processed. In the daily work of the hospital, this request was issued using forms on paper. The very simple observations we made were that lots of forms are used in organisations, and when you complete a form you almost never proceed to put it in the drawer of your desk. Rather, when you have completed a form you send it on to somebody else. That other person is likely either to add some more information to it and send it on again, or he or she will leave it on his desk for a while and later do something to it, more or less immediately.

If you trace the itineraries that are taken by forms in the organisation, you will see certain standard paths, but also cases where a form goes on a unique route. The degree of standardisation of the itineraries depends on the character of the organisation and the topic that the form addresses. In the particular cases we looked at first, there were very strongly standardised paths for the forms.

They went from the ward where the patient and the doctor were located to the chemical laboratory and from there back to the ward, as well as to central files, to accounting, and so on.

A reasonable way of describing this data processing service was therefore to draw a flow graph which showed how these forms, or these packages of information, were created in one part of the organisation, and how they were sent on from workstation to workstation or from person to person, and how finally the information ended up in data bases which were more or less longlived. For the purpose of a systems designer, and for the purpose of dialogue with the end users, it was very convenient to use an *ad hoc* graphical language to describe the flow paths. We used it for specification work. We also implemented a tool whereby we could input such a flow graph into the computer, and interpret it in prototype style in order to show what the intended new services would be like when imple-

mented on personal computers. After this prototype had been debugged in cooperation with the users, it could be rapidly transferred to everyday use.

The software for supporting the information flow is also an example of another principle that we inherited from our previous AI activities. In AI research projects there are a large number of cases where people write special purpose tools which operate on a special purpose language. They have identified some aspect of the application domain or some aspect of a technical system which lends itself to a concise description in a *special purpose language*. Then they implement an interpreter or a compiler for that language. Indeed LISP has been characterised as a very high level implementation language. It is a very convenient tool if you want to implement a number of such special purpose languages, and enable them to work nicely together. We brought the same practice into the more mundane domain of office systems.

Another such tool, which we also found quite convenient, was dialogue software supporting *dialogue transition networks* (Hägglund 1980, Hägglund and Oskarsson 1975). You can think of them as a tool for writing adventure games. There seem to be quite a number of applications where the user needs to navigate in a dialogue space. Each point in the space, or each node in the dialogue transition network, consists of one interaction. In each node of the network, the user is presented with one printout, or one screenful from the computer. Also for each node the allowable inputs from the user were defined. The present node and the user input together determine the next node to be used in the dialogue.

This dialogue pattern re-emerged from time to time. Of course if the network is very simple you can implement it easily in any programming language and environment, but quite often we needed to have a fairly large number of such nodes, and a large space for the user to navigate in. Then it was convenient to have a tool where you could interactively build the network and interpret it. This brought several advantages, including the advantage of rapid prototyping. We would let the system designer and the end user sit together at the terminal. The end user could try his hand (or her hand) navigating in the dialogue network to get things done. If some aspect of the system seemed to be counterintuitive or difficult to understand, then the system designer, who was the other person at the terminal, could go in and very rapidly modify the network and let the user try again.

Yet another tool, and the last one which I will describe, was a tool for distribution of modules to subsystems (Sandewall *et al.* 1981). This need arose especially as an extension of the information flow service. In the organisation we looked at there were a number of workstations, and hence a number of users. There were also a number of separate services, each of which could well be described as an information flow. So we had a kind of a matrix structure for the software. Each user, or each user workstation, was affected by a number of the services, and each service affected several workstations.

Because of the 2-dimensional structure of the problem, we had a difference between how we wanted to cut the software at development time, and how we wanted to cut it at run time. At development time it was reasonable to develop each of these information flow services, one at a time. We could take a bird's eye view of how one particular kind of information package would flow through the organisation. At run time, however, we wanted instead to have the software which was relevant for one particular user in his or her workstation, or in his or her workspace in the central computer.

The appropriate tool for the development phase, therefore, was a *development environment* which contained one information flow model, including all the components for all the different users which were involved along its flow path. When the model had been debugged, we wanted to decompose it into the pieces which belonged to the individual workers along the path, and distribute those pieces to the end user environments of the respective participants. Conversely, since each end user environment needed contributions from several flow paths, it needed to receive contributions from several such development environments. This defined how the general purpose module distribution system should work. It should allow one to generate a contribution from one environment and send it to another environment where it could be nicely integrated into the right places.

