AR-004-637

DEPARTMENT OF DEFENCE

DEFENCE SCIENCE AND TECHNOLOGY ORGANISATION

ELECTRONICS RESEARCH LABORATORY

REPORT

ERL-0372-RE

# MANAGING SOFTWARE COMPLEXITY

P.F. Calder

S U M M A R Y

This report examines the methodologies and tools available to the manager and programmer for assisting in the development of large software projects.

POSTAL ADDRESS: Director, Electronics Research Laboratory,
Box 2151, GPO, Adelaide, South Australia, 5001.

TABLE OF CONTENTS

## 1. INTRODUCTION

The brief history of computing projects is littered with the wreckage of software developments which cost too much, took too long or were never completed satisfactorily. On the other hand, many other projects were very successful. This variability in software implementation has concerned software designers, and slowly the process of developing programs has become better understood(ref.1). The term software engineering has come into use, implying a concrete design methodology which is guaranteed to produce desired results. This implication is not fully justified, but it is true to say that the practice of a proven design methodology increases the probability that a software project will be completed successfully.

The size and complexity of software projects is growing exponentially, not only because of the increased sophistication and functionality of computer systems, but also because computers are being used to perform tasks previously carried out by other types of hardware. For example, computers are being included in systems to provide control capability performed manually before. The use of computers in modern tactical fighters and commercial aeroplanes is indicative of this trend. It is also true in the case of a tactical Command, Control and Information System (CCIS), where computers are being applied to areas which previously have had little more assistance than a typewriter.

Any software project, large or small, is developed within a framework of system management and computer resources which include computers, language compilers, source code editors. In recent years this framework has been termed the 'Software Development Environment'. This environment involves not only writing programs, but also project management, system design, system integration, system and program documentation, testing and software module management. In a large project, these tasks are often full time demands, and staff must be allocated to perform them. It was the inability to recognise the differences between small projects requiring few staff, and large ones requiring many staff, that resulted in failures and costly over-runs during the 1960s. A large project is not simply a small project scaled up(ref.2).

As these differences were recognised, techniques began to evolve from the primary need to provide control over software development. These techniques were aimed at ensuring that:

(a) the right problem was being solved,

(b) progress was visible to managers,

(c) areas experiencing difficulty became apparent early, enabling staff to be added, redeployed or removed, or the requirements to be renegotiated,

(d) system design decisions were documented, minimising the time new staff needed to 'come up to speed',

(e) components of the system fitted together properly.

However, it is only in the last few years that design methods, techniques and software tools have begun to come together to form an integrated software development methodology(ref.3).

This report has been prepared in response to Army Research Request: (1159/82): Architectures for Automated Tactical Command and Control Systems.

## 2. AIM

The aim of this paper is to discuss the various software methodologies and development tools that are available for large software development projects.

## 3. THE IMPORTANCE OF USER REQUIREMENTS

Most software methodologies take as their starting point a statement of user requirements. The methodologies then assist in providing a disciplined way of turning the requirements into reality. It is extremely important, then, to have an accurate and detailed statement of what the computer system has to do, right from the start of the project.

It is inevitable that some details of the requirement will not be exactly what is required. However, significant changes to requirements during the course of a project can lead to confusion, cost escalation and time delays. Of course, there is great pressure to accommodate the latest thinking into the developing system, particularly if new concepts in user requirements may mean that the current design will fall short of user's needs. Some compromise may be necessary by deferring some new options until a subsequent software revision.

The design methodology should include early exposure of the user to the system as the project progresses. Detailed amendments to the requirements can then be identified before significant effort has been expended on them. The means of managing limited changes to the requirement should be part of the methodology, so that the effects on the project may be monitored. The cost of incorporating changes into the software under development increases dramatically towards the end of the project, as figure 1 shows.

The development of the user requirement is outside the scope of this paper, which deals only with managing software complexity. However, the fact remains that the user requirements must include as much detail as necessary to define, without ambiguity, exactly what assistance the system is to give the user. This high level of detail is not easy to specify, particularly for a tactical CCIS.

Computers are being introduced into the tactical information scene for the first time. Users have been constrained in the past to perform functions achievable by essentially manual techniques. Any system designed to support current manual requirements will not satisfy for very long once users begin to appreciate the enhanced capability of the new equipment. Of course, the use of new equipment often requires different ways of doing things, which can affect other functions and so on. The introduction of computers to this 'virgin' environment significantly alters the tasks being performed in, often, unpredictable ways.

Early CCIS projects (notably the US Tactical Operations System) failed because they were too encompassing, and aimed at a moving target of user requirements. Large software projects such as these should not be embarked upon until the user requirements have matured sufficiently to be frozen. However, reaching this point often requires considerable user education, and effective exposure to different techniques and procedures. This can be achieved quite effectively, through the use of a 'test bed', or through 'rapid prototyping'.

A test bed involves setting up software to demonstrate as many of the staff functions as necessary to give the user a familiar environment in which to decide what his requirements are. Rapid prototyping is often used to provide examples of alternative user interfaces to particular staff procedures, so that the most promising can be incorporated into the test bed.

A third approach would be to develop a system in independent software modules, (or groups of routines supporting particular functions), to requirements which have been identified so far, and expand the modules in stages as further requirements become clear.

Current research is directed towards software automation tools which will be able to accept user requirements in some standard form and produce programs directly(ref.4). Automation forces the software designer to expend a great deal of effort on the requirement and specification phase of a project because even program-writing computers cannot accept ambiguous inputs (refer to Section 6). But, even without automation, the time spent at the beginning of a project to get the requirements right will pay off right throughout the development.

It is essential, therefore, to have a comprehensive, detailed and accurate set of user requirements available BEFORE a major software development project is entered into.

The requirements must be in terms that the user can identify with and be assured that they represent his needs. The fewer changes to requirements that need to be applied to software being developed, the more likely that the project will be completed successfully.

## 4. MANAGING SOFTWARE CONTRACTS

The management of a software project can be viewed from two important aspects:

(1) from the customer's point of view, and

(2) from the contractor's point of view.

The customer wants to monitor the software development to satisfy himself that the project will meet the requirements, be on time and be within cost limits.

The contractor wants to deliver a product which the customer will be happy with, but the contractor must at the same time, meet milestones and keep costs under control.

The customer must understand clearly the problem to be solved, so that a comprehensive user requirement document can be written, for issue with the Request for Tender (RFT). The tighter the specification of the problem, the easier it will be for potential contractors to evaluate the effort required, and identify the areas of risk. As time means money, any requirement which is not clearly defined in the specification will have a higher contingency factor added than one which has a straight forward and obvious solution path. It is reinforced again that the best requirements specification, from both customer and contractor point of view, is one which contains no areas of uncertainty.

A widely publicised briefing can be one way of conveying the general requirements and extent of a project to industry, so that companies may assess their desire to be involved before providing formal expressions of interest.

After an RFT has been sent to companies which have expressed interest in the project, sufficient time must be allowed for them to become familiar with the problem, define their approach and prepare a response. This time could be somewhere between three and six months, depending on the complexity of the project.

When all tender responses have been received, the customer then has the major task of evaluating the responses and selecting the most suitable.

Unfortunately, the selection will seldom be clear cut, with each tendered approach having its own advantages and disadvantages. A set of selection criteria should be established against which each response may be evaluated in a manner which will be fair to all parties. Contract cost is obviously a significant factor, but whole-of-life costs should also be considered to account for maintenance charges. Guide lines for tender evaluation have been laid down in the Commonwealth Purchasing Manual(ref.5) and these should be followed where possible. Specialist agencies in Defence such as Defence Science and Technology Organisation or Engineering Design Establishment could be asked to assist in the evaluation process.

During tender evaluation, clarification may be needed from tenderers to explain their written response. This must be carefully handled to avoid giving opportunities to particular companies to change their bid. Clarification should relate solely to their formal response to the RFT and be offered to all contenders.

The RFT should specify the management and technical staff positions needed, and request that the tender response provide the names and qualifications of the personnel who will fill those positions. Of particular interest in projects which are predominantly software based, is the ability of the personnel which are designated in the tender response as being available to work on the task. Variations in staff during the course of the contract should be monitored to ensure that the standard of ability agreed to at contract signing is maintained.

The contract which is finally signed should detail the necessary mechanics for monitoring the course of the project. These would include:

(1) Defining a point of contact for formal exchanges between customer and contractor on such matters as contract interpretation or requirements clarification. Contractors may be given many opinions on questions of interpretation, but only one should carry the contract seal of approval.

(2) Defining the means for providing day-to-day contact with the customer on detailed requirements. User representatives should be encouraged to be involved in day-to-day discussions with the contractor at the working level, particularly for providing advice during the design process. The user needs to educate the contractor into the fine details of the problem so that a more useful solution may be produced. It is too late after software has been delivered to say "but that's not what I really wanted". The customer will need to budget for providing personnel for this support, including any travel required.

(3) Defining milestones related to specific demonstrated capabilities, and tests which must be satisfied before payment of the milestone will be approved.

(4) Defining a change control mechanism for managing changes to the requirement which may occur as the project progresses.

It is important that progress on the task be visible to the customer, so that problems in implementation may be identified early and steps taken to rectify the situation. The methodology employed by the contractor should provide most of this visibility, and provision should be made for the customer to attend design reviews and major structured walk-throughs. Regular progress meetings should be designated, with progress being demonstrated at the meetings.

Inevitably in a large software project, it will become clear that some of the requirements may not be achievable. This could come about for a number of reasons, including misunderstanding of the original requirements, changes to other requirements having adverse side effects, or simply that the achievement of the requirement may be too costly. The contractor should be free to refer to the customer to confirm the importance of the particular requirement, or to see if the restriction may be relaxed. In some cases, the relaxing of a constraint in a non-critical area may mean the success or failure of a project. Obviously, if the requirement is a critical one, then the customer has the right to insist on adequate performance.

Computer projects are rarely software based only, unless software is being developed to run on existing equipment. The usual case involves both the selection of suitable computer hardware which would interface to the user environment, and the development of software to perform the user function. However, conflicts can arise in setting the boundary between hardware and software, particularly when some functions could be executed in either software or special purpose hardware. For example, Ethernet protocols could be implemented in a special purpose microprocessor, thus relieving the host of responding to bit transitions on the net. Alternatively, the host could manage the protocols itself, if it had plenty of spare capacity available to satisfy the speed requirements. This kind of conflict can only be resolved by considering <u>both</u> the hardware and the software aspects of the project. Optimising on the basis of hardware only or software only could lead to problems in achieving adequate performance. Areas where such judgements must be made should be identified as early as possible in the project, as performance may be effected, as well as having adverse cost implications. The contractor should be obliged, very early in the project, to justify his choice of computer equipments in relation to the application environment, the required performance and the software workload. This justification should be subject to customer approval before the project should be allowed to proceed.

At specific points in the task, software auditing should be carried out by qualified auditors (possibly under separate contract) to verify the quantity and quality of the work done to date, and to relate the work back to the user requirements. The software documentation standards called for in the contract can make this task easier, especially if comments included in software modules directly relate to specific requirements.

At this point, it is worthwhile to note that a new US Department of Defence Military Standard on Defence System Software Development (DOD-STD-2167, JUN 85)(ref.6) has been released, to supersede DOD-STD-1679A as the reference applying to the development of embedded computer software. The standard thoroughly specifies the formal documentation required for each of the phases of the software life cycle (see Section 5) and also covers the following topics:

(1) basic coding standards and in-line documentation,

(2) formal and informal test requirements,

(3) interface requirements,

(4) data base design requirements,

(5) baseline management,

(6) configuration control,

(7) software project planning, and

(8) reviews and audits.

There are a number of topics which are left for the contractor to specify in the Software Development Plan (which should form part of his tender response). These include:

(1) structured requirements analysis tool or technique,

(2) program design language,

(3) formal procedures to control all changes to baselined documents and program materials,

(4) formal problem reporting and change system,

(5) procedures to generate periodic status reports on all products in the developmental baseline, and

(6) software quality program.

The following sections of this report discuss topics which concern the software contractor more than the contract supervisor. Nevertheless, the steps needed to implement a software project must be understood by the contract supervisor in order that his management may be effective.

## 5. SOFTWARE DEVELOPMENT METHODOLOGIES

A software methodology provides a disciplined approach to the controlled development of a system from user requirements through in-service maintenance, ie the whole life cycle of the system. It identifies all the aspects of a project which need addressing, and defines procedures and mechanisms for ensuring that the project will be manageable and well documented. A useful tutorial on methodologies is found in reference 7.

