RAL-85-007

Science and Engineering Research Council
Rutherford Appleton Laboratory
CHILTON, DIDCOT, OXON, OX11 0QX

RAL-85-007

Sixth Annual Lecture of the C&CD of IEE 'Software Engineering'

R W Witty

December 1984

RAL 85 007

SIXTH ANNUAL LECTURE OF THE C&CD OF IEE

"SOFTWARE ENGINEERING"

11th December 1984

by

Dr R W Witty

ABSTRACT

This report is a written version of the Sixth Annual Lecture of the Computing & Control Division.

Future IT products can be expected to be more complex than those of today and thus to place greater demands upon the software in them and the people building them. The IT industry must meet this challenge even though there is a growing recognition that system development techniques are inadequate for the systems of today, let alone those of tomorrow.

It is accepted today that the development of a substantial new software system carries a number of significant risks and it is by no means uncommon for such systems to be delivered late, over-budget and incapable of meeting the complete requirements of the purchaser. Some systems, after considerable expenditure of human effort and money, fail to materialise at all.

Software engineering may be considered as having two major goals for the future:

- improved quality ie satisfying criteria such as performance, reliability, security, on-schedule delivery and meeting the needs of the user;

- improved productivity reducing cost, not just of the development but of the life-cycle as a whole, including maintenance and future evolution.

The industrialised nations are all currently increasing the scale of their research and development programmes in the search for improved software engineering methods, skills and tools. A major UK initiative is being led by the Alvey Directorate. This report gives a summary of the Alvey Strategy.

The main objective of the report is to try to impart a practical feel for what software engineering methods and tools might be in common industrial use 5 years from now. The actual lecture contained both live and recorded demonstrations of current research prototypes.

SIXTH ANNUAL LECTURE OF THE C&CD OF IEE

"SOFTWARE ENGINEERING"

by

Dr R W Witty

CONTENTS

Continued ....

Continuation of CONTENTS.

A. INTRODUCTION

1. SOFTWARE ENGINEERING: A DEFINITION

Software Engineering was first recognised as a distinct subject at the 1968 NATO Conference [8].

The following introduction to Software Engineering is taken from Boehm [1].

"Our definition of **software engineering** is based on the definitions of **software** and **engineering** given in the current edition of **Webster's New Intercollegiate Dictionary** [Webster, 1979]:

- **Software** is the entire set of programs, procedures, and related documentation associated with a system and especially a computer system.

- **Engineering** is the application of science and mathematics by which the properties of matter and the sources of energy in nature are made useful to man in structures, machines, products, systems, and processes.

Since the properties of matter and sources of energy over which software has control are embodied in the capabilities of computer equipment, we can combine the two definitions above as follows:

- **Software engineering** is the application of science and mathematics by which the capabilities of computer equipment are made useful to man via computer programs, procedures, and associated documentation.

1.1 Discussion

This definition of software engineering contains two key points which deserve further discussion. First, our definition of software includes a good deal more than just computer programs. Thus, learning to be a good software engineer means a good deal more than learning how to generate computer programs. It also involves learning the skills required to produce good documentation, data bases, and operational procedures for computer systems.

The second key point is the phrase "useful to man". From the standpoint of **practice**, this phrase places a responsibility upon us as software engineers to make sure that our software products are indeed useful to people. If we accept an arbitrary set of specifications and turn them into a correct computer program satisfying the specifications, we are not discharging our full responsibility as software engineers. We must also apply our skills and judgment to the job of developing an appropriate set of specifications, and to the job of ensuring that the resulting software does indeed make the computer equipment perform functions that are useful to society. Thus, concerns for the social implications of computer systems are part of the software engineer's job, and techniques for dealing with these concerns must be built into the software engineer's practical methodology, rather than being treated as a separate topic isolated from day-to-day practice.

From the standpoint of **learning**, the phrase "useful to man" implies that the science and mathematics involved in software engineering covers a good deal more than basic computer science. For something to be useful to people, it must satisfy a human need at a cost that society can afford. The science and mathematics of human economics presented in this book provides an opportunity to learn some ways to handle the cost and human-needs aspects of a software engineering problem, and to integrate them with the computer science aspects.

2. SOFTWARE TRENDS: COST

The way we perform software engineering determines the cost and the quality of the software produced. This makes software engineering important because of the following two trends:

1. Software is a large and increasingly costly item.
2. Software makes a large and increasing impact on human welfare.

These trends are covered in the next two sections.

The annual cost of software in the US in 1980 was approximately 40 billion dollars, or about 2% of the Gross National Product. Its rate of growth is considerably greater than that of the economy in general. Compared to the cost of computer hardware, the cost of software is continuing to escalate along the lines predicted in Fig. 3-2 [Boehm, 1976].

By now, the trend in Fig. 3-2 has become so pronounced that we can often consider the hardware as a kind of packaging for the software, which is the portion of the computer system which largely determines its value. Thus, today, the computer system that we buy as "hardware" has generally cost the vendor about three times as much for the software as it has for the hardware. Most thorough "hardware" procurements are primarily software purchases, as the buying evaluations place more weight on the software aspects than on the hardware aspects. (For an example, see Chapter 15.) A number of new computer systems (for example, Amdahl, Magnuson, Cambridge, and National Semiconductor) offer a product that is largely IBM software repackaged for a different mainframe. And the IBM software rental for a basic IBM 4331 system can be greater than the rental cost for the hardware [Lundell, 1979].

With respect to the overall computer and information processing industry of the future, computer software will be the dominant portion of an industry expected to grow to 8.5% of the Gross National Product by 1985 [Dolotta and others, 1976] and to 13% of the GNP by 1990 [Steel, 1977].

This growth in demand for software creates a tremendous challenge for the software engineering profession. The challenge is twofold: first, to significantly increase software development productivity; and second, to increase the efficiency of software maintenance. As shown in Fig. 3-2, the portion of efforts spent on software maintenance is greater than that spent on software development. The data point for 1978 comes from a recent survey of 487 data processing installations, in which the mean percentages of effort were 43.3% for development, 48.8% for maintenance, and 7.9% for other miscellaneous activities [Lientz-Swanson, 1978].

3. SOFTWARE TRENDS: SOCIAL IMPACT

The growth in the demand for software results largely from the fact that as computer hardware becomes increasingly inexpensive, reliable, and plentiful, we find it more and more advantageous to automate the machine-like portions of human jobs.

Figure 3-3 illustrates this trend. It summarizes the results of three studies ([AFIPS-Time, 1971; Boehm, 1973; Dolotta and others, 1976]) on the growth of computer usage and its human impact. The most striking implication of the studies is that, by 1985, roughly 40% of the American labor force will be relying on computers and software to do their daily work, without being required to have some knowledge of how computers and software work. Thus, this 40% of the labor force will be trusting implicitly in the results produced by computer software.

Computers and software are making an even deeper impact on our personal lives. With every passing day, more and more of our personal records, bank accounts, community services, traffic control, air travel, medical services, and national security are being entrusted to the hopefully reliable and humane functioning of computers and software. And the potential threats to our personal welfare via computer crime [Parker, 1976], massive data banks [Westin-Baker, 1972; Ware and others, 1974], or computer systems that make people think and act like computers [Weizenbaum, 1976; Docherty 1977], become more and more difficult to contain.

This increasing impact on human welfare presents several tremendous challenges for the software engineering profession. They are to develop and maintain software which ensures that computer systems are:

- Extremely reliable
- Humane
- Easy to use
- Hard to misuse
- Auditable

and that keep people, rather than computers, in the driver's seat.

These challenges, plus the economic productivity and maintainability challenges identified in the previous section, provide the main motivation for the goals of software engineering discussed in the next section "[1].

4. THE PROBLEMS OF SOFTWARE PRODUCTION

The simple life cycle of any man-made article, not just software, is:

1. Identify Requirement for product

2. Design the product

3. Manufacture the product

4. Maintain the product

5. Scrap the product.

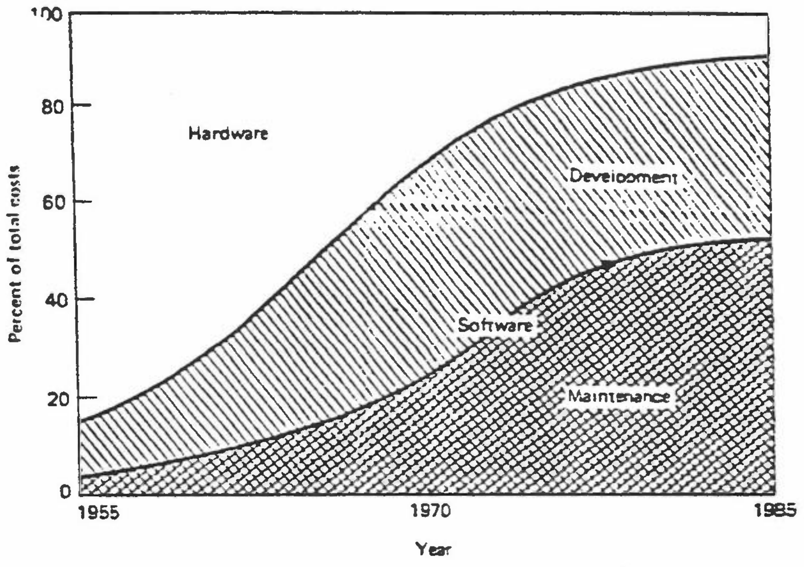


FIGURE 3-2 Hardware/software cost trends

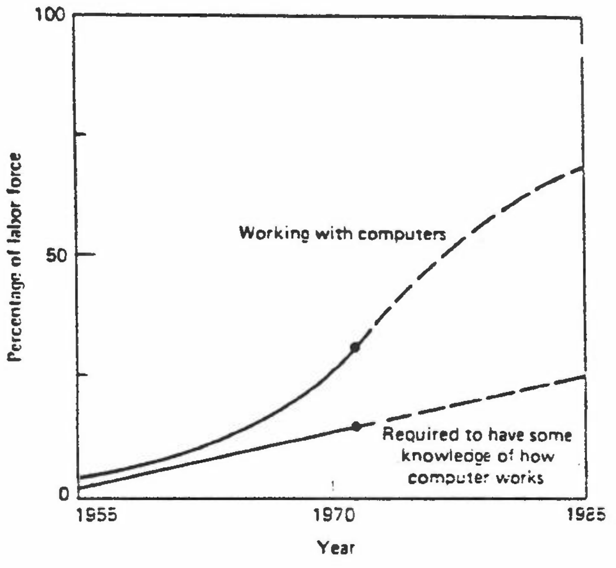


FIGURE 3-3 Growth of reliance on computers and software

In Software Engineering the manufacturing process is not a problem if manufacturing means replication of an item. Copying a magnetic tape or disk can be done quickly, easily and reliably; this contrasts markedly with say motor car production or VLSI chip production.

For a software product the greatest proportion of life cycle cost is associated with the maintenance phase. In the worlds of motor cars and computer hardware 'maintenance' means 'restoration to original condition', faults being mainly due to physical wear and tear on components.

Software, by its very nature, cannot wear out. To first order, the only type of fault which can occur in a software product is a design fault. Thus correct software maintenance does the exact opposite of 'restoration to original condition', if the fault is cured, because a new version of the product has to be built. 'Software maintenance' is a euphemism for 'rectification of design faults'.

Software faults can be trivial at the lexical level whilst having disasterous consequences. A very famous example is that of a missing comma which caused the loss of a deep space exploration probe. The offending error was of the following nature

```
DO 99 I = 1,10 which is a FORTRAN repetition statement was
               mistyped as

DO 99 I = 1 10 which, by chance, became the valid assignment
               statement

DO99I = 110
```

A software error delayed the first launch of the Space Shuttle, a very public mistake; software errors in the USA's World Wide Command and Control System have had significant publicity when incoming missile attacks have been falsely detected.

Maintenance is also usually taken to include development work in which a correctly operating product is changed to reflect some new customer requirement.

Rectification and development often take up 50% of the total life cycle cost of a software product.

This figure is so high because today's software is error prone, fails to meet its requirements specifications and is difficult to change/develop. All of these problems are design problems.

B. ALVEY SOFTWARE ENGINEERING PROGRAMME

1. GOALS AND OBJECTIVES

1.1 The Central Goals

Future IT products can be expected to be more complex than those of today and thus to place greater demands upon the people building them. The IT industry must meet this challenge, even though there is a growing recognition that system development techniques are inadequate for the large systems of today, let alone those of tomorrow.

It is accepted today that the development of a substantial new computer system carries a number of significant risks and it is by no means uncommon for such systems to be delivered late, over-budget and incapable of meeting the complete requirements of the purchaser. Some systems, after considerable expenditure of human effort and money, fail to materialise at all.

Skilled programmers are a scarce resource which is not being used efficiently. The industry is fragmented by organisation, by language and by target computer. One result of the consequent lack of commonality of environment or concentration of resources is that many programmers are not provided with even the simplest programming aids, let alone sophisticated ones. The economies of scale necessary to justify their introduction have not been perceived to exist.

Despite these problems, the UK does not lag behind other countries in software engineering, except perhaps the USA. The UK is certainly regarded as the leader in Europe in this field. Efforts to improve software engineering practice are crucial if important developments in technology are not to be wasted or cast into disrepute through poor production methods. The UK must not allow other countries to overtake it, for if it does, UK research work will be exploited by other countries to the detriment of our industry.

Software engineering may be considered as having two major goals for the future:

- improved quality ie satisfying criteria such as performance, reliability, security, on-schedule delivery and meeting the needs of the user;

- improved productivity ie reducing cost, not just of the development but of the life-cycle as a whole, including maintenance and future evolution.

Current software practice is centred on the programming process, and depends strongly on the skills, experience and resources of individual workers. Significant problems frequently result from inadequate effort being devoted to the front end of a development, notably concept formation, requirements definition, and design. Although there have been some efforts to study these problems, as well as interesting advances in both design verification and code verification, relatively little work has been devoted to integrating all of the stages into a common framework useful in production environments. Significant improvements in software productivity will be achieved when the current practice of repreated 'reinvention of the wheel' is replaced by the widespread re-use of prefabricated components. In the future then, software practice will tend to focus more on methodology, design, and component reuse and less on individual programming skills.

System design must include not just software design, but also hardware considerations. A narrow view of software engineering as just a collection of techniques to produce efficient software is not adequate. Software engineering should be aimed at the development of high quality systems, ie reliable, secure, efficient and easy to use, in a way that integrates hardware and software-based design criteria. In the future it must become information system engineering, not just software engineering.

1.2 Major Objective

To help achieve the general goals of improved Quality and Productivity the Software Engineering component of the Alvey Programme is focussed towards a strategic goal - that in 1989 the UK should be a world leader in Information System Factories (ISF). This goal is highly ambitious and competitive, as are the goals of the Japanese 5th Generation Project. The ISF objective implies a series of sub goals both in technology and timescale. The Alvey Software Engineering component will be judged on its ability to show that UK industry has increased both its software development productivity and software product quality as a result of striving to achieve the ISF. The strategy given in this document outlines the route towards the ISF with planned interim spin-offs so that productivity and quality gains may be achieved prior to the emergence of the ISF.

What is meant by an Information System Factory? Today, the production of most application-specific hardware/software systems - such as a banking network, a corporate management information system or production control system - does not in general make great use of development tools. In that sense it is not capital intensive. The application-specific part of the Information Technology industry is characterised as a cottage industry. It is predicted that it will not remain so for long, indeed the Japanese are already building 'software factories'. To stay competitive in producing large, reliable, application-specific systems, IT companies will have to make a large investment in some kind of production facility. Exactly the same criteria will apply to manufacturing software products. This expensive facility - part hardware, part software, part stored knowledge - is an Information System Factory.

1.3 What Will Happen in Any Case

The Alvey SE strategy is based on the prediction that the production of application-specific information systems will cease to be a cottage industry and become a capital-intensive industry.

