ISSUES, OPPORTUNITIES AND CONCEPTS IN THE TEACHING OF PROGRAMMING TO NOVICE PROGRAMMERS AT THE UNIVERSITY OF LINCOLN: THREE APPROACHES

Amanda Jane Bird

A thesis submitted in partial fulfilment of the requirements for the degree of PhD

School of Education, University of Sheffield

December 2004

For Darren, Charlotte, Megan and Amelia
and
Annalie Moreton

Summary

This thesis describes three small-scale, computer-based approaches developed and used by the author in her teaching of programming concepts to novice programmers, using Pascal as a first language, within a higher education context.

The first approach was the development of a piece of tutorial CAL, the second was the development of an on-line help system and the third the development of a pattern language. For the first two, the author created the product. For the pattern language, she designed the template. These three approaches are described and the results obtained outlined. The work also looks at the kind of research methodologies and tools available to the author and present a rationale for her choices of method and tools.

This work also briefly reviews some learning theories that could be used to underpin the design, use and evaluation of CAL. The thesis looks at a range of topics associated with the teaching of programming and the use of CAL. It looks at issues around the psychology and human aspects of learning to program, such as confirmatory bias and vision. It looks at other research efforts aimed at developing software to support inexperienced programmers, including new programming languages specifically designed to teach programming concepts and sophisticated programming support environments.

The work briefly reviews various types of CAL and their uses. It also examines some key projects in CAL development from the 1960s onwards, with particular emphasis on UK projects from the early 1970s to the late 1990s. It looks at what conclusions can be drawn from examining some of the many CAL projects in the past.

Finally, the work reviews the various strands of the author's research efforts and presents a brief overview and some initial suggestions for the teaching of programming to novice programmers.

Table of contents

Contents i
Introduction to Chapter 1 1
Studying the teaching of programming 1
The author 7
Structure of thesis 14
Chapter 1: Conclusion 16
Introduction to Chapter 2 17
Goals of the research effort 17
Educational research methodologies 20
  Action research 21
    Stages in action research 26
Educational research tools and methods 27
  Qualitative and quantitative research 28
    Little q and big Q 30
  Survey methods 33
    Semi-structured interviews 35
    Participant observation 36
    Diaries 38
    Focus groups 38
  Sampling strategies 39
  Internal and external validity 42
The value of educational research 47
The concept of reflexivity 50
Outline of the author's research 55
The concept of patterns and pattern languages 58
  Patterns, software development and teaching 61
Chapter 2: Conclusion 64
Introduction to Chapter 3 65
Learning to program versus programming to learn 65
Computing and programming curricula 66
A programming syllabus 68
Elements of structured programming 70
  Edsger Dijkstra's article on goto 71

Table of contents

The algorithmic toolkit 73

Constructs 73

Algorithms 74

Variables 75

Modularity and parameters 75

Some comments on programming 77

Which language should be the first programming language? 79

At what age should students learn to program? 86

Pascal 90

The teaching of programming 92

Human aspects of learning to program 96

The psychology of programming 96

Personality and aptitude tests 99

Personality tests 100

Aptitude tests 102

Mental models 103

Visual cognition 109

Change and inattention blindness 110

Confirmatory bias 114

Metaphor and analogy 115

Learning styles 117

The work of Howard Gardner 118

The work of Kolb 119

The work of Gregorc 119

Chapter 3: Conclusion 121

Introduction to Chapter 4 123

Issues in debugging 123

The affective aspect of debugging 127

Debugging and large scale programming 130

Approaches to teaching programming using software 131

Editors 132

Intelligent Tutoring Systems 134

Designing new programming languages 135

Table of contents

Software and program visualisation 137
Algorithm animation 141
Code layout 142
Software visualisation and teaching programming 146
Programming Environments 148
Other approaches 151
Programming by Demonstration 151
Extreme programming 153
Chapter 4: Conclusion 154
Introduction to Chapter 5 156
Roots of learning theories 156
Behaviourism 157
The work of Thorndike 157
The work of Pavlov 159
The work of Hull, Mowrer and Tolman 161
The work of B F Skinner 162
Cognitive psychology 166
The work of Vygotsky 167
The work of Bruner 169
Constructivism 171
Constructionism 175
Chapter 5: Conclusion 177
Introduction to Chapter 6 178
A note on terminology 178
Classifying types of CAL 178
Drill and practice CAL 185
Simulations, modelling and games 187
Tutorial CAL 189
Intelligent CAL 190
Microworlds 193
CAL for collaborative and on-line learning 196
Background to the tutorial CAL development 201
Gathering feedback on CAL use within the Department 213

