ZEN AND THE ART OF MODULAR ENGINEERING

INFORMATICA 1/90

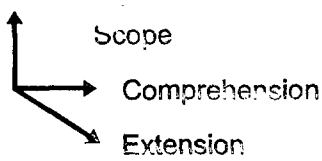
Keywords: modular engineering, modularization, software

Brian R Kirk MSc MBSC,
Robinson Associates, United Kingdom

Presented at the 1st Int'l Modula-2 Conference, October 12-13, 1989 Bled-Yugoslavia

INTRODUCTION

As time passes computing components became an integral part of more and larger systems. The diagram shows our dilemma



The **scope** of what is requested seems only to be limited by what can be imagined. Somehow as designers we must gain **comprehension** of all the implications of the whole system in all its states. And most taxing of all we must provide accurate solutions that support **extensions** to match the requirements as they evolve.

This paper offers an approach to coping with the dilemma, the title encapsulates the concepts.

Zen	looking inside in a search for understanding
Art	a fine skill
Modular	separate parts designed to be cohesive
Engineering	designing and building practical machines

A real product is used as an example of how modules and machines made from modules can provide reuse and extension of existing software, even when there are difficult constraints on the implementation.

The objective of the paper is to pass on the experience learned whilst engineering a large real time software system. In particular the approaches used to divide and conquer the complexity and inherent concurrency may be of interest to implementors of high integrity systems. In all cases the pragmatic approach to the finding of practical solutions is described. The language Modula-2 has been used as the programming notation. It is now hard to conceive or believe that such a large system could have been created so effectively with any other available language. Particularly in a form that can be understood and extended with ease.

THE PRODUCT

Often it is necessary to update an existing product and give it a new image. In our case the requirement was to take a paper-tape based multi-axis machine tool and to match it to the current marketplace. The extensions included CAD, graphics, a file system, a printer, remote controlled operation - and all this with either English, French, German, Italian and Russian interaction with the user - see Figure 1.

The **form** of any design is a product of its designers interpretation of its requirements and constraints. In this case the constraints were formidable ...

1 the need to support all existing functionality

2 the impossibility of all but minor modification to existing software (some sources were lost!)

3 the need for a real-time response on the display, CNC, remote link and language translation

4 the need to interact in ad-hoc ways with 3 existing computers

5 only having a RAM memory of one third the size of the whole program

6 the need to make all the new software resilient to power failure for continuous operation

7 the Client's prior choice of DOS and GEM for filing, graphics and multi-tasking

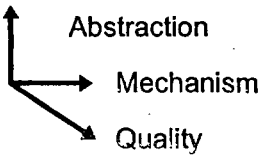
The completed software is large, it contains:

3 programs with 15 overlays
150 modules
2000 messages each in 5 languages
2 Mbytes of executable code
30 Mbytes of source code

It was developed by a team of 6 people over a period of 2 years. Had we realised initially the full scope of the requirements and the implications of the constraints we might never have started. Only the rigorous use of modular engineering concepts and carefully coordinated implementation in Modula-2 by a team of professional software engineers made the whole project feasible.

MODULAR ENGINEERING

Engineers analyse problems using concepts and then synthesize their solution by organising some physical form, in this case the software part of the system. The diagram shows the main criteria