The main reason has to do with software quality, in the widest sense of the word. Expectations of software quality, both within the industry and without, are very low. Today, programmers expect to have lots of bugs in their code, and the public expect computers to send them stupid invoices. This situation is not confined to the UK; it is worldwide. British standards of software quality are relatively high, while low in an absolute sense. This situation cannot last indefinitely. In the hardware field, one manufacturer (Tandem) has grown spectacularly by offering high reliability at a premium. This has been done against a background of hardware from IBM and others which is already highly reliable. The incentives to do the same in software, and the potential payoffs, must be much higher given the current poor quality of software. It seems highly likely that someone soon will "do a Tandem" in software, and either keep the method to himself or sell it very expensively. The Japanese are certainly trying, as are the Americans and the French. Without concerted action, the UK is bound to become an importer of this technology. If the UK is prevented from importing such technology then the industrial consequences could be very serious.

A number of other current trends are leading towards the 'capitalisation' of the software industry - the growing complexity of software systems, which demands new techniques and computer assistance to manage it, the dawning awareness of the importance of project and programming support environments, and the emergence of software packages which demand new skills to integrate them in particular applications. Finally, there is the emergence of non-Von Neumann architectures and VLSI, which are inevitably mixing the software and hardware design problems, making both more complex. All these are creating larger and more complex problems, which cannot be solved without a radically new level of automation and mechanical assistance.

2. THE CHANGING NATURE OF SYSTEM DEVELOPMENT

2.1 Summary

The expected changes that will result in the most significant increases in cost-effectiveness of software development over the next ten years are the following, listed in approximate order of expected impact.

In the short term

1. incremental changes in programmer productivity through the more widespread use of design methodologies and tools

2. the coming together of methodologies and tools for the entire development life-cycle within integrated project support environments (IPSEs)

3. growing standardisation of development methodologies as a consequence of 2.

4. further refinement of suitable high-level programming languages appropriate to the integrated development methodologies

5. growing interest in, and use of, formal specification methods and extension to animation

6. automatic software generation techniques in limited form, probably first in the area of commercial systems built around Data Dictionaries.

In the medium term

7. spread of powerful networked, personal workstations

8. consolidation of the use of formal specification methods coupled with verification and growth in use of (semi-) automatic software generation

9. development of reusable software and hardware modules, rigorously tested and formally documented

10. second generation IPSEs adapted to support activities 8 and 9 above, coupled with greater use of higher-level languages.

And in the longer term:

11. the consolidation of the developments above into Information System Factories, coupled with the use of Intelligent Knowledge Based Systems, to provide 'automatic' assisted system development from user requirements expressed in high-level terms appropriate to the application rather than the implementation.

The crucial, and inter-related, technical developments underlying those changes will be:

1. integrated system (software and hardware) development methodologies supported by programming tools, administrative procedures and management information in an integrated environment

2. formal specification, leading to 'animation' and verification

3. reusable software and hardware components

4. automatic software generation

5. measurement and quality assurance and certification

3. STRATEGY

3.1 Summary

The Alvey SE Programme has as its long term objective the creation of the Information Systems Factory. This is predicated on technical progress in the two crucial areas of:

1. QUALITY
2. PRODUCTIVITY

To ensure continuous benefit during the period preceeding the achievement of the ISF the SE Programme proposes a strategy which encourages intermediate levels of technology transfer by encouraging not just research but:

i. Exploitation: efforts to ensure that existing methods are effectively used and their benefits gained by industry as a whole, and continuing efforts to bring the fruits of research out into industrial use, with the associated investment and training.

ii. Integration: development of integrated methodologies and sets of tools for hardware and software development covering all phases of the system life-cycle.

iii. Innovation: research and development to extend the methodologies and techniques of software engineering.

To give a feel for the activities which will be covered by innovation, integration and exploitation figure 1 shows the system development life cycle subdivided into

1. Methods and processes - how things are developed.

2. Management - monitoring and control of methods and processes.

3. Environment - the workplace, tools and equipment with indications of where in the classification various key elements of the strategy occur.

STRATEGY	Innovation and Understanding	Integration and Implementation	Exploitation and Evaluation
Methods and Processes	Specification V & V Reliability Quality Metrics Reusability	Blend techniques into life cycle method for both hardware and software	Measure use of IPSE
Management	Models of development and mainte-nance processes and methods	Integrate development methods with management techniques	Evaluate use of IPSE
Environment	Influence on Productivity and Quality MMI, IKBS, DCS	Build IPSEs	Make IPSE available via Centres

Figure 1

3.2 Integration

The second major need identified is for Integrated Project Support Environments (IPSE). The common understanding of an IPSE is that it should contain a compatible set of specification, design, programming, building and testing tools, supporting a development methodology that covers the entire life-cycle, together with management control tools and procedures, all using a central project database. That is already a very demanding requirment, exceeding that of the Ada APSE, but even then it does not go far enough. It does not cover multiple-language development; it does not cover mixed hardware and software development; it does not cover reusable components.

3.2.1 The 2nd Generation IPSE

The second generation IPSE contains two major components not found in the 1st generation IPSE:

(1) Database-based tool set (rather than file-based) eg CADES.

(2) Support for geographically distributed project teams e.g. Newcastle Connection.

The 2nd generation IPSE software will run on new hardware; developments in cheaper CPU power, cheaper, high resolution colour graphics, and non keyboard input-output devices, for instance, will facilitate productivity gains due to improved man-machine interaction. The 2nd (& 3rd) generation IPSE will require new hardware based components such as:

1. Single user workstation costing £5K with A3 black and white 2K x 2K pixel graphics.

2. Colour single user workstation costing £10-20K with

   - 2K x 2K pixel A3 screen
   - 10 MIPS power CPU
   - 32K microcode store
   - 10 Mbytes physical memory
   - 32 bit arithmetic and data paths
   - 32 bit virtual address space per process
   - hardware cache, paging, floating point
   - hardware graphics support
   - sophisticated i/o devices.

3. 100 Mbits/sec local area network.

4. Gateway to high speed (greater than 1 Mbit/sec) wide area communications.

5. LAN servers for files and databases.

6. High quality, cheap print server eg. laser printer.

7. Full-generality distributed operating system.

8. Sophisticated man-machine interface.

9. Integrated with Office Automation and Corporate Applications Systems.

3.2.2 <u>The 3rd Generation IPSE</u>

The 3rd generation IPSE (or ISF), containing knowledge bases and 'intelligent' tools, requires significant research which must begin now if the 1989 target date for the Information System Factory is to be met.

An Information System Factory will probably consist of six main subsystems:

1. specification and prototyping facilities
2. a Software Development Environment
3. a facility for CAD of VLSI and hardware development
4. a database or knowledge base of available software and hardware components
5. the communication systems, both local and wide area, to facilitate co-operative development
6. project management aids.

3.3 <u>Innovation</u>

The current list of research priorities includes:

i. Software Development Methods

- Formal Specification
- Verification and Validation
- Reusable Components
- Metrics
- Quality Assurance and Certification

ii. Project Management

- planning and estimating
- progress and productivity measurement
- budgeting
- standards control

iii. IPSE

- items already indicated above are relevant
- evaluation experiments to test changes in productivity and quality due to use of IPSE in the industrial context
- MMI, VLSI/CAD etc from other Alvey areas but relating to IPSE construction

3.4 National Quality Certification Centre

The primary medium term payback activity is seen as the creation of a National Quality Certification Centre (NQCC) for software products and components. The NQCC must build up an international reputation. This will involve the adoption of state of the art techniques on a continuous basis. The commercial benefit of NQCC approved software products in an international market is potentially extremely valuable. As the mass market for software products develops consumers will buy NQCC approved products rather than unapproved products. The rapid establishment of such a national capability could give the UK a significant commercial advantage.

C. FORMAL METHODS

1. INTRODUCTION

Software systems, even small ones, cannot be completely tested. This is due to a variety of reasons including

1. the extremely large number of independent internal states

2. the extremely large number of different possible inputs

3. the inability to generate valid tests.

An example of 2) is that, at present hardware speeds, it takes a time period greater than the predicted life time of planet earth to check that an arithmetic unit correctly adds together all possible pairs of numbers from its finite input set.

Examples of 3) include the inability to test realistically that part of the software which handles nuclear power station crises or navigates cruise missiles over enemy territory. The 'computer overload' problem which beset Neil Armstrong as he attempted the first lunar landing fell into this category.

Testing is necessary and useful but will always suffer from the problem of lack of completeness which is enshrined in the famous remark "program testing can be used to show the presence of bugs, but never to show their absence!" [11]. Thus modern software engineering is trying to improve Quality by theoretical methods (formal methods) which attempt to prove (in the strict mathematical sense) properties of programs and systems.

A mathematically based proof that a well specified property of a program holds for all inputs is more valuable than an assumption that the property is true for all inputs, given that testing shows it to hold for a small subset of the input domain. In the finite and discrete world of software, extrapolation is not always a sound technique!

2. A SIMPLE EXAMPLE

The following fragment of code appears to be simple enough in outline (3 simple assignments executed in sequence).

```
A:=A+B;
B:=A-B;
A:=A-B;
```

Without a higher level specification it is not immediately obvious what this code fragment achieves and therefore there is no way of knowing if it is correct, for correctness in software means proving that a program implements the required specification.

If the code fragment is analysed formally by assuming the final values of the variables 'A,B' are 'x,y' then the code fragment can be 'symbolically executed' backwards to derive the initial values of 'A,B'.

$$\left\{\begin{array}{l} B_i=x \\ A_i=y \end{array}\right\}$$ <u>PRE CONDITION</u>

$$\left\{\begin{array}{l} A_i+B_i-A_i-B_i+B_i=x \\ A_i=y \end{array}\right\}$$

$$\left\{\begin{array}{l} (A_i+B_i)-((A_i+B_i)-B_i))=x \\ (A_i+B_i)-B_i=y \end{array}\right\}$$

(1) $A_t:=A_i+B_i;$

$$\left\{\begin{array}{l} A_t-(A_t-B_i)=x \\ A_t-B_i=y \end{array}\right\}$$ 'B' changed so substitute LHS;

(2) $B_f:=A_t-B_i;$

$$\left\{\begin{array}{l} A_t-B_f=x \\ B_f=y \end{array}\right\}$$ 'A' changed so substitute LHS;
'B' unchanged

(3) $A_f:=A_t-B_f;$

$$\left\{\begin{array}{l} A_f=x \\ B_f=y \end{array}\right\}$$ <u>POST CONDITION</u>

Therefore $A_f = B_i$ where 'i' is the initial value,
$B_f = A_i$ 'f' is the final value.

Therefore Code fragment swops values between A,B.

Specification: Swop the values held in A,B
s.t. $A_f = B_i$ and $B_f = A_i$
without use of temporary storage

Implementation:

```
A := A+B
B := A-B
A := A-B
```

(Contrast the equivalent function using a temporary storage variable

```
T:=A;
A:=B;
B:=T;).
```

Note that the predicates which describe the initial and final states of the computation (ie the values of the variables in the above example) are collectively called the pre and post conditions. A program 'P' can be thought of as a process which causes a state, with precondition 'I', to be changed into a state, with postcondition 'F'. This is often written

[I]P[F]

3. A SECOND EXAMPLE

Here is a second, more complex example involving iteration to perform integer division by the process of repeated subtraction.

Specification: Given X>=0 and Y>0, find a program P to compute Q and R such that X=Q*Y+R and 0=<R<Y.

this can be re-expressed as a requirement to prove that P satisfies the following pre and post conditions

[0=<X,0<Y]P[X=Q*Y+R,0=<R<Y]

where P is

```
begin

Q:=0;
R:=X;

while  R>=Y  do


    begin

      R:=R-Y;
      Q:=Q+1;

    end

end
```

```
P                                                   O=<X,          O<Y

begin                          X=X,                 O=<X,          O<Y

                               X=O*Y+X,             O=<X,          O<Y
  $Q_i$:=O;
                               X=$Q_i$*Y+X,           O=<X,          O<Y
  $R_i$:=X;
                               X=Q*Y+$R_i$,           O=<$R_i$,        O<Y

  while  $R_i$>=Y  do
                                                         X=$Q_i$*Y+$R_i$,  O=<$R_i$,  O<Y

      begin                    X=$Q_i$*Y+$R_i$  [loop invariant]

                               X=$Q_i$*Y+Y+$R_i$-Y                If O<Y and O=<R and Y
                               X=($Q_i$+1)*Y+($R_i$-Y)             constant then 'R:=R-Y'
                                                          must always decrease R
      $R_f$:=$R_i$-Y;                                           so that eventually R
                               X=($Q_i$+1)*Y+$R_f$                 will become smaller
                                                          than Y and the loop
      $Q_f$:=$Q_i$+1;                                           will terminate.

      end                      X=$Q_f$*Y+$R_f$

                                                         X=$Q_f$*Y+$R_f$,  O=<$R_f$<Y

end                            X=$Q_f$*Y+R,                  O=<$R_f$<Y

                               X=$Q_f$*Y+R,                  O=<$R_f$<Y
```

Note: see appendix B.2 for proof by mechanical theorem prover.

4. DISCUSSION

The above two very simple examples illustrate several points about formal methods.

1. Proving consistency between a program and its specification requires both program and specification to be expressed in a mathematically tractable notation if the formal rules of mathematics and logic are to be securely employed. Hence the Alvey SE Programme's emphasis on improving specification techniques.

2. One man's specification is another man's program ie a hierarchy of specifications and proofs can be constructed.

3. Proof need not only tackle functional correctness. Termination, security, safety, performance etc may also be specified and formally manipulated.

4. Proofs need to be built on reusable theories. All good proofs use shortcuts called theorems. The first example assumed much of the theory of arithmetic. It assumed that there were no bounds on the values of the variables; however computers are finite machines so that it is invalid to assume, for instance, that 'A:=A+B;' can be executed for any values of A,B. If A,B are both of the same order as the largest number which the machine can represent then their summation cannot be represented and the program will malfunction. This type of error is quite common in practice. The need to make explicit such restrictions and reason about them is one way in which formal methods improve the quality of software by error prevention at the design stage.

5. Even simple proofs (and manipulations) are tedious, lengthy and tricky and hence themselves often contain errors. However because of their formal basis such manipulations can today be checked for accuracy by computer based software tools called 'proof checkers'. The proof of 'A Second Example' is complex although the program P is fairly small. Appendix B.2 contains a mechanically generated proof of the key elements of the 'second example' program. Current research is striving to find cost effective ways to increasingly automate the creation and construction of proofs themselves to speed up the process even more. For instance in the ML/LCF system[9] the human prover expresses not the detail of the proof but only the 'MetaLevel' tactics to guide the computerised prover. ML/LCF also allows the construction and reuse of theorems to speed up the mechanical proof process. (See overleaf.) The IOTA system is one of the world's most advanced mechanical verification systems. It exhibits a significant degree of tool integration, being built on a data base foundation [12] (see overleaf).

5. QUALITY CERTIFICATION

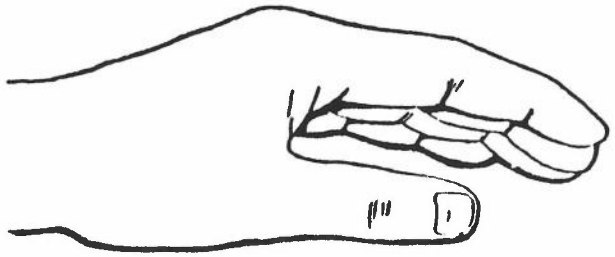
Quality is the major goal for the Alvey SE Programme. It is of vital commercial importance. Customers will tend to buy software which is 'high quality'. It is a challenge to the SE profession to be able to demonstrate and measure the quality of its products. This is currently beyond the state of the art.

Current quality techniques are based on testing eg the current certification of Ada compilers. Testing is inadequate for the reasons discussed earlier in section C.1. The drive towards quality certification is being led more by customers than suppliers.

Both customers and suppliers see proof-based formal methods as the most satisfactory route to certification which gives an additional boost to work in this area.