Table of contents

The author's own CAL 218
    Other research on novice programmers 221
The author's own Pascal Help system 223
    Use of the help system 225
    The academic year 2000 - 2001 227
Overview of work up to 2001 235
Chapter 6: Conclusion 240
Introduction to Chapter 7 242
The concept of patterns 242
The author's research design 247
    The author's data gathering approaches 249
        Questionnaires 250
        Focus groups 252
        Observation 253
The institutional context for the author's research 256
Evaluating student programs – an overview 259
    Programmer profiles 259
    Software metrics 260
    Criterion referencing 261
    Automated grading of student programs 263
The author's results 269
    The initial questionnaire 269
    Focus groups 275
    Observation 276
    Exit questionnaire 277
    Unit results 279
Chapter 7: Conclusion 280
Introduction to Chapter 8 282
A note on viewpoint and voice 282
The research: an overview 282
The next stage: the help system 297
A pattern language for novice programmers 299
What of ethnography and the TLTP? 300

Table of contents

Chapter 8: Conclusion 301
Introduction to Chapter 9 302
Suggestions for the teaching of programming 302
A view of the author 302
Conclusion to Chapter 9 306
Conclusion to this work 306
Appendix A 308
Introduction to Appendix B 308
Positivism 308
Hermeneutics 309
  Phenomenology and phenomenography 310
  Ethnography and ethnomethodology 311
Appendix B: Conclusion 313
Appendix C 314
Introduction to Appendix D 315
Review of selected Pascal texts 315
Appendix D: Conclusion 321
Introduction to Appendix E 322
Experts versus novices 322
Appendix E: Conclusion 330
Introduction to Appendix F 331
The Stanford Arithmetic project 331
TICCIT and PLATO 337
Appendix F: Conclusion 341
Introduction to Appendix G 342
The NCET 1967 – 1973 342
  The work of the NCET up to 1969 342
The work of the NCET 1969 - 1973 346
The work of NDPCAL 347
MEP (Microelectronics Project) 353
CTI 358
The TLTP 360
The ILT (Institute for Learning and Teaching) 366

Table of contents

Appendix G: Conclusion 367
Appendix H 369
Appendix I 423
Appendix J 425
Appendix K 426
Appendix L 430
Introduction to Appendix M 432
Appendix M: Conclusion 446
Appendix N 447
Bibliography 458
Index 512

List of figures

Figure 1: The logon screen 205
Figure 2: The time domain 207
Figure 3: A sample CAL screen 208
Figure 4: Sample multiple choice questions screen 211
Figure 5: Resource rankings 227
Figure 6: Correct lecture notes should be given for reference 229
Figure 7: Practical work helps the learning process 230
Figure 8: In order to learn the student must work as well 230
Figure 9: Learning is more effective through practical examples 231
Figure 10: Responses to statements about learning 232
Figure 11: Resource rankings 234
Figure 12: Ages of student respondents 270
Figure 13: Some of the post 16 qualification noted by respondents 270

Volume 1

Chapter 1 Teaching and learning programming: introduction and background

Introduction to Chapter 1

This chapter outlines the impetus and context for the author's research in the teaching of programming. Firstly, it advances arguments for the study of teaching programming to novice programmers, looking at some of the issues around software development. It then describes the author's experience of learning to program and focuses on the reasons for her interest in issues around this topic. It places the author's research in an institutional and sector context, briefly outlining some key issues in these two areas. Finally, it outlines the material covered in subsequent chapters.