In order to get that principle to work properly we needed to pick up yet one other key method from artificial intelligence software techniques. Namely, it was necessary to store data, and descriptions of data and procedures, in an integrated way in a data base. In traditional programming we are used to storing data and programs as text files, which means that manipulation of the procedures and manipulation of special purpose languages becomes fairly difficult.

What we do instead in the AI style software technology is to define a data base in which different kinds of formulae can be stored, and where executable procedures are just one kind of data.

This is also the kind of structure that we needed in these development systems or development environments. In them, we needed to manipulate forms descriptions, information flow descriptions, descriptions of end user environments, descriptions of the topology of the total data processing system that one was operating against, and so on. Based on these various special purpose descriptions, this environment must be able to extract the structures that could be sent out to the target workstations for run-time use.

A common idea in this list of key methods is that we used a system development environment, a kind of computer aided design system for software in order to support the various tools.

7.3 Some lessons from AI-based software technology

This section looks retrospectively at the experience that was gained during this period. What did the software look like? What were the bottlenecks? What were the difficulties which had to be overcome in later generations of the system? Secondly, I want to look over these methods from several ordinary points of view: the point of view of the programmer, the point of view of the end user and so on.

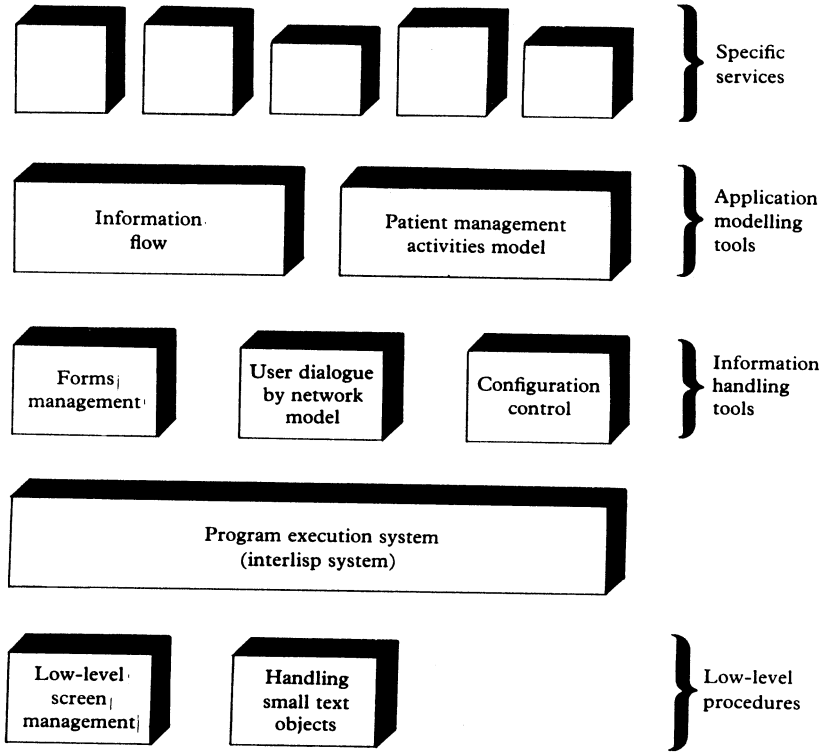
First, let us consider the software that was built. We thought while we were doing the work that we did a fairly decent job as programmers. Things were done according to the best prescriptions of structured programming, project management and so on. Yet after a number of years, of course, the accumulated set of software was fairly large and fairly difficult to extend and to work with. We went back to try to identify where the complexity arose. Which were the parts of the system that were difficult to develop and to maintain? For this analysis, we needed a model, shown in Figure 7.1, of the total structure of the system.

At the heart, we had the program execution system, which was a variety of the INTERLISP system. This provided the database in which descriptions and programs could be stored. There were various low level routines subordinate to it for executing services such as screen management, handling small text objects and so on. Immediately above the level of the program execution system, we had a number of information handling tools, such as forms management, command dialogue, configuration control, sending things to other environments and so on. Above it again, we had the level of

application modelling tools such as the information flow tool that I described, and a few other, similar ones. Finally, on the highest level we had the specific services, so, for example, each particular use of the information flow model would be located on the top level.