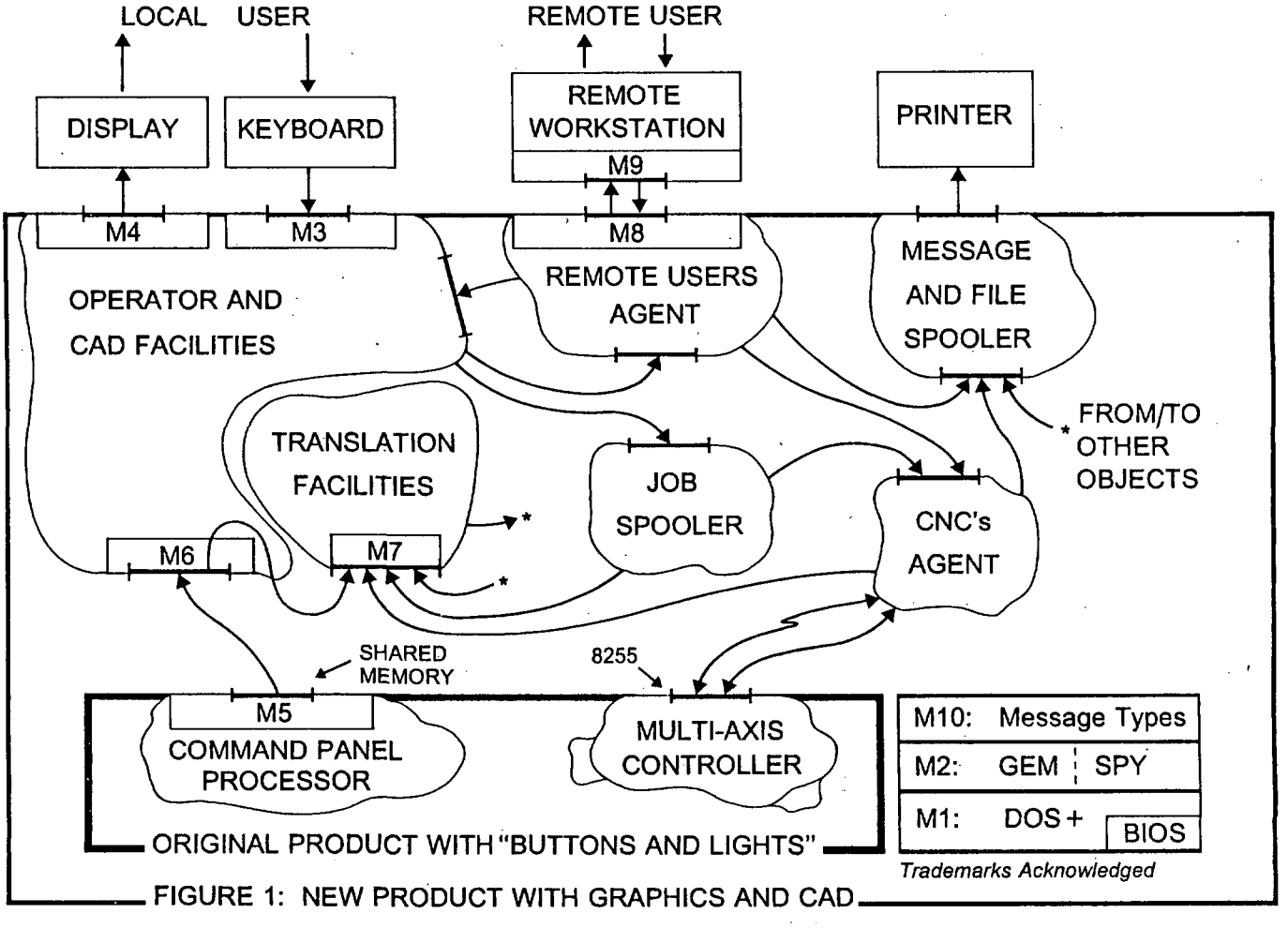


The **abstractions** we use to analyse and model the problem have evolved over the past 40 years of computing, these include

Names	for instructions, data and locations
Macros	to encapsulate and reuse the text of sequences of instructions or data
Procedures	to encapsulate and reuse sequences of instructions at runtime
Control Structures	to encapsulate the flow of control
Classes	to encapsulate evolutionary definitions in a reusable and extensible way
Modules	to encapsulate whole components, hiding information and/or ownership
Extensible Modules	to encapsulate objects which have statically related definitions
Delegating Objects	to encapsulate objects which are dynamically related and extensible. An object which cannot provide a requested method delegates it to another object which can.

Languages provide a means to express solutions to problems in terms of these abstractions, for example, Assembler, Algol, Simula, Modula-2, Oberon and Delegate. The trend in abstraction is towards an object oriented approach because this minimises the distance between the problem and its programmed solution: "the solution is a simulation of the problem". In practice we have found Modula-2 an adequate language for expressing both modules and delegating objects, which are message driven tasks consisting of modules.