THE LCF SYSTEM

Meta Language
(ML)



Object Language
(OL)

Meta Languages

- Edinburgh ML
  - Fully higher-order functional programming language
  - Eager evaluation (i.e. non lazy)
  - Polymorphic type discipline
  - Raising and handling of exceptions
- Standard ML
  - A standard for future ML implementations
  - Combines Cardelli's ML with Burstall's Hope
  - Very careful design by Milner and colleagues
  - Hope-like data-structures and pattern matching
  - Value passing with exceptions (failures)
- Unix ML
  - Enhanced standard ML implemented under Berkeley Unix
  - Input-output and separate compilation (modules)
  - Good performance (comparable to Pascal)
  - Implementation almost complete (by Cardelli of Bell Laboratories)

LCF = ML + OL

- Security: No false theorems thanks to ML's type discipline
- Automation: Both data-directed (forward) and goal-directed (backward) strategies
- Generality: PPLAMBDA good for inductive reasoning about recursive definitions
- History
  - 1969: Scott lays foundation for PPLAMBDA
  - 1971: Stanford LCF
  - 1973–: Edinburgh LCF
  - 1981–: Cambridge LCF
- Case studies of proof
  - Simple compiling algorithms (Cohn)
  - FP-systems } Leszczylowski
  - Balanced trees }
  - Parsing algorithms (Milner, Cohn)
  - Substitution and unification (Paulson)
  - Set-theory (Schmidt)
  - Hoare-logic (Sokolowski)
  - Hardware correctness (Moxon, Gordon)

Object Languages

- Edinburgh PPLAMBDA
  - Polymorphic Predicate LAMBDA-calculus
  - Terms: Polymorphic typed λ-calculus
  - Formulae: Restricted predicate calculus
- Cambridge PPLAMBDA
  - Enhancement of Edinburgh PPLAMBDA
  - Full formula structure of predicate calculus (negation, existential quantification, disjunction etc.)
  - More general induction rule
- Other Logics
  - LSM (Logic of Sequential Machines): Extension of PPLAMBDA with terms from CCS
  - Constructive type theory (Gothenburg)
  - Classical higher-order logic
  - Classical set-theory

FORMAL PROOF IN LCF

University of Cambridge

Representing a logic in ML

- Theorems are an abstract type thm
- Axioms are predefined values of type thm
- Inference rules are ML functions which return theorems
- Example: The inference rule

$$\frac{\vdash A \wedge \vdash B}{\vdash A \wedge B}$$

Is represented by the ML Function

$$\underbrace{\text{CONJ}}_{\text{ML Name}}:\ \underbrace{\text{thm} \times \text{thm} \rightarrow \text{thm}}_{\text{ML Type}}$$

- Derived rules can be programmed in ML
- Example: The derived rule:

$$\frac{\vdash A_1, \ldots, \vdash A_n}{\vdash A_1, \ldots, \vdash A_n}$$

Is programmed as:

```
letrec CONJLIST thl=
if null thl        then fail
if null (tl thl) then hd thl
              else CONJ (hd thl , CONJLIST (tl thl))
```

Forward Proof

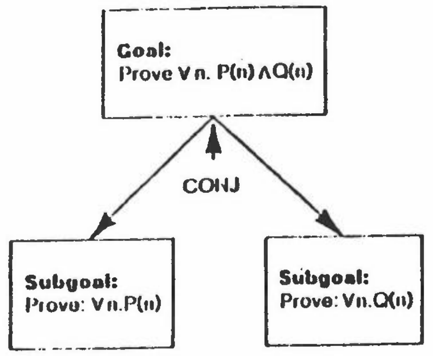
- Move forward from axioms via inference rules
- Example:

Formal Proof	**How it is generated in LCF**
th1 by Ax1	let th1 = Ax1 ;;
th2 by Ax2	let th2 = Ax2 ;;
th3 by rule R1, th1, th2	let th3 = R1 (th1, th2) ;;
th4 by rule R2, Ax3, th3	let th4 = R2 (Ax3, th3) ;;

- Disadvantages of forward proof
  - Tedious and low level
  - User must explicitly generate every step
  - Machine interactions do not reflect natural proof-finding activity

Goal-directed Proof

- How it works
  - Set up a goal
  - Generate subgoals
  - Prove subgoals
  - Hence prove goal
- Example:



- Proof strategies can be programmed in ML
- Example:

Informal strategy	**Representation in ML**
To prove goal G try strategy S1, if this doesn't work try S2. In either case then try S3 followed by S4 followed by S5 repeatedly	let S = S1 ORELSE S2 THEN S3 THEN S4 THEN REPEAT S5 ;; SG;;

Lecture Notes in Computer Science

Edited by G. Goos and J. Hartmanis

160

The IOTA Programming Systen

A Modular Programming Environment

Edited by R. Nakajima and T. Yuasa

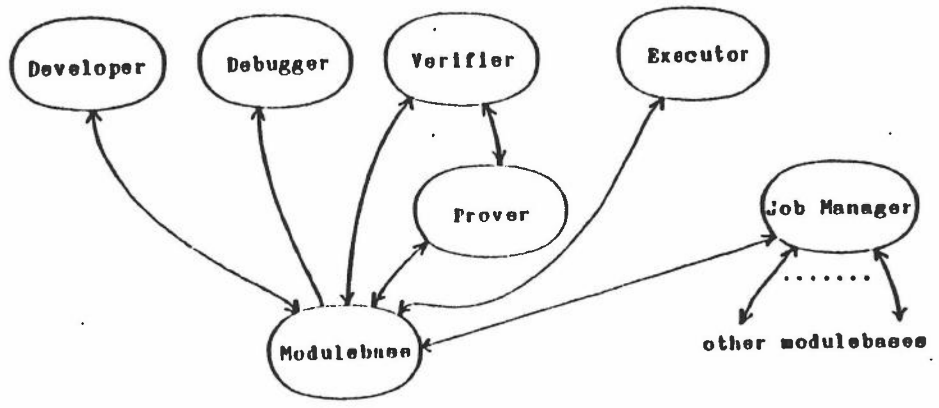
Introduction

This monograph describes the work of the project of IOTA which began in 1976 and was completed in 1983. The initial motivation of the work derived from an attempt to formalize the concepts of abstraction and to develop a mechanizable verification method for programming with modules. The scope of the project later extended to include the design of a programming and specification language IOTA for modular programming, and then to develop a total programming system.

The language IOTA supports modular programming with abstraction and parameterization mechanisms, and module specification in a many-sorted first order logic. An informal introduction to the language is given in Chapter 1, together with a formal view of the program development with the language. A formal definition of the language is presented in Appendices.

The IOTA system is a *modular programming system* which is intended to provide an integrated environment to enhance the goal of modular programming. One of its most important design objeqtives is to enable the programmer to concentrate on a single module at a time. Although the interfaces between modules are supposed to be kept minimal not to increase the complexity of the problem, objects belonging to different modules are related with one another. Experiences with some interactive LISP systems suggests that a module should be able to processed, debugged and/or verified, independently, without waiting for completion of other modules immediately after its creation. By putting together thus constructed reliable modules, a reliable software can be efficiently and naturally generated.

The IOTA system is constituted of five major subsystems: Developer, Debugger, Verifier, Prover, and Executor. These subsystems are highly integrated into an interactive system over a database of modules (*modulebase*) which maintains all information necessary for each subsystem. The organization of the modulebase is given in Chapter 6.



← : data flow

The Developer is the main interface between the user and the system. Modules are input and modified with the Developer. When input, the source text of a module is analyzed both syntactically and semantically, translated into an inner representation, and stored into the modulebase. Modifications on modules are proceeded directly on the inner representation through the Developer. In addition, the Developer provides the programmer with various information necessary to develop modules. The design principles and the system description of the Developer are presented in Chapters 2 and 4, respectively. The Debugger is an abstraction-oriented dynamic debugging aids.

The Verifier is an interactive verification system specially designed for modular program development. It also manages information concerning ongoing verification and, with help of the Prover, verifies that the program realization of each module meets its specification. Chapter 1 introduces a formalization of abstraction concepts as a basis for module verification, and presents the verification method which is implemented by the Verifier. Chapter 2 includes discussion on the role of the verification and specification embedded in the whole programming environment. The detailed behavior of the Verifier is presented in Chapter 6.

The Prover is an interactive proof system for a many-sorted first-order logic. Chapter 3 discusses the major difficulties with machine proofs for realistic module verification and suggests a solution which is adopted in the Prover as its proof strategies. The features of the Prover are given in Chapter 7.

The Executor generates executable codes from the inner representation of each module, loads them, and executes them. The main implementation issue with this subsystem is separate processing of modules, especially of those which are type-parameterized. The solution in IOTA is discussed in Chapter 5.

While these subsystems work on a single modulebase, there is an inter-modulebase utility system called the Job Manager, which supports cooperation of more than a single programmers by managing data transmission between multiple modulebases. The details of its functions are presented in Chapter 8.

In Chapter 9, we show how we solve a problem in the language IOTA and how the IOTA system supports development, verification, and execution of modules.

The IOTA system currently runs on DECSystem 20 and M-series (IBM-compatible machines by Fujitsu and Hitachi). Transplantations to VAX-11 and Eclipse-MV are planned. Further information is available from the editors.

Acknowledgement

The editors thank Tatsuyuki Akiyama, Makoto Murata, Hayao Nakahara and Takashi Sakuragawa for their contribution to the project.

D. INFORMAL METHODS

1. INTRODUCTION

The world currently seems to have an insatiable demand for software. Even if the number of software engineers increases dramatically, along with productivity and quality it will still not fulfill the demand.

As the world in general becomes more familiar with Information Technology the sophistication and ability of end users will increase. With better packages and tools much of today's programming will be 'off loaded' to end users. This trend is well set now with the advent of Visicalc type products, report generators, word processors, office automation, IKBS expert systems and 'fourth generation' languages and applications generators.

Not all software products will need to be of the highest quality. Thus in parallel to the highly professional, 'precision engineering' approach, typified by the formal methods based techniques, there are emerging 'informal' tools and methods which allow a highly experimental, exploratory style of development.

One could argue that this ad-hoc approach is the traditional one; the difference is the degree of computer based tool support currently being developed which makes this approach somewhat more cost-effective. The different approaches are not competitors; flight critical control software is unlikely to get CAA certification if it has been developed by a 'suck it and see' approach; no manager is likely to submit his Visicalc spreadsheets for BSI Quality Certification. The tools themselves and their associated methods, are increasingly likely to be of high quality and based on sound theory.

2. COMPONENTS AND PACKAGES

The informal approach involving end users will significantly stimulate both the package market (as Visicalc and its successors have demonstrated) but also the new Component market which has yet to emerge in large scale terms.

3. SMALLTALK

An example of the computer based, informal approach to software construction is the Smalltalk system [10], a famous piece of research work done at Xerox's Palo Alto Research Centre (PARC).

Smalltalk uses the object-oriented approach. Objects can be thought of as small autonomous programs which communicate between one another by sending messages (data). Communicating networks of objects can be constructed to form applications programs. Efficient software production is achieved by the ability to quickly and easily replicate objects. An object is not described explicitly but a more general description defines a class of objects. An individual object is thus built as an instance of a class.

Class descriptions can be easily modified and reused to speed experimentation. This approach is only made feasible by the recent development of high powered, sophisticated single user computers with high resolution displays and good man-machine interfaces.

Because of this high degree of computer based support by closely integrated sets of software tools, the Smalltalk system exhibits many of the properties of the envisaged IPSE developments.

E. NEW SOFTWARE TOOLS

1. TRADITIONAL TOOLS

The traditional software tools used to construct a program are

1. Free form TEXT EDITOR to input and edit source code

2. A BATCH MODE COMPILER to produce relocatable modules

3. A LINK EDITOR to combine relocatable modules into an executable binary program.

Most editors have no knowledge of the meaning of the data on which they operate ie they operate on a string of meaningless, independent characters. This enables one editor to be used to input and manipulate arbitrary files but also this freedom enables the user to input bad data such as source code containing syntax errors.

Most compilation systems are not designed for interactive working. Even if the only change to a program is that 'A+B' becomes 'A-B' then one or more modules must be recompiled and the entire program relinked. This is expensive in programmer time and machine resources causing wasteful recompilation of unaltered source code.

It is very unusual for a compiler to accept input expressed in more than one programming language, yet the world's software is written in several major languages and hundreds of minor ones. This lack of ability to mix languages results in poor reuse of existing code and encourages wheel reinvention. Although compilers for each language exist they cannot be made to work together; their actions are independent, not integrated.

2. NEW TOOLS

Modern software tools are beginning to exhibit the following characteristics.

1. Designed for interactive use

2. Knowledge of language syntax

3. Knowledge of language semantics

4. Designed for integrated working as member of toolset.

2.1 <u>Syntactic Tools</u>

A 'syntax knowledgeable' editor is a synthesis of editor, compiler front end and text formatter which

(a) ensures syntactic correctness

(b) offers templates to speed typing and assist the programmer. Such an editor contains within it details of the syntax of the relevant programming language which the programmer is currently using. These details, toegether with interaction with the programmer, ensure that the only text the programmer inputs conforms to the syntactic rules of the language. This speeds production because the subsequent compilation will never fail due to syntax errors.

(c) automatically formats the program text according to predefined layout rules of indentation, fonts etc (pretty printing).

The sophistication of interaction with the programmer is usually increased with such editors by generating templates. Once the editor has recognised the overall syntactic form which the programmer wishes to input a skeleton of the complete form is immediately generated, saving much typing. Usually the template will contain place holders which the programmer then fills in to complete the structure. It works like this. The programmer wishes to input a completely new procedure or subroutine. He types 'proc' by which time the syntax analyser can predict the programmer's intention and, unprompted, generates the following text

```
procedure   <NAME> <ARGS>;
  var <VARIABLE NAMES>;
  const <CONSTANTS>;

begin
      <STATEMENTS>
end   [<NAME>];
```

in which the lower case letters are items in the language which become part of the program (and save the programmer having to type them) and the upper case letters in angled brackets are place holders showing the names and positions of language elements which the programmer must further define. These tools greatly increase productivity and reduce compilation costs.

2.2 Semantic Tools

More sophisticated tools are being built based on a closer integration of the editor/compiler/proof checker technologies. These are called 'semantic' tools because the checks and aids they provide require a deeper analysis of the program under construction than simple syntactic correctness.

Integration of the editor/compiler functions enables immediate detection of the use of undeclared variables and guidance on the likely causes and remedies for such errors.

Integration of the editor/compiler/proof checker technologies allows partial or symbolic execution to take place during construction. This prevents the programmer making such errors as mistyping 'A' for 'B' in

```
C:=A-A;         (s.b.  C:=B-A)
E:=D/C;         (division by zero error detected)
                (by substituting E:=D/ (A-A=O)  )
```

2.3 Databases

As tools begin to operate on databases rather than files it will become increasingly possible to track, trap and prevent errors of more global significance which arise out of module interface inconsistencies, wrong versions of modules being used as well as automating the configuration, construction and change management of large complex systems which are too big ever to recompile completely.

3. TOOL DEMONSTRATIONS

The 'Blit' and 'Smalltalk' video demonstrations, backed up by the live demonstrations will show in practice much of the philosophy outlined in this paper.

In particular the demonstrations will show

1. increased productivity by allowing multiple activity threads for one individual via multiple virtual terminals (windows) to a multi programming operating system dedicated to a single, real user.

This uses

  a. powerful single user system workstation

  b. bit mapped, high resolution, raster graphics

  c. mouse

  d. windows.

2. improved productivity via highly interactive, highly integrated toolsets with individual tools having significant 'knowledge' of syntax and semantics.

This uses

  a. mouse driven editors operating uniformly across all tools

  b. syntax knowledgeable editing

  c. place holders, templates, power typing

  d. reusable, tailorable code (Smalltalk classes)

  e. recursively activatable browsers integrated closely with other tools.

3. informal development methods allowing reusability, incremental developments, rapid prototyping, animation and object orientated programming

4. local area networks to create physically distributed but logically integrated computing systems

  a. Newcastle Connection

  b. Servers.

Tools under current development but not demonstrated include

1. spelling checkers for documentation as well as code
2. grammar checkers for documentation as well as code
3. style analysers for documentation as well as code
4. quality metrics for documentation as well as code
5. proof generators
6. proof checkers
7. version control
8. configuration management
9. project cost measurement
10. project progress measurement
11. project management aids
12. integration of software design with hardware design
13. integration of software and hardware design with office automation and other corporate applications of information technology.

4. HOST-TARGET WORKING

These new tools are more expensive to produce and operate making this new style of production more capital intensive. However the quality and productivity gains they will permit will mean that those producers failing to make the necessary investment will suffer declining competitive ability.