In this software architecture, the complexity was quite clearly located on the level of information handling tools. The extent to which it was concentrated there was very surprising. The program execution system was given and did not offer any problems. The lowest level consisted of a few individual routines which could be written easily by any competent programmer, e.g. a competent undergraduate in computer science. Given a specification, he would go away and write the program and come back with it completed. The application modelling tools were also surprisingly simple. In the case of information flow, for example, the actual work of implementing the information flow model, given that the forms management system and the other underlying tools existed, was quite small

Figure 7.1: Software architecture for Linköping Office Information System



(Sandewall 1979). This has later been confirmed in the continuation of the project, where the idea of information flow has been transferred to software companies who built such information flow handlers on top of their forms management software and database software. Again the same story repeated: it was a small piece of software work. All the complexity arose on the level of the information handling tools. We therefore took as one of our goals in the research project to look over that level, as well as the total structure of the software, in order to reduce the complexity.

We now think the key to dealing with that complexity is to recognise an essential similarity between many of the office services. Essentially, in the office environment again and again we are building special purpose "mumble management systems" where *mumble* is a variable. We write telegram management systems that we call computer mail systems; appointment management systems that we call computer calendars; management systems for managing duties and promises which we call tickler files, and so on. The key idea in each of these is that you have a system which allows you to store data in the computer, and to put it in and bring it out piecewise. So another good term would be *systems for piecewise operations* on data – put the data in, move it around a bit and print it out in various projections and selections. In fact, this general framework also covers some types of software which are not usually thought of as office software, for example, computer aided design systems.

The first observation was that we have a lot of mumble management systems around. The second observation was that the essential similarity between many of those services are hidden by the terminology: not only by the names we assign to these systems, but also because there are a lot of unnecessary differences between their user interfaces. The end users repeatedly complained that the sets of operations were difficult to remember because "the same thing was called different names in different systems". This indicated an awareness and a perception from the end users that there was really "the same thing" in different software. So sometimes you have to write PRINT, sometimes TYPE, sometimes DISPLAY, and sometimes SHOW in order to see something on the terminal. If you want the information on a printer you say LIST or PRINT or maybe OUTPUT. Entering data is called ENTER or CREATE or NEW, or maybe you go into the editor even if you have new data. DELETE can be called REMOVE and I think several of us have sometimes searched frantically for the magic word that you need to get out of a

piece of software. Should you say EXIT or STOP or QUIT or OK or BYE or whatever?

Clearly some kind of normalisation is needed here. Ideally, we should not have several separate systems in parallel to each other at all. Notice the reason that we have all the different names for the "same" operation is not that somebody intentionally created those alternative list of names in order to confuse the user. The reason is that we actually have parallel sets of software doing similar things. There is one whole program which does the operations in the first column of Figure 7.2, another program which does the operations in the second column, and so on. Instead of that software architecture, we would like to have a single piece of software which supports a matrix organisation, where the different operations with standardised names are listed in one dimension: PRINT, LIST, ENTER, EDIT and so on. The other dimension lists the various contexts or the various data environments. Ideally again, we would like to write the procedure for each of those operations just once.

In practice, that is not entirely possible. But there are going to be some differences (although perhaps marginal ones) between how you print out one entry in the mail system, and how you print out one entry in the address directory. The differences may depend on the structure of the data, but they may also depend on the character of the

Figure 7.2: An interactive command thesaurus

Application				
1	2	3	4	5
PRINT	TYPE	PRINT	DISPLAY	SHOW
LIST	PRINT	LIST	COPY	OUTPUT
ENTER	ENTER	CREATE	NEW	EDIT
EDIT	UPDATE	EDIT	CHANGE	EDIT
DELETE	REMOVE	DELETE	DELETE	DELETE
...	...	...	...	...
EXIT	STOP	END	QUIT	OK

application. So just having one procedure for each of the commands is not rich enough. What we can do instead is to say that for each context C and for each operation OP there should be one operator definition procedure. Often the operation definition procedure for similar contexts C or similar operations OP will be roughly the same.

In the worst case you have one procedure for each combination of context and operation. In practice we can rationalise it. In some cases several operations require a similar procedure and then we can form the abstraction from them. In some other cases the procedures associated with two different operations differ in some details. Then what you want to do is to write a joint procedure which covers the similarities, and also to have small attachments to those procedures which account for the differences between the operations.

Another thing about this repertoire of standardised operations is that many of them involve a traversal. In many cases the data structure is some kind of tree. Many of the standard operations make a scan over that tree and do something locally in each node. As an extension of the abstraction process, many of the operations can be characterised by a canned procedure which just traverses the tree, plus information which is specific to each operation. The add-on information specifies what to do in the leaf of a tree, what to do if the sub-tree you are trying to traverse does not exist and so on. For example, if you make a data access and you are trying to access a piece of information which does not exist, you will just give up. On the other hand, if you try to put the information into a part of the tree, and the place where you are trying to put it does not exist, then it should be created and inserted into the tree so that you can put your data there.