Most methodologies assume the existence of a well defined user requirement, and then describe the steps necessary to implement it and verify that all requirements have been met. The previous section of this paper has already emphasised the importance of user requirements and this cannot be overstated.

The life cycle of a project can be conveniently represented by a 'waterfall' diagram as in figure 2, showing how one phase leads into another.

A methodology should address all phases of a project, and include:

(1) Requirements Analysis, Definition and Verification,

(2) Design,

(3) Implementation,

(4) Quality Assurance,

(5) Testing and verification,

(6) Maintenance - corrective and adaptive,

(7) Documentation,

(8) Budgeting,

(9) Personnel deployment,

(10) Project review,

(11) Scheduling,

(12) Configuration management.

Some of these topics for example, documentation, apply to all phases of the life cycle, while others may apply to different parts of a project at different times. For example, testing may result in changes to the specification which may require some modules to be redesigned.

By laying down procedures related to both the technical and managerial aspects of a software project, the progress of the project can be monitored. Figure 3 defines the relationships between the various elements that make up a software development environment(ref.8). A methodology makes work progress visible through reviews at a number of intermediate checkpoints, so that problems can be identified and corrective action taken. The effect on the project can also be clearly seen.

A methodology should be simple, and easy to learn by the various project members. The methodology should be well documented, so that not only original project members but also new recruits can be made aware of their individual and collective responsibilities.

The most widespread methodology in use today particularly in US, is that advocated by Yourdon, Constantine and deMarco(ref.9,10,11), and promoted throughout the world through seminars and workshops. A more readable text on Structured Systems Design will be found in reference 10. Use is made of Data Flow Diagrams, Structure Charts and a Data Dictionary.

The SPECTRUM methodology used by the Australian Department of Defence for administrative computing projects(ref.12) incorporates the Yourdon/deMarco and Jackson(ref.13) structured techniques of analysis, design and programming. SPECTRUM is described in a set of manuals which support each aspect of a task. The manuals do not include any automated tools to support the methodology, but describe a non-automated approach to project development.

A methodology which has a large measure of support in the UK Ministry of Defence is called MASCOT - Modular Approach to Software Construction Operation and Test(ref.14). It is particularly useful for implementing real time stand-alone multiprogramming computer systems, since MASCOT describes, not only the program implementation procedures, but also the run time support requirements.

MASCOT is programming language independent. However, many executive and monitor routines have been written in Coral 66, as well as a library of procedures. As Ada will become the official UK MOD programming language from 1986, MASCOT is expected to be used for Ada program development. Some of the multitasking features of MASCOT are now included in the Ada programming language, and the use of MASCOT to construct Ada programs is described in reference 15.

The US DoD, as part of its Ada program, is currently going through a public design process which will result in the definition of a preferred methodology. The last description document is called 'Methodman'(ref.8).

While the advantages and disadvantages of a particular methodology may be debated at length, it is absolutely essential that a proven methodology be chosen before any significant software development project is undertaken. Project managers should ensure that the principles of the chosen methodology are adhered to during software development.

## 6. THE IMPACT OF SOFTWARE TOOLS ON METHODOLOGIES

Current methodologies have developed from the need to provide rules and procedures to support each aspect of the life cycle process. The use of these procedures requires education of all the project members so that they apply the rules properly. This adds a burden to the work load of each project member, and the administrative overhead can become significant. There is therefore a need to provide automated support to each phase of the project, following the concepts enunciated in the methodology.

Structured Development Forum is a user's group devoted to advancing structured analysis and design techniques. Semi-annual meetings are held on the west coast of USA, and close ties are maintained with methodology tools available. Further information can be obtained from Structured Development Forum, Atlantic Systems Guild, 353 West 12th St, NY, 10014.

Software tools currently available tend to support single phases of the life cycle, this most often being the coding and implementation phase. Recent advances in tool concepts are leading towards the integration of several individual tools into a composite software development environment supporting a number of phases.

The emergence of these integrated tools will have a significant impact on methodologies, as several development phases will be incorporated into the tool itself. In particular, automated tool research is aiming at providing the capability to generate programs directly from the task specification. This approach has a number of significant advantages:

(a) The task specification directly leads to the code and is the only entry point.

(b) Because the specification can be executed, its validity can be checked immediately.

(c) Documentation requirements are reduced, as the code is regenerated from the corrected specification.

(d) Automatic translation will lead to correct code, so that no time and effort need be expended in testing and verifying code.

Automation forces the software designer to expend a great deal of effort on the requirement and specification phases of a project. Engineers will spend all of their time creating a specification and evaluating results.

Using such tools places very heavy emphasis on writing a complete and consistent specification in a formal language. This language must define what a program has to do, rather than how it will do it. Some progress has been made in developing suitable specification languages(ref.4), but much remains to be done.

How specifications can best be made complete and unambiguous is not yet settled. One school of thought advocates a graphical approach(ref.16), defining a hierarchical decomposition interactively on a workstation. The major problem with this 2-dimensional depiction is the complexity of the display produced for a system of realistic size. The alternative approach uses textual representation. Some combination of both approaches appears to offer a viable long term solution(ref.17).

Until these 'program writing' tools become more generally applicable, the majority of software development will continue to be performed using the traditional life-cycle approach, and making use of software tools designed to support specific phases. However, the chosen methodology should allow for the incorporation of new tools when they become available.

## 7. DATA MANAGEMENT

A significant factor in the management of software complexity, particularly for a tactical CCIS, is the management of the data and design of the data structures, used by the system. The importance of data management is recognized by many software design methodologies in that the identification of data elements and the way they are manipulated is given high priority in the early software design phases. Data element analysis and data flow diagrams are two such design techniques(ref.9,10,11).

The development of Data Base Management Systems (DBMS) as efficient self-contained software packages has been of considerable assistance in removing the complexities of data structure implementation from the design of the application software which uses those data structures. Designers can therefore devote more time to making sure the data structures meet the design requirements, and that the software meets the user's needs.

Most modern DBMS include more than just a means of storing data. They include the following major software capabilities:-

(a) data definition language

(b) data dictionary

(c) data access language

(d) screen forms manipulation package

(e) data back-up procedures

(f) data storage and retrieval methods.

All application software should be written to access data via the DBMS and should follow the standard interface to the DBMS for data access and update. This disciplined approach helps to keep data usage visible and therefore more manageable. It is recommended that a DBMS be utilized in any tactical CCIS in order to assist in managing the complexity of the software development task.

Modern data base management is tending towards the relational data base model first proposed by E.F. Codd in 1970. This model has the advantage of simplicity, elegance and flexibility, when compared to others such as the network or hierarchical models. A discussion of the relative merits of DBMS models will be found in Appendix II. The relational data base model is recommended for all tactical CCIS software development.

A DBMS can be implemented as a software package which can be called by the application programs. Alternatively, the DBMS could be implemented on a special purpose processor, called a Data Base Machine (DBM) which runs in parallel with the application processor. In either case, the separation of functionality through use of a DBMS aids the management of the software development, and allows more freedom for choosing a system which best suits the system requirements. Data Base Machines are further discussed in Appendix II.

One data base access language is rapidly becoming a standard for relational data bases. This is Structured Query Language (SQL) which can be used either for screen-driven direct queries of the data base, or by application programs for implementing a specific user function. It is recommended that SQL be chosen as the query language for all tactical CCIS developments.

While a DBMS may not be applicable to all software development projects, their use in a tactical CCIS software development is seen as essential if the data design and application software interface are to remain manageable. The choice of a suitable relational DBMS will significantly reduce the cost of developing tactical CCIS software, but at the possible price of requiring additional computing power.

## 8. ADA PROGRAMMING LANGUAGE

The US DoD initiated, in 1980, the development of a programming language which would become the standard one used throughout the US Department of Defence. This aim has been achieved, in that from 1985, all new projects for the US Armed Forces are obliged to use Ada as the programming language for embedded computer systems. However, the change to the use of Ada will not occur overnight, as some existing systems will continue to be developed in a non-Ada language for some time to come.

It would seem sensible for Australia to adopt a similar policy on a standard language for command, control and information systems, particularly as few tactical computer systems have yet entered service. There are many advantages which would flow to the Australian Defence Forces through the use of one programming language.

(a) Programmer training. Less training would be required when moving programmers from project to project, both because the same language would be used, and also because the programming environments would be similar (Refer to the next section on Ada Programming Environments).

(b) Software reuse. A standard language will allow the development of libraries of standard software modules which could be used in many projects.

(c) Software maintenance. Programs written in Ada are much easier to maintain than in other languages. In addition, Ada features are so comprehensive that machine language routines (which are notoriously hard to maintain) are unnecessary.

However, such a policy is not without its difficulties:

(a) A number of existing or proposed Defence information systems are based on US software packages, with modifications incorporated by Australian support centres to bring the packages into line with Australian requirements. Changing to Ada would not be feasible unless the US supplier released an Ada version. In that event, all the Australian modifications would need to be reapplied to the new Ada version. If it was desired to keep to the old version, then support from the supplier would probably fall to zero as the old language expertise was phased out. It would seem sensible to build up an Ada programming team at the Australian support centre, so that the transition could be completed before US support for the old system ceased.

(b) The number of programmers experienced in Ada will remain low for some time until the use of Ada becomes more widespread in Australia. Ada is included in most Computer Science curricula, so that new graduates have at least some knowledge of the language. Ada is a defence sponsored language. Commercial organizations will have little need to convert to Ada unless they intend responding to defence software contracts. However, if Ada lives up to expectations, it will be increasingly used in both defence and non-defence applications. This process could be hastened by the Australian Defence Department sponsoring educational programs to increase the availability of Ada programmers, and encourage the more general use of Ada in industry. Industry seems reluctant to lead, but prepared to follow.

When the decision to use Ada is made for a project, there will probably already exist some software written in a non-Ada language. This software could be rewritten in Ada, thus incorporating any new advance in program design that Ada permits. Alternatively, the old program could be automatically translated into Ada, statement for statement, and therefore not taking advantage of new language constructs that Ada supports. It would seem sensible to directly translate into Ada small programs which were not critical to the project, but to redesign and rewrite in Ada, programs which are critical, or have a long lifetime. Maintaining Ada programs which do not reflect good Ada style would be difficult and void one of the major advantages of Ada. Some tools are beginning to emerge to aid in the translation process(ref.18), and they can be useful in some circumstances. A discussion of the utility of Pascal to Ada translation is included in Appendix I.

The problems that may occur by introducing Ada as a standard language are transient ones, although they may require effort in the short term. The advantages of introducing Ada, on the other hand, are long term ones, with benefits which will increase with time. The longer the move to Ada is deferred, the more significant will be the effect on a larger number of developing projects, and the harder it will be to introduce the standard. It is recommended that Ada be adopted as the ADF standard computer language without delay.

## 9. ADA PROGRAMMING ENVIRONMENT

As well as specifying Ada as a programming language, the US DoD also attempted to define a programming environment which would provide a set of common facilities which would be independent of the computer on which they would run.

Reducing the proliferation of support environments will allow for more effective transfer of managers and programmers from one project to another, as well as provide for more uniform supervision of contracts by the DoD.

The development of an Ada Programming Support Environment (APSE) was intended to be guided by a series of documents which would provide a progressively more detailed definition. However, an APSE is far more complex to define than is a language, and general agreement on what features to include has been slower to achieve than for the language itself. The APSE features are outlined in the latest document called 'Stoneman'(ref.19).

The aim of the APSE is to define:

(a) a set of user-friendly tools to improve both management control of life-cycle processes and programmer productivity,

(b) a configuration control data base, and

(c) a Kernel APSE (KAPSE) interface to enhance portability.

The APSE provides tools for Ada program development. These include editors, libraries, library managers, debuggers, frequency and timing analysers, file copiers and other file handling tools, data base tools, and program loaders. Some of these will be discussed in the next section.

The APSE tools must be designed to run on different machines(ref.20). The machine dependent routines are defined as a separate Kernel APSE which supports a standard interface to the APSE tools. The KAPSE defines a number of Ada packages which provide the interface services. The package bodies are implemented to support a particular machine environment. In this way, the APSE tools can be easily ported from one machine to the other.

The first APSE to achieve validation was the US Army's Ada Language System (ALS) developed under contract by Softech, for a VAX running the VMS operating system. The US Navy and US Air Force are also developing their own APSE.

The 'Stoneman' APSE requirements document does not refer to any specific software methodology. However, it is expected that APSEs will emerge which will support individual methodologies.