To maximise the use of capital investment in new software production facilities and in staff training, it is increasingly clear that a single design and production system, called the Integrated Project Support Environment, will come to be used in any given area of activity. This is called the host because it will have to generate binary programs which execute on a wide range of 'applications' processors, the targets.

Microprocessors are cheap but have limited capability. It is not economic to develop software on the same microprocessor system as will be used in the eventual application. One is hardly likely to be able to develop high quality, proven and safety certified software to control the brakes of motor cars on the same configuration as will be installed in the actual cars. Large, powerful computers will be needed to run sophisticated software design aids whilst small, cheap microprocessors will be installed in the cars. Hence host-target working will become the norm in future for anyone who wishes to reuse software and software tools.

F. CONCLUSION

1. METHODS, SKILLS, TOOLS

Software design and development requires a harmonious blend of effective methods, good tools, and people skilled and trained to use them.

This paper began with the definition that "software engineering is the application of science and mathematics by which the capabilities of computer equipment are made useful to man via computer programs, procedures and associated documentation" [1].

The Alvey Directorate is helping to stimulate the development, in the UK, of new methods and tools for software development, but no government sponsored research programme can equip engineers with the skills to use the new tools and methods. Nor can a government research programme instill into engineers, especially in such a new discipline as software engineering, the traditions of professional conduct and standards, of social responsibility and commercial propriety.

With the ever widening role of software in social and industrial life, with the increasing integration of 'hardware', 'software' and 'systems' design, development and construction techniques, perhaps the time has come to prepare a path to lead the youthful programming enthusiast through education and training towards the skill levels which the UK needs to badly today and beyond this, via continuous, life time retraining to the highest standards of skill and conduct befitting the title of 'engineer'.

R E F E R E N C E S

1. **BOEHM, B W**
Software Engineering Economics
Prentice Hall

2. **WITTY, R W**
The Software Technology Initiative
Final Report 1981-1984
SERC, Oct 84

3. **TALBOT, D E**
Alvey Software Engineering - a strategy overview
ALVEY DIRECTORATE, Nov 83

4. **TALBOT, D E & WITTY, R W**
Software Engineering Strategy
ALVEY DIRECTORATE, Nov 83

5. **RELIABILITY & METRICS ADVISORY PANEL**
Software Engineering: Software Reliability & Metrics Programme
ALVEY DIRECTORATE, July 84

6. **FORMAL METHODS ADVISORY PANEL**
Software Engineering: Programme for Formal Methods in Systems Development
ALVEY DIRECTORATE, April 84

7. **DIGNAN, A**
Software Engineering/IKBS: Strategy for Knowledge Based IPSE Development
ALVEY DIRECTORATE, August 84

8. **NAUR, P & RANDELL B (Eds)**
Software Engineering: Report on a Conference Sponsored by the NATO Science Committee, Garmisch, Germany, 7-11 Oct 1968.
Scientific Affairs Division, NATO, Brussels, Jan 69.

9. **GORDON, M, MILNER, R & WADSWORTH, C**
Edinburgh LCF
Lecture Notes in Computer Science Vol 78
Springer-Verlag, 1979

10. **GOLDBERG, A & ROBSON, D**
Smalltalk-80 : The Language and its Implementation
Adison-Wesley, 1983

11. **DAHL, O J, DIJKSTRA, E W & HOARE, C A R**
Structured Programming
Academic Press, London 1972

12. **NAKAJIMA, R & YUASA, T**
The IOTA Programming System
Lecture Notes in Computer Science Vol 160.
Springer-Verlag, 1983

A P P E N D I X

A. HUMOROUS (?)

1. Laws of Project Management

2. IBM's New Operating System

3. Real Programmers Don't Use PASCAL

LAWS OF PROJECT MANAGEMENT

1. No major project is ever installed on time, within budgets, with the same staff that started it. Yours will not be the first.

2 Projects progress quickly until they become 90 per cent complete, then they remain at 90 per cent complete forever.

3. One advantage of fuzzy project objectives is that they let you avoid the embarrassment of estimating the corresponding costs.

4. When things are going well, something will go wrong.

   - When things just cannot get any worse, they will.

   - When things appear to be going better you have overlooked something.

5. If project content is allowed to change freely, the rate of change will exceed the rate of progress.

6. No system is ever completely debugged: Attempts to debug a system inevitably introduce new bugs that are even harder to find.

7. A carelessly planned project will take three times longer to complete than expected; a carefully planned project will take only twice as long.

8. Project teams detest progress reporting because it vividly manifests their lack of progress.

IBM

Data Processing Division

Date: January 30, 1979

Programming Announcement

NEW OPERATING SYSTEM

Because so many users have asked for an operating system of even greater capability than VM, IBM announces the Virtual Universe Operating System - OS/VU.

Running under OS/VU, the individual user appears to have not merely a machine of his own, but an entire universe of his own, in which he can set up and take down his own programs, data sets, systems networks, personnel, and planetary systems. He need only specify the universe he desires, and the OS/VU system generation program (IEHGOD) does the rest. This program will reside in SYS1.GODLIB. The minimum time for this funciton is 6 days of activity and 1 day of review. In conjunction with OS/VU, all system utilities have been replaced by one program (IEHPROPHET) which will reside in SYS1.MESSIAH. This program has no parms or control cards as it knows what you want to do when it is executed.

Naturally, the user must have attained a certain degree of sophistication in the data processing field if an efficient utilization of OS/VU is to be achieved. Frequent calls to non-resident galaxies, for instance, can lead to unexpected delays in the execution of a job. Although IBM, through its wholly-owned subsidiary, The United States, is working on a program to upgrade the speed of light and thus reduce the overhead of extraterrestrial and metadimensional paging, users must be careful for the present to stay within the laws of physics. IBM must charge an additional fee for violations.

OS/VU will run on any IBM x0xx equipped with Extended WARP Feature. Rental is twenty million dollars per cpu/nanosecond.

Users should be aware that IBM plans to migrate all existing systems and hardware to OS/VU as soon as our engineers effect one output that is (conceptually) error-free. This will give us a base to develop an even more powerful operating system, target date 2001, designated "Virtual Reality". OS/VR is planned to enable the user to migrate to totally unreal universes. To aid the user in identifying the difference between "Virtual Reality" and "Real Reality", a file containing a linear arrangement of multisensory total records of successive moments of now will be established. It's name will be SYS1.est.

For more information, contact your IBM data processing representative.

International Business Machines Corporation
1133 Westchester Avenue
White Plains, New York 10604

"Real Programmers Don't Use PASCAL"

Back in the good old days — the "Golden Era" of computers, it was easy to separate the men from the boys (sometimes called "Real Men" and "Quiche Eaters" in the literature). During this period, the Real Men were the ones that understood computer programming, and the Quiche Eaters were the ones that didn't. A real computer programmer said things like "DO 10 I=1,10" and "ABEND" (they actually talked in capital letters, you understand), and the rest of the world said things like "computers are too complicated for me" and "I can't relate to computers — they're so impersonal". (A previous work [1] points out that Real Men don't "relate" to anything, and aren't afraid of being impersonal.)

But, as usual, times change. We are faced today with a world in which little old ladies can get computers in their microwave ovens, 12-year-old kids can blow Real Men out of the water playing Asteroids and Pac-Man, and anyone can buy and even understand their very own Personal Computer. The Real Programmer is in danger of becoming extinct, of being replaced by high-school students with TRASH-80's.

There is a clear need to point out the differences between the typical high-school junior Pac-Man player and a Real Programmer. If this difference is made clear, it will give these kids something to aspire to — a role model, a Father Figure. It will also help explain to the employers of Real Programmers why it would be a mistake to replace the Real Programmers on their staff with 12-year-old Pac-Man players (at a considerable salary savings).

LANGUAGES

The easiest way to tell a Real Programmer from the crowd is by the programming language he (or she) uses. Real Programmers use FORTRAN. Quiche Eaters use PASCAL. Nicklaus Wirth, the designer of PASCAL, gave a talk once at which he was asked "How do you pronounce your name?". He replied, "You can either call me by name, pronouncing it 'Veert', or call me by value, 'Worth'." One can tell immediately from this comment that Nicklaus Wirth is a Quiche Eater. The only parameter passing mechanism endorsed by Real Programmers is call-by-value-return, as implemented in the IBM\370 FORTRAN-G and H compilers. Real programmers don't need all these abstract concepts to get their jobs done — they are perfectly happy with a keypunch, a FORTRAN IV compiler, and a beer.

* Real Programmers do List Processing in FORTRAN.

* Real Programmers do String Manipulation in FORTRAN.

* Real Programmers do Accounting (if they do it at all) in FORTRAN.

* Real Programmers do Artificial Intelligence programs in FORTRAN.

If you can't do it in FORTRAN, do it in assembly language. If you can't do it in assembly language, it isn't worth doing.

STRUCTURED PROGRAMMING

The academics in computer science have gotten into the "structured programming" rut over the past several years. They claim that programs are more easily understood if the programmer uses some special language constructs and techniques. They don't all agree on exactly which constructs, of course, and the examples they use to show their particular point of view invariably fit on a single page of some obscure journal or another — clearly not enough of an example to convince anyone. When I got out of school, I thought I was the best programmer in the world. I could write anunbeatable tic-tac-toe program, use five different computer languages, and create 1000-line programs that WORKED. (Really!) Then I got out into the Real World. My first task in the Real World was to read and understand a 200,000-line FORTRAN program, then speed it up by a factor of two. Any Real Programmer will tell you that all the Structured Coding in the world won't help you solve a problem like that — it takes actual talent. Some quick observations on Real Programmers and Structured Programming:

* Real Programmers aren't afraid to use GOTO's.

* Real Programmers can write five-page-long DO loops without getting confused.

* Real Programmers like Arithmetic IF statements — they make the code more interesting.

* Real Programmers write self-modifying code, especially if they can save 20 nanoseconds in the middle of a tight loop.

* Real Programmers don't need comments — the code is obvious.

* Since FORTRAN doesn't have a structured IF, REPEAT ... UNTIL, or CASE statement, Real Programmers don't have to worry about not using them. Besides, they can be simulated when necessary using assigned GOTO's.

Data Structures have also gotten a lot of press lately. Abstract Data Types, Structures, Pointers, Lists, and Strings have become popular in certain circles. Wirth (the above-mentioned Quiche Eater) actually wrote an entire book [2] contending that you could write a program based on data structures, instead of the other way around. As all Real Programmers know, the only useful data structure is the Array. Strings, lists, structures, sets -- these are all special cases of arrays and can be treated that way just as easily without messing up your programing language with all sorts of complications. The worst thing about fancy data types is that you have to declare them, and Real Programming Languages, as we all know, have implicit typing based on the first letter of the (six character) variable name.

OPERATING SYSTEMS

What kind of operating system is used by a Real Programmer? CP/M? God forbid -- CP/M, after all, is basically a toy operating system. Even little old ladies and grade school students can understand and use CP/M.

Unix is a lot more complicated of course -- the typical Unix hacker never can remember what the PRINT command is called this week -- but when it gets right down to it, Unix is a glorified video game. People don't do Serious Work on Unix systems: they send jokes around the world on UUCP-net and write adventure games and research papers.

No, your Real Programmer uses OS\370. A good programmer can find and understand the description of the IJK305I error he just got in his JCL manual.A great programmer can write JCL without referring to the manual at all. A truly outstanding programmer can find bugs buried in a 6 megabyte core dump without using a hex calculator. (I have actually seen this done.)

OS is a truly remarkable operating system. It's possible to destroy days of work with a single misplaced space, so alertness in the programming staff is encouraged. The best way to approach the system is through a keypunch. Some people claim there is a Time Sharing system that runs on OS\370, but after careful study I have come to the conclusion that they were mistaken.

PROGRAMMING TOOLS

What kind of tools does a Real Programmer use? In theory, a Real Programmer could run his programs by keying them into the front panel of the computer. Back in the days when computers had front panels, this was actually done occasionally. Your typical Real Programmer knew the entire bootstrap loader by memory in hex, and toggled it in whenever it got destroyed by his program. (Back then, memory was memory — it didn't go away when the power went off. Today, memory either forgets things when you don't want it to, or remembers things long after they're better forgotten.) Legend has it that Seymore Cray, inventor of the Cray I supercomputer and most of Control Data's computers, actually toggled the first operating system for the CDC7600 in on the front panel from memory when it was first powered on. Seymore, needless to say, is a Real Programmer.

One of my favorite Real Programmers was a systems programmer for Texas Instruments. One day he got a long distance call from a user whose system had crashed in the middle of saving some important work. Jim was able to repair the damage over the phone, getting the user to toggle in disk I/O instructions at the front panel, repairing system tables in hex, reading register contents back over the phone. The moral of this story: while a Real Programmer usually includes a keypunch and lineprinter in his toolkit, he can get along with just a front panel and a telephone in emergencies.

In some companies, text editing no longer consists of ten engineers standing in line to use an 029 keypunch. In fact, the building I work in doesn't contain a single keypunch. The Real Programmer in this situation has to do his work with a "text editor" program. Most systems supply several text editors to select from, and the Real Programmer must be careful to pick one that reflects his personal style. Many people believe that the best text editors in the world were written at Xerox Palo Alto Research Center for use on their Alto and Dorado computers [3]. Unfortunately, no Real Programmer would ever use a computer whose operating system is called SmallTalk, and would certainly not talk to the computer with a mouse.

Some of the concepts in these Xerox editors have been incorporated into editors running on more reasonably named operating systems — EMACS and VI being two. The problem with these editors is that Real Programmers consider "what you see is what you get" to be just as bad a concept in Text Editors as it is in women. No the Real Programmer wants a "you asked for it, you got it" text editor -- complicated, cryptic, powerful, unforgiving, dangerous. TECO, to be precise.

It has been observed that a TECO command sequence more closely resembles transmission line noise than readable text [4]. One of the more entertaining games to play with TECO is to type your name in as a command line and try to guess what it does. Just about any possible typing error while talking with TECO will probably destroy your program, or even worse — introduce subtle and mysterious bugs in a once working subroutine.

For this reason, Real Programmers are reluctant to actually edit a program that is close to working. They find it much easier to just patch the binary object code directly, using a wonderful program called SUPERZAP (or its equivalent on non-IBM machines). This works so well that many working programs on IBM systems bear no relation to the original FORTRAN code. In many cases, the original source code is no longer available. When it comes time to fix a program like this, nomanager would even think of sending anything less than a Real Programmer to do the job — no Quiche Eating structured programmer would even know where to start. This is called "job security".

Some programming tools NOT used by Real Programmers:

* FORTRAN preprocessors like MORTRAN and RATFOR. The Cuisinarts of programming — great for making Quiche. See comments above on structured programming.

* Source language debuggers. Real Programmers can read core dumps.

* Compilers with array bounds checking. They stifle creativity, destroy most of the interesting uses for EQUIVALENCE, and make it impossible to modify the operating system code with negative subscripts. Worst of all, bounds checking is inefficient.

* Source code maintenance systems. A Real Programmer keeps his code locked up in a card file, because it implies that its owner cannot leave his important programs unguarded [5].

THE REAL PROGRAMMER AT WORK

Where does the typical Real Programmer work? What kind of programs are worthy of the efforts of so talented an individual? You can be sure that no Real Programmer would be caught dead writing accounts-receivable programs in COBOL, or sorting mailing lists for People magazine. A Real Programmer wants tasks of earth-shaking importance (literally!).

* Real Programmers work for Los Alamos National Laboratory, writing atomic bomb simulations to run on Cray I supercomputers.

* Real Programmers work for the National Security Agency, decoding Russian transmissions.

* It was largely due to the efforts of thousands of Real Programmers working for NASA that our boys got to the moon and back before the Russkies.

* Real Programmers are at work for Boeing designing the operating systems for cruise missiles.

Some of the most awesome Real Programmers of all work at the Jet Propulsion Laboratory in California. Many of them know the entire operating system of the Pioneer and Voyager spacecraft by heart. With a combination of large ground-based FORTRAN programs and small spacecraft-based assembly language programs, they are able to do incredible feats of navigation and improvisation — hitting ten-kilometer wide windows at Saturn after six years in space, repairing or bypassing damaged sensor platforms, radios, and batteries. Allegedly, one Real Programmer managed to tuck a pattern-matching program irto a few hundred bytes of unused memory in a Voyager spacecraft that searched for, located, and photographed a new moon of Jupiter.