The general framework was therefore to identify those operations which had the character of tree traversal, and to form a joint abstraction for them. We then associated the general purpose tree traversal procedure with that abstraction. With the lower level nodes in the abstraction tree for operations we associated the detailed procedures which specify how to handle the leaf, and how to handle a missing daughter in the tree, and so on.

Using the design principles that I have now outlined, we were able to rewrite the editor level of the system. After a number of iterations over complex systems, we were able to build a compact system which served quite well in the office environment, and which implemented these key methods which we carried with us as our methodological baggage.

Clearly none of the services which I have described contains any particular intelligence. If we look back on the figure of the structure of the software (Figure 7.1), the logical place to put the kind of intelligence facilities that are used in expert systems and other AI systems is on the level of application modelling tools. Things like rule-based reasoning, and maybe also truth maintenance logically belong there. We therefore thought it was encouraging that what we did on this level was relatively simple and turned out to be easy to implement. That showed that in terms of the complexity constraints of the total system, there was a lot of available "space" there. After we had brought everything into good order up to the level of this information management system, we were able to provide the standard office services very easily. We felt we had plenty of room for adding more intelligence.

This observation provides the key to the strategy I want to propose for introducing AI techniques and expert system techniques into conventional data processing. If you feel that today's expert system packages do significant things for you, if they are worth their price and if they are worth the effort of introducing them, then fine, go ahead. If it is instead your feeling that the expert system technology is not yet mature for the applications you have in mind, then maybe you wish to wait a few years before putting it into wider application in your own operations. At the same time you may be looking for ways of positioning yourself, and positioning your software so that those same AI techniques can be introduced easily and flexibly when the time arrives.

The style of software architecture which is described in this paper may then fit in nicely. It has a lot of merit in itself, so it is worth doing quite apart from whether you are going to use AI later on. But, at the same time, it puts you in a much better position for introducing AI techniques later on, than if you pursue the traditional COBOL-based or similar software engineering methods.

7.4 New approaches to systems development

In the final part of this essay I want to review the proposed software strategy from a number of additional viewpoints. After all, if we introduce a new software strategy we must know what we are doing. We must know how it affects everything else that is important in the operation of data processing.

Software engineering. Let us first consider how this strategy can be

seen from a general engineering viewpoint. The term "software engineering" has been introduced in order to suggest similarities with other branches of engineering, and to suggest that this is a branch of engineering (which is not entirely obvious to other engineers). It has also been introduced in order to encourage us to borrow ideas and principles from other branches of engineering.

One of the standard engineering methods, which is very often recommended, is modularity. It is considered to be a good idea to recognise recurrent phenomena and recurrent needs, and to package solutions for those needs into a standardised format. Usually modularity has been identified with the use of subroutines or other similar things, which means that modules are components which can be assembled into larger systems. Now I claim that, in engineering in general, this is not the only important aspect of modularity.

One example of subroutine style modularity is shown in Figure 7.3. Here we have built an aeroplane from standardised components. Of course, the components do not always fit the purpose precisely, but at least something has been built.

Figure 7.4 shows an example of the other kind of modularity, which is sometimes referred to as a *universal tool*. Here the designer of this tool has identified one class of applications in the workshop and has built a tool which supports whatever is common