When planning to use Ada as the programming language for a project, it is equally important to select an environment which will support the standard APSE, as well as incorporate the tools needed to support the chosen methodology.

## 10. SOFTWARE DEVELOPMENT TOOLS

It is only relatively recently that automated software tools have emerged from the research laboratories and have been offered as reliable, robust products in their own right. Obviously such tools have to perform flawlessly in support of a project, or no-one would ever be prepared to use them. A large project could suffer badly if some tool around which the project was structured, was found to be unreliable and prone to failure.

When a particular tool is not commercially available, it may be necessary to build one using general purpose packages such as relational data base systems, screen management software and a structured data base query language. While such special tools may not be very efficient, they may, nevertheless, be very useful. Such a tool may be needed, for example, to support the requirements phase in the form of a requirements dictionary. The purchase of tools, training of staff in their use, development of special tools and their maintenance will require the permanent allocation of staff for this purpose. Software tools are not unlike 'tools-of-trade' in many other disciplines, and the importance of such tools should never be in dispute, or underestimated.

It must be realised that the use of software tools to support a methodology will impose a significant load on the computer and its storage devices during development. This development load must be taken into account when sizing the computer and its peripherals to be used for the programming support environment.

As mentioned in Section 5, software support tools are appearing to support one or more phases of the software development life cycle. This section will discuss the type of assistance that may be provided by these tools.

## 10.1 Requirements tools

Requirements tools assist in turning a user requirements document into a definitive statement of requirements on which to proceed. This process is termed 'structured analysis' in references 9,10,11. Requirements analysis is not a well understood area, and Section 5 noted that consensus on the best method has not yet emerged.

(a) Directly executable specifications are one technique for providing immediate feedback to the user and designer that the system is on the right track(ref.4).

(b) Rapid prototyping is a method useful for quickly providing the user with examples of user interfaces, such as screen menus and formats, as well as menu switching for controlling the task flow. These tools go part way along the path that leads to a test bed, but uses a general purpose approach(ref.21).

(c) Requirements specification languages provide a formal way of capturing the data flows, tasks and procedures which are inherent in the user requirement. Formalizing the needs helps check for consistency and completeness of the specification(ref.22).

(d) Data flow diagrams are a very useful 'tool' for documenting a system from the point of view of data, rather than of control or of functionality. Users can quite readily follow these diagrams, and quickly identify any deficiency in the basic analysis. Data flow diagrams allow the system to be more logically partitioned into conveniently managed slices.

(e) Requirements dictionary is a data base containing details of each identified requirement, and in which program modules it is addressed. This tool may be application dependent and may need to be configured for a particular task. It is a useful tool for ensuring that each requirement has actually been addressed in some module, and can be referred to by name.

## 10.2 Design tools

The functions of a design tool are to capture the logic and structure of a program at a level higher than code, and to devise a model of the proposed solution to the problem. Cross referencing with the requirements dictionary can check for completeness and consistency.

These tools include Program Design Languages, Design Data Dictionaries and graphically based design aids for representing structures and relationships.

A number of Program Design Languages are available, but those based on Ada are of particular interest(ref.23,24,25). The process of program creation is traditionally divided into two phases - design and implementation. The purpose of design is to define suitable modules, with implementation actually producing the code. Ada contains elaborate constructs to support the implementation of program modules or packages. Using Ada as a PDL as well as an implementation language may seem absurd. However, the higher levels of a system can be written in Ada, leaving details of procedures and packages open(ref.26). In some environments, this skeleton program can be executed after a fashion to check for consistency across module interfaces.

Anna (for Annotated Ada) is an Ada based PDL from Stanford University(ref.27). It achieves executable Ada extensions that permit programs to be verified. The sequences "-|" and "-:" are ignored by an Ada compiler, but are used to indicate Anna statements to an Anna compiler. Programs are designed in Anna, and then Ada code is added to implement the required functions. Anna statements are thus interspersed with Ada code and become high level comments to the Ada program. The Anna statements can be stripped from the Ada program to provide documentation.

A feature of Ada is its information-hiding achieved through the use of package specifications and a separate package body. The package body can be changed or improved at any time without affecting its calling process, so long as it fulfills the specification. The package body can thus be kept invisible from the user. However, to understand the visible services an Ada package provides, the user often needs to study the package body. Anna permits the addition of syntactic information to the visible specification of Ada packages, thereby providing an understanding of how the package body works.

Software design tools form a basic part of software methodologies. It is important to choose design tools which are integrated or at least compatible with the requirements tools discussed above. In particular, the requirements dictionary and design dictionary have much in common and the ability to cross reference from one to the other is very valuable.

## 10.3 Coding tools

Tools to assist the programming phase of the software development process were among the first to appear, and so are the most numerous and best developed. Nevertheless, only about 20% of a large software project is devoted to producing code. Early coding assistance provided was in the form of assembly language assemblers, high level language compilers and high level language interpreters. Text editors subsequently emerged as valuable tools to support programming. Since then, a number of tools have appeared to assist the coding process, and are discussed in this section.

### (a) Source code editors

Initially program editors were line based to be usable on the printer-type terminals then available. As video display terminals began to proliferate, so editors changed to give programmers more visibility to the edited text, considerably improving productivity. These screen-based editors are now a relatively mature and sophisticated product, and are an essential tool for any programming environment.

The next step in sophistication that is now becoming available is the syntax-directed, or context sensitive editor. These editors are language specific, and contain details about the structure of the language being used. Using windows and menus, the programmer may select a particular language construct. The editor then places the basic structure into the program. The programmer then adds coding for his particular application to complete the code segment. This assistance is of particular importance for the Ada language where the number of possible statement types is very large. Only one very familiar with the language could expect to remember all the language constructs, and perhaps all the details of only some of the more often used ones. The editor thus behaves as an intelligent aide-memoire which can directly influence the way programs are generated.

It is possible also to select pre-defined package definitions via the editor, which will not only facilitate program creation, but ease the documentation burden on the programmer. For example, when a program is initially created, a standard one or two page package description header could be invoked, with the programmer filling-in entries to describe his application. In large software developments, it is essential to provide copious comments in programs, not only for the programmer's benefit, but also for later maintenance. It is not uncommon for up to 80% of a source code file to be comments. Readability is enhanced if comments are not interspersed with procedural code.

An extension of the 'intelligent' editor concept is the provision of a choice of, not only language constructs, but complete code segments. These segments would be defined by examining successful programs and building up a library of code segments. An index system related to function would be used to locate an appropriate segment from within the editor.

An important feature of these language sensitive editors is that they can directly check the program for syntax errors, that is errors in the use of the language. This means that most programming errors would be found before programs were submitted to the compiler. In an integrated editing/compilation environment, the need to perform a syntax check by the compiler could be avoided altogether, as these checks would have already been performed by the editor.

The syntax checking process needs to be user-involved, to allow for partly completed language statements. These would occur particularly when code was moved from one part of the program to another. Automatic syntax checking could become irritating if the programmer were to be continually reminded of errors he already knew about and was in the process of correcting.

(b) <u>Source code managers</u>

Source code control tools maintain a history of every change made to a program module: which programmer made the change, when it was made, and why it was made. Keeping a record of the status of program modules, and a history of their development is very important in large projects when the number of modules may be in the hundreds, or even thousands. Reference to a module's history is valuable during both development and testing, when the reasons for module modifications need to be understood by a programmer who may not have written the original program or performed any of the modifications himself. This situation would apply particularly to the maintenance phase of the life cycle.

When modules are initially created, the first 'version' must be registered to the source code manager (SCM). Whenever a registered module is to be modified, a programmer must indicate his intention to the SCM which records the fact that he is in the process of modifying a program module. The source code module will then be unavailable for editing by others. When the module is returned to the SCM, its version number is updated and the module can then be made available for others to use.

Saving a copy of every version of every module used in a project could be very expensive in file storage, particularly for large projects. To alleviate this problem SCMs usually store only the differences between one version and the next. Any version can thus be reconstituted by applying the differences to the first version through to the required version. This reduces the storage requirements for previous versions and results in an overhead of about 40% above that required for keeping the latest version of each program. This overhead is not significant when compared with the advantages gained by using the SCM.

The SCM tool can operate on any ASCII source file, including source code, specifications or documentation. Its use goes beyond the coding phase and helps keep track of all aspects of the project. In fact, the SCM provides a very powerful management tool by which progress on project deliverables, right down to individual module level, can be made visible to managers, giving a better picture of where attention should be focussed.

For example, if the number of versions of a module exceeds some norm, the reasons could be investigated to see if some problem were developing. Perhaps requirements have been changing, or too many coding errors have been found.

The SCM history files can have very important legal implications for projects involving life-critical functions such as passenger aircraft control computers. Crash investigators may need to refer to the history files of the modules that were installed on the computers in the crashed aircraft, to help define the cause of the accident.

A Source Code Manager is essential for any large software project where the coordination of many programmers writing many software modules, must be managed efficiently.

(c) Software release builder

During the course of project development, complete software packages will need to be delivered to meet various project milestones, each package containing specific modules of specific versions. This current release of software would be sent to the customer, while expanded and enhanced versions of modules would be being developed for the next release. There exists, therefore, the need for a tool which can reconstruct a software package release by firstly identifying the modules and versions which comprise the release, secondly, recompiling all the identified source modules and thirdly, verifying that the resultant code image is identical to the one that has been tested and approved for release. With a large number of modules involved in a project, the administrative load imposed by managing progressive software releases can be very high and prone to error. It can be very time consuming trying to identify which module version number is wrong when code images do not match, unless such a tool is used to enforce the software release discipline. For large software efforts, such a tool is indispensible.

(d) Rebuild Optimiser

Program modules form a hierarchy, with one module calling one or more other modules. Once a module is fully debugged and tested, it can be placed in the project library where it can be used by other modules. However, during program development, program modules will change. This will affect all those modules which call the changed modules. Depending on the extent of the changes, modules higher up the hierarchy may need to be recompiled and rebuilt, or only rebuilt. These dependencies need to be managed by a software tool called a Rebuild Optimiser.

A Rebuild Optimiser is a tool which, from a stored table of module names and calling dependencies, assists in the task of compiling and task building only these modules which are affected by changes to modules comprising the rebuild.

In Ada, for instance, modules consist of a package specification and a package body. If the body is changed, then all modules which call the body must be rebuilt to incorporate the new body. All modules which call the rebuilt module must also be rebuilt and so on up the calling hierarchy chain. If the package specification is changed, then all modules which call the changed module directly must be recompiled. Then all modules which call recompiled modules must be rebuilt, following on up the calling dependency chain.

It can be seen that keeping track of these dependencies manually would be a very complex task and very prone to error or omissions. It can also be appreciated that the rebuild process could be time consuming when a large number of modules was involved. Without a Rebuild Optimiser, there would be a tendency to rebuild or recompile ALL the program modules but, while this would ensure that all required modules were covered, the time taken could be prohibitive. An Ada compiler is required to perform a large amount of checking, and this slows down the compilation process. To compile modules unnecessarily would waste a considerable amount of time, as well as put an unnecessary load on both people and computers. A Rebuild Optimiser is seen as essential in a software project.

The creation and maintenance of the dependency data base could be a manual process, or it could be automatic by linking the process to the Source Code Manager. When a modified program module was returned to the Source Code Manager, the dependencies could be compared with the existing ones and the stored record adjusted accordingly. This would avoid relying on a manual procedure to identify any change to the dependencies, a process which could easily be in error.

(e) Static analysers

A static analyser takes the source code as input, and examines it for certain types of errors. These include:-

(i) variables that are used but never set,

(ii) variables that are set but never used,

(iii) code that can never be reached.

Static analysers were developed to support FORTRAN source code, and to make up for the lack of data type, mixed mode arithmetic or parameter calling checks in FORTRAN compilers. Modern language compilers such as Ada already include these checks so they remove these sources of error from correctly compiled code. However, static analysers are still needed to detect the above three enumerated errors, and also to document the program module by providing the following information:

(i) Calling tree,

(ii) Global variable cross reference,

(iii) Breakdown by statement type,

(iv) Local variable cross reference.

(f) Frequency analysers

A frequency analyser uses the source program as input and, using a set of test inputs to the program module, produces a histogram of the number of times each code statement is executed. In this way, it can be seen where the main program loops are, so they can be examined and reduced to the minimum required to perform the function. The results of a frequency analysis should not be used in isolation, as some very frequently executed statements could be very fast, while some used less frequently could be much slower. It can be counter productive to spend a great deal of effort on sections of the code which will not significantly effect the performance.

