Chapter 4.2

Research and Practice: Software Design Methods and Tools

Barbara Kitchenham[1] **and Roland Carn**[2]

[1]*National Computing Centre, Oxford Road, Manchester, UK*
[2]*Reliability Consultants Limited, Fearnside, Little Park Farm Road, Segensworth, Fareham PO15 5SH, UK*

1 Introduction

This chapter will briefly review the current state of programming in commercial and engineering practice. It will attempt to summarize the work reviewed in the preceding chapters and to place it in perspective against the industrial scene. From this perspective some ways forward will be suggested.

The central message of the chapter will be that there is a need to relate cognitive mechanisms and human errors to errors committed during the software development process which are manifested as faults in the software.

We will start this chapter by reviewing current practice, the nature of programming, and the causes of human error. We will then describe some of the popular methods and tools for software design and specification, and highlight some of their advantages and disadvantages. We will conclude by discussing some of the challenges that study of experienced programmers offers to cognitive psychology, and the benefits that the software engineering community might expect from such studies.

Psychology of Programming
ISBN 0-12-350772-3

The tone of this chapter will be necessarily forward looking and speculative, in contrast to the soundly based work so far discussed. Little new work will be presented.

Most of the academic literature concentrates on the characteristics of novice programmers, and on how novices become experts. However, the characteristics of the professional programmer and of the expert programmer are of more interest to the commercial and economic community. Who, then, is a programmer and what is a program and what is programming?

Programming is practised at different levels by different people. Almost anyone can write a program and get it to work but there are dramatic individual differences between programmers. There are the novice programmers who have either never written a program or who have written only a few relatively trivial programs. Amateurs are often computer buffs for whom the technique is more important than the program or the application. Amateurs can be expert programmers and can sometimes write very high-quality systems. The casual programmer is professional in some, often technical, area but not a professional programmer. He is interested in the application; in using the computer to solve a professional problem. The class of professional programmers or software engineers is distinguished from the others in that the members of the class earn their living by building software systems. Generally, they have had formal training in programming and have a greater experience than the other groups.

However, it is not clear who the professional programmer is. In the commercial and financial industry, DP staff include programmers, analysts and designers. In the scientific and engineering industry staff are referred to as software engineers, software designers and programmers. It is recognized in these industries that there are differences in the skills, experiences and responsibilities associated with the job titles and this is reflected in the rewards attached to the job titles. The situation is further confused by the inconsistent use of the titles from organization to organization. So that a programmer in one organization simply translates detailed designs into code and tests the result for executability while in another organization the programmer is responsible for finding and implementing a computer solution to the user's problem.

The practising software engineer is responsible for the specification, design, coding, testing, implementation, documentation and maintenance of the software component of a system or equipment which contains software. The professional programmer as a practising engineer needs to understand the software development process. He will use this understanding to manage and control the errors that are committed during software production. He will then know what risks are associated with the use of his software. The users of the software take responsibility for it and are held accountable for their decisions.

Software does not exist in isolation from the hardware that it animates or from the environment with which it interacts. Software engineers work with professionals in other disciplines ranging from bankers and teachers to space scientists and military personnel.

The term software is a very broad term encompassing the executable machine code, the source code in a high-level language, the design documents, the test and management documents, and the user manuals for operating and maintaining the code. It refers, therefore, to the whole of the non-physical component of a computing system.

The definition of what is or is not a program is not always clear. In some situations, especially where the computing system is small, the term program is used to refer to the whole of the software. In other situations the term refers to a small subroutine of only a few lines of code. In practical terms a program could be defined as the smallest set of instructions that can be compiled and executed to perform a single coherent function. In a system of any size, it might be very difficult to identify such units when 'include' files are used and common service routines are shared.

The tasks required to create a program are not entirely clear. There is general agreement that there is some sort of thinking to be done to understand the user's requirement. Somehow one achieves a design and expresses it in a computer language. Finally one compiles, tests and tweeks it until the user can be persuaded to accept it, or not.

