Be positive

One striking feature of the computer business of the early 1990s is the way in which even quite large organisations tie themselves to software development systems or languages which claim to be of universal or nearly universal application. The most recent of these languages (C) dates from the 1960s (even ADA was required to be based on a 1960s language, and ADA is not greatly used except for military software). Experience has shown that replacing a general purpose language by a language that closely matches the requirements of a system can save anywhere between 10% and 80% of the development costs. These savings come about because the initial designs do not need to be compromised to fit the language limitations, the specifications are simpler to translate into programs and the resulting software is intrinsically easier to commission and more reliable in service.

By comparison with major systems developments, compilers are not very expensive to write. In particular, the cost of a compiler to create a language for a specific project is very much less than the cost of a general purpose compiler. If you have a system for creating compilers then they become even cheaper. The aim of such a compiler system is to produce a very high level language which is closely tied to the application. This enables the application developers to concentrate on what needs to be done, leaving it to the compiler to create the means. This is not a fairy-tale idea. It was more than ten years ago that I first applied this approach to a project. Although the lack of a suitable compiler generator meant that the high level compiler had to written by hand, and some "hand compilation" was required. It was a dramatic success: a high quality, reliable and efficient application was produced by a team of people whose main experience and skills were in the field of the application, not the computer.

This "project specific compiler" approach proved to be effective for projects as small as five man years; with automation of the compiler generation, it would be suitable for even smaller projects. Such an automatic system was planned but was never completed because, by this time, I was no longer convinced that a programming language was an appropriate way to represent a computer program.

To my mind, conventional programming systems, based on program source stored in text files using a (sometimes complex) rigid syntax to define the meaning of each statement are an anachronism. This type of system squeezes all the various layers of definitions, commentary, data and program structures into a linear stream of bytes. So much argument goes on over the relative merits of different programming languages, that little attention has been paid to the more fundamental question: is a programming language the best way of writing and representing a computer program?

To some extent, programmers try to overcome this one dimensional restriction by laying out their text in paragraphs with indentation to highlight the structure. Effectively, this is a one and a half dimensional representation, although such a layout does not usually have any "significance": such layouts are purely decorative.

Modern computer displays are not just capable of two dimensional representations (although even that could be an improvement over the conventional one dimensional representation or a program) but, by the use of suitable windowing software, can represent multiple layers of information. Nor are such displays limited just to characters: the characters can not only be in various sizes, styles or colours, they can be complemented with boxes, lines, directed lines and a whole range of graphical symbols.

Doubtless everybody will have their own ideas about the ideal development environment, but my priorities for a new software development system are:

- Productivity
- Representing the user interface
- Representing the overall system structures
- Internal documentation
- Representing the structures for internal, filed and transferred data
- Guided access to common facilities
- Representation of algorithms

Productivity

It may seem strange to put productivity at the top of the list. But, if all other factors such as standards of documentation, quality, functionality and performance of the software are kept constant, then productivity is a good measure of the effectiveness of the development system.

More importantly, with all the additional computing power available now, with all the development tools that can be applied, the most optimistic estimate of software productivity improvement that I have seen from major US software developers is that productivity has been stable for the past ten years. This may be true if the cost per byte is taken as the basis, but a 1992 byte is, on average, worth much less than a 1982 byte. My own impression is that the decline in productivity has been dramatic, and, in many cases, it has been accompanied by a parallel decline in quality.

The reasons are not hard to find. When access to computing resources was limited, software developers were constrained to spend most of their time thinking and designing. With easy access, and fast compilation turnaround, this thoughtful process has tended to be replaced by a "hack it and see" approach. Attempts to apply strict methodologies have not always been successful at reversing this tendency.

Representing the user interface

Historically, for software developed to be accessed directly by a "user", the user interface has usually been the weakest part of the systems. There is no point in providing ingenious facilities, unless the user is able to access them in a coherent, easily understood way.

For applications programs, the facilities provided and the method of providing them should be derived from the user interface.

NOTE!! User manuals, reference manuals and on-line help are all part of the user interface and should be conceived, created, and updated all at the same time.

Representing the overall system structures

However impatient a software developer may be to get in and start hacking, he or she must be restrained. The development system must only allow code and structures to be created for those parts of the system whose interfaces are well defined. This is the point at which a software development can be safely split into parts to be created separately.

There is a second level where development work can always be done, regardless of the state of the project. Utility software, by its very nature, is not, and should not be, tied to a particular application.

Internal documentation

The representation of the overall structure is only the first level of the internal documentation. The problem with most programming languages is that the only built-in documentation is the "comment" which tends to be a bit of an afterthought. Most development organisations have standards for including appropriate comments at the head of a routine, and many have software to process these comments into formal documentation. This is an untidy, unreliable, unsatisfactory bodge. I do it this way myself: at the moment there is no readily available alternative.

The documentation should, of course, be maintained by the development system. This enables much closer control over the documentation, as well as making it easier to create, update and access.

> The definition of data structures and the code to process them must be subsidiary to the documentation and not the other way round.
>
> The descriptions of data structures and routines must be available where they are used as well as where they defined. In particular, the descriptions of routines must be parameterised in the same way as the routine itself, so that a description of routine is appropriate to the context in which it appears and uses the actual parameter names rather than the formal parameter names.
>
> The expression of the data structures and the expression of the code must be explicit and easy to read. The verbosity implied by this will not adversely effect either the time to create the code or the size of the source files: an intelligent source file editor will be able to fill in much of the padding required, and there is no reason to store the source in character format.