The **mechanisms** are simply ways of achieving something. For example in Figure 1, modules M5 and M6 provide an interface between various tasks in the two processors. In our first implementation M5 replaced the old graphics card driver and sent equivalent messages to M6. This made it possible to reuse the vast majority of the original software



Trademarks Acknowledged

FIGURE 1: NEW PRODUCT WITH GRAPHICS AND CAD

with minimal changes. Of course M6 completely hid the protocol and a rather nasty dual port RAM interface from all the new software. This technique was much too slow in practice and was later replaced by a set of records and update flags at agreed fixed positions in the shared memory. By using modules on each side to encapsulate the mechanisms it became possible to change the mechanism separately from the rest of the system. We found that a good test for the quality of a module's interface was to consider how much it would need to change if the mechanism it encapsulated but not necessarily the functionality, has to change.

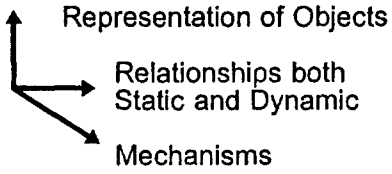
The **quality** of the implementation is the third main factor. Engineers differ from computer scientists in that they are faced with many practical constraints and exceptions yet their solution must be effective in actual use. For example the machine tool can cut diamonds and diamonds are valuable. The clients are not impressed by large diamonds that unfortunately have the wrong shape due to software errors. About 15% of the modules we wrote were **test harness** modules which either exercised the modules under test or acted as dummy modules for uncompleted parts of the system. Sometimes we wrote modules to provide rough prototypes of parts of the system that were poorly specified or particularly difficult to achieve. By isolating these areas adequate solutions were found quickly and the risk to the whole system minimised.

Sometimes the structure or quality of existing software was too risky to incorporate into the product. In these cases we 'reverse engineered' the software. This involved analysing the code to discover what the intended requirements were, we then made the requirements self-consistent. The software was then redesigned in line with the system model, mechanisms and modules. This concept provides clean maintainable software rather than horribly bodged incongruous coding - it also takes less effort.

The system was constructed as a 'pile of machines' implemented with programs, processes and modules. Always striving to verify that the partially complete system had 100% correct functionality within itself. This policy of **stepwise construction** of the system provided visibility of progress, a practical means to assess quality and confidence for our Clients.

CRITERIA FOR MODULARISATION

The main reason that we partition systems into subsystems and modules is to encapsulate our comprehension and thus extend our capabilities. This is achieved by using abstraction to separate out distinct parts of the problem. These abstractions are then implemented by building logical machines on top of physical ones to mechanise the abstraction in a form, and at a cost, which is appropriate to the user. In the past the criteria for modularisation were influenced by the 'everything is a hierarchy' view of programming, latterly the use of 'information hiding' as a criterion has been much more useful. What is really needed is a set of criteria that maximise the separation of ...



At the same time we wish to optimise

- ease of comprehension
- ease of development by teams
- flexibility for extension possibilities

Perhaps the fundamental criterion is that each separate part, be it active, object or component module, should be testable. If it is not certain that something can be tested before it is built then there is little point in building it because there is no possibility for quality assessment or control.

Looking back on our projects we can now see the actual criteria that have been most effective, they include encapsulation of reuse, adaption, concurrency, consistency and mechanisms.

REUSE

It is a fact of life that reuse of what already exists is often essential. Usually the reason is short-term economic optimisation (this is rarely justified in practice!) but sometimes it is just not possible to reimplement old parts of a new system because there is not enough time or the knowledge is no longer available. In any case if a product is still 'alive' it certainly will need to be extended to match its behaviour and performance to the evolving needs of its users. This needs to be done with minimal modification of existing parts but the aim is to inherit the functionality and system model from the existing system. Unfortunately the mistakes and constraints are also inherited, the main disadvantages of standardization.

There are some classic examples of reuse in Figure 1. The whole of the old machine software is reused except for two modules that provide a new interface to the software extensions, eg module M5. More typical ones are M1 and M2 which provide 'cleaned up' interfaces to DOS and GEM. Indeed GEM provides both graphics and multiprogramming scheduling machines built on-top-of DOS. The GEM constraints of supporting only 4 programs (not tasks) and of round-robin scheduling were inherited by the system and distorted its form, reducing productivity.