(g) Timing Analysers

A Timing Analyser (sometimes called a Performance Analyser) evaluates which parts of the program consume the most time. These analysers need to be carefully designed to avoid influencing the performance of the program under test.

It is often found that most of the execution time of a program is consumed in only a small part of the code. This tool helps to identify those parts of the program which need to be streamlined to execute as efficiently as possible, and which parts have little influence on the execution time. The tool can save wasting a lot of time improving code which is of no real importance.

Frequency and Timing analysers could be considered to be programming or testing tools depending on how they are used. They could be used quite effectively by the programmer before formally passing his module to the testing staff for integration testing in conjunction with other modules. These tools are primarily designed to help the programmer write better code.

## 10.4 Testing tools

Program testing consists of a scattered collection of rules of thumb, coverage measures and testing philosophies. Currently there is no tool which can take as input a program requirement, and produce verification that a particular program meets that requirement. Research is being directed towards this goal, but there is no testing procedure yet available that can guarantee that a program will perform exactly as required.

However, functional testing theory has reached the stage where specific types of faults can be detected using a functional analysis tool(ref.28). The fundamental idea in functional testing is that a program computes one or more functions, and to test the program, each function must be tested over functionally important test cases. However, until this tool is available, testing will continue to rely on methods and techniques which are less than perfect.

Program testing can be performed using a top-down or bottom-up approach, or both(ref.29). Top-down testing is useful when lower level modules are not yet available, either because algorithms are still being developed or interface hardware is not yet ready. Simple routines (or stubs) which simulate the operation of a particular module or group of modules can be written and linked with the top level calling modules. In this way the high level logic of the system can be confirmed at an early stage. Eventually the stubs would be replaced by the modules they represent.

Bottom-up testing is often used for these lower level modules, with groups of modules replacing the stubs developed for the top-down tests.

Initially, the programmer subjects his module to a range of tests by writing test routines which will provide an appropriate set of inputs which should produce a desired set of outputs. When the module's performance is satisfactory, the program is passed to the next level of testing where it will be integrated with a number of associated routines to form a larger module. This is then tested in a larger environment to detect errors before being passed on to further integration tests. At some point, the lower level modules will be integrated with the top level modules, and the complete integrated system test can then begin.

Module testing usually detects programming errors, integration testing will detect design errors, while acceptance testing will determine specification errors. That is, errors made early in the project may be the last to be detected and often the most expensive to correct. Top down testing helps to determine errors in the interpretation of user requirements, and provides the user with an early verification of the man/machine interface.

Testing is required during initial module development, but if a program module is ever modified, <u>all</u> testing involving that module should be <u>repeated</u>. Modification can lead to other errors and side effects which can be totally unexpected.

The effort required to thoroughly test program modules, particularly mission critical ones, should not be underestimated. The staff allocated to testing can often exceed the number allocated to program development. Several tools are available to assist the testing process.

(a) Debugger

A debugger is the most basic of testing tools, although modern debuggers have become very sophisticated. This tool allows the programmer to execute his program in single statements at a time and view the results, or to execute groups of code statements, then view the results. He can interactively monitor, control or modify the execution process of the program under test.

Modern debuggers work in a window environment, so that results can be viewed in one window while the executing statements are displayed in another window. Switching to the editor for program correction can be easily done before testing the program again.

A debug tool is essential in any program development environment.

(b) Path coverage

In mission critical software, and this must include most CCIS software, it is essential to know that every possible data flow path through a program has been tested. This involves setting up test inputs in a form that all paths should be covered, and then executing the routine under control of the path coverage tool. This tool provides an analysis of which statements have been executed and which paths have been followed. Some contracts demand documented evidence that all possible paths have been executed at least once. This is difficult and very time consuming to prove manually.

(c) Test manager

A test management tool maintains a data base of test cases and expected results for various software modules and suites used in the system. This tool can assist in writing the test specifications for each part of the system and in keeping a record of where failure to meet the test requirements has occurred. It could be linked to the source code manager tool to ensure that modules that have been modified are also retested, or else reasons for not testing need be given and recorded.

(d) Environment simulator

Instead of testing an embedded computer in its external environment (which may or may not yet exist), it is often very much worth while to provide a simulated external environment which will exercise the computer system with actual or simulated inputs. A command and control system simulator might be another computer which sends and receives representative tactical messages through a simulated or actual communications link. The simulator could also include human operators manning screens in some instances, with automatic recording of results for later analysis.

(e) Target machine emulator

A software emulator is sometimes necessary, particularly in cases where the computer which is used for software production is different from the one which will run the generated code. This will normally be the case for Ada program development, where the host computer could (for example) be a VAX, while the code generated is to run on (for example) a 68000 microprocessor. The machine code instructions of the target machine can be emulated on the host, enabling programs to be run as if they were on the target machine. This kind of testing is useful for finding logical program errors, but is not so useful where real time interrupts must interact with the program.

A host/target environment has always been advocated for the Ada language support system, with Ada compilers being able to generate code suitable for a number of target machines, including the host itself. A significant amount of debugging can be carried out on code which runs on the host, enabling logical errors to be detected. Later, code for the actual target machine would be generated, and then run using the target instruction emulator. Finally, the code can be loaded into the target machine itself and tested in a hardware/software environment simulator. Later, the complete hardware/software suite can be linked to the system in which it is embedded, and full integration tests run.

(f) In-Circuit Analyser

Considerable testing can be done using emulators and simulators, but once the software is loaded into its actual execution machine, faults can still occur. In fact, the 'obvious' errors have already been found by previous testing, with the more subtle ones still unknown. An In-Circuit Analyser(ref.30) assists the analyst to peer into the target computer and observe the action and reaction of the software to its environment.

In-Circuit Analysers provide a high speed interface to the target computer address and data bus, and to selected signals which are part of the operating environment. The simplest 'state' analyser decodes the instructions appearing on the bus and displays the execution sequence. More sophisticated 'symbolic' analysers recognise processor functions such as entry and exit from procedures, or when particular variables are changed. From the symbolic trace, the user gains a picture of program and data flow without being inundated with too much information. Timing and path coverage can also be provided.

The problem of adequately testing computers which form part of a larger system is a real one, and will often involve construction of special purpose software and hardware particular to the application. Adequate funding must be provided to perform these integration tests properly.

## 10.5 Software maintenance

Once a system has been accepted by the customer, the period of software maintenance begins. This involves:

(a) correction of any errors in the delivered code,

(b) correction of mis-understandings in the user requirement (by either customer or sponsor),

(c) minor enhancements to the system as a result of operational use,

(d) major enhancments due to increased expectations.

It has been estimated for the US DoD, that 80% of the total software life cycle effort is devoted to software maintenance. The whole thrust of the DoD Ada initiative has been to provide a language which will make program maintenance easier, and therefore cheaper.

Invariably, the programmer who writes a module is not the one who maintains it. The maintainer requires good documentation if he is to apply modifications and corrections with confidence that no side effects will be caused. It can help considerably if the original design specifications for a module are available to him. The documentation required also includes a self documenting language such as Ada, the use of standard comment header pages at the start of every program module, copious blocks of comments within the code, and the record of the module's development history. However, interspersing one or two lines of comments within procedural code can make the program logic hard to follow, and defeat the purpose of having the comments there at all.

The initial problem for the maintainer is to understand the module which needs to change. He may well need to look at and understand other modules which have some bearing on the modification, particularly for large changes. When sufficient understanding has been reached, the process of design, implementation, debug and test must be undertaken in as rigorous a manner as occurred when the program was originally produced. Thus, all the tools used for production must be capable of assisting the maintenance team too.

## 10.6 Office administration tools

All software projects generate large amounts of documentation in the form of specifications, manuals, minutes of meetings, progress reports etc. These documents often include diagrams and charts, some of which are required to be used in management presentations on overhead projectors.

An integrated package containing a word processor, graphics manager and a spread sheet is a very useful tool in the software development environment, in conjunction with a good quality, fast printer capable of printing graphics as well as text.

Other tools are also available which assist project documentation. These include a spelling checker, index generator, synonym dictionary and an "English teacher" for finding cliches, trite phrases, sexist language and 'dead wood'. It can also give a measure of difficulty of understanding of the document based on indicators such as the length of sentences.

It is essential for project management to be aware of the administrative tools that are available to assist the project, and to select the ones that are appropriate for their method of working. This process of tool selection is an on-going one and requires continual process of review, selection of new tools and staff training.

## 11. CONCLUSIONS

This report has discussed the means and methods currently available for developing software for large information systems. It is concluded that the use of a disciplined methodology is essential if the complexity of modern software systems is to be managed effectively. Full recognition must be made that the use of computer based tools to help in the software development process is not a luxury but an absolute necessity. Methodologies and design tools by themselves do not ensure success, but their intelligent application to a software project will significantly improve the likelihood that a project will reach a satisfactory conclusion.

## 12. RECOMMENDATIONS

(1) A well disciplined approach defined by a comprehensive methodology is essential for managing effectively large software projects. Large projects are much more than small projects scaled up.

(2) Before undertaking any task involving significant software development, it is imperative that user requirements be defined as accurately and as detailed as possible. Projects fail when requirements are ill defined and variable.

(3) Rapid prototyping or a test bed approach is recommended for reducing uncertainty in user requirements before the software design is frozen.

(4) Military Standard "Defence System Software Development (DOD-STD-2167)" should be followed, particularly if embedded software is involved.

(5) A methodology should be chosen which allows for the introduction of new software tools when they become available.

(6) Maximum use should be made of modern software tools to assist all phases of the software life cycle.

(7) Only extremely reliable and well supported software tools should be used for software development.

It is recommended that (at least) the following software tools be used during software developments:-

(1) Requirements dictionary,
(2) Rapid prototyping tool,
(3) Design dictionary,
(4) Syntax directed source code editor,
(5) Source code manager,
(6) Software release builder,

(7) Rebuild optimiser,
(8) Static analyser,
(9) Frequency analyser,
(10) Timing or performance analyser,
(11) Source code debugger,
(12) Path coverage tool,
(13) Test manager,
(14) Target machine emulator,
(15) Office administration tools such as word processors, graphics managers, general purpose data base managers.

(8) The permanent allocation of staff to select, develop and maintain software tools and train staff in their use, is seen as essential for any large software development project.

(9) A relational data base management system (DBMS) is recommended for use in all tactical command control information systems to simplify program development and reduce software maintenance problems.

(10) A relational query language such as SQL is recommended for adoption as the standard for all tactical command control information systems. The use of a general purpose relational data base is recommended for quickly developing special data dependent tools such as requirements and design dictionaries where these tools are not available commercially.

(11) The Ada programming language should be chosen as the standard language in all Australian developed defence projects, and for the support of as many systems purchased overseas as practicable.

(12) An Ada Programming Support Environment should be used for Defence related projects.

(13) The translation of existing software to Ada is useful in simple cases, but is not recommended for critical packages, or for programs which will have a long maintenance lifetime. Except in simpler cases, existing software should be redesigned and rewritten in Ada.

## 13. ACKNOWLEDGEMENT

The author is grateful to Mr P. Dart, for preparing Appendix I, based on his experience with the Ada programming language while at the Naval Combat Data Centre, Fyshwick, Canberra.