2 Current practice

Most commercial and industrial software systems are such that several different people are involved in their development. The development takes place in an organization which has socio-economic goals and structures. It is a social as well as a technical and commercial exercise. The engineering of software to meet the functional, performance and commercial goals of the organization is managed very much like other engineering activities.

The waterfall model described by Boehm (1981) is widely used as the basis for managing the development of software. This is a device that enables managers to do their management tasks. It is not based on any understanding of the process of software creation nor does it reflect the real activity of the people involved in the process (cf. Visser, 1988). As a consequence the history of software application is littered with failed, over-run and over-spent projects.

The creation of software remains in practice essentially a craft industry. Even in very large projects involving billions of pounds, and millions of lines of code where hundreds of software engineers are involved, design and programs remain the work of single individuals. Curtis (1989) has identified the 'super-designer' who is central to the success of large projects.

There have been several attempts at imposing a discipline on the process, which have been motivated by professionalism and by the need to manage the process. The structured methods have proved popular and successful partly due to the marketing zeal of their originators. Many of the structured methods are supported by graphics which have an intuitive appeal to humans whose primary modality is vision, particularly when several relationships need to be considered in parallel. Mathematical methods are currently being proposed because of their conciseness and the ability they offer of reasoning with the symbols of their arcane notation. However, it is not yet clear that any of these methods are closely related to the actual process of software creation.

In the pursuit of efficiency and effectiveness various tools have been developed and used to automate the repetitive and labour-intensive aspects of computing. Languages, compilers, environments and other specialized tools are in reasonably common use. They are valuable and are born of practical experience out of necessity. It is interesting to note that these tools are themselves software – thinking tools for thinking. More effective tools could be built if we understood the creation process and knew what tools were needed to support human weaknesses and magnify human strengths.

The often cited rate of technical change in software is all in the areas of new and more complex applications and the supporting hardware technology: faster processors, bigger, faster memories, more powerful computers. The organizational impact and the whole life cost of the software are now the most significant costs of computing. In spite of new developments, Cobol is still the most widely used language. The use of new methods presents too great a risk and too great a cost for the perceived return. In the scientific and engineering industries the situation is a little different. Here retraining is beginning to emerge as an issue. In anticipation of demand from the defense industry, some companies have invested heavily in ADA, VDM and Z.

Closer inspection of the software production process suggests that it is an engineering discipline like any other engineering discipline. It is not as mature as electrical or chemical engineering or even agriculture. Nonetheless, it is *not* an art form. Before software engineering can mature as an engineering discipline, practitioners need a better understanding of the process by which software is created in response to a demand and of the risks and errors which are associated with the process.

2.1 Nature of programming

The conventional view of the software creation process treats it as an engineering process that progresses from conception to realization through the stages of requirement definition, specification, design, construction (coding and unit testing), integration and operation. Conventional software engineering wisdom is that software is created in a top-down manner through progressive stages from the abstract requirement definition to the concrete realization of the final code. To this simple linear model is added the feedback and control necessary for management of the process. Thus each phase is verified to be in some sense a correct implementation of the previous phase, the specification is verified to meet the requirement, the design is verified to meet the specification and so on. Quality control also verifies that the further phase was reached by the agreed application of methods and procedures. Various stages of testing during the integration phase validate that the product meets the perceived need. Typically, validation and verification at each phase are associated with rework as faults and inadequacies are discovered.

Perhaps the most obvious characteristic of software creation is iteration. Iteration occurs at each level in the top-down model. It also occurs in an uncontrolled way in the verification-rework cycle and the testing-rework cycle. Our inability to predict and control these iterations is a major contributor to the chequered history of software applications. We need to understand the management and engineering tasks at each phase of the software development lifecycle in terms of the psychological activities and stages by which the software is created.

In practice, a computer system and the software that it contains is required to solve someone's information-processing problem. Thus a program is intended to achieve a specific goal, such as providing information for a decision or controlling some other mechanism, by manipulating data. The program is the set of instructions addressed to the computing engine, which is a perfect idiot, for performing the desired manipulations. Declarative descriptions of the data structures and the rules for applying the necessary manipulations are subservient to the procedural structure. The procedural structure is the product of decomposing the program goal into subgoals at progressively more detailed levels until the means to achieve the detailed goal can be expressed in a language that, after translation (compilation), can be interpreted by the computer. Until the advent of advanced architecture machines, multiprocessor systems and concurrent processing, the underlying computing engine was always assumed to be a Turing machine with a von Neuman architecture.