Figure 7.3: Subroutine style modularity

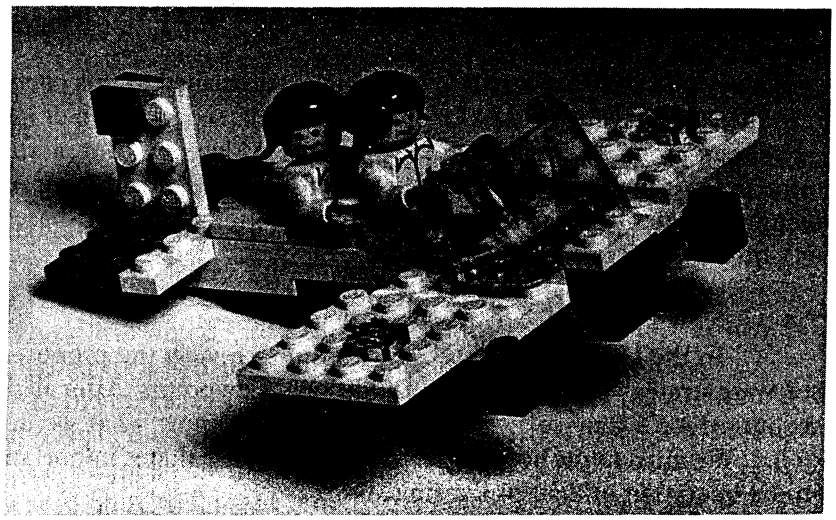
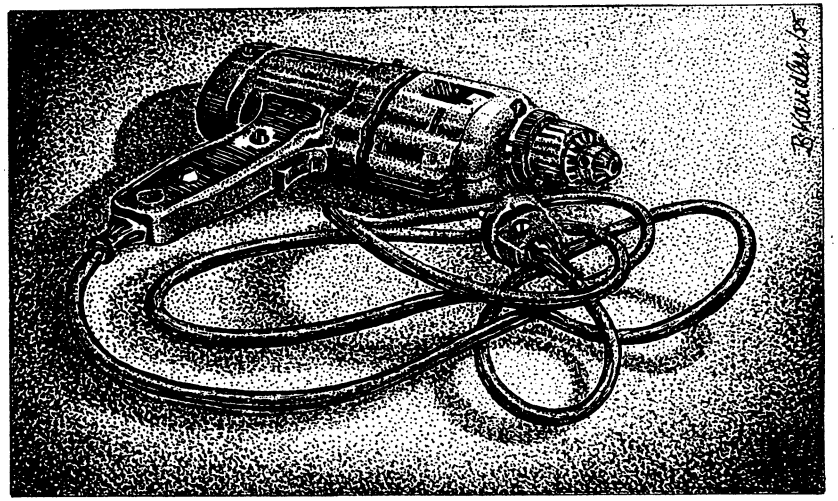
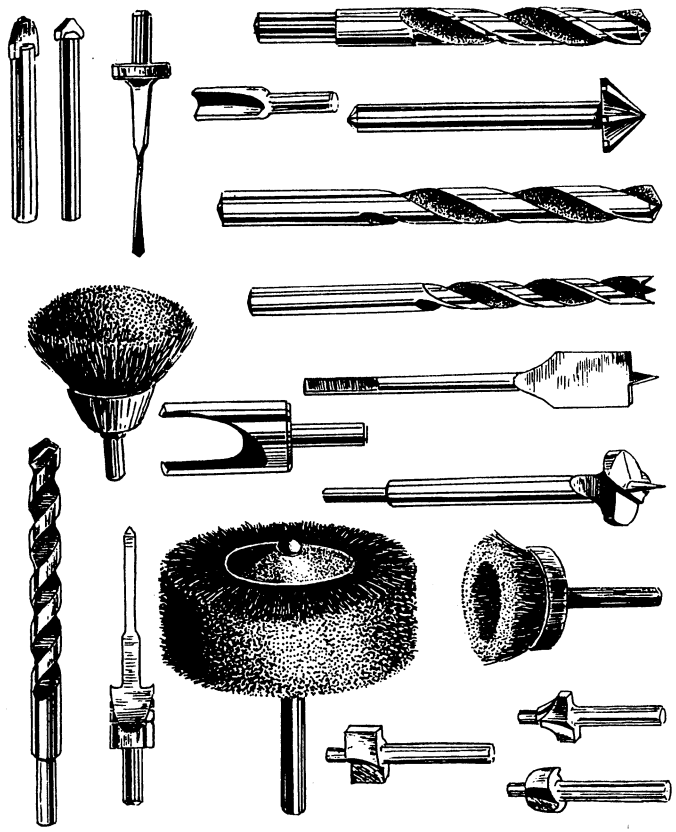


Figure 7.4: universal tool





for that whole class of application. The common tool here, the universal tool, contains an electric motor, and a cord, and controls so that you can turn on and turn off the motor. A wide flexibility is guaranteed because we can make various attachments to the chuck: a drill, a polisher, and so on. Almost everything that is significant for the use of this tool is concentrated in the exchangeable parts. On the other hand consider the total cost of the system: the major cost is in that part which is common. Another thing which is very important is that there must be a device (in this case the chuck) which allows you to easily and safely swop the changeable parts.

What has been shown here is *not* the same kind of modularity as we had in the previous case: it is not the modularity which arises because you have standardised components. Component modularity also arises because we may have used, e.g., a standard, off-the-shelf cord in the production of the basic tool. But the significant modularity, for the purpose of this discussion, is the modularity which is seen by the owner of the tool: the end user who is able to easily change his general tool to satisfy different needs. In the software engineering domain, this kind of exchangeability is not easily supported by the use of sub-routines, but it is very similar to what we have for example in the forms management system. The forms handler corresponds to the motor parts of the tool, and as we attach different forms descriptions to it, we plug in whatever corresponds to these various attachments. Of course the spreadsheets systems use a very similar principle. The basic machinery for administrating the contents of a screen and doing the dynamic update corresponds to the general tool, and the so-called templates that you plug in correspond to the attachments.