## REFERENCES

| No. | Author | Title |
|---|---|---|
| 1 | Hosier, W. | "Pitfalls and Safeguards in Real Time Digital Systems with Emphasis on Programming". IRE Transactions on Engineering Management Vol EM-8, No 2, pp.99-115, June 1961 |
| 2 | Brooks, F. | "The Mythical Man-Month". 1st Edition, Addison Wesley, MA, 1975 |
| 3 | Freeman, P. and Wasserman, A. (Editors) | "Tutorial on Software Design Techiques". 4th Edition, IEEE Computer Society Press, Silver Spring, MD, 20910, 1983 |
| 4 | Rochmore, J. | "Knowledge-based Software Turns Specifications into Efficient Programs". Electronic Design, Vol 33, No 17, pp.105-112, July 25, 1985 |
| 5 | Commonwealth Dept of Local Government and Administrative Services | "Commonwealth Purchasing Manual". December 1983 |
| 6 | US Department of Defence | "Defence System Software Development". Military Standard DOD-STD-2167, US Department of Defence, Washington, DC 20301, June 1985 |
| 7 | Wasserman, A. | "Information Systems Design Methodology". Journal of the American Society of Information Science, Vol 31, No 1, pp.5-24, January 1980 |
| 8 | Wasserman, A. and Freeman, P. | "Ada Methodologies: Concepts and Requirements". University of California, San Francisco, CA, USA SIGSOFT SOFTWARE ENG. NOTE (USA), Vol 8, No 1 pp.33-50, January 1983 |
| 9 | Yourdon, E. and Constantine, L. | "Structured Design Fundamentals of a Discipline of Computer Program and System Design". 1st Edition, Prentice Hall, NJ, 1979 |
| 10 | Page-Jones, M. | "The Practical Guide to Structured Systems Design". 1st Edition, Yourdon Press, NY, 1980 |

| No. | Author | Title |
|---|---|---|
| 11 | deMarco, T. | "Structured Analysis and System Specification". 1st Edition, Prentice Hall, NJ, 1979 |
| 12 | Department of Defence | "SPECTRUM Project Management/Systems Development Methodology". Department of Defence Circular Memorandum 95/85, August 1985 |
| 13 | Jackson, M.A. | "Principles of Program Design". 1st Edition, Academic Press, NY, 1975 |
| 14 | Jackson, K. and Simpson, GPCAPT H. | "MASCOT - A Modular Approach to Software Construction Operation and Test". RRE Technical Note No 778, October 1975 |
| 15 | Fichenscher, G. | "Use of the MASCOT Philosophy for the Construction of Ada Programs". RSRE Rep 83009, October 1983 |
| 16 | Schindler, M. | "Through Automation, Software Shapes Itself to the Task in Hand". Electronic Design, Vol 33, No 17, pp.87-38, July 25, 1985 |
| 17 | Martin, J. and McClure, C. | "Diagramming Techniques for Analysts and Programmers". 1st Edition, Prentice Hall, NJ, 1984 |
| 18 | Wolfe, A. | "AI Tools Automate Software Translation". Electronics, Vol 58, No 38, pp.59-61, September 1985 |
| 19 | US Department of Defence | "Requirements for Ada Program Support Environment". 'Stoneman' Report, US Department of Defence, February 1980 |
| 20 | Narfelt, K.H. and Schefstrom, D. | "Towards a KAPSE Database". IEEE Computer Society 1984 Conference on Ada Applications and Environments, pp.42-51, October 1984 |
| 21 | Leach, D., Parge, M. and Satko, J. | "A Methodology for Rapid Prototyping". IEEE 4th Annual International Conference on Computers and Communications, Scottsdale, AZ, pp.53-61, March 1985 |
| 22 | Gutag, J.V., Horning, J.J. and Wing, J.M. | "The Larch Family of Specification Languages". IEEE Software, Vol 2, No 5, pp.24-36, September 1985 |

| No. | Author | Title |
|---|---|---|
| 23 | Sammet, J. et al | "PDL/Ada - A Design Language based on Ada". ACM Ada Letters, Vol II, No 3, pp.19-31, November 1982 |
| 24 | Texas Instruments, Inc | "TI Ada PDL Manual". Texas Instruments Inc, Equipment Group, McKinney |
| 25 | Manuel, O. and Bonnet, C. | "Ada as a Programming Design Language for a Telematic Services Project". IEEE Computer Society 1984 Conference on Ada Applications and Environments, pp.89-94, October 1984 |
| 26 | Intermetrics Inc | "BYRON Program Development Language and Document Generator Users' Manual". Intermetrics Inc, Cambridge, MA, October 1985 |
| 27 | Luckham, D. and von Henke, F. | "An Overview of Anna, a Specification Language for Ada". IEEE Computer Society 1984 Conference on Ada Applications and Environments, pp.116-127, October 1984 |
| 28 | Howden, W.E. | "The Theory and Practice of Functional Testing". IEEE Software, Vol 2, No 5, pp.6-17, September 1985 |
| 29 | Miller, E.F. and Howden, W.E. (Editors) | "Software Testing and Validation Techniques". IEEE Computer Society Tutorial, 1978 |
| 30 | Ableidinger, B., Agawal, N. and Nobles, C. | "Real-Time Analyser Furnishes High Level Look at Software Operation". Electronic Design, Vol 33, No 22, pp.117-131, September 19, 1985 |
| 31 | US Department of Defense | "Military Standard: Ada Programming Language". ANSI/MIL STD 1815A, US Department of Defense, Washington, DC |
| 32 | Tompkins, H. | "In Defense of Teaching Structured COBOL as Computer Science". SIGPLAN NOTICES, Vol 18, No 4, pp.86-94, April 1983 |
| 33 | Mayoh, B. | "Problem Solving with Ada". 1st Edition, Wiley, NY, 1981 |
| 34 | Wolfendale, G. (Editor) | "Data Base Management Systems". Proceedings of the Joint ANU/ACS One Day Seminar held at the Computer Centre of the ANU, 17 November 1976, Australian National University Press, Canberra, 1977 |

| No. | Author | Title |
|---|---|---|
| 35 | Tsichutyis, D. and Lochoresky, F. | "Data Base Management Systems". 1st Edition, Academic Press, Florida, 1977 |
| 36 | Date, C. | "An Introduction to Database Systems". 2nd Edition, Addison Wesley, MA, 1977 |
| 37 | Deen, S.M. | "Fundamentals of Database Systems". 1st Edition, MacMillan, NY, 1977 |
| 38 | Malabarba, F. | "Review of Available Database Machine Technology". Naval Data Automation Command, Washington, DC, Unpublished paper, June 1984 |
| 39 | Pallard, J. | "The Database Machine is Now a Reality". Computer World, Australia, Vol 8, No 3, pp.26-27, July 19, 1985 |
| 40 | Myers, E. | "Database Machines Take Off". Datamation, Vol 31, No 10, pp.52-63, May 15, 1985 |
| 41 | Epstein, R. | "Why Database Machines". Datamation, Vol 29, No 7, pp.139-144, July 1983 |

THIS IS A BLANK PAGE

# APPENDIX I

# AUTOMATIC PASCAL TO ADA TRANSLATION

## I.1 Introduction

An installation that has decided to convert to using a new programming language has three basic options. Firstly, it may continue to use the old programs, and therefore the old languages, until they become redundant. Secondly, they may convert the old programs to the new language, either automatically from the old source, or manually from some level of the old specification. Thirdly, they may throw out the old programs altogether and begin afresh. These options are not incompatible; some combination of all three is likely to be used in any real situation and many factors need to be considered in choosing what combination will provide the greatest cost effectiveness in the long run.

This appendix is concerned with the merits and difficulties involved in the use of automatic translation from the programming language Pascal, to the programming language Ada. In considering this particular option, it will be necessary to make some comments on how this form of conversion would fit in with the overall conversion process.

This appendix is more detailed than that required for a report of this type. However, the details have been included here to provide some background to the assertion that:

(a) Ada program structures can be significantly different from that of Pascal, and

(b) the many advantages of the modern features of Ada could not be utilized by direct mapping from Pascal into Ada.

The comments on Ada included in this Appendix were prepared by Mr P. Dart, based on his experience with the language while at Naval Combat Data Centre, Fyshwick, Canberra.

## I.2 The software lifecycle

Although the topic at hand is one of automatic translation, there may arise some difficulties that could be circumvented 'handrulically'. Manually making changes of other than trivial complexity to the translator output will cause problems if upgrading of the original program is desired at any stage. The lifecycle of the translated program is critical in this context. If the program is expected to be in use for a period that is likely to include a number of upgrades, then a choice must be made as to whether the original programs or the translation must be changed. If the original source is to be upgraded then the danger in other than a trivial change to the translation is obvious. Also, the original source must be kept around for the life of the program.

If the original source is not to be upgraded, then it may be discarded (unless desired as backup). In this case, the translator output must be in an easily readable form. This requirement is not insignificant; producing comprehensible translations (suitable for upgrading by a programmer unfamiliar with the program) may be an order of magnitude more difficult problem than producing merely executable ones. Some of the reasons for this will be revealed in conjunction with the examples below.

## I.3 Pascal and Ada

The "Programming Language Ada Reference Manual"(ref.31) contains the following quote:

"...the language includes facilities offered by classical languages such as Pascal as well as facilities often found only in specialised languages."

The first part of the statement reveals why it is a relatively simple task to produce a Pascal to Ada translator. Almost every Pascal construct has a direct counterpart in Ada, allowing a simple mapping from one to the other. The output of using a translator that simply took advantage of this mapping, though, is unlikely to be considered good under any criterion other than correctness. This final statement will be justified in the next section.

## I.4 Specific language features

Following are a series of examples illustrating various features of the Ada programming languages. These features are discussed in relation to their Pascal counterparts, if any.

The first example given is an adaptation of a Cobol program that appears in the April 83 issue of "Sigplan Notices"(ref.32). Professor Howard E. Tompkins presents the original to highlight shortcomings in Pascal.

The next 3 examples are based on code that appears in 'Problem Solving with Ada' by Brian Mayoh(ref.3 ). The originals have been corrected and improved, but the resulting code is not intended to represent the best possible Ada code.

### I.4.1 Example 1

```
with text_io; use text_io;

procedure Future_Day_of_Week is

  type T-Day is (sun, mon, tue, wed, thu, fri, sat);
  subtype T_Positive is integer range 0..integer'last;

  package Day_Io is new Enumeration_Io(T_Day);
  use Day_Io;
  package Positive_Io is new Integer_Io(T_Positive);
  use Positive_Io;

  day: T_Day;
  elapsed_days: T_Positive;

begin
 put("Enter day of the week as three letters:");
 new_line;  -- to flush the buffer
 loop
  begin
   get(day);
   exit;
  exception
   when Data_Error =>
    put_line("Input not recognized as a day of week");
    put("Please try again:");
    new_line;     -- to flush the buffer
```

```
28   end;
29  end loop;
30  put("Enter future elapsed days:");
31  new_line;  -- to flush the buffer
32  loop
33   begin
34    get(elapsed_days);
35    exit
36   exception
37    when Data_Error =>
38     put_line("Input not recognized as a positive number");
39     put("Please try again:");
40     new_line;     -- to flush the buffer
41   end;
42  end loop;
43  put("Future day is a ");
44  put(T_Day'val(T_Day'pos(day)+elapsed_days)rem(T_Day'pos(T_Day'last)+1)));
45  new_line;
46 end Future_Day_of_Week;
```

This example is largely an exercise in input handling. To achieve the same functionality in Pascal would be a long, tedious task in comparison. Some form of search on an array of packed arrays could be used to convert the input string to an ordinal value, and the 'repeat read until no errors' loops would be considerably more complex. The declaration on line 5, which would probably be thought to appear in a similar form in the Pascal equivalent, would in fact be of little use since conversion from an ordinal value to an enumerated type in Pascal is a troublesome task. Also, it should be noted that if some other circular enumerated type was to be used instead of days of the week (eg months of the year), only line 5 and the I/O strings need be changed. (The naming conventions could have been chosen more generically if this was desired). Line 44 is the key to the flexibility of this Ada program. The necessary constants and functions are direct attributes of the enumerated type, thereby allowing the functional style of statement in line 44 that is likely to appear in many Ada programs.

Since a Pascal program performing the same task as Example 1 would be significantly more complex, the automatically translated version of such a program using a straight forward mapping would also be far more complex. For this reason, any translations of this sort would not reap the benefits in readability, changeability, and efficiency that would result from a direct coding in Ada.

### I.4.2 Example 2

```
 1 with Text_Manipulator;
 2 procedure Edit_Data is
 3  use Text_Manipulator;
 4  T: Text;
 5 begin
 6  T := Get_Text;
 7  loop
 8    Correct(T);
 9  end loop;
10 exception
11  when No_More_Input=>
```

```
  begin
   Renumber(T);
   Put_Text(T);
  end;
end Edit_Data;
```

Example 2 is a simple main program consisting of a simple loop. If written in Pascal, a 'while' loop would be used. In fact, for the Ada version, as it stands, a pre-tested loop would be more readable. The form used in example 2 would be useful if a number of different terminating conditions could arise; the exception handler would then have the corresponding number of 'when' clauses. This, then is another example where the sort of loop control used in Pascal would not map into the style suggested for a direct Ada coding.

An Ada package 'Text_Manipulator' is introduced in this example. One of the greatest advantages of the Ada language is its packaging facilities, and full use of them must be made to gain full benefits from the language. Packaging is non-standard in Pascal. Versions that do have some form of separate compilation facility vary from version to version, and generally only provide a very restricted form. A translator that took Pascal source en masse would not be able to take full advantage of the package facilities in Ada, and Ada libraries may already contain code that could be used in an Ada version of the Pascal source.