There are several problems associated with this view of programming. It fails to account for the order of magnitude differences in productivity between individual programmers. There is evidence, discussed by Visser and Hoc in Chapter 3.3, that the actual activity of programmers does not conform to the top-down model. There is no indication of how to control iterations. No account is given for the introduction of faults into the program. There is no way to determine whether a program is complete or correct because completeness and correctness are operationally undefined. We can only inspect designs and code to determine completeness and correctness. Although low-level errors can be detected by compilers and by executing the code serious errors of commission or omission in requirement, specification or design can often remain undetected and can be very expensive to correct. There is no way to determine which problems are hard and which are easy because we do not understand the mechanisms of program creation and cannot classify problems in terms of their solutions. For similar reasons there is no way to determine which design or program code is better than another. No account is given of the group dynamic and social factors that influence programming, discussed earlier by Curtis and Walz, in Chapter 4.1.

2.2 Cognitive models and software production

In order to create an effective piece of software a series of conceptual or cognitive models have to be created, co-ordinated and communicated to and among the members of the development team (and their managers). The nature, quality and integration of these models is critical to the eventual success of the software system.

The cognitive models illustrate and arise from the varying viewpoints that users and developers have of a software system:

The users' conceptual models of their problems/requirements determine a system's goals, and the way in which it will be used. A user's model must be communicated to and understood by the system designers/analysts. The programmers and designers need a conceptual model of the behaviour of the specific computing device or system on which the software will be executed. The programmers and designers also need a model of the virtual computing device that underlies the language in which the program is to be written.

These conceptual models can be related to the levels of abstraction known as the top-down development method and to the waterfall model of development. Users and designers create a set of requirements which a software system must provide. Designers/analysts create an information-processing model to solve the user's problem, which must be communicated to the programmer. Programmers create a textual model of the designers' information processing model, in the programming language. The textual representation of the system is then (automatically) compiled into machine code.

Only when a designer or a programmer has assimilated appropriate conceptual models of the programming task does he or she move from the ranks of novice to the ranks of expert software engineer. Thus, acquiring appropriate conceptual models underlies the conversion from novice to expert.

2.3 Cognitive models and human error

Human performance and human error result from the same cognitive activities. These activities operate at three levels:

(1) exercising a skill, where routine actions are performed effortlessly;

(2) organizing a collection of ideas and perceptions around a kernel concept, arrived at, perhaps, by analogy;

(3) solving problems, by setting goals and organizing activities into plans to achieve those goals.

Errors that occur when using a skill give rise to 'slips' when perceptual or motor patterns interfere with each other. Errors that occur at the insight level, give rise to incorrect identification of objectives and of key elements in the problem solution, and invalid assessments of the relationships among those elements. Errors at the problem-solving level result in incorrect goal structures and inappropriate or invalid plans. Errors relating to the last two levels are failures of knowledge and/or logic, and give rise to 'mistakes'.

Cognitive activities of each kind are needed to specify, design and program computers, and are the origin of the faults that are introduced into software systems during their production.

From this point of view a good development method is one which facilitates performance while minimizing errors. Good methods facilitate:

(1) problem understanding, specification, and elaboration;

(2) development of solutions (plans) that satisfy the problem requirements (goals) and satisfy complementary goals relating to different view points;

(3) controlled search of the solution space;

(4) recognition and control of incompleteness and inconsistency;

(5) movement among levels of abstraction/detail without loss of important information.

3 Software design

A software design is a plan for converting a specification into executable code, i.e. the means by which a problem description is turned into a problem solution.

Software design involves:

(1) Structural design which determines the components of software systems in terms of their required functionality and their links to one another.

(2) Algorithmic design which determines how each structural component delivers its required functionality. It is algorithmic design that is usually equated with programming in the way it has been described by Pair in Chapter 1.1.