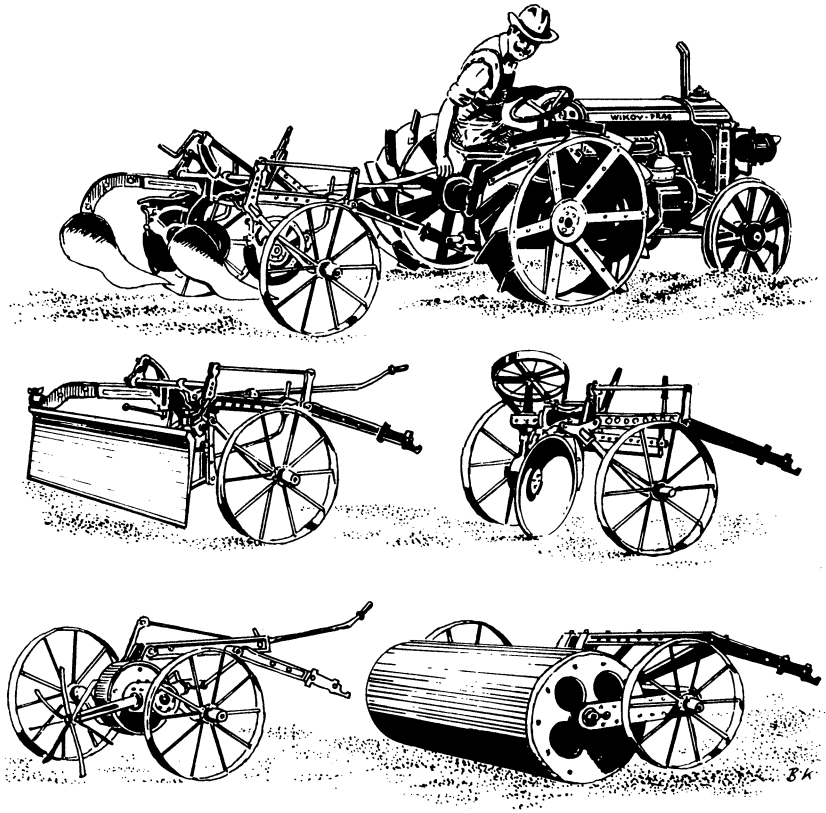
This general engineering principle has not been properly observed in the discussion of software engineering principles. There are plenty of other examples from other branches of engineering. Fighter aeroplanes provide an example, with bombs, robots and other weapons as attachments. Figure 7.5 shows another example, a tractor with all its attachments.

The tractor analogy is interesting from an additional point of view when we discuss office systems. Quite often one makes parallels between office automation and factory automation. There is frequent reference to the investment for each worker in manufacturing industry and the investment for each worker in the office, and it is suggested that office productivity should be increased by making offices more similar to industry.

This trend is actuated by the choice of terms such as "office worker" and maybe also "knowledge worker". The use of the term "worker" may be because people wish to apply the methods of Taylor, or may be due to socialist romanticism, but in either case I believe the analogy with industry is not the only relevant one. There have been objections to whether industrialisation of offices is a desirable development, and fears have been voiced that work in the office could become as boring as work in manufacturing industry is often thought to be.

There is, however, another branch of society which has an even larger investment per worker, namely, agriculture. Certainly in farming we don't have the scenario of very routine work that is usually encountered along the assembly line. The individual farmer understands very well the domain that he is working in: the growing

Figure 7.5: Another universal tool



of the crops, the effects of the weather, and so on. At the same time he has at his disposal a number of very powerful tools whereby he can fairly independently manage his work.

I therefore want to suggest that we should not look for the "office worker" in the future, but rather for the "information farmer": the individual person who does independent work using very powerful tools. The tractor that the farmer has as one of his tools would then correspond to general purpose software tools into which the end user can plug his particular templates or descriptions of the job to be done.

Data processing. Although we have examined the issues of general engineering principles and the view of the end user, it is also necessary to say something about the data processing perspective.