## I.4.3 Example 3

```
with Line_Manipulator;
package Text_Manipulator is
 use Line_Manipulator;

 type Text is private;
 No_More_Input: Exception;

 function Empty return Text;
 function Get_Text return Text;
 procedure Put_Text(T: Text);
 procedure Correct(T: in out Text);
 procedure Renumber(T: in out Text);
private
 type Text is array (1..20) of Line;
end Text_Manipulator;

with Text_Io, Line_Manipulator; use Text_Io,Line_Manipulator;
package body Text_Manipulator is
 package int-io is new integer_io(integer); use int_io;

L_Index, P_Index: Position;
L:Line_Manipulator.Line; C: Character;
Zero: constant integer := character'pos('0');

function Empty return Text is
begin
 return (1..Text'last => (1..Position'last=> Ascii.LF));
end Empty

function Number return integer is
 M : integer;
begin
 M := 0;
 loop
```

```
  C := Next_Character;
  if C in '0'..'9' then
   M := M * 10 + character'pos(C)-Zero;
  else
   return M;
  end if;
 end loop;
end Number;

procedure Correct(T:in out Text) is
begin
 L := Get_Line;
 L_Index:=Number;
 if C = ':' then
  T(L_Index) := Get_Line;
 else
  T(L_Index..Text'last) :=
   (L_Index..Text'last => (1..Position'last => Ascii.LF));
 end if;
exception
 when others => raise No_More_Input;
end Correct;

procedure Put_Text(T: Text) is
begin
 L_Index := 1;
 loop
  if T(L_Index)(1) /= Ascii.LF then Put-Line(T(L_Index)); end if;
  exit when L_Index = Text'last';
  L_Index := L_Index + 1;
 end loop;
end Put_Text;

procedure Renumber(T: in out Text) is
 U: Text; M: integer;
begin
 M := 2; U := Empty;
 for L_Index in Text'range loop
  exit when M >= Position'last - 1:
  if T(L_Index)(1) /= Ascii.LF then
   U(M) := T(L_Index); M := M + 2;
  end if;
  end loop;
  T := U;
 end Renumber;

 function Get_Text return Text is
  T: Text;
 begin
  T := Empty;
  L_Index := 2;
  loop
   T(L_Index) := Get_Line;
   L_Index := L_Index + 2;
  end loop;
 exception
  when others => return T;
 end Get_Text;

end Text_Manipulator;
```

Lines 1 to 15 of Example 3 are the interface specification to the 'Text_Manipulator' package. The ability to hide the internal representation of a data type is demonstrated by lines 5, 13 and 14.

A problem that most Pascal programmers are familiar with is that of the 'N+1' constant. Often, a program has the constant N, but requires the constant 'N+1' for a later type declaration. Ada solves this problem by allowing constant expressions. A simple example is on line 23, but these expressions may be used as array bounds in declarations (eg in the form: TypeId'last+1), or anywhere else desired.

Aggregates and slices, as used on lines 27, 51 and 52 are another feature of Ada that a translator would find difficulty making use of. Together with attributes, these features encourage a functional style of programming that is impossible in Pascal. The ability to take a slice of an array allows an Ada compiler to make use of a 'block move' instruction (if any), on the target machine, and so can be useful in terms of efficiency. (There are versions of Pascal, such as UCSD, that have a 'generic' copy procedure to allow this efficiency. The difficulty of mapping the use of procedures such as this into a strongly typed language like Ada can be left to the reader's imagination).

Lines 72 to 77 are an example of a dual exit loop; a combination of the 'for' and 'while' loop constructs in Pascal. The function 'Get_Text' also demonstrates a different (though not necessarily desirable) form of control. Neither of these forms would result from the translation of Pascal source.

## I.4.4 Example 4

```
 1 package Line_Manipulator is
 2  type Line is new String(1..80);
 3  subtype Position is integer range 0..Line'last;
 4  Limit, Spaces: integer;
 5  function Get_Line(L: Line);
 6  procedure Put_Line(L: Line);
 7  function Line_to_Character return character;
 8  function Next_Character return character;
 9  procedure Insert (Old_Line: Line);
10 end Line_Manipulator;
11
12 with Text_Io; use Text_Io;
13 package body Line_Manipulator is
14  Index: Position; L: Line;
15
16 procedure Insert (Old_Line: Line) is
17 begin
18  Limit:= 1;
19  while L(Limit) /= Ascii.LF and Limit < Position'last loop
20   Limit:= Limit + 1;
21  end loop;
22  L(1..Limit):= Old_Line(1..Limit);
23  Index:= 0;
24 end Insert;
25
26 function Next_Character return character is
27 begin
28  if Index < Position'last then Index:= Index + 1; end if;
29  return L(Index);
30 end Next_Character;
31
```

```
32 function Line_to_Character return character is
33  C: character;
34 begin
35  loop
36   C:= Next_Character;
37   case C is
38    when 'a'..'z' | Ascii.LF=> return C;
39    when others           => null;
40   end case;
41  end loop;
42 end Line_to_Character;
43
44 function Get_Line return Line is
45  C: character;
46 begin
47  Limit:= 0;Spaces:= 0;
48  loop
49   get(C);
50   if Limit < Position'last then
51    Limit:= Limit + 1;
52    L(Limit):= C;
53    if C = ' ' then Spaces:= Spaces + 1; end if;
54   end if;
55   exit when C = Ascii.LF;
56  end loop;
57  Index:= 0;
58  return L;
59 end Get_Line;
60
61 procedure Put_Line(L: Line) is
62 begin
63  Limit:= 1;
64  loop
65   put(L(Limit));
66   exit when L(Limit) = Ascii.LF or Limit >= Position'last;
67   Limit:= Limit + 1;
68  end loop;
69 end Put_Line;
70
71 end Line_Manipulator;
```

Some of the gains in readibility, and therefore changeability through the use of Ada instead of Pascal are visible is lines 3, 19 and 28 of example 4. The use of the 'last attribute often saves the declaration of unnecessary constants, and allows simple changes to be made at the right place; where the declaration of the data type is made.

Line 38 demonstrates the use of ranges in the Ada case statement. This particular example could be duplicated in Pascal with an if statement and the set 'in' operator, but a more complicated series of case labels would require a series of nested if-then-elses in Pascal. A translator producing a similar series of if-then-elses in Ada would not give the Ada compiler the opportunity to produce a jump table (as often used in implementing a case statement) and would therefore be less efficient than the hand coded Ada equivalent.

Another loop control construct appears on lines 64 to 68. This form of 'middle-tested' loop is not available in Pascal and usually requires the use of boolean(s). The result is likely to be less readable and less efficient.

## I.4.5 Example 5

The next example is of a 'CSP' style 'sort' of an Ada string and is only likely to be of interest to someone with reasonable understanding of Ada.

```
with text_io; use text_io;
with TSort; use TSort;

procedure Test_TSort is
begin
 put_line(Sort("edcbaabcde"));
end Test_TSort;


package TSort is

function Sort(O_String: in string) return string;

end TSort;


with text_io; use text_io;

package body TSort is

subtype T_Sort_Element is character;

task type Sort_Unit is
 entry Store(Sort_Element: in T_Sort_Element);
 entry Retrieve(Sort_Element: out T_Sort_Element);
end;

type A_Sort_Unit is access Sort_Unit;

task body Sort_Unit is
 Next: A_Sort_Unit;
 O_Sort_Element, Temp_Sort_Element: T_Sort_Element;
 Count: Natural:= 0;
 Unit_Name: constant string:= "Sort_Unit";

begin
 Select_Loop: loop
  select
   accept Store(Sort_Element: in T_Sort_Element) do
 Temp_Sort_Element:= Sort_Element;
  end Store;
  if Count = 0 then
   O_Sort_Element:= Temp_Sort_Element;
  else
   if Count = 1 then
 Next:= new Sort_Unit;
   end if;
   if Temp_Sort_Element >= O_Sort_Element then
 Next.Store(Temp_Sort_Element);
   else
 Next.Store(O_Sort_Element);
 O_Sort_Element:= Temp_Sort_Element;
   end if;
  end if;
  Count:= Count + 1;
 or
  when Count > 0 =>
accept Retrieve(Sort_Element: out T_Sort_Element) do
 Sort_Element:= O_Sort_Element;
   end Retrieve;
Count:= Count - 1;
exit Select_Loop when Count = 0;
Next.Retrieve(O_Sort_Element);
  or
   terminate;
  end select;
 end loop Select_Loop;
exception
 when others=>
  put_line(Unit_Name & "- exception"); raise;
end Sort_Unit;

function Sort(O_String: in string) return string is
 SortL: A_Sort_Unit;
 Out_String: string(1..O_String'last):= (others => ' ');
 Unit_Name: constant string:= "Sort";

begin
 if O_String'Length = 0 then
  return "";
 else
  SortL:= new Sort_Unit;
  for I in O_String'range loop
   SortL.Store(O_String(I));
  end loop;
  for I in Out_String'range loop
   SortL.Retrieve(Out_String(I));
  end loop;
  return Out_String;
 end if;
exception
 when others =>
  put_line(Unit_Name & "- exception"); raise;
end Sort;

end TSort;
```

This last example is not meant to be a practical Ada program. It does work and is a useful demonstration of the tasking facilities in Ada. No more than a few versions of Pascal make even a token gesture at providing tasking facilities. Thus no translator would make use of Ada's tasking facilities in translating from Pascal. The question that this leaves us with is: Which, if any, Pascal programs, if written in Ada, would make use of the tasking facilities? In fact there are likely to be very few, but one important example is that of simulation. A simulation program written in Pascal would have to provide many of the facilities already available in Ada from scratch.

There are many features (particularly with respect to control) that have not been pointed out in the above examples. These should be evident to any Pascal programmer who scans this code. The author has noted from his own experience that for an (experienced) Pascal programmer. Ada seems like an extended form of Pascal. This effect soon wears off, as the style of the Ada language itself becomes evident. Much of this style comes from the use of Ada facilities not available in any form in Pascal; eg aggregates, slices exception handlers. A more functional style seems in order; often, what is written as a procedure in Pascal because of control information that needs to be passed back, can be written as a function in Ada with the control handled by exception handlers (if need be).

## I.5 Conclusion

It has already been stated that (correct) automatic translation of Pascal to Ada is unlikely to be a complex task. The above discussion suggests, though, that many advantages of the Ada language would be lost if output code from a translator was to be used for any significant period of time. Some of the problems in translation mentioned above could be circumvented through the use of complex semantic analysis in the translator, but this is unlikely to be a total (or simple) solution.

The place for automatic translation of Pascal to Ada would seem to be only in the early stages of the conversion of an installation to Ada from Pascal. The old programs could be translated to get the installation as close as possible to an 'Ada only' site, while work is under way to rewrite the programs in Ada. It is important here to realize that rewriting in this context means producing a completely new design, since a design with Pascal in mind is unlikely to suit Ada. Thus only language independent phases of the original design would be of use in rewriting. Priority should be given to the most 'dynamic' of the translated programs. If a program is likely to require modification in the near future, then it should have been rewritten by the time the need arises.

Automatic translation from Pascal to Ada is likely to be most cost effective if used in this limited scope.

# APPENDIX II

## DATA MANAGEMENT IN A TACTICAL COMMAND INFORMATION SYSTEM

A Command Control and Information System (CCIS) is required to store data about its operation and environment, in order to assist staff to make decisions to support the commander's objectives. Consequently, a major task which must be performed is the collection, processing and distribution of this data so that it adequately meets the needs of the staff. Data in a CCIS must be recognised as a valuable resource which must be managed effectively to make maximum use of a limited commodity.

Systems which need to store only a small amount of data can do so quite effectively through the use of sequential or random access files under the computer's operating system. However, data requirements have a habit of growing as systems mature, so data management may quickly become a significant part of the support effort of even a small system.

Without proper data management, systems with medium to large data storage requirements become unwieldy, as the effort to support a large number of files requires many online computing demands. It is under these circumstances that a Data Base Management System (DBMS), is essential.

### II.1 Data Base Management Systems

Data bases are used to store information about objects in the real world and their relationship to each other. In addition, mechanisms must be provided to retrieve this information in an effective manner so that the information can be of use. The definition in(ref.34) summarises this quite nicely. A data base is there defined as:

"a generalised integrated collection of data which is structured on natural data relationships so that it provides all necessary access paths to each unit of data in order to fulfil the differing needs of all users".

A number of advantages flow from a well organised data base(ref.35).