The current plan for the Galileo spacecraft is to use a gravity assist trajectory past Mars on the way to Jupiter. This trajectory passes within 80 +/-3 kilometers of the surface of Mars. Nobody is going to trust a PASCAL program (or a PASCAL programmer) for navigation to these tolerances.

As you can tell, many of the world's Real Programmers work for the U.S. Government — mainly the Defense Department. This is as it should be. Recently, however, a black cloud has formed on the Real Programmer horizon. It seems that some highly placed Quiche Eaters at the Defense Department decided that all Defense programs should be written in some grand unified language called "ADA" ((C), DoD). For a while, it seemed that ADA was destined to become a language that went against all the precepts of Real Programming — a language with structure, a language withdata types, strong typing, and semicolons. In short, a language designed to cripple the creativity of the typical Real Programmer. Fortunately, the language adopted by DoD has enough interesting features to make it approachable — it's incredibly complex, includes methods for messing with the operating system and rearranging memory, and Edsgar Dijkstra doesn't like it [6]. (Dijkstra, as I'm sure you know, was the author of "GoTos Considered Harmful" — a landmark work in programming methodology, applauded by PASCAL programmers and Quiche Eaters alike.) Besides, the determined Real Programmer can write FORTRAN programs in any language.

The Real Programmer might compromise his principles and work on something slightly more trivial than the destruction of life as we know it, providing there's enough money in it. There are several Real Programmers building video games at Atari, for example. (But not playing them — a Real Programmer knows how to beat the machine every time: no challenge in that.) Everyone working at LucasFilm is a Real Programmer. (It would be crazy to turn down the money of fifty million Star Trek fans.) The proportion of Real Programmers in Computer Graphics is somewhat lower than the norm, mostly because nobody has found a use for computer graphics yet. On the other hand, all computer graphics is done in FORTRAN, so there are a fair number of people doing graphics in order to avoid having to write COBOL programs.

THE REAL PROGRAMMER AT PLAY

Generally, the Real Programmer plays the same way he works -- with computers. He is constantly amazed that his employer actually pays him to do what he would be doing for fun anyway (although he is careful not to express this opinion out loud). Occasionally, the Real Programmer does step out of the office for a breath of fresh air and a beer or two. Some tips on recognizing Real Programmers away from the computer room:

* At a party, the Real Programmers are the ones in the corner talking about operating system security and how to get around it.

* At a football game, the Real Programmer is the one comparing the plays against his simulations printed on 11 by 14 fanfold paper.

* At the beach, the Real Programmer is the one drawing flowcharts in the sand.

* At a funeral, the Real Programmer is the one saying "Poor George. And he almost had the sort routine working before the coronary."

* In a grocery store, the Real Programmer is the one who insists on running the cans past the laser checkout scanner himself, because he never could trust keypunch operators to getit right the first time.

THE REAL PROGRAMMER'S NATURAL HABITAT

What sort of environment does the Real Programmer function best in? This is an important question for the managers of Real Programmers. Considering the amount of money it costs to keep one on the staff, it's best to put him (or her) in an environment where he can get his work done.

The typical Real Programmer lives in front of a computer terminal. Surrounding this terminal are:

* Listings of all programs the Real Programmer has ever worked on, piled in roughly chronological order on every flat surface in the office.

* Some half-dozen or so partly filled cups of cold coffee. Occasionally, there will be cigarette butts floating in the coffee. In some cases, the cups will contain Orange Crush.

* Unless he is very good, there will be copies of the OS JCL manual and the Principles of Operation open to some particularly interesting pages.

* Taped to the wall is a line-printer Snoopy calendar for the year 1969.

* Strewn about the floor are several wrappers for peanut butter filled cheese bars — the type that are made pre-stale at the bakery so they can't get any worse while waiting in the vending machine.

* Hiding in the top left-hand drawer of the desk is a stash of double-stuff Oreos for special occasions.

* Underneath the Oreos is a flowcharting template, left there by the previous occupant of the office. (Real Programmers write programs, not documentation. Leave that to the maintenance people.)

The Real Programmer is capable of working 30, 40, even 50 hours at a stretch, under intense pressure. In fact, he prefers it that way. Bad response time doesn't bother the Real Programmer — it gives him a chance to catch a little sleep between compiles. If there is not enough schedule pressure on the Real Programmer, he tends to make things more challenging by working on some small but interesting part of the problem for the first nine weeks, then finishing the rest in the last week, in two or three 50-hour marathons. This not only impresses the hell out of his manager, who was despairing of ever getting the project done on time, but creates a convenient excuse for not doing the documentation. In general:

* No Real Programmer works 9 to 5 (unless it's the ones at night).

* Real Programmers don't wear neckties.

* Real Programmers don't wear high-heeled shoes.

* Real Programmers arrive at work in time for lunch [9].

* A Real Programmer might or might not know his wife's name. He does, however, know the entire ASCII (or EBCDIC) code table.

* Real Programmers don't know how to cook. Grocery stores aren't open at three in the morning. Real Programmers survive on Twinkies and coffee.

THE FUTURE

What of the future? It is a matter of some concern to Real Programmers that the latest generation of computer programmers are not being brought up with the same outlook on life as their elders. Many of them have never seen a computer with a front panel. Hardly anyone graduating from school these days can do hex arithmetic without a calculator. College graduates these days are soft — protected from the realities of programming by source level debuggers, text editors that count parentheses, and "user friendly" operating systems. Worst of all, some of these alleged "computer scientists" manage to get degrees without ever learning FORTRAN! Are we destined to become an industry of Unix hackers and PASCAL programmers?

From my experience, I can only report that the future is bright for Real Programmers everywhere. Neither OS\370 nor FORTRAN show any signs of dying out, despite all the efforts of PASCAL programmers the world over. Even more subtle tricks, like adding structured coding constructs to FORTRAN have failed. Oh sure, some computer vendors have come out with FORTRAN 77 compilers, but every one of them has a way of converting itself back into a FORTRAN 66 compiler at the drop of an option card — to compile DO loops like God meant them to be.

Even Unix might not be as bad on Real Programmers as it once was. The latest release of Unix has the potential of an operating system worthy of any Real Programmer — two different and subtly incompatible user interfaces, an arcane and complicated teletype driver, virtual memory. If you ignore the fact that it's "structured", even 'C' programming can be appreciated by the Real Programmer: after all, there's no type checking, variable names are seven (ten? eight?) characters long, and the added bonus of the Pointer data type is thrown in — like having the best parts of FORTRAN and assembly language in one place. (Not to mention some of the more creative uses for #define.)

No, the future isn't all that bad. Why, in the past few years, the popular press has even commented on the bright new crop of computer nerds and hackers ([7] and [8]) leaving places like Stanford and M.I.T. for the Real World. From all evidence, the spirit of Real Programming lives on in these young men and women. As long as there are ill-defined goals, bizarre bugs, and unrealistic schedules, there will be Real Programmers willing to jump in and Solve The Problem, saving the documentation for later. Long live FORTRAN!

* Real Programmers don't wear neckties.

* Real Programmers don't wear high-heeled shoes.

* Real Programmers arrive at work in time for lunch [9].

* A Real Programmer might or might not know his wife's name. He does, however, know the entire ASCII (or EBCDIC) code table.

* Real Programmers don't know how to cook. Grocery stores aren't open at three in the morning. Real Programmers survive on Twinkies and coffee.

REFERENCES

[1] Feirstein, B., "Real Men don't Eat Quiche", New York, Pocket Books, 1982.

[2] Wirth, N., "Algorithms + Data Structures = Programs", Prentice Hall, 1976.

[3] Ilson, R., "Recent Research in Text Processing", IEEE Trans. Prof. Commun., Vol. PC-23, No. 4, Dec. 4, 1980.

[4] Finseth, C., "Theory and Practice of Text Editors — or — a Cookbook for an EMACS", B.S. Thesis, MIT/LCS/TM-165, Massachusetts Institute of Technology, May 1980.

[5] Weinberg, G., "The Psychology of Computer Programming", New York, Van Nostrand Reinhold, 1971, p. 110.

[6] Dijkstra, E., "On the GREEN language submitted to the DoD", Sigplan notices, Vol. 3 No. 10, Oct 1978.

[7] Rose, Frank, "Joy of Hacking", Science 82, Vol. 3 No. 9, Nov 82, pp. 58-66.

[8] "The Hacker Papers", Psychclogy Today, August 1980.

[9] sdcarl!lin, "Real Programmers", UUCP-net, Thu Oct 21 16:55:16 1982

A P P E N D I X

B. SERIOUS

1. SIGGRAPH Video Review

2. Mechanical Theorem Proving

3. Expansion of Alvey SE Strategy

SIGGRAPH VIDEO REVIEW

Human-Computer Interaction: the Focus of Two New Issues

Two new issues (12 & 13) of the SIGGRAPH video review (SVR) have recently been completed. What is unique is that they are organized around a single theme: human-computer interaction. The tapes are an edited compilation of the video sessions at CHI '83, the 1983 Conference on Human Factors in Computing Systems. Each tape is one hour long, and contains a number of titles which illustrate important aspects of interaction and input. Techniques, user interface management tools, technologies and sample applications are examples of topics covered. The examples are short, and were chosen to give as good an overview as possible as to the current state-of-the-art in interaction. The collection should be of interest to systems designers, researchers, students and managers.

The appearance of these tapes as part of the SVR is in keeping with the charter of SIGGRAPH, and the recommendations of the SIGGRAPH-sponsored 1982 Workshop on Graphical Input and Interaction Techniques. The CHI '83 video sessions, organized by Sara Bly, Michael Harris and Donald Patterson collected, for the first time, a wide assortment of high-quality tapes on the subject. The conference was sponsored by SIGCHI and the Human Factors Society. But since it was held in cooperation with (among others) SIGGRAPH, and since SIGGRAPH had in place a mechanism for tape editing and distribution, it was natural that an edited version of the session should be distributed by SVR. The tapes have been edited for SVR by Bill Buxton, Copper Giloth and Raul Zaritsky with the cooperation of Tom DeFanti.

NEW!

Contents of Issue 13:
Edited 3/10/84
1. Blit (Bell Labs)
2. The Movie Manual Project (MIT)
3. The Office of the Professional (Imperial College)
4. Put That There (MIT)
5. Program Visualization (CCA)
6. Magnetic Fusion Experiment Control Center (Lawrence Livermore Labs)
7. Sketchpad (MIT) (not shown at CHI '83)

NEW!

Contents of Issue 12:
Edited 3/10/84
1. Rapid Prototyping Using Flair (TRW)
2. Towards A Comprehensive UIMS (U of T)
3. Cousin Interface System (CMU)
4. Tiger System Demonstration (Boeing)
5. Video Games by Example (Atari)
6. Mockingbird (XEROX)
7. SSSP Demo (U of T)
8. Selection-Positioning Task Study (U of T)

Contents of Issue 1:
Edited 5/15/80
1. TOPES—Bell Laboratories
2. Newswhole—University of Toronto
3. VideoCel—Computer Creations, Inc.
4. Sunstone—Ed Emschwiller
5. Voyager 2—J. Blinn et. al.
6. Information International Inc. Demo Reel
7. DNA with Ethidium—N. Max et. al.

Contents of Issue 2:
Edited 8/30/81
1. The Compleat Angler—T. Whitted
2. Vol Libre—L. Carpenter
3. JPL/Saturn—J. Blinn et. al
4. Peak—N. Snitly
5. Doxorubicin/DNA—N. Max et. al
6. Digital Effects Demo Reel
7. MAGI/Synthavision Demo Reel
8. Spatial Data Mgt. System—C. Herot et. al.
9. Pantomation—T. DeWitt et. al.
10. Artifacts—The Vasulkas

Contents of Issue 3:
Edited 8/30/81
1. CTS Flight Simulator—Evans and Sutherland
2. Time Rider—JVC
3. Imagination—Acme Cartoon Company, Inc.
4. Dubner Demo Tape
5. Vidsizer—Dan Franzblau
6. Zgrass Paint Demo—Giloth et. al.

Contents of Issue 4:
Edited 8/30/81
1. Abel Demo Reel—W. Kovacs et. al.
2. Image West Demo Reel
3. Ohio State Computer Graphics Research Group Terrain Model—C. Csuri et. al.
4. Computer-Assisted Dance Notation—T. Calvert et. al.
5. The GRIP—75 Man-machine Interface—University of North Carolina Computer Science Department
6. Graphics Interactions at NRC—M. Wein et. al. National Film Board of Canada

Contents of Issue 5:
Edited 10/22/82
1. Evans & Sutherland Demo '82
2. The Tactical Edge—Evans & Sutherland
3. Carla's Island—Nelson Max, LLL
4. Aurora Demo
5. Digital Effects Sampler '82
6. Real Time Design, Inc. Zgrass Demo
7. Marks & Marks Demo

Contents of Issue 6:
Edited 10/22/82
1. Abel '82 Demo Reel
2. Galileo—Jim Blinn, et. al., JPL
3. Mimas/Voyager II—Jim Blinn, et. al., JPL
4. Non-Edge Computer Image Gen.—Grumman
5. Disspla Animation—ISSCO
6. Tomato Bushy Stunt Virus—Arthur Olson
7. Interactive Raster Graphics Sampler—UNC
8. Ron Hays Music-Image Sampler

Contents of Issue 7:
Edited 11/7/82
1. Triple-I Digital Scene Simulation Reel
2. TRON reference—Disney
3. MAGI/Synthavision '82 Demo
4. Videocel '82—Computer Creations
5. Cranston-Csuri Demo Reel
6. Four Seasons of Japan/Expo '85—NHK
7. Acme Cartoon Company Samples '82
8. ADAM—Arthur Olson and T. J. O'Donnell
9. 1982 Experimental Works—Texnai C.G.L.
10. Sorting Out Sorting Excerpt—U. Toronto

Contents of Issue 8:
Edited 10/27/83
1. Smalltalk—Xerox Corp.
2. Lisa—Apple Corp.
3. Warpitout—Veeder
4. Soma—Gillerman
5. Act III—Winkler & Sanborn
6. Laser Show at SIGGRAPH '83—Heminover & Rollefstad

Contents of Issue 9:
Edited 10/27/83
1. Economars Earth Tours—Upson
2. Toyo Links Demo
3. Antics—Abe
4. Japan Computer Graphics Lab, Inc.
5. Bo Gehring Demo
6. Omnibus Video, Inc.
7. Translation Part 3—Moran
8. Julia I Excerpts—Peitgen & Saupe
9. Space Simulator—Galicki
10. Marks & Marks/Novocon
11. Solid Modeling—Zaritsky & Herr

Contents of Issue 10:
Edited 10/27/83
1. When Mandrills Ruled...-Watterberg
2. Cranston-Csuri
3. Ohio State University—Zeltzer & Van Baerle
4. Pan Optica Preview '83—Gordon
5. Ray Tracing—Barr & Lorig
6. Pacific Data-Images
7. NHK Special Programs Division
8. Humanon—Francois
9. Light & Shadow—Nakamae
10. University of North Carolina Sampler
11. Benesh Notation—Singh
12. Blooming Stars Excerpt—Genda

Contents of Issue 11:
Edited 10/27/83
1. Star Trek II Genesis—Paramount/Lucasfilm
2. Non-Edge CIG—Grumman
3. Digital Effects Demo
4. The Cube CUBE—Gerhard
5. SPN—SEIBU Productions Network
6. Symmetry Test 11A—Newell
7. Composite News—Burson
8. A/V Tour at SIGGRAPH '83—Veeder & Morton
9. Shirogumi Sampler
10. Movie Maker—IPS, Inc.
11. Pixel Play—Nakajima
12. Growth/Mysterious Galaxy—Kawaguchi
13. Digital Harmony—Whitney Sr. et. al.

Two new hours of videotape from SIGCHI '83 have been edited and duplicated to form issues 12 and 13 of the SIGGRAPH Video Review. Each issue is on videotape and is one-hour long. The material in the tapes is in full color and represents advanced applications of computer graphics technology, both hardware and software.

Both ¾" U-matic and VHS formats are available. We do not make Beta or ½" reel-to-reel tapes. PAL and SECAM tapes also are not available.