Structural design may involve many levels of system decomposition before components are ready for algorithmic design (e.g. system-subsystem-module-procedure). It is an understanding of the nature of structural design that distinguishes an experienced program designer from a novice. (This is reflected in programmer's grading structures, where the ability to take over structural design responsibility distinguishes the system designer from the system programmer.)

All software systems are information-processing systems, they accept data as input, manipulate that data (often within the context of other previously input data), and provide data as output. From a technical viewpoint, design must, therefore, address three issues:

(1) definition of the data to be held by the system;

(2) definition of the process by which inputs are manipulated (i.e. the order in which system components are invoked);

(3) definition of the states that the system can assume and what transformations are permitted between states.

It is often the case that one issue dominates the other for a particular application (problem type). Data definition dominates many commercial systems. Process and/or state definition dominates many scientific, operating and embedded systems.

3.1 Requirements of design methods and tools

A design method needs to provide:

(1) a method and/or heuristic guidelines indicating how to create a design;

(2) a method for recording the design unambiguously (i.e. a notation);

(3) a method of, or procedures for, verifying that a design is correct (verification involves ensuring that the design fulfils the terms of the specification, is internally consistent and does not prevent or hinder further system development, i.e. coding and testing.)

Design tools support design methods, but more importantly support design as part of a software development process. Tools, therefore, have additional requirements to:

(1) support design configuration and change control;

(2) interface with specification, coding, and testing tools;

(3) make the design and the design process visible both for management and quality control.

3.2 Major design methods

Today, existing design methods split into two main types:

(1) mathematical methods (often referred to as 'formal methods');

(2) structured methods.

The mathematical methods are aimed at specification activities. They are intended for use in determining the system specification and component specification. The issue of structural design is not usually addressed explicitly.

They provide a mathematical specification notation and use the classic methods of mathematical proof to verify that a program is consistent with its specification. The process of design becomes that of 'reification'. This is a process of refining data structures and their associated operations from the abstract forms used in the specification to forms capable of being expressed in programming languages. For example, a specification might define a data item as a sequence but a programming language might only handle arrays. Reification ensures that the transformation of the data item from sequence to array is done in a way that can be proved to be correct.

In terms of the three requirements of a design method discussed above, the mathematical methods perform well with respect to providing an unambiguous notation and the possibility of verification. However, they do not offer many rules or heuristics for the *creation* of a specification.

There are many design approaches which come under the general heading 'structured'. They include:

(1) Entity-relationship modelling which considers the objects about which the system collects and maintains data. This approach is used for database design (Veryard, 1984), and is now being used extensively in the design of 'object management systems' needed as part of software engineering environments. [NB. An object management system is one that handles a variety of different object types, e.g. 'files', 'tools', etc., as well as the 'records' which a database system would handle.]

(2) Data-flow analysis and design, which at the analysis level, models the flow of data through a system and at the design level specifies the system structure in terms of the calling relationships among modules. Data-flow analysis is used to identify the functional and data components in a system and the order in which the functional components are invoked (DeMarco, 1978). Structured design methods associated with data-flow analysis refine the functional components into modules and define the calling relationships between modules (Yourden and Constantine, 1978).

(3) Data structure methods, which are based on analysing the structure of the input and output data. They are usually used to design individual functional components (Jackson, 1975), but have been extended to cover systems analysis (Jackson, 1983).

(4) Object-oriented design, which is based on identifying the objects of interest in a system and treating those objects as abstract data types with associated operations to interrogate or alter their state (Pressman, 1987). This approach is very closely related to the entity-relationship modelling approach.

Unlike the mathematical methods, the structural methods are particularly useful for the specification of system structure.

In terms of the criteria for design methods, the structural methods vary, but they are all weak on the issue of verification methods. The structured design approach of Yourden and Constantine (1978) offers a number of heuristics with which to assess a particular design based on the principles of coupling and cohesion (strength). The coupling principle advises designers to minimize the links among modules by restricting them, whenever possible, to direct calling links, and minimizing the use of common data structures. The cohesion principle suggests that modules should, whenever possible, perform a single, well-defined function.