ADAPTION

When creating large systems it always pays to make the software part as portable as possible. Conventionally this is done by providing 'device driver' modules which abstract away particular

physical characteristics at the lowest level and offer a clean logical software interface instead. The client modules then use the clean interface so making it portable and also improving the flexibility for hardware machine choice. Typically these modules are hidden in 'the BIOS' but any new devices can have their drivers written in Modula-2.

Modules M3 and M4 are good examples in practice. M3 extends the normal DOS keyboard driver to support the storing of a keyboard history and also the alternative keyboard layout and coding needed to support Russian. Module 4 had to be rewritten to match a non-standard graphics display controller.

CONCURRENCY

The sequential instruction by instruction execution of programs by CPUs has unfortunately led generations of programmers to presume that concurrency does not exist. They inherently try to coerce the concurrency of the problem into a single stream of CPU instructions. The liberation from this mental straight-jacket is inherent in data flow diagrams which show the flow of information between processing activities. Their use has broken the curse of the flowchart which deems its uses to think only in terms of sequential control flow. Figure 1 takes the concept of a DFD further, it shows the flow of information between naturally concurrent objects in the system, be they logical or physical objects. By initially analysing the problem in terms of the concurrent objects it contains we get some very clear benefits ...

1 the implementation can be a simulation of the problem

2 the objects can be allocated to or shared between the processes/processors depending on their individual performance needs of throughput and response time

3 once the shared modules and interfaces between objects are defined and designed the implementation of the objects can be developed separately by members of a team

4 synchronisation and communication between objects can be optimised separately to suit the needs of the objects, see modules 5 and 6, 8 and 9 later

5 the system can be constructed incrementally by providing dummy objects as 'stubs'

Creating a concurrent-object-information-flow-diagram as the 'top' level of the analysis and design process gives a clear overview of the whole system. It is the equivalent of the hardware engineers 'system block diagram' and the architects initial building design sketches, nothing really new.

Figure 1 provides many examples. The soft shapes enclose the logical objects in the system and the arrows the information flows. The objects are in fact allocated processor time in a variety of ways

- in the original product each has its own processor

- in the extended product they share 3 tasks on another processor. The objects share the tasks but were originally written and tested separately, during optimisation they were coalesced to save memory overheads

- time critical objects such as timers are activated physically by CPU interrupt events

To provide the quasi-concurrency that the objects require the modules M1, M2 and M10 are used. M1 provides a clean interface to DOS and M2 multiplexes DOS to create 3 separate program environments. M10 provides datatype definitions for messages sent between the programs. Note that this module requires special version control treatment because it is shared by separate **programs**, if it is changed then all programs that use it need to be remade. It is noteworthy that no 'real time' operating system was used, objects being either co-operatively scheduled or event driven by real time events depending on their needs.

CONSISTENCY

The possibility for automating consistency checking is perhaps the greatest benefit of using non-permissive languages, 'C' and Assembler are permissive! Pascal introduced strong data type checking, Modula-2 has introduced the possibility of explicit control of visibility of module contents combined with an environment which automates inter-module consistency checking at both compile time and run-time. Even greater support is needed when modules are shared between separately compiled programs because a change made to satisfy one program may have nasty knock on effects which are inconsistent for the other programs. We tackled this problem by extending the PVCS Version Control system using batch command files to automate the consistent updating of modules shared between programs.

There are some examples of modules which enhance consistency in Figure 1. Module M10 contains a set of datatypes which define the format of messages passed between objects in the system. Variant records are used to overlay data of differing types over the same memory area. The generic data format is consistent with existing GEM messages so that new message formats become extensions to the existing ones. Module M6 contains a set of datatypes which define records in a shared dual port memory. It also hides access procedures which provide a synchronised read/write protocol with the Command Panel Processor, guaranteeing consistent atomic access to each whole record of fields. **Incidentally static compile time checking was helpful but we also found that dynamic runtime type checking of both subrange values and enumeration values was essential.** The reason for this is that the other processor and its interface module M5 were programmed in assembler code, of course without compile or runtime type checking. By having the

benefit of runtime checking in the extended software it was possible to quickly detect and locate errors in the existing system.