The ¾" tapes are one-hour long. One issue fits on one tape. Thus, the two new issues occupy two tapes. At the ACM SIGGRAPH member price of $50/tape, both issues come to $100. The non-member prices is $60/tape, so both are $120. Educational institutions may use the member price. For overseas airmail, please include an extra $10/tape, or $20 for the set. Similarly, all 13 issues come to $650 for members, $780 for non-members, plus $130 in additional postage for overseas airmail if necessary.

The VHS videotapes are two hours long. Two issues are on each tape, except for issue 7. Issues 1 & 2, 3 & 4, 5 & 6, 8 & 9, 10 & 11 and 12 & 13 are each $50 for members, and $60 for non-members. Issue #7 is $40 for members, and $50 for non-members.

The same surcharge of $10/tape applies for overseas postage. Thus the new issues, 12 and 13 are $50 for members and $60 for non-members. All 13 issues are $340 for members, $410 for non-members, with an additional $60 for overseas airmail.

Ordering information:
1. You MUST send a check payable in U.S. funds drawn on a U.S. bank. I will return purchase orders unfilled. Return airmail postage is included in the price for North American orders. If you are in an extreme rush, include your Federal Express number.
2. Make the check payable to SIGGRAPH.
3. Send check and order to: SIGGRAPH c/o Tom DeFanti, UIC/ EECS, Box 4348, Chicago, Illinois 60680.
4. Write or call for clarifications. My phone number is (312) 996-5485. I do not loan copies or provide press copies.
5. For best results, include a statement specifying which tapes you want and in which format. Or, include this form and circle the appropriate items below:

Tape format: ¾" or VHS

Issues wanted: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13

Name and address of person to receive tapes: ____________
____
____

B.2 Mechanical Theorem Proving

The following is the output from using a mechanical theorem prover to tackle the Second Example on Section C.3 (page 17).

```
ERIL version R1.0          Rutherford Appleton Laboratory - Informatics Division
-------------------------Simple verification example-------------------------


                              Welcome To

                   EE E E E E    RR R R R      II      LL
                   EE            RR     RR     II      LL
                   EE            RR     RR     II      LL
                   EE E E E      RR R R R      II      LL
                   EE            RR     RR     II      LL
                   EE            RR     RR     II      LL
                   EE E E E E    RR     RR     II      LL L L L L
                   ------------------------------------------------
                   Equational Reasoning: an Interactive Laboratory




  Pre-release version  -  All rights reserved  -  Jeremy Dick  -  Nov 1984
```

```
ERIL version R1.0          Rutherford Appleton Laboratory - Informatics Division
-------------------------Simple verification example-------------------------
COMMANDS:                                h     Help (menu display)
                                         f     Finish session
a    Apply equalities                    n     anNotate console log
b    Build system                        o     reOrder equalities
c    Copy equalities                     q     Quit to PROLOG
d    Display                             r     Remove equalities
e    Escape to new shell                 s     Superpose equalities
l    Load algebra/equalities             w     Write equalities to file
m    Move equalities                     x     eXecute a command file
-----------------------------------------------------------------------------
ENTER command:                                   (h for Help)
>b
```

ERIL is a computer program for reasoning about equations. We use it here to prove two equations:-

$$0.q + x = x$$

and $(q+1).x + (r-x) = q.x + r$

To do this, we use well-known laws of addition and multiplication.

```
ERIL version R1.0          Rutherford Appleton Laboratory - Informatics Division
-------------------------Simple verification example-------------------------
SYSTEM CONFIGURATION:                                      Console Log : on/OFF
E Q U A L I T Y   S E T S        Display        Sort     Order    Equality count
Label..Name...............Type........Trace....File.....File.....Total..Current
R   Rules                 =>        On    Off    bylhs    kbord       0      0
C   Confluent Set         =>        On    On     bylhs    ____        0      0
H   To be proved          =?=       On    On     ____     ____        0      0
-------------------------------------------------------------------------------
CONFIGURATION OPTIONS:                Label   change equality set defaults
                                      n       create New equality set.
                                      r       Reset configuration
                                      t       set screen Title
                                      c       Console log on/off
                                      h       Help
                                      f       Finish configurations
-------------------------------------------------------------------------------
ENTER option:
>f
```

This information shows that ERIL has been configured to process three sets of equations:-

R the set of rules provided for describing the properties of addition and multiplication

C an additional set of rules to aid in the generation of new rule from the given ones

H the set of hypotheses to be proved.

The rules in R and C will be constantly applied to the hypothesies until enough rules have been generated to prove them.

```
ERIL version R1.0        Rutherford Appleton Laboratory - Informatics Division
-------------------------Simple verification example-----------------------
R     Rules          0
--------------------------------------------------------------------------
C     Confluent Set  0
--------------------------------------------------------------------------
H     To be proved   0
--------------------------------------------------------------------------
ENTER command:                                    (h for Help)
>l H console
ENTER To be proved :        (f to finish)
H1:
>0*q + x  =  x.
 - - - - - exprtype consulted 1816 bytes 1 sec.
 - - - - - renamed consulted 1388 bytes 0.783337 sec.
 - - - - - reduced consulted 4304 bytes 2.21667 sec.
H1:   0*q+x =?= x
H2:
>(q + 1)*x + (r - x)  =  q*x + r.
H2:   (q+1)*x+(r-x) =?= q*x+r
H3:
>f.
```

Here we type in the two equations we wish to prove.

```
ERIL version R1.0        Rutherford Appleton Laboratory - Informatics Division
-------------------------Simple verification example-----------------------
LOAD OPTIONS:                         Label   load equality set
                                      a       load Algebra
                                      h       Help
                                      f       Finish load option
--------------------------------------------------------------------------
ENTER option:
>R verax
ENTER label of set to receive new equalities:   (f to Finish)
>R
 - - - - - matched consulted 1100 bytes 0.716677 sec.
 - - - - - replaced consulted 668 bytes 0.700003 sec.
REDUCING  H1:   0+x =?= x
REDUCING  H1:   x =?= x
PROVED    H1:   x =?= x
CHOOSE function precedence:
a   - < +  gives min(X)+X1 => X1-X
b   + < -  gives X-X1 => min(X1)+X
r   Reject
>b
REDUCING  H2:   (q+1)*x+(min(x)+r) =?= q*x+r
H2:   (q+1)*x+(min(x)+r) =?= q*x+r
```

Here we are loading rules from a file. (Some of these apply to the hypotheses.)

```
ERIL version R1.0         Rutherford Appleton Laboratory - Informatics Division
--------------------------Simple verification example--------------------------
R      Rules           7
R1:    0+X  => X
R2:    1*X  => X
R3:    0*X  => 0
R4:    X-X1  => min(X1)+X
R5:    min(X)+X  => 0
R6:    X+X1+X2  => X+(X1+X2)
R7:    X*X1+X2*X1  => (X+X2)*X1
-------------------------------------------------------------------------------
C      Confluent Set  0
-------------------------------------------------------------------------------
H      To be proved    2
H1:    0*q+x  =?=  x
     { PROVED }
H2:    (q+1)*x+(r-x)  =?=  q*x+r
     { (q+1)*x+(min(x)+r)  =?=  q*x+r }
-------------------------------------------------------------------------------
ENTER command:                                   (h for Help)
>m
ENTER label of equality to move:                 (f to Finish, h for Help)
>R all
MOVING Rall:
ENTER label of destination set:                  (f to Finish)
>C
ENTER label of set to receive new equalities:    (f to Finish)
>R
```

These are the rules describing the properties of addition and multiplicati

These are the hypothesies. (One already proved!)

These commands start the generation of new rules that are consequences of the given ones.

```
ERIL version R1.0         Rutherford Appleton Laboratory - Informatics Division.
--------------------------Simple verification example--------------------------
H      To be proved    2
H1:    0*q+x  =?=  x
     { PROVED }
H2:    (q+1)*x+(r-x)  =?=  q*x+r
     { (q+1)*x+(min(x)+r)  =?=  q*x+r }
-------------------------------------------------------------------------------
PROVED     H1:    x  =?=  x

PAUSE:
ENTER Continue or Halt  (c/h) :
>c
```

The program notices straight away that the first hypothesis has been proved

```
MOVING R1:   0+X => X   TO C
C1:   0+X => X
 - - - - - superpos consulted 6260 bytes 3.06669 sec.
MOVING R2:   1*X => X   TO C
C2:   1*X => X
MOVING R3:   0*X => 0   TO C
C3:   0*X => 0
MOVING R4:   X-X1 => min(X1)+X   TO C
C4:   X-X1 => min(X1)+X
MOVING R5:   min(X)+X => 0   TO C
C5:   min(X)+X => 0
MOVING R6:   X+X1+X2 => X+(X1+X2)   TO C
C6:   X+X1+X2 => X+(X1+X2)
MOVING R9:   min(X)+(X+X1) => X1   TO C
C7:   min(X)+(X+X1) => X1
MOVING R14:  X+0 => X   TO C
C8:   X+0 => X
MOVING R21:  min(0) => 0   TO C
C9:   min(0) => 0
MOVING R20:  min(min(X))+X1 => X+X1   TO C
C10:  min(min(X))+X1 => X+X1
REDUCING  C10:  X+X1 => X+X1
MOVING R29:  min(min(X)) => X   TO C
C11:  min(min(X)) => X
MOVING R32:  X+min(X) => 0   TO C
C12:  X+min(X) => 0
MOVING R35:  X+(min(X)+X1) => X1   TO C
C13:  X+(min(X)+X1) => X1
MOVING R7:   X*X1+X2*X1 => (X+X2)*X1   TO C
C14:  X*X1+X2*X1 => (X+X2)*X1
CHOOSE function precedence:
a    + < *  gives (1+X)*X1 => X1+X*X1
b    * < +  gives X+X1*X => (1+X1)*X
r    Reject
>a
REDUCING  H2:   q*x+x+(min(x)+r) =?= q*x+r
REDUCING  H2:   q*x+r =?= q*x+r
PROVED    H2:   q*x+r =?= q*x+r
```

The processing continues ....

```
ERIL version R1.0         Rutherford Appleton Laboratory - Informatics Divisio
-------------------------Simple verification example-----------------------
H      To be proved   2
H1:    0*q+x =?= x
    { PROVED }
H2:    (q+1)*x+(r-x) =?= q*x+r
    { PROVED }
---------------------------------------------------------------------------
PROVED    H2:   q*x+r =?= q*x+r

PAUSE:
ENTER Continue or Halt  (c/h) :
>h
MOVE FINISHED
```

Both now proved.

```
ERIL version R1.0         Rutherford Appleton Laboratory - Informatics Division
-------------------------Simple verification example------------------------
R      Rules          7
R59:   (1+X)*X1 => X1+X*X1
R60:   (X+1)*X1 => X*X1+X1
R43:   X+(X1+min(X+X1)) => 0
R65:   X*X1+(X2*X1+X3) => (X+X2)*X1+X3
R17:   min(X+X1)+(X+(X1+X2)) => X2
R54:   X+(X1+(min(X+X1)+X2)) => X2
R67:   min(X*X1)+(X+X2)*X1 => X2*X1
----------------------------------------------------------------------------
C      Confluent Set  13
C9:    min(0) => 0
C1:    0+X => X
C2:    1*X => X
C3:    0*X => 0
C4:    X-X1 => min(X1)+X
C8:    X+0 => X
C11:   min(min(X)) => X
C5:    min(X)+X => 0
C12:   X+min(X) => 0
C6:    X+X1+X2 => X+(X1+X2)
C7:    min(X)+(X+X1) => X1
C13:   X+(min(X)+X1) => X1
C14:   X*X1+X2*X1 => (X+X2)*X1
----------------------------------------------------------------------------
H      To be proved   2
H1:    0*q+x =?= x
    { PROVED }
H2:    (q+1)*x+(r-x) =?= q*x+r
    { PROVED }
----------------------------------------------------------------------------
ENTER command:                                      (h for Help)
>f
CONFIRM exit from ERIL (y/n):
>
```

This is the final state of affairs.

B.3 Expansion of Alvey SE Strategy

ALVEY SOFTWARE ENGINEERING PROGRAMME

1. GOALS AND OBJECTIVES

1.1 The Central Goals

Future IT products can be expected to be more complex than those of today and thus to place greater demands upon the people building them. The IT industry must meet this challenge, even though there is a growing recognition that system development techniques are inadequate for the large systems of today, let alone those of tomorrow.

It is accepted today that the development of a substantial new computer system carries a number of significant risks and it is by no means uncommon for such systems to be delivered late, over-budget and incapable of meeting the complete requirements of the purchaser. Some systems, after considerable expenditure of human effort and money, fail to materialise at all.

Skilled programmers are a scarce resource which is not being used efficiently. The industry is fragmented by organisation, by language and by target computer. One result of the consequent lack of commonality of environment or concentration of resources is that many programmers are not provided with even the simplest programming aids, let alone sophisticated ones. The economies of scale necessary to justify their introduction have not been perceived to exist.

Despite these problems, the UK does not lag behind other countries in software engineering, except perhaps the USA. The UK is certainly regarded as the leader in Europe in this field. Efforts to improve software engineering practice are crucial if important developments in technology are not to be wasted or cast into disrepute through poor production methods. The UK must not allow other countries to overtake it, for if it does, UK research work will be exploited by other countries to the detriment of our industry.

Software engineering may be considered as having two major goals for the future:

- improved quality ie satisfying criteria such as performance, reliability, security, on-schedule delivery and meeting the needs of the user;

- improved productivity ie reducing cost, not just of the development but of the life-cycle as a whole, including maintenance and future evolution.

Current software practice is centred on the programming process, and depends strongly on the skills, experience and resources of individual workers. Significant problems frequently result from inadequate effort being devoted to the front end of a development, notably concept formation, requirements definition, and design. Although there have been some efforts to study these problems, as well as interesting advances in both design verification and code verification, relatively little work has been devoted to integrating all of the stages into a common framework useful in production environments. Significant improvements in software productivity will be achieved when the current practice of repreated 'reinvention of the wheel' is replaced by the widespread re-use of prefabricated components. In the future then, software practice will tend to focus more on methodology, design, and component reuse and less on individual programming skills.

System design must include not just software design, but also hardware considerations. A narrow view of software engineering as just a collection of techniques to produce efficient software is not adequate. Software engineering should be aimed at the development of high quality systems, ie reliable, secure, efficient and easy to use, in a way that integrates hardware and software-based design criteria. In the future it must become information system engineering, not just software engineering.

In the short term, the UK cannot afford grossly inefficient utilisation of its scarce skilled programming resource. Introduction of simple tools on a wide scale is an essential first step in increasing programmer productivity, and also in the educational process needed to prepare for the later exploitation of more sophisticated methodologies and tools.

1.2 Major Objective

To help achieve the general goals of improved Quality and Productivity the Software Engineering component of the Alvey Programme is focussed towards a strategic goal - that in 1989 the UK should be a world leader in Information System Factories (ISF). This goal is highly ambitious and competitive, as are the goals of the Japanese 5th Generation Project. The ISF objective implies a series of sub goals both in technology and timescale. The Alvey Software Engineering component will be judged on its ability to show that UK industry has increased both its software development productivity and software product quality as a result of striving to achieve the ISF. The strategy given in this document outlines the route towards the ISF with planned interim spin-offs so that productivity and quality gains may be achieved prior to the emergence of the ISF.

What is meant by an Information System Factory? Today, the production of most application-specific hardware/software systems - such as a banking network, a corporate management information system or production control system - does not in general make great use of development tools. In that sense it is not capital intensive. The application-specific part of the Information Technology industry is characterised as a cottage industry. It is predicted that it will not remain so for long, indeed the Japanese are already building 'software factories'. To stay competitive in producing large, reliable, application-specific systems, IT companies will have to make a large investment in some kind of production facility. Exactly the same criteria will apply to manufacturing software products. This expensive facility - part hardware, part software, part stored knowledge - is an Information System Factory.

1.3 Subgoals and Directions

Defining a concrete and ambitious strategic objective crystallises a number of more general but worthwhile aims as subgoals along the route to the main goals. Many questions about the balance and direction of the whole programme can be judged by their contribution to the main goals.