We have seen how some data processing services could be immersed into a general purpose editor. We could also see, in the course of our research project, how various other data processing services in the office could be nicely characterised by special purpose languages, for example, the information flow application. But what about the remaining services? Will these two techniques cover every kind of data processing in the office?

There is a very interesting paper by Boehm who organised students to write and implement a few medium size office applications (Boehm 1980). They measured how much time was spent on different activities, and how many lines of software were generated for various activities. The interesting result was that the "job at hand" was served by only 2%–3% of the number of lines of code. The algorithm for doing the work that was the purpose of the program was a small fraction of the code! Everything else was for things like general housekeeping of the data, error handling, describing the data that needed to be operated on, checking for incorrect input from the user, supporting the user with a convenient data entry language, and so on and so on.

In the terms that I introduced earlier, I would say that 98% of that special purpose program did information management and only 2% did the things which were application specific. So from this point of view maybe text editors, forms handlers and so on are just the limiting cases of a general principle, namely the cases where the algorithm is 0% rather than 2%. This suggests that the right way to organise this kind of software should be to build the information management system first, in order to cover 95 or 100% of the task. When an algorithm needs to be added, it should be possible to plug it in. A good information management system should be able to receive "plug in" procedures which account for the things that distinguish different applications. This of course confirms again the idea of using software with "plug in" modularity.

Language design. Another perspective from which to look at this software technology is from the perspective of programming languages. In the literature, there have been a very large number of papers about how to design good programming languages. These papers reflect several different schools. In particular there is the ALGOL-like school, the school which started in FORTRAN and ALGOL which has later on generated PASCAL, Ada and so on. There is another school of thought which is represented by incremental languages such as LISP, APL, MUMPS, to some extent BASIC, and more recently PROLOG.

Almost all this normative or even moralistic literature about what is a good programming language comes from the first of these two schools. We may speculate why that is so. With respect to LISP, which has been used in AI projects, my explanation is that the users have often been graduate students who were interested in the various aspects of the LISP system they were using as a tool. But the research leaders have often discouraged the study of the LISP as such, because they saw a danger of losing sight of the main issue, which was the design of AI. Therefore, over the years, there has been pressure to discourage too much delving about in the tools. In the area of classical programming languages, there has not been the same kind of restraint, because there the programming language was *the* topic of interest and therefore was considered to be the appropriate thing to write about.

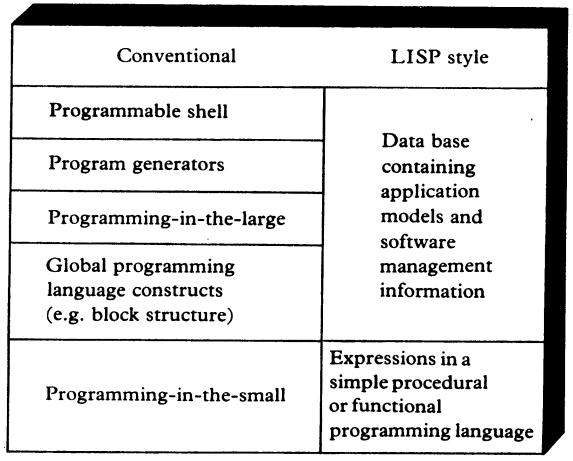
There is therefore a philosophical position about programming languages which is quite widespread in actual work but which has not been as often articulated in the literature. Figure 7.6 illustrates one important difference between the two schools of thought. In the LISP environment, the key idea when you build a system is to first build a data base where you can store expressions. These expressions can be, e.g., expressions in logic, or expressions which describe forms in a forms management system, or descriptions of data corresponding to the Data Divison in a COBOL program. But they can also be procedures, as another special case.

When we receive complaints about the bad, parenthesis-oriented syntax of LISP programs, compared to the syntax in, for example, ALGOL with all its syntactic sugar, the explanation for the difference is that the notation used in LISP is not specially designed for procedures. It is more general than that. We have a basis which is able to store expressions in a data base. On top of that, the data base is used for storing procedure definitions, application modules, software management information, application data, and so on and so on in an integrated way.

In the classical school, the programming language performs both the service of "programming in the small" and "programming in the large". That is, it allows us to express the contents of procedures, conditional statements, statement sequencing, loops and so on, and it also allows us to define block structure and other global structures. On top of that there will be module management. In sophisticated systems there may be program generators, and there may be a command system for the operating system or a programmable shell.