All the methods, apart from object-oriented design, offer graphical notations that are intended to assist in the early process of design creation, but which are ambiguous and incomplete as representations of a design. The JSP notation for algorithm design (Jackson, 1975) is the least ambiguous and is supported by a design notation ('schematic logic') formal enough to drive automatic code generators.

In addition, the JSP method offers a very well-defined procedure for design creation and refinement. Unlike most design approaches JSP is teachable and repeatable – most programmers who use the method on the same problem will produce similar programs. This is not the case for any other methods, including the mathematical methods.

It is interesting to note that the strengths of the structural methods complement the weaknesses of the formal methods and vice versa. It is, therefore, surprising that there is no effort among software engineers to integrate the two approaches. This may be because historically (with the exception of Mascot) the structured methods have been developed by the commercial data-processing community and the formal methods have been developed by the scientific and real-time communities. It may also be a result of an emphasis by software tool builders on solving specific technical problems (e.g. imprecise specifications, or unreadable system descriptions) rather than investigating the underlying causes of the technical problems.

3.3 Tool support for the major design methods

Tools to support structured methods include:

(1) data dictionaries which identify each named item in a system;

(2) graphics systems;

(3) object-oriented languages with complex support environments, such as Smalltalk (Kay and Goldberg, 1976).

Tools to support the mathematical models include structured editors which ensure that specifications are syntactically correct, specification animators, specification refinement aids and programs that check (or even generate) proofs (Lindsay, 1988).

However, most of these tools are experimental and, with the exception of syntax checkers and animators, are not widely available to practitioners.

Work being undertaken to base tool design for formal methods support on detailed task analysis is a particularly interesting development (Masterson *et al.*, 1988a,b). Masterson and his colleagues adopted the knowledge acquisition techniques used in the development of knowledge-based systems (KBS) and the ideas of user-centred design (Monk, 1985) to gain an understanding of how an expert creates a specification.

In common with most software development activities, writing a specification in a formal language involves an iterative cycle of exploring the problem space and then representing it in the language. Masterson *et al.* investigated the way experts explored the initially ill-defined problem space, and the way the experts refined the problem space by their attempts to express it clearly. They used the information to guide the design of tools to assist formal specification, and have developed tools to support VDM (Jones, 1986), and Z and are in the process of designing tools to support Lotos.

They describe the process of knowledge acquisition related to creating Lotos specifications in some detail (Masterson *et al.*, 1988a). Lotos (ISO/DIS8807, 1987) is a language that allows the specification of concurrency by use of a process algebra derived from CCS (Milner, 1980) and CSP (Hoare, 1985).

The interviews took the form of asking Lotos experts:

(1) about their experience with the language, in particular the way in which they structured the task of writing a specification;

(2) to look at a Lotos specification and work out what it did, providing a commentary of what they were looking at and why;

(3) what features they would like to see in tools.

The interviews indicated the relative importance of features of the specification during the creation of the specification (i.e. understand the processes before the data typing), and the ordering of activities (i.e. identify and elaborate the component processes, before assembling the top-level behavioural description; only consider data typing after the construction of most of the process algebra). In addition, the style of specification that achieved maximum clarity was identified (i.e. no more than two levels of embedding; structuring and specification around the 'normal' case, using simple synchronizations).

Masterson *et al.* conclude that tools to support Lotos should include: a means of representing the specification based on graphics and cross-referencing among gates, syntactic aids, specification reorganization aids, data-typing aids, library aids, an animator, and a temporal logic aid.

Tools that integrate all the various aspects of software development are called SEEs (software engineering environments), and have been subject to extensive research efforts in Europe and the USA during the past five years.

Such tools place stringent requirements on their designers to provide a consistent, usable interface for a variety of users (i.e. not just software engineers, but software project managers and quality managers as well). In addition, SEEs maintain permanent records of existing designs, and so could be designed to assist the use of past experience which Visser and Hoc noted as a usual expert design strategy (Chapter 3.3).