By using these techniques it is possible to guarantee consistency at a system level even between modules shared by programs on the same processor or with 'foreign' modules on other processors. Much of the existing software was written in 'C'; by putting a veneer of Modula-2 over it it became possible to maintain strong type checking.

OWNERSHIP AND MECHANISMS

Perhaps the most powerful criteria is that of ownership. Here the concept is that the module owns a mechanism such as how a filing system is structured or how a protocol works. The interface, or external visibility, of the mechanism is minimised. Its clients are only informed of its information and services on a 'need to know' basis. When trying to partition a system, an object or program for modules, the main criteria are not only to encapsulate the ownership of mechanisms and/or information but also to guarantee that its behaviour and performance can be verified. If modules are well designed in this way then the mechanisms can evolve to meet the requirements with minimal changes to the client modules. The underlying architecture of objects, programs and module relationships should be resilient to particular choices of mechanisms chosen for an implementation.

Nearly every module shown in Figure 1 owns a separated mechanism. Module M7 owns the mechanism for converting to message values into a text string in either English, French, German, Italian or Russian. Initially the string was searched for in a linear way from floppy disk, just to get a prototype working quickly. Of course the mechanism was not fast enough for an interactive graphics based product. The implementation module was redesigned and rewritten 3 times, each time with a faster mechanism until the product performance goals were achieved. The definition module and client modules did not change at all.

Another pair of modules, M8 and M9, own the interprocessor protocol used to reliably convey commands and information between the machine and a remote workstation. The modules were written and exhaustively tested in isolation from the system. The protocol was specified as a finite state machine and implemented using a CASE statement and a variable of an enumerated type to represent the state. The list of meaningful names in the enumeration type made possible the creation of a readable program which could benefit from strong type checking. To improve the performance of the remote channel the mechanism for activating the protocol was changed from 'application polling' to 'timer event' activation. Once again this was achieved without altering the client modules.

BENEFITS OF THE EXPERIENCE

1 Partitioning by **concurrency of objects** leads to a very clean system design 'the solution is a simulation of the problem'.

2 The most useful criteria for modularisation are reuse, adaption, concurrency, consistency and ownership of mechanisms.

3 It is important to separate out parts that are likely to change often, such as the user dialog text.

4 Stepwise refinement is relevant to improving mechanisms without changing their functionality. Also product specifications evolve dynamically and it is necessary to consciously adapt and refine the program architecture to suit the new facilities required.

5 Modula-2 is an excellent language for engineering large systems in a contolled way. It is flexible enough to express both object oriented and structured programming concepts. It provides for both compile and runtime checking.

6 Specialised languages like SmallTalk would have been inappropriate for the solution; languages like 'C' are just too inherently lax and unreliable for such large projects.

7 The problem of controlling the integrity separate programs which share modules can easily be solved in the development environment.

8 Designing the system as a " pile of interacting machines" provides for great productivity, flexibility and portability.

ACKNOWLEDGEMENTS

Prof N Wirth - for separating out the concepts
B B McGibbon, S Doyle, J Teague - for help with system design
D Gifford, T Harris, D Fox - for reliable programming
C Diegez, P Zanolari, M Wakely - for modifying the existing parts
Dr B Schumacher, K Pferdekaemper, P Locati, M Bertoli - for project direction
J Woodhouse, S Doyle, G Luker - for typesetting.

REFERENCES

1 Dijkstra E W, May 1968'
"The structure of the 'THE multiprogramming system"
Comm of the ACM, N5, Vol 11.

2 Wirth N, April 1971
"Program Development by Stepwise Refinement"
Comm of the ACM

3 Parnas D L, Dec 1972
"On the criteria to be used in Decomposing Systems into Modules"
Comm of the ACM

4 Maruichi, Uchiki and Tomoro, 1987
"Behavioural Simulation based on Knowledge Objects"
Dept Electrical Engineering, Keio University, 3-14-1 Hiyoshi, Yokohama 223, Japan

5 Stein L A, Liebermann H, Ungar D, 1989
"A shared view of sharing: The Treaty of Orlando"
Object Oriented Concepts ... ISBN 0-201-14410-7
Addison Wesley