In the conventional programming language architecture there are these different layers which are stacked on top of each other. In the LISP environment, on the other hand, the lowest level, the expression level, is intended only to correspond to the local aspects of the programming language – "programming in the small". Everything else is programmed in that language, in an integrated way. I think it is important to keep this correspondence in mind when you compare

Figure 7.6: Programming language and systems



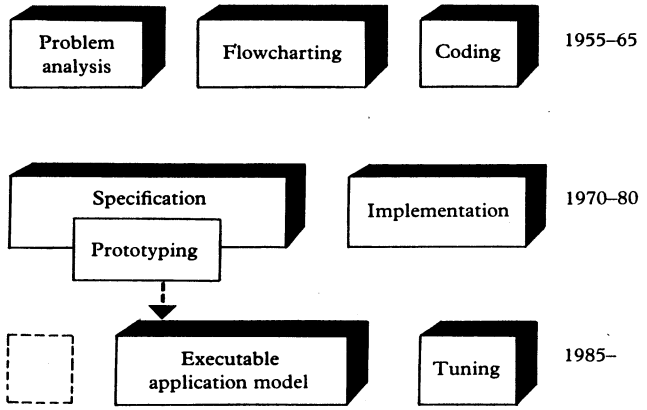
programming languages, for example, LISP to PASCAL or LISP to Ada. Don't compare all of kernal LISP with all of Ada because then you are comparing structures that were intended for different ranges of notational service.

Development methodology. Yet another perspective to take on this software technology is with respect to the stages of development of a piece of software. We are used to thinking about two stages – specification and implementation. In the debate about rapid prototyping, which is yet another novel technology, we sometimes encounter the question of where prototyping fits in. Is prototyping a part of the specification phase or is it a part of the implementation phase? It is sobering, in such a discussion, to remember that the distinction between specification and implementation has not been there always.

If we go back to the early days of data processing: the 50's and early 60's before procedural languages were in widespread use, we would rather have used the stages shown in Figure 7.7. There was problem analysis, followed by flow charting where flow charting was of course done "off line" and not in direct connection with a computer. Instead it involved the preparation of drawings which could be used as a specification for the person who wrote the machine code.

These three stages therefore do not correspond directly to the stages of specification and implementation. Some of what we do now in specification would have been done in flow charting earlier,

Figure 7.7. Development methodologies



but flow charting also overlaps with what we now do in implementation in a procedural language.

If such a switch has occurred before, it may occur again. My proposal is that the methodology from now on is likely to be increasingly one where we have an initial thinking phase and a phase of dialogue with users. Then there is a phase where we develop an *executable application model* in the computer: something which runs in the computer aided design system for software. For example, in the case of information flow modelling, this stage would be the stage where the information flow model is built up in the computer and tested in co-operation with the users. When this executable application model has been finished and judged satisfactory, there is either transition to practical use, using automatic software tools, or a short stage of manual tuning in order to speed up the system. But the essential design work would then be done in the software design system or design environment, by building and checking the executable application model.

7.5 Conclusions

To summarise very briefly the message of this chapter is that editors or information management systems are a key element in software architecture. They are important as end user tools. They are also important as parts of system development environments, because during system development we are going to edit or modify the application model continuously. Every step towards extended use of such information management systems in practical data processing operations today is valuable in itself, but it also paves the road for increased use of AI technology over the years to come.

References and suggestions for further reading

Boehm, B.W. (1980). Developing small-scale application software products: some experimental results. In *Information Processing 80*, ed. S.H. Lavington, pp. 321–326. Amsterdam: North Holland.

Hägglund, S. (1980). Contributions to the development of methods and tools for interactive design of applications software. Ph.D. thesis, Department of Computer and Information Science. Linköping, Sweden: Linköping University.

Hägglund, S. and Oskarsson, Ö. (1975). IDECS2 User's Guide. Report DLU 75/3, Datalogilaboratoriet, Uppsala, Sweden: Uppsala University.

Sandewall, E. (1979). A description language and pilot-system executive for *information transport systems*. In *Proc. Fifth International Conference on Very Large Data Bases*. Rio de Janeiro.

Sandewall, E. (1982). Unified dialogue management in the carousel System. In

System development environments

P... ACM Conference on Principles of Programming Languages. Albuquerque, NM.

Sandewall, E. Strömberg, C. and Sörenson, H. (1981). Software architecture based on communicating residential environments. In *Proc. Fifth International Conference on Software Engineering*. San Diego.

Strömfors, O. and Jonesjö, L (1981). The implementation and experiences of a structure-oriented text editor. In *Proc. ACM SIGPLAN/ SIGOA Symposium on Text Manipulation. SIGPLAN Notices*, Vol. 16, No. 6.