Most software engineering environments concentrate on providing support for technical processes. However, the Programmer's Apprentice is an important exception (Rich and Waters, 1988). The long-term goal of the Programmer's Apprentice is to develop a theory of how expert programmers analyse, synthesize, modify, explain, specify, verify and document programs. It is intended to apply artificial intelligence techniques to automate the programming process. The work centres on two basic principles, the assistant approach and inspection methods (cliches); and two main technical advances, the Plan calculus and a hybrid reasoning system (Cake).

The assistant approach is based on the idea of assisting programmers rather than replacing them. It suggests that each programmer should be provided with a support team in the form of an intelligent computer program that can take over routine programming tasks. It assumes a substantial amount of communication between the programmer and the assistant based on a substantial body of shared knowledge of programming techniques. The Programmer's Apprentice is viewed as an *agent* in the software development process not a tool.

Inspection methods are the ways that expert programmers construct programs from basic components (i.e. plans or cliches). Inspection methods are based on the assumption that experts construct programs by inspection rather than reasoning from first principles. In analysis by inspection, properties of a program are deduced by recognizing occurrences of cliches and referring to their properties. In synthesis by inspection, program construction is achieved by recognizing cliches in specifications and selecting an appropriate implementation cliche.

The Plan calculus is a formal representation for programming cliches. It is part of the hybrid knowledge reasoning system called Cake. Cake allows users to manipulate frames and cliches and to use logical and mathematical reasoning.

Although it is clear that plans are not sufficient to account for expert performance (see Chapter 4.2), the principles underlying the Programmers' Apprentice appear to offer many facilities that actively support expert behaviour. Petrie and Winder (1988) observed that experts develop their own pseudocode to represent partial solutions in different notations and at different levels of detail. Visser (1988) observed that experts do not follow a strict top-down approach. These aspects of expert behaviour are supported by the flexibility of representation handled by Cake, as well as the ability to store and reuse expertise using libraries of cliches.

4 Cognitive issues

The cognitive tasks involved in software design are mainly those concerned with problem solving but also include learning issues.

4.1 Problem solving

Problem-solving tasks raise a number of issues which might be best investigated from the viewpoint of cognitive psychology:

(1) Selecting the best design method for a particular software application.

(2) Identifying design notations that are an integral part of the design creation. This involves investigating the extent to which the search for a problem solution is assisted by the notations used to describe the problem and its solution. From a software engineering viewpoint, currently only the JSP techniques match a problem definition with a method of problem solution.

(3) Identifying design notations that enhance a programmer's ability to recognize a correct problem solution. This involves investigating the extent to, and ease with, which a design can be verified for completeness and correctness.

4.2 Learning

A designer must obviously learn the basic design notations and approaches, but also needs to determine the most appropriate approach for a particular problem class.

A major controversy surrounding the mathematical methods is whether they can be learnt and used by more than a minority of software engineers. In this debate, the issue of declarative as opposed to procedural approaches raised by Pair (Chapter 1.1) is also relevant.

The mathematical (and indeed the object-oriented) methods are declarative. However, the current generation of software designers in industry have been brought up with the procedural programming languages. This means that the difficulties that software engineering researchers experience in gaining acceptance for new approaches may not be blind ignorance or laziness as is often asserted, but a genuine cognitive difficulty caused by the interaction of an unfamiliar notation and an unfamiliar underlying approach to design representation.

This presents a challenge to the cognitive psychology community to extend their work on skill acquisition of novices to retraining skilled staff. The concept of representation and processing systems developed by Hoc would seem to be a fruitful approach to this problem.

Learning tasks include not only learning the basic method, but also tasks that arise from the fact that in large software developments different people are involved in different stages of the development process. Thus, a designer may need to read and understand a specification produced by another person before producing a design, and a design may, in turn, be passed on to other people for coding and testing. Thus, a notation must be not only be unambiguous, it must also be readable and understandable.

The mathematical notations usually force precision at the expense of redundancy (although the Z method provides specifications that include both a mathematical and natural language section). The structured methods use graphical notations and natural language, and so provide redundancy at the risk of ambiguity.

In addition, a software designer is often faced with the task of incorporating new functions into an existing system, which involves 'learning' a system to the extent that new functionality can be safely incorporated.