(1) the amount of redundancy in the stored data can be reduced to that essential for failsafe operation,

(2) problems of inconsistency in the stored data can be avoided (to a certain extent),

(3) stored data can be shared,

(4) standards can be enforced,

(5) security restrictions can be applied,

(6) data integrity can be maintained through appropriate authorization,

(7) conflicting user requirements can be balanced,

(8) data independence can be achieved ie the way data is stored is independent of how the data relationships are defined.

To achieve these advantages, it is essential for a data model to be defined describing the data to be handled by the data base system. The data model is a set of guidelines for the representation of the logical organisation of the data. It is a pattern showing how the named logical units of data are organised in their relationships to each other, without saying anything about how data is physically stored.

Programs supporting a particular application make use of a logical 'view' of the data base, that is, logical view encompasses as much of the data model as the application is concerned with. Data models and views are completely logical. This is, no implementation in terms of physical storage structure is implied. In fact, it is possible to implement the data base in a number of different ways, all supporting the same data model.

The prime advantage of data independence, is that physical data structures can be changed without needing any change to the logical view, and therefore to the application program. Program maintenance is simplified, and system implementers have more freedom to add new data fields, or to rearrange data structures to fine tune data manipulations. In addition, new hardware technologies can be introduced without causing application reprogramming.

A Data Base Management System then, is a set of procedures and data structures that isolate the applications from the details of creation, retrieval, storage, modification, security and physical storage structure of computerised data bases. The important consideration is that no matter how the data are organised or reorganised, the DBMS is always able to provide an application with the same view.

The DBMS is thus an interface between the application program and the physical copies of the data. Although a DBMS is by no means a necessity in an information system, it is becoming increasingly clear that a DBMS is required for effective management in an information systems environment.

DBMS's provide a number of valuable service facilities to support applications:

(1) Performance optimisation: facilities to evaluate performance and tune the data base to optimise response;

(2) Concurrent useage: the use of the data base by more than one user at a time;

(3) Data protection: protection against loss or damage of data in the data base and the protection of the confidentiality of the data from unauthorised persons;

(4) Data organisation: a disciplined approach to managing the data base.

## II.2 Data Models

In the tactical environment, staff have a perception of the totality of events and occurrences on the battlefield, which corresponds to the real world. As an aid to thought processes and communication with others, this real world is modeled by storing certain information about it. Classes of objects are defined, with each object class having certain attributes. For example, object type UNIT may have attributes UNIT NAME and LOCATION. The set of possible values of an attribute is called a domain. A set of attributes that uniquely determine an instance of an entity is called a key.

Having decided on the objects about which information is to be stored, it is necessary to also define the relationships between these objects, in the form of a data model.

Relationships may be considered to be either attribute relationships, or associations. Attribute relationships describe characteristics of an object and are of interest only so long as the object exists. For example, the status of a unit. Associations, on the other hand, describe a relationship between objects, as for example, the association defining the superior/subordinate relationship between two units.

Data models can be distinguished mainly as to how they represent relationships among data. Most data models handle attribute relationships in similar ways. However, associations are handled in different ways. The two main approaches are relational, or network. (A third category, hierarchical, is a special case of the network approach).

A network model consists of record types and links. Record types are used to represent the relationships among attributes, while the links represent the associations between entity sets.

In a relational model, all attribute relationships and all associations are simply represented as relations.

These relationships are as important as the data itself, and can be stored as part of the data base in the form of a data dictionary. This data dictionary is used by application programs for obtaining a 'view' of the data that is applicable to them.

## II.3 Relational or Network Data Base

Historically, the network approach to data bases was the first to come into widespread use, since it evolved quite naturally from files representing record types and the ability to cross link them. The CODASYL data base proposal(ref.36) was defined in 1969 by the Data Base Task Group formed with the US Defence Department to determine a common approach. A Data Description Language (DDL) and a Data Manipulation Language (DML) was defined and these have been used successfully since about 1971.

In 1970, E.F. Codd(ref.37) proposed a model for a generalised relational data base system to provide data independence and data consistency which are difficult to achieve in the network approach. This model has been expanded and improved by Codd and is now regarded by many as the future of all data base systems. Data structures in a network data base are designed to meet specific application access requirements, and if the access requirements change, then the data structures need to be changed as well. However, Codd's relational model is free from such considerations as access paths are universal - any data item value can be retrieved from one or more relations with equal ease.

The relational model is simple, elegant and flexible. This has in part been achieved by making the physical structure of data beneath the relational model independent of the logical design. Information about the real world can be represented faithfully, without having to introduce artificialities or contortions. The data structures can evolve gracefully as applications change or expand. The logical structure of the data can be easily understood, minimising errors. Relations are represented as tables, and this is the only logical structure that need be considered. The relational model provides wide freedom to the applications programmer by enabling him to access any data item value in the data base directly, rather than by its relative position or by a pointer.

## II.4 Data Query Language

As part of Codd's model, he developed a relational calculus based on the standard operations of set theory. This formed the basis of his relational query language called Data Sub-language ALPHA(DSL ALPHA). Since that time, a number of other access languages have been developed, including SEQUEL (Structural English Query Language) and SQUARE (Specifying Queries as Relational Expressions).

These data query languages can be used directly from an interactive terminal, or called from application programs written for specific purposes. This language provides for the definition of user or application views of the data base as well as for the retrieval of data. A query language is often included as part of a DBMS.

The simplicity of use of the relational model is an important consideration. The tabular structure and ability to retrieve by value ("find UNITS with EQUIP = MEDIUM TRUCK") are simple and intuitive. This allows a less experienced staff to implement a data base application, and very often allows them to complete the work more quickly.

Because data entities and record types are not always known from the start of a project, and this is particularly true for an evolving CCIS, the ability to easily redefine data and data relationships becomes very important. This is a feature of relational data base systems.

From these considerations, it becomes clear that relational DBMSs have many characteristics which make then well suited for systems requiring data base storage.

Structured Query Language (SQL) has become a defacto standard among relational DBMSs, and many suppliers offer this support. The US Army is considering the adoption of SQL as its standard query language for relational data bases in an effort to reduce proliferation of languages, and consequent increased software maintenance costs.

The use of a standard query language provides for a common programming interface to the DBMS. This improves the portability of application programs across projects.

A DBMS can be implemented in software on a host or in special purpose data base machine (DBM) (see next section). A standard query language would allow for both of these DBMS implementations, without changes to the application software. This would allow projects to be initially developed using a software DBMS, and then to migrate to a DBM when the system grows beyond the initial capacity.

In a distributed CCIS, the choice of a host based DBMS or a DBM could be based on the tactical level of the supported headquarters.

## II.5 Data Base Machines

A DBMS can be regarded as simply a complex program which runs under the control of a computer's operating system. The operating system interacts with users logged onto the computer, and schedules their data base requirements by allowing them slices of time for access to the DBMS program. Not all tasks involve data base searches, but, in an information systems environment, all tasks compete with the DBMS for CPU time and disc access. These computer resource demands by a DBMS can be significant, and can reduce the computer's response to all users.

In an attempt to alleviate this conflict, separate data base machines have been developed for performing this DBMS function (figure 34). The data base machine (DBM) forms part of a dual processing system whereby the host computer accepts data base requests from a user or application program, and passes them onto the DBM for processing. The DBM returns the result to the host for dissemination to the calling program or user.

There are several advantages of a DBM.

(1) A DBM is normally a special purpose computer optimised to perform specific data base functions. Data base machines can perform data base operations much more quckly than a host computer.

(2) Relational data base management requires large programs and consumes a large percentage of the capacity of the host. By moving these tasks to a DBM, the host is free to perform other tasks, either by accommodating more users or applications or by performing current tasks faster.

(3) Several different hosts can share one DBM. The host computers may be from different manufacturers, but can share the data base through using the same host/DBM interface.

(4) A DBM can provide enhanced performance at a cheaper cost than upgrading an existing host. A smaller host can be used, or the lifetime of an existing host can be extended.

(5) A DBM can provide a shared data base resource as a data base server to all computers in a computer network.

(6) The DBM technology can be upgraded with minimal effect on the hosts.

A complete DBMS is formed by combining the hardware and software of a DBM with interface software running on the host. The DBM implements all the facilities of a relational data base management system, while the interface software on the host provides facilities to manage communication with the DBM, including language processors, report generators and screen handlers. The division of tasks between the DBM and host must be properly designed so that the communications link between them does not become a bottleneck. Early DBMs failed because of this bottleneck, but modern versions seem to have overcome this problem(ref.39,40,41).

DBMs appear to be quite well suited for use in a tactical CCIS. The development of the CCIS can be made simpler as more effort can be devoted to the data model which will satisfy the user requirements. Several application processors can share the DBM via a local area network, with access authorisations defined by the DBM software. Data base backup and recovery techniques are contained within the DBM system. Data storage can be duplicated on one DBM, or two DBMs may be required for higher reliability.

## II.6 Data Base Implementation

A DBMS provides an interface between data users and the data storage system. The principle of data independence ensures that the way data is stored on the storage devices is independent of the user's view of the data. Data models are defined which, in a relational data base, describe logical relationships without dictating the storage techniques. Underlying storage techniques can be changed, if necessary, without requiring any changes to data models or application programs.

Within a DBMS, therefore, a number of different data files may be in use to support the different relationships defined in the model.

Files are a named collection of records containing the occurrences of one or more record type. A particular record may belong to one <u>physical</u> file, but to many <u>logical</u> files.

Files may be grouped into six categories - serial, sequential, indexed, direct, inverted, list. Many variations of these six categories are possible to provide different features. Many text books(ref.34,35,36) are available which examine these techniques in detail.

The detailed implementation of a relational data base will involve some of the above categories. It is important to realise however, that a relational data base description does not define how the physical data is to be stored, just how it is to be logically viewed.

The early implementations of relational data base systems were noted for their large memory requirements and slow speed of operation. Modern relational DBMS seem to have overcome these limitations to some extent using techniques such as bit arrays and tree searching methods.

A relational data model may be efficiently designed at the start of the development of a CCIS. However, as new relations are added during development, new access paths need to be superimposed on the existing physical storage structure. This may result in some slow responses to these new relations. At some stage during development, the physical structure may need to be reorganised to integrate the enlarged requirement into a comprehensive design. However, the ability to add relations temporally, even if somewhat inefficiently, during development, is useful.

The physical implementation of a relational data base in a tactical environment would need to be done with more care than in a support environment. Tactical application programs will allow only a defined type of access to the data base and so the access paths become well defined. Under these conditions, the data base structure can be designed to provide maximum performance. Those access paths which are used frequently could be provided with maximum efficiency, while those used infrequently would receive a lower priority of response. Alternatively, some types of retrieval would not be required 'instantly', particularly those used for planning purposes in an HQ, and therefore could have their performance sacrificed in order to better support more critical tasks.