Representing the structures for internal, filed and transferred data

I have spent much of the past ten years designing computer systems, systems software and applications. In those ten years, I have spent very little time designing code. Mostly, I have designed the overall functionality of the system and then the data structures. At this stage, the code tends to design itself as the code exists merely to maintain the data structures.

This has turned me into a firm believer in data driven design. This is possibly the reason for my scepticism over object oriented programming. For me, code exists solely to maintain data structures or transfer information from one data structure to another. The code is so intimately linked to the data that the concept of creating artificial linkages of the C++ variety seems out of place.

The data structures should be able to incorporate not just the simple data elements (integers, floats, strings, pointers etc.), which are supported directly or indirectly by the processor and "normal" programming languages, but also more complex constructions ("molecules"?). These molecules include such constructions as linked lists and sets (where the the elements are distributed across a number of structures) and indirection tables, hash tables, indexes which serve to hold together, sort or access a collection of structures. There are, of course, the more mundane molecules such as queues, stacks, buffers and sieves which can form parts of larger structures. Each molecule should, of course, be defined with its own code to create, process, maintain and remove the molecule. The system must provide facilities for the creation of new molecules.

Guided access to common facilities

Standard data structures and their associated code, utility routines and standardised user interface, communications and database modules are all candidates for incorporation into applications programs. Providing a guided tour of such facilities available is essential.

The same guided access must be available for structures and code defined for a specific application.

This guided access works in two ways. First of all it provides a topic search for suitable routines, and when using a routine, ensures that all parameters required (even implicit parameters such as buffer sizes) are supplied and are of suitable types (selecting if necessary a suitable variant of the routine). Secondly, where there is a reference to a routine in the source, the system displays, not a cryptic call to the routine with parameters, but the description of the routine incorporating the parameters:

for example	copy input string to work buffer in upper case
rather than	strcpy_UC (input_string, work_buffer, len_work_buffer)

Used with the conventional compiler, the former would give serious syntactic problems: effectively the name of the procedure is "copy (string) ..... to (string) .... in upper case". With this new system, however, there is no problem: the internal representation of the call is a formal (not textual) representation of the call: strcpy_UC itself provides the visible form "copy $1 to $2 in upper case" where the development system editor substitutes the first and second parameters into the text when it is displayed.

The most important implication of this approach, is that all definitions of routines (whether called or expanded into inline code) are maintained using a definition file (which applies to all modules of a particular project) rather than individual header files for each module, which may, by accident or design, have different definitions in different modules. It also implies that the cross referencing of modules is maintained by the source editor of the development system, rather than the compiler, linker or loader. Cross referencing is, therefore, always available and always up to date.

Representation of algorithms

Classically, the representation of algorithms has been in "code". For most of the world, something written in code is intended to be incomprehensible. The source code of computer programs has always had a tendency to be incomprehensible, whether intentional or not. In the old days, every character of the source had to be typed or punched. This encouraged cryptic constructions. So many identifiers were used on large developments that the name itself was frequently buried in a host of qualifiers giving the usage, the author and other useful information. With a development system which can keep track of all these things for you, and which can expand identifiers from a few keystrokes as they are typed, there is no need to encrypt identifiers.

The same improvements can be made to the syntax of programming structures. If the source is not maintained as a character file, then we can put an end to obscure syntax. Each form of structure has its own distinctive window. The conditional action window (IF) has a region for the condition and a region for the conditional actions. Windows corresponding to the ELSEs and ELSEIFs of conventional programming languages are treated as repeated sections of the conditional window: in such a structure, it is impossible to become confused as to which ELSE belongs to which IF. Other standard structures are treated in an analogous way, but, because they do not need to be expressed in written syntax, are freed from many of the arbitrary restrictions typical of conventional programming languages.

To avoid not being able to "see the wood for the trees", which is a problem inherent in structured programming where there are more than two or three levels of nested structures, the content of a program structure (including any nested structures) is first displayed as a simple descriptive text. This text can be expanded into the full description of the structure as required, thus revealing the next level of structures down. Any section of a programming module can, therefore, be viewed (or printed) in any level of detail desired.

Common operations can be collected into procedures, but, unlike conventional languages, the appearance of a call to a procedure is defined by the procedure itself, not the programming language.

Other requirements

To meet the needs of those unfortunate enough to have to work with more than one architecture of computer, there are two other requirements.

The first is the support for optional sections of program which are only linked into programs when the option is required. The second is for assembly language routines, where the algorithms are expressed in pseudo-code and for each pseudo-code statement there is one or more assembly language statements for each processor supported. This latter facility helps to ensure that all processor versions are updated at the same time.

Side effects

Because the meaning of a program does not have to be forced into a textual form but is stored as a formal representation, the appearance of a program on the display, or when listed, can be in any language, English, French, German or even Japanese. For this reason, utility routines need to be supplied not only with appropriate translations of the words used in the call to the routine, but different syntaxes for the call as well.

Replies please.

What is wrong with your present software development environments?

Would a visual multi-layer representation of a software system be valuable?

How would YOU do it?

How much would you pay for it?