The learning task is often made more difficult by the fact of variability among experts noted by Visser and Hoc. Understanding a design developed by another designer often involves understanding a solution that you yourself would never have envisaged, given the particular problem statement. This implies that software engineers should aim to develop design methods and tools that reduce variability.

5 Discussion

A major problem for software designers is identifying the appropriate design approach for a particular application area. We need better criteria for evaluating our methods, and those criteria must include factors such as how well the method supports the cognitive activities involved in software design, both from the viewpoint of initial design creation and from the viewpoint of design readability.

Another major problem is that of the quality of software products. Software development is a human-intensive process which is therefore subject to human errors. We need to ensure that our methods and the tools that we use to support those methods support human problem-solving activities while minimizing the opportunity for 'mistakes' and 'slips'.

Therefore, when developing tools to support methods, it is necessary to consider the cognitive tasks that must be supported as well as the technical tasks. Masterson *et al.* (1988a,b) are developing intelligent tools to support formal methods, which were designed after a detailed analysis of the tasks performed by an expert. There would seem to be a wide application for this approach to tool development.

The software industry as a whole needs to overcome its retraining problems. The new techniques being developed in the universities and the research laboratories are very different from those used by software practitioners. The knowledge of novice skill acquisition that cognitive psychologists have developed must be extended to the area of expert retraining. Unless serious efforts are made to retrain existing staff, the adoption of new methods, particularly the mathematical methods, will be severely delayed.

References

Boehm, B. W. (1981). *Software Engineering Economics.* Englewood Cliffs: Prentice-Hall.

Curtis, B. W. (1989). Video Presentation. Psychology of Programming workshop, Warwick University, Warwick, UK.

DeMarco, T. (1978). *Structured Analysis and System Specification.* Englewood Cliffs: Prentice-Hall.

Hayes, I. J. (Ed.) (1987). *Specification Case Studies.* Englewood Cliffs: Prentice-Hall.

Jackson, M. A. (1975). *Principles of Program Design.* New York: Academic Press.

Jackson, M. A. (1983). *System Development.* Englewood Cliffs: Prentice-Hall.

Jones, C. B. (1986). *Systematic Software Development Using VDM.* Englewood Cliffs: Prentice-Hall.

Kay, A. and Goldberg, A. (1976). *Smalltalk 72. Instruction Manual.* Palo Alto: Xerox Research Centre.

Lindsay, P. A. (1988). A survey of mechanical support for formal reasoning. *Software Engineering Journal,* **3**, 1.

Hoare, C. A. R. (1985). *Communicating Sequential Processes.* Englewood Cliffs: Prentice-Hall.

ISO/DIS8807 (1987). Information processing systems - open systems interconnection - LOTOS - a formal description technique based on temporal ordering of observational behaviour.

Masterson, J. J, Ishaq, K. P. and Hockley, A. T. (1988a). An approach to providing support tools for Formal specification. *FORTE '88,* University of Stirling.

Masterson, J. J., Ishaq, K., Patel, S., Norris, M. T. and Orr, R. A. (1988b). Intelligent tools for formal specifications. *Proceedings in Software Engineering,* **88.**

Milner, A. J. (1980). *A Calculus of Communicating Systems.* Berlin: Springer-Verlag.

Monk, A. (Ed.) (1985). *Fundamentals of Human-Computer Interaction.* New York: Academic Press.

Petrie, M. and Winder, R. (1988). Issues governing the suitability of programming languages to programming tasks. *In* D.M. Jones and R.L. Winder (Eds), *People and Computers.* Cambridge: Cambridge University Press.

Pressman, R. S. (1987). *Software Engineering. A Practitioner's Approach,* 2nd edn. New York: McGraw-Hill.

Rich, C. and Waters, R. C. (1988). The Programmer's Apprentice: a research overview. *Computer,* **21,** 11.

Veryard, R. (1984). *Pragmatic Data Analysis.* Oxford: Blackwell Scientific.

Visser, W. (1988). Giving up a hierarchical plan in a design activity. INRIA Rapports de Recherche.

Yourden E. and Constantine, L. (1978). *Structured Design.* Yourden Press.