1.3.1 Use

An equally important part of the programme is to create a climate in which advanced software engineering methods are in demand. This has started with a programme of investment in and use of today's technology, with associated training, measurement and evaluation. This needs to be supported by education in 'formal methods' to prepare for the widespread introduction of specification and verification.

1.3.2 Measurement

The ISF will only succeed if it can bring radical improvements in software quality and productivity. These two concepts are notoriously difficult to pin down, and certainly it is not currently known how to measure them. The programme is facing up to the difficult task of developing metrics for quality and productivity. Subgoals must be set for achievement. Performance must be reported against these goals. Finally it must be possible to measure the impact of an Information System Factory. (See Reliability & Metrics Strategy [ref 3].)

1.3.3 Distributed Working

Immediate use must be made of Wide and Local Area Networks to link co-operating designers and programmers tackling common development and production tasks. This requires investment in the use of current network technology. Measurements must be made of the improved performance flowing from these investments.

1.3.4 Research

Substantial research tasks must be undertaken with the goal of incorporating successful outcomes into products from 1985 onwards. This requires co-operation between industry and universities. The correct balance must be struck between basic research, development, practical experimentation and importing other people's ideas. The goal of having a commercially viable Information System Factory by 1989 will provide a focus for research. It tips the balance towards practical experiments, development, and a readiness to exploit other people's ideas, rather than concentrating solely on basic research for "scheduled breakthroughs".

1.3.5 Short term exploitation

A number of subgoals must be established and met along the way to the selling and exporting of tools over the coming years. The establishment of strong sales organizations in the key markets is vital. Some companies have started already in software products. Many more must make this investment. The programme can make an extremely valuable contribution through support and direction of this investment. This is being achieved through close cooperation with other Government schemes which are more specifically aimed at product development and marketing, thereby constructing a smooth 'pipeline' from Alvey R & D through to product sales.

1.3.6 <u>Standards</u>

The programme must support the development of standards. These will vary from major international activities eg. ISO language standards, through to informal, Alvey specific, tools interfaces. Close cooperation must be established with other Government and Industry standards initiatives. It is anticipated that the increasing use of formal methods will improve the foundation and creation of standards.

1.4 <u>What Will Happen in Any Case</u>

The Alvey SE strategy is based on the prediction that the production of application-specific information systems will cease to be a cottage industry and become a capital-intensive industry.

The main reason has to do with software quality, in the widest sense of the word. Expectations of software quality, both within the industry and without, are very low. Today, programmers expect to have lots of bugs in their code, and the public expect computers to send them stupid invoices. This situation is not confined to the UK; it is worldwide. British standards of software quality are relatively high, while low in an absolute sense. This situation cannot last indefinitely. In the hardware field, one manufacturer (Tandem) has grown spectacularly by offering high reliability at a premium. This has been done against a background of hardware from IBM and others which is already highly reliable. The incentives to do the same in software, and the potential payoffs, must be much higher given the current poor quality of software. It seems highly likely that someone soon will "do a Tandem" in software, and either keep the method to himself or sell it very expensively. The Japanese are certainly trying, as are the Americans and the French. Without concerted action, the UK is bound to become an importer of this technology. If the UK is prevented from importing such technology then the industrial consequences could be very serious.

A number of other current trends are leading towards the 'capitalisation' of the software industry - the growing complexity of software systems, which demands new techniques and computer assistance to manage it, the dawning awareness of the importance of project and programming support environments, and the emergence of software packages which demand new skills to integrate them in particular applications. Finally, there is the emergence of non-Von Neumann architectures and VLSI, which are inevitably mixing the software and hardware design problems, making both more complex. All these are creating larger and more complex problems, which cannot be solved without a radically new level of automation and mechanical assistance.

2. THE CHANGING NATURE OF SYSTEM DEVELOPMENT

2.1 Summary

The expected changes that will result in the most significant increases in cost-effectiveness of software development over the next ten years are the following, listed in approximate order of expected impact.

In the short term

1. incremental changes in programmer productivity through the more widespread use of design methodologies and tools
2. the coming together of methodologies and tools for the entire development life-cycle within integrated project support environments (IPSEs)
3. growing standardisation of development methodologies as a consequence of 2.
4. further refinement of suitable high-level programming languages appropriate to the integrated development methodologies
5. growing interest in, and use of, formal specification methods and extension to animation
6. automatic software generation techniques in limited form, probably first in the area of commercial systems built around Data Dictionaries.

In the medium term

7. spread of powerful networked, personal workstations
8. consolidation of the use of formal specification methods coupled with verification and growth in use of (semi-) automatic software generation
9. development of reusable software and hardware modules, rigorously tested and formally documented
10. second generation IPSEs adapted to support activities 8 and 9 above, coupled with greater use of higher-level languages.

And in the longer term:

11. the consolidation of the developments above into Information System Factories, coupled with the use of Intelligent Knowledge Based Systems, to provide 'automatic' assisted system development from user requirements expressed in high-level terms appropriate to the application rather than the implementation.

The crucial, and inter-related, technical developments underlying those changes will be:

1. integrated system (software and hardware) development methodologies supported by programming tools, administrative procedures and management information in an integrated environment

2. formal specification, leading to 'animation' and verification

3 reusable software and hardware components

4 automatic software generation

5 measurement and quality assurance and certification

These are discussed more fully below.

2.2 <u>Integrated System Development Process</u>

One view of the system development life-cycle is the following:

REQUIREMENT SPECIFICATION		
OVERALL DESIGN		typically costs
DETAILED DESIGN		20-50%
CONSTRUCTION		total development
TESTING		budget
OPERATION		- not usually quoted
RECTIFICATION & EVOLUTIONARY DEVELOPMENT	euphemistically called maintenance	typically costs 50-80%

Many design methodologies and software tools exist and are in sporadic use today, but the state of the art leaves much to be desired. For however good some of the tools may be, there are two serious problems.

First, they do not support a development methodology or capture any data relevant to the management of the development process. Second, most tools support coding activities but fail to support the life-cycle in its entirety, or even fail to be compatible with other relevant tools. There is a need for more tools to assist with software specification, design, testing, rectification and development, as well as with management of software projects; and there is a need to integrate them into a coherent life-cycle support environment built on a database.

Recently there has been widespread recognition of these problems, with a resulting effort to develop better tools; a prime example is the growing work on the Ada Programming Support Environment (APSE), which should lead to a qualitative and quantitative improvement over today's state of the art. Viewed in the wider context of software engineering advances generally, two important short term benefits from such work should be increased programmer productivity in the technical tasks of project development and increased management awareness and control, leading to better decision making and costing. Moreover, the growth in use of integrated project support environments (IPSE) should provide the framework within which subsequent advances, such as improved specification and verification methods, can take place. This last point argues for a need for flexibility in IPSEs. They must not be closed systems incapable of accommodating improved techniques as these are developed elsewhere.

Whilst there are a number of issues still under debate, there does seem to be fairly widespread agreement on certain key characteristics that these environments will display.

First, and of crucial importance, there will be far less emphasis on the actual source text of the program than there is at present. Typical current practice focuses far too much attention on the source code representation of a program and far too little on other representations - expressions of requirements and various levels of specification. The tools which are most commonly employed are those concerned with manipulating and testing the source code representation. Yet most software projects that are 'unsuccessful' by some measure have already gone irretrievably wrong by the time that the first line of source code has been written. If there is to be real progress on the issues of effectiveness and cost then attention must be shifted from the code to requirements and design, and projects must be far more concerned with the 'higher level' representations. (Note that such a shift of attention is entirely compatible with an aproach which emphasises re-use of existing components rather than always developing everything from scratch.)

Second, the environments will support a high degree of project visibility and traceability. At any stage of a project all relevant information will be readily available and there will be a proper basis for measurement of progress and detection of problems. For any identifiable activity there will be a record, not only of the end product of that activity, but also of the decisions (both positive and negative) which were taken during that activity.

Third, the environment will support various kinds of control. Management control, access control and configuration control all play an important part in addressing software effectiveness and software costs.

Reviewing the three issues above - emphasis on 'higher level' representations, visibility and control - leads to an inevitable conclusion: any given project employing such an environment must follow a defined methodology. This is not to say that the environment offers only a single methodology, but it is necessary for any given project to employ some defined methodology, and it is necessary for the supporting environment to 'recognise' this methodology (or at least certain aspects of it). Really, it is the methodology which addresses the issues of software quality and cost. The degree to which these issues are addressed depends upon the quality of the methodology and how well it is supported.

2.3 Formal Specification

The first qualitative change that will occur in system development will be the use of formal specification techniques. It is a large leap from today's practice to automatic program generation on a large scale, to proving theoretically that systems meet their requirments, to easy re-use of system components; but in each case the first step is formal specification.

Today's functional specifications are written in English, often with a liberal sprinkling of design detail in the difficult parts.

Specifications written in natural language have the major defects that:

a. they are imprecise, ie are subject to conflicting interpretations

b. they may be logically inconsistent without the fact being apparent

c. they are apt to be incomplete

d. they cannot be used for (mechanically assisted) formal reasoning.

Use of natural language does not force the specifier to be precise at all times. In some cases he may be unaware of imprecision, which thus slips through; in other, he may decide to gain precision and by default the method chosen will probably be to take some design decisions and specify the requirement in terms of an implementation. Neither result is satisfactory.

The development of formal specification techniques should ultimately overcome these difficulties and lead to complete, precise specifications which do not contain any unnecessary design detail. Experience has already shown that efforts to translate natural langauge specifications into a logical form show up inconsistencies, ambiguities and omissions.

During the development of the complete specification, particular specifications can be 'animated' in the sense that their logical consequences can be explored. Questions such as What will happen if..? can be answered precisely, and the specification improved or modified as appropriate. In this way, purchaser and supplier can gain the clearest understanding of the system requirements. Another use would be simulation of critical aspects of the system, for example the user interface of a Command and Control System, so as to give the customer an early understanding of them.

Ultimately, formal methods can provide a very clear contractual basis for the statement of requirements and thus help to avoid disputes about whether the system meets the requirements or not.

Numbers are hard to come by, but it is probably fair to say that most computer systems have to change after only a limited period of operation because the true operational requirements are, with the benefit of hindsight, perceived to be different from those originally requested. It has been argued that that is not so much a problem as an inherent characteristic of the real world which must be catered for in the development process. Systems must be designed to be capable of evolution. A rigorous path from specification through to implementation, with all the steps recorded, is essential if newly understood requirements can be fed in again at the beginning of the process without requiring a complete rewrite of the system.

2.4 Re-usable System Components

Today, hardware is thought of in terms of components whose behaviour is well understood and which can be put together in a number of ways to build a system. In the future, software will more and more come in packages until it too can be regarded as providing a set of component parts out of which software or mixed hardware/software systems can be constructed.

A number of trends will work together to bring this about. First, packaged software will account for a higher proportion of software sales to meet the enormous need for inexpensive software for personal, home or other small computer systems. Tailor-made software will be too expensive for this market and suppliers competing to reduce production costs will find it necessary to use mass production techniques, eg standardised design techniques, specialised tools, integrated project development environments.

Second, skilled programmers are a scarce resource and will continue to be so. Development techniques which provide a path away from today's labour-intensive methods will permit levels of production control and documentation adequate to the development of truly re-usable software.

Third, design by components appears to offer the only solution to the problem already encountered today that some systems are so large and complex that their operation is hard to comprehend, their performance impossible to predict and their design impossible to optimise. Design in terms of components may permit only theoretical sub-optimisation but in practice this may be vastly superior to what could be obtained otherwise; and the ability to predict performance and cost, in advance of implementation, will be a major benefit.

2.5 Automatic Software Generation

Automatic software generation is in use now in a limited way, and is a very powerful technique for producing commercial software of the type that consists of simple, repetitive processing applied to a complex database. The development of data dictionaries in the commercial sphere, the trend to put the structure of applications into the database rather than into the programs, will encourage automatic software generation so that it can be expected to be a common technique in transaction processing within five years.

The experience thus gained, coupled with advances in specification techniques and the availability of a wide variety of software components, will subsequently enable automatic software generation to be applied to increasingly more complex processing tasks. It is here that Intelligent Knowledge Based Systems can be expected to make their greatest contribution to Software Engineering, in particular in determining and enforcing consistency of specifications and of the transitions from requirements specification to design and from design to implementation.

2.6 Measurement and Quality Assurance & Certification

Today it is difficult to predict the costs and timescale of a software development project, to measure the progress and productivity of the project team and to measure the quality of the finished product.

The widespread adoption of integrated project support environments built on database technology will facilitate new research on the quantitative aspects of software development. A significant improvement in management effectiveness, productivity and product quality will occur when respectable metrication is introduced into the software development process.

Quantitative assessment of the benefits of new tools and techniques will provide a major stimulus to the introduction of further new techniques and further research, as hard-headed senior managers will be more easily persuaded to make the necessary investment funds available when presented with reputable, quantitatively argued cases with measurable pay offs.

Metrics and formal methods are the keys to effective quality assurance. The developers of good quality assurance techniques will enjoy a significant commercial advantage. The current shaky reputation of software will mean that the ever broadening range of customers will gravitate towards products bearing something akin to the BSI kite mark for simple products and components. Customers wanting more sophisticated products will favour suppliers who can offer independent, top class quality assurance and certification as part of the legal contract.

3. STRATEGY

3.1 Summary

Consultation has shown that there is very strong agreement in industry, Government and the academic community on the technical directions that the software engineering programme should take.

The Alvey SE Programme has as its long term objective the creation of the Information Systems Factory. This is predicated on technical progress in the two crucial areas of:

1. PRODUCTIVITY
2. QUALITY

To ensure continuous benefit during the period preceeding the achievement of the ISF the SE Programme proposes a strategy which encourages intermediate levels of technology transfer by encouraging not just research but:

i. Exploitation: efforts to ensure that existing methods are effectively used and their benefits gained by industry as a whole, and continuing efforts to bring the fruits of research out into industrial use, with the associated investment and training.

ii. Integration: development of integrated methodologies and sets of tools for hardware and software development covering all phases of the system life-cycle.

iii. Innovation: research and development to extend the methodologies and techniques of software engineering.

To give a feel for the activities which will be covered by innovation, integration and exploitation figure 1 shows the system development life cycle subdivided into

1. Methods and processes - how things are developed.

2. Management - monitoring and control of methods and processes.

3. Environment - the workplace, tools and equipment with indications of where in the classification various key elements of the strategy occur. Figure 1 is a summary which is expanded in the following sections.

STRATEGY	Innovation and Understanding	Integration and Implementation	Exploitation and Evaluation
Methods and Processes	Specification V & V Reliability Quality Metrics Reusability	Blend techniques into life cycle method for both hardware and software	Measure use of IPSE
Management	Models of development and mainte-nance processes and methods	Integrate development methods with management techniques	Evaluate use of IPSE
Environment	Influence on Productivity and Quality MMI, IKBS, DCS	Build IPSEs	Make IPSE available via Centres

Figure 1

3.2 Exploitation

Today, most small to medium projects in the UK (and elsewhere) confine their use of tools to simple text editors, compilers or assemblers, linkers and simple debugging aids. Occasionally, a design technique such as Jackson Structured Programming or MASCOT will be used.

In the rare cases that a more systematic approach is felt to be necessary, and additional tools to support the approach are required, they are commonly developed ad hoc, on a project specific basis, and thrown away when the project is complete. The high cost of this approach both in terms of tool development and in re-training staff to use new tools and techniques for each project, has militated against the systematic use of tools throughout the UK software industry.

There is therefore great scope for improvements in software quality and productivity, in the short term, by encouraging the widespread acceptance and use of even simple tools, of the kinds currently in use in more restricted environments. Three things are needed to bring these improvements about:

i. provision of a set of tools in a standard, compatible form

ii. measurement of the effect (positive or negative) on quality and productivity due to the new methods, tools and training.

iii. education of both management and software staff; management must be shown that investment in tools does pay off, and software staff must be educated in the systematic development and production methods that enable the tools to be used cost-effectively.

A short term attack on these three factors will be crucial not only in bringing about the much-needed improvements in quality and productivity, but also in providing wide appreciation of the nature and benefits of software engineering and the demand for more advanced techniques.