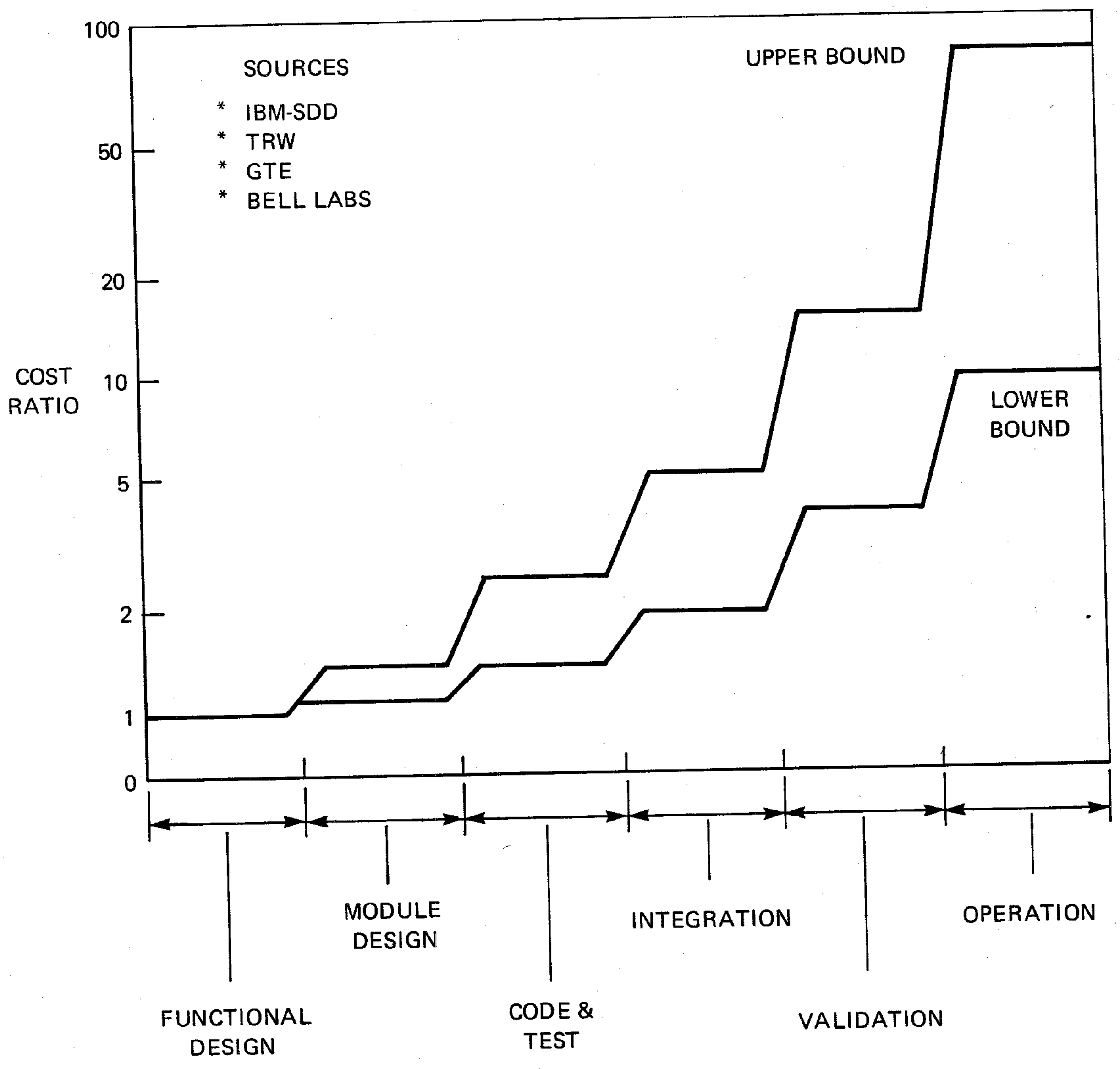

Figure 1. Cost of correcting a mistake in requirements

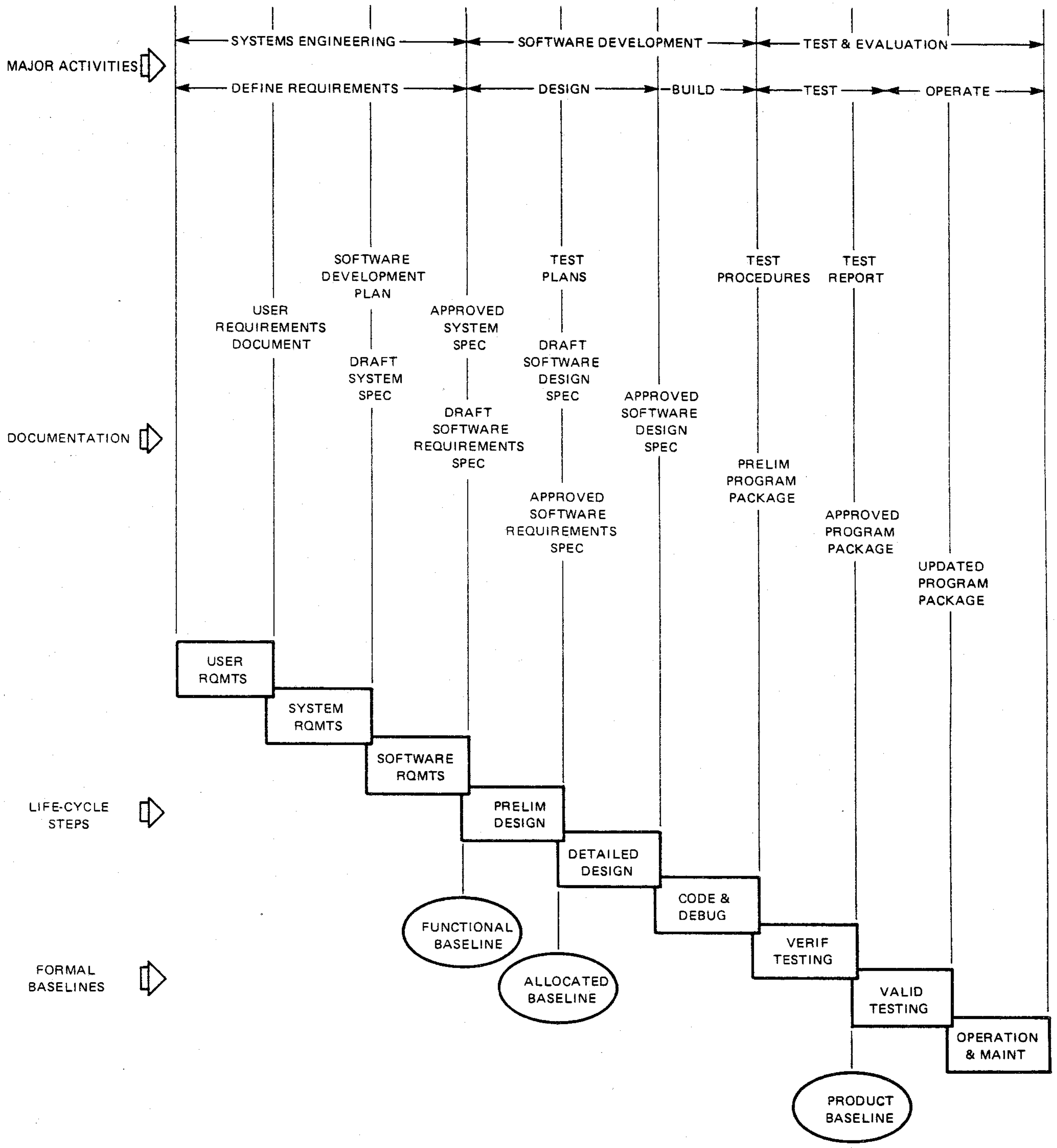

Figure 2. Software lifecycle diagram

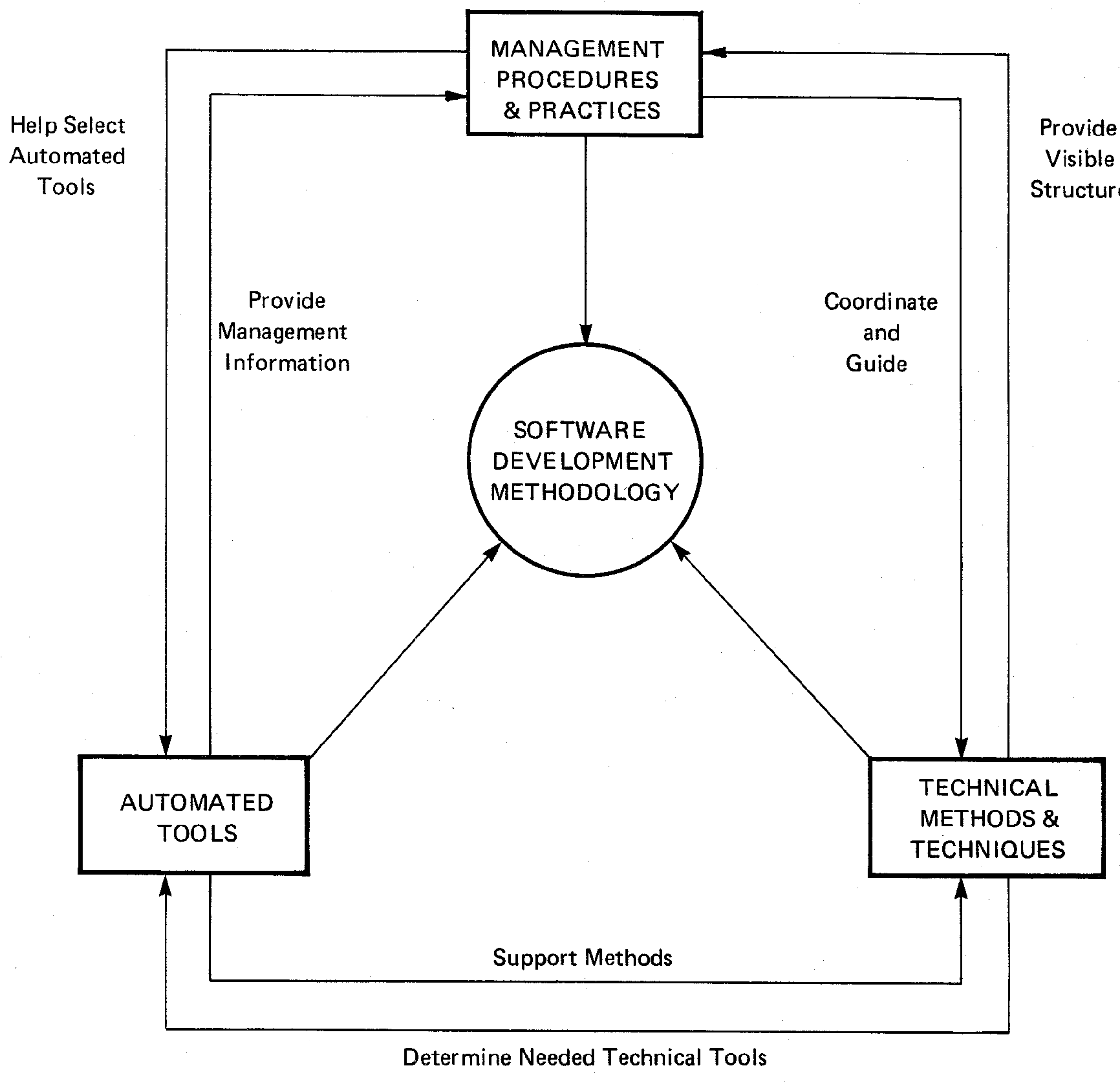
MANAGEMENT PROCEDURES & PRACTICES
Help Select Automated Tools
Provide Management Information
Provide Visible Structure
Coordinate and Guide
SOFTWARE DEVELOPMENT METHODOLOGY
AUTOMATED TOOLS
TECHNICAL METHODS & TECHNIQUES
Support Methods
Determine Needed Technical Tools

Figure 3. Software development methodology

DOCUMENT CONTROL DATA SHEET

Security classification of this page: UNCLASSIFIED

| | |
|---|---|
| **1 DOCUMENT NUMBERS** | **2 SECURITY CLASSIFICATION** |
| AR Number: AR-004-637 | a. Complete Document: Unclassified |
| Series Number: ERL-0372-RE | b. Title in Isolation: Unclassified |
| Other Numbers: | c. Summary in Isolation: Unclassified |

**3 TITLE**

MANAGING SOFTWARE COMPLEXITY

| | |
|---|---|
| **4 PERSONAL AUTHOR(S):** | **5 DOCUMENT DATE:** |
| P.F. Calder | July 1986 |
| | **6** 6.1 TOTAL NUMBER OF PAGES 47 |
| | 6.2 NUMBER OF REFERENCES: 41 |

| | |
|---|---|
| **7** 7.1 CORPORATE AUTHOR(S): | **8 REFERENCE NUMBERS** |
| Electronics Research Laboratory | a. Task: ARM 83/109 |
| | b. Sponsoring Agency: Army Research Request 1159/82 |
| 7.2 DOCUMENT SERIES AND NUMBER Electronics Research Laboratory 0372-RE | **9 COST CODE:** 442981/126 |

| | |
|---|---|
| **10 IMPRINT** (Publishing organisation) | **11 COMPUTER PROGRAM(S)** (Title(s) and language(s)) |
| Defence Research Centre Salisbury | |

**12 RELEASE LIMITATIONS** (of the document):

Approved for Public Release

Security classification of this page: UNCLASSIFIED

Security classification of this page: UNCLASSIFIED

13 ANNOUNCEMENT LIMITATIONS (of the information on these pages):

No limitation

14 DESCRIPTORS:

| | |
|---|---|
| a. EJC Thesaurus Terms | Systems analysis<br>Systems management<br>Computer programming |
| b. Non-Thesaurus Terms | Program development systems<br>Spectrum |

15 COSATI CODES:

05010
09020

16 SUMMARY OR ABSTRACT:
(if this is security classified, the announcement of this report will be similarly classified)

This report examines the methodologies and tools available to the manager and programmer for assisting in the development of large software projects.

Security classification of this page: UNCLASSIFIED