In the longer term, continued efforts will be needed to ensure that the results of research are developed and exploited. The 'development gap' between research and production has been a problem in Britain for many years, and the software engineering programme will tackle it directly by commissioning innovative research and development projects rather than just funding research. Moreover, the main objective of the proposed 'Software Production Centre' (see section 3.6) is to make advanced tools directly available to British industry for experimental evaluation and genuine production work.

3.3 Integration

The second major need identified is for Integrated Project Support Environments (IPSE). The common understanding of an IPSE is that it should contain a compatible set of specification, design, programming, building and testing tools, supporting a development methodology that covers the entire life-cycle, together with management control tools and procedures, all using a central project database. That is already a very demanding requirment, exceeding that of the Ada APSE, but even then it does not go far enough. It does not cover multiple-language development; it does not cover mixed hardware and software development; it does not cover reusable components.

There is certainly no agreement that one particular programming language meets all foreseeable needs, though there are individual proponents of this view for different languages. There is also considerable investment in software in the languages of the 60s and 70s, which will tend to prolong their life for reasons of compatibility, cost of re-training and so on. This multiplicity of languages, coupled with a recognition of the need to move towards re-usable software, argues for multi-language IPSEs where systems can be built out of components in a variety of languages.

Similar considerations apply to mixed hardware and software systems. It is clear that there are enough similarities between the hardware and software design processes, and the administrative and management procedures appropriate to them, for there to be benefit in using one IPSE for hardware and software development. Furthermore, it is important that the requirements analysis, functional specification and much of the design work can be done independently of decisions whether particular modules should be implemented in hardware or software. For such modules, their function must be defined, their place in the overall design established and their performance requirements known; economic, timescale and other criteria may then be used to determine how they should be implemented.

A fully integrated IPSE as just described is exactly the Information System Factory that is the major objective of the programme. It is a long term objective, but it is important to be clear about what the objectives are in order to see how to move towards them, and in particular to determine the role of UNIX and Ada APSE developments in this process.

One conclusion that emerges strongly is that there is still a great deal of research and development to be done before such an integrated PSE can be built. Two important areas needing R & D are:

i. formal, rigorous methods of specification of requirements, and techniques to express designs and determine how far they meet their specifications for performance, reliability, correctness etc;

ii. methods of structuring software or hardware system components for wide re-use; the nature of their interfaces to each other, the appropriate types of global design to incorporate them; how to document them; how to search for and locate them.

The Alvey programme will include one or more evolving IPSEs, which not only bring together existing tools and procedures to improve development cost-effectiveness in the shorter term but are also capable of incorporating new techniques that emerge from relevant R & D projects.

3.3.1 ISF and the Three IPSE Generations

The Integrated Project Support Environment (IPSE) is a major product objective of the programme and a crucial mechanism for blending together the results of individual research projects. The blending process is itself a research topic. The blending might well prove to be more important that any of its constituents when judged in terms of commercial success. The programme will proceed as follows.

(1) Commission development and creation of three generations of IPSE:

```
1st)                  ) file
2nd) generation IPSE  ) database
3rd)                  ) knowledge base
```

(2) Versions of each generation of IPSE to be sited in SPC (section 3.5) and NQCC (section 3.6) and selected organisations where IPSE impact on quality and productivity can be monitored and reported.

(3) Cooperate with and incorporate aspects of other Alvey areas towards ISF eg CAD for VLSI, high resolution displays, expert systems for programmers.

3.3.2 The 1st Generation IPSE

UNIX will be used as the basis for:

(1) The 'Exploitation' Tools Propagation exercises.

(2) The 1st generation (file based) IPSE.

UNIX is rapidly becoming a de facto standard over a very wide range of systems and organisations and therefore offers the prospect that:

- There will be many developments for UNIX which can be taken advantage of by the Alvey programme.

- The market for UNIX-based development environments and tools is large and growing.

These factors should minimise the amount of tool integration and development needed to improve today's UNIX environment into a genuine 1st generation IPSE.

This is not to say that the Alvey programme is endorsing UNIX as a standard; UNIX will be used as a starting place. Nonetheless, it is envisaged that an active UNIX community will come into being in the early years of the programme, supported by communications network facilities.

3.3.3 The 2nd Generation IPSE

The second generation IPSE contains two major components not found in the 1st generation IPSE:

(1) Database-based tool set (rather than file-based) eg CADES.

(2) Support for geographically distributed project teams e.g. Newcastle Connection.

As (1) and (2) above are somewhat orthogonal it is expected that several approaches will be attempted, including the evolutionary development of the 1st generation, UNIX-based IPSE as well as the 'clean sheet', non UNIX attack, possibly via intermediate steps which will contain one, but not both, of the distribution and database components.

The 2nd generation IPSE software will run on new hardware; developments in cheaper CPU power, cheaper, high resolution colour graphics, and non keyboard input-output devices, for instance, will facilitate productivity gains due to improved man-machine interaction. The 2nd (& 3rd) generation IPSE will require new hardware based components such as:

1. Single user workstation costing £5K with A3 black and white 2K x 2K pixel graphics.

2. Colour single user workstation costing £10-20K with

   - 2K x 2K pixel A3 screen
   - 10 MIPS power CPU
   - 32K microcode store
   - 10 Mbytes physical memory
   - 32 bit arithmetic and data paths
   - 32 bit virtual address space per process
   - hardware cache, paging, floating point
   - hardware graphics support
   - sophisticated i/o devices.

3. 100 Mbits/sec local area network.

4. Gateway to high speed (greater than 1 Mbit/sec) wide area communications.

5. LAN servers for files and databases.

6. High quality, cheap print server eg. laser printer.

7. Full-generality distributed operating system.

8. Sophisticated man-machine interface.

The Programme will stimulate the UK production of hardware suitable for the 2nd Generation IPSE as described above. It is important to effect this development in a short enough timescale to prevent UK manufacturers being eclipsed by the USA and Japanese industries in this large market. Such machines will be available in 1984/5 for about £5-10K.

3.3.4 The 3rd Generation IPSE

The 3rd generation IPSE (or ISF), containing knowledge bases and 'intelligent' tools, requires significant research which must begin now if the 1989 target date for the Information System Factory is to be met.

It is envisaged that the ISF will be defined as much by market and economic realities as by any technical goals; it will (almost by definition) embody the most cost-effective ways of producing application-specific IT systems available at the time.

An Information System Factory will probably consist of six main subsystems:

1. specification and prototyping facilities

2. a Software Development Environment

3. a facility for CAD of VLSI and hardware development

4. a database or knowledge base of available software and hardware components

5. the communication systems, both local and wide area, to facilitate co-operative development

6. project management aids.

How far advanced these six subsystems are by 1989, and how closely integrated together, depends on technical advances which are hard to predict. Markets will exist for the separate components as well as the unified ISF. The following sketches are probably optimistic in their assumed rate of technical progress, but help to define the aims.

1. Specification and Prototyping Facilities

Specifications of the system under development will be held internally in a formal, machine-manipulable form (which is central to the integration of the whole ISF - since it is used by all its subsystems). There will be extensive facilities to convey these specifications to people such as system designers and the eventual users of the application system - by animating the specifications, producing small prototype systems, question-answering and so on. Completeness and consistency of the specifications will be checked automatically.

2. Software Development Environment

This will go beyond present-day environments in supporting all phases of the software lifecycle and in relating them back to the formal specifications. It will be tailored to support one of several different development methodologies - depending on the application area - and will support a defined style of project management.

3. Facility for CAD of VLSI and Hardware Development

With the emergence of special-purpose hardware architectures implemented in VLSI, the need arises for functions to migrate between software and hardware during the lifetime of an application system. So CAD of VLSI cannot be considered as a separate problem. A VLSI-implemented system must meet the same formal specifications as a software system, and pass the same tests, and vice versa. Therefore the software development and the VLSI CAD facility need to be centred around the same specification method and must communicate with one another.

4. Database of Available Components

To compete effectively in making IT application systems, it will be increasingly necessary to re-use existing software and hardware components. These components will be very diverse - a component could be a software product, an integrated circuit, a sub-routine or fragment of code, an algorithm, a man-machine interface device or one of a set of formal theories about data structures. A database of such components will hold some information common to all of them, to answer questions such as: What does it do? (i.e. its formal specification) What environment does it require? (language, storage space, power requirements, inter-connections etc) and Can it be adapted to perform a slightly different function? Some components will be general purpose and some will be application specific. Initially such information will be held in a database and searched in various ways; but in the longer term there is a need for automatic reasoning based on the data; this broadens the requirement to an intelligent knowledge based system (IKBS).

5. Communications Systems

A key feature of the ISF is the facility to allow groups of designers and programmers to work co-operatively, even when geographically distributed. This will mean a requirement for high bandwidth communication between co-operating processes, both within site, and between sites. It must be possible to implement the ISF in a distributed manner. It is expected that whilst the basic communications technologies of local and wide area networks will exist to allow this to occur, nevertheless considerable developments will be required to meet the special needs of the ISF, expecially in handling interactive high resolution colour graphics.

6. Project Management Aids

Project planning, management and control methods will be developed. When supported by a comprehensive collection of tools, these management techniques will provide both professional managers and technical staff with the ability to effectively plan and control all aspects of the software development process throughout the life cycle. These management tools must be intimately integrated into the development process to ensure that all the appropriate parameters can be realistically measured.

Thus an ISF, with all six subsystems implemented to a greater or lesser extent, will be an essential prerequisite to compete in producing medium to large scale information systems in the late nineteen eighties. It will represent a major capital investment for anyone intending to compete in the field.

The discussion so far has concentrated on the development of large, complex application systems; however, similar remarks apply equally to the small systems market. To remain competitive in producing IT products, companies will have to use advanced specification and prototyping tools, application development aids and libraries of components to produce better systems faster. So analogous small-scale Information System Factories may well dominate the small systems field, although market forces will drive them more to a low cost, high volume règime. The greater dynamism and adaptabiliy of this sector means that new approaches are always rapidly emerging and can be rapidly tested in that market. Therefore the Alvey programme will by no means ignore the small systems market; producing and supporting small scale Information System Factories will be an important activity in its own right, as well as a testbed for ideas to be used in the large-scale systems market.

3.3.5 Concluding Remarks on Integration

Thus the strategy for producing the three generations of IPSE requires a controlled set of concurrent and overlapping research and development activities. It is important that the 1st and 2nd generation IPSEs are produced, not just the 3rd generation ISF, because major gains are expected in software productivity and quality from their UK installation and exploitation as well as export sales.

3.4 Innovation

The Director (SE) will initiate a programme of research to ensure that the scenario in section 2 (The Changing Nature of Software Development) is realised in the UK. This will require a balanced programme of directed contract work and responsive funding. The universities will play a significant part in this work and the SE programme expects to work closely with other funding bodies and initiatives. The Director (SE) will sometimes let competition develop between research teams as well as organising collaborative projects, some between companies and some including universities as well.

The SE research programme will overlap significantly with other areas (this is a good thing) and the Alvey Directorate will ensure intra programme coordination.

The three key points to be made about innovation are that

i whilst the general directions in which innovation is needed are known it would be premature now to try to pick winners and ignore rival approaches;

ii research projects are often on too small a scale to provide an adequate testing ground for a new technique;

iii the scale of UK research must be increased to compete with our international rivals.

Thus the programme will back a number of promising approaches to (for example) specification, and test them out on life-size projects rather than attempt to evaluate them in terms of their apparent success in small-scale use. This approach not only offers a better chance of selecting useful techniques, it also starts to bridge the 'development gap' by bringing research results out into a development environment.

The current list of research priorities includes:

i. Software Development Methods

- Formal Specification
- Verification and Validation
- Reusable Components
- Metrics
- Quality Assurance and Certification

ii. Project Management

- planning and estimating
- progress and productivity measurement
- budgeting
- standards control

iii. IPSE

- items already indicated above are relevant
- evaluation experiments to test changes in productivity and quality due to use of IPSE in the industrial context
- MMI, VLSI/CAD etc from other Alvey areas but relating to IPSE construction

The list of research priorities will be regularly reviewed and, if necessary, modified. In addition to the above list which sketches out some of the work required to achieve the programme's major goals and objectives it will also fund a small amount of longer term and/or more fundamental research to maintain a balance between targetted development and pure research. The theoretical underpinnings of software engineering are considered to be of vital importance - a thorough 'understanding' must precede the expensive construction of the sophisticated ISF-type environments.

3.5 National Quality Certification Centre

The primary medium term payback activity is seen as the creation of a National Quality Certification Centre (NQCC) for software products and components. The NQCC must build up an international reputation. This will involve the adoption of state of the art techniques on a continuous basis. The commercial benefit of NQCC approved software products in an international market is potentially extremely valuable. As the mass market for software products develops consumers will buy NQCC approved products rather than unapproved products. The rapid establishment of such a national capability could give the UK a significant commercial advantage.

The concepts behind the NQCC are currently in their infancy with only communications protocols and programming language compilers being 'certified'. The NAG library quality control reputation shows the potential benefit of extending this concept.

The NQCC should provide a realistic focus for much speculative research and development work.

The NQCC cannot spring into existence overnight. It is envisaged that early in the programme one or more R & D centres will be established to develop quality assurance and certification techniques. At least one centre's medium term aim will be to transform itself into the Alvey Quality Certification Centre. If the AQCC can establish a national reputation then the move to genuine 'national institution' status, possibly as an independent, revenue earning body, should rapidly follow.

3.6 Software Production Centre

The SE programme will establish a Software Production Centre. This will not be a research project but a working factory funded to exploit and incorporate the latest technology. The facilities of the centre will be made available to software producers for genuine production work.

This will enable large organisations to try out 'real' new techniques before making the necessary in-house investment. It will also enable small companies to experience the benefits of new technology which they could otherwise never afford.

The SPC will be aimed at producing software which will pass the tests laid down by the National Quality Certification Centre.

Technically, the SPC is to be a multi-lingual, database foundation, integrated project support environment. It will act as the focus for the practical embodiment of much research and development work.

It will support not only the development of new software but the maintenance and evolutionary development of existing products. To this end, it will be 'multi-lingual', ie it will be capable of developing systems in, say, Cobol, Fortran and Coral as well as eventually, say, Ada and Prolog. It will also be 'multi-lingual' in the sense that any one of its software products can be constructed from components coded in several different languages. Such a requirement will stimulate the development of re-usable software components and maximise the return on investment in existing software.

The technology contained within the Software Production Centre could be exported into the sites of the participating organisations by:

a. replication of hardware and software components on the site

b. network access from the site to the Centre

c. a combination of a and b.

The running of both the NQCC and the SPC will be contracted out to industry.

3.7 Development Programme

The NQCC and Software Production Centre require a research and development programme to feed them with new technology. The Director (SE) will initiate a medium term R & D programme to ensure that the goals given in section 1 are realised throughout the UK. This will require a balanced programme of directed contract work and responsive funding.

3.8 Software Components Brokerage

The Director (SE) will examine the desirability and feasibility of a centre for software components and products. It will operate by holding specifications, code etc in a database accessible only via the Alvey network. Dissemination of components will be only by FTP (file transfer). Participants will lodge their components and products in the database with distribution at a charge. This scheme should encourage collaboration, technology transfer and the creation of reusable software components and products, the idea being that it will be quicker and cheaper to get a subroutine from the brokerage than to reinvent it.

Two types of software products are envisaged as being handled by the Brokerage:

a. Packages for sale to the public. These should ideally have been approved by the NQCC.

b. Reusable software components. These too should ideally have NQCC approval but will not be on sale generally. They will be available to the 'trade', ie to those software developers who will attain increased productivity by using existing components rather than developing their own and who will contribute components of their own manufacture. The Software Production Centre should be a major source of, and customer for, these components.

With seed money from the Alvey programme to assist its launch this should become a commercially viable operation.

3.9 **Product Stimulus**

Industry must produce products. The sales of such products are one important evaluation criterion for the Alvey Programme, other government initiatives and the health of the industry. However, short term sales figures will not be the dominant factor for Alvey Programme assessment.

The programme will work collaboratively with other industry and Government initiatives, such as the Software Products Scheme, to ensure a smooth transition from Alvey-supported research into more market orientated activities. This will help to avoid the creation of an Alvey development gap. Conversely, the programme will welcome input from such initiatives which perceive market pressures having implications for the programme's strategy and priorities.