Studying the teaching of programming

"*There are only two commodities that will still count in the 1990s. One is oil and the other is software. And there are alternatives to oil.*" (Bond, in Norris and Rigby 1992 p 1) It is a well-rehearsed truism to say that software is a crucial part of the life of industrialised societies in the twenty-first century. To review all the areas in which software plays a central or critical role would be rather clichéd but it is obvious that software, of many different types, is economically and culturally entrenched in many societies across the globe, forming the basis of activities in finance, economics, health care, recreation and communications.

The author does not have figures on total spending on software across all sectors, either in the UK or globally but one example of a growing software market is that of spending on software licenses to support e-commerce. In 2003, US companies from small (under 100 staff) to large (over 500 staff) spent a total of 14, 544 billion dollars on those licenses, compared to 3,140 billion dollars in 1999. (http://faculty.insead.fr/adner/projects/E-procurement%202.pdf 13/11/2003) As another example, in the UK, in 2002, more than 2 billion pounds was spent on computer games, more than on renting videos or going to the cinema. (http://www4.thny.bbc.co.uk 13/11/2003)

Chapter 1 Teaching and learning programming: introduction and background

Campbell-Kelly (1995) notes

> *"The 1980s saw dramatic growth in the software industry of 20 percent a year or more, so that the annual revenues of U.S firms had grown to $10 billion by 1982 and to $25 billion by 1985..."* (Campbell-Kelly 1995 p 74)

2002 saw a downturn in software revenues worldwide but the top three software companies, Microsoft, IBM and Oracle, had revenues of 25.9, 13.1 and 6.9 billion dollars respectively. The global software market declined by 5% in 2002 but was still worth $152 billion. (http://www.ovum.com/go/content/c,36416 19/11/2003)

Given that individual programmers and teams of programmers must write all the software developed and sold, the question is then, how good are those programmers? Programmers must surely be as fallible as any other human being and as variable. There must be good programmers and indifferent, or even bad, programmers. For every programmer passionate about the creation of software there must be several, or many, for whom it is just a job or a route to financially rewarding stock options.

Finding and training skilled programmers is not a trivial consideration for any company that needs to develop software. The arguments about the software crisis of the 1970s are generally familiar. As computing power became more widely available in the 1960 and 1970s, software could not be developed quickly enough to meet the demands of increasingly computerised businesses.

> *"Larger and more complex new software programs had to be developed for an increasing number of tasks. Literally, the "art of computer programming" became something in high demand and a whole new profession of computer programmers had to be created.* [] *Software programming at that time was widely considered an extremely complex task. The resulting low-quality products were exceedingly expensive, difficult to handle and hard to maintain."* (http://www.eduard-rhein-stiftung.de/html/T2002_e.html 20/1/2003)

The development of structured programming techniques and the idea of software engineering were at least partly responses to the perceived problems of developing software of appropriate quality within reasonable time scales and budgets. (www.cse.unsw.edu.au/~cs3111/Lectures/ 2003S2SA/newIntro.pdf 20/11/2003)
Any review or discussion of software development methods will almost certainly describe at least some of notable failures of software production: faulty software that

has life-threatening potential such as Therac-25 (http://courses.cs.vt.edu/~cs3604/lib/Therac_25/Therac_1.html 07/01/2004) or fly by wire systems (http://seeri.etsu.edu/SECodeCases/ethicsC/DeathByWire.htm 07/01/2004). (For discussion of software failures see also Norris and Rigby 1992 pp 1 – 9)

There are also software failures directly attributable to elementary human mistakes, such as the crash landing of the Mars Climate Orbiter, popularly supposed to be due to the mixing of metric and imperial measurement by two teams working on the software. A more considered view of the failure suggests that a lack of adherence to NASA software development and project procedures was the root cause of the problem. (http://www.space.com/news/mco_report-b_991110.html 07/01/2004)

It is not just that software projects deliver faulty or poor quality software – in some cases the projects overrun on time scales and budgets by staggering amounts. Almost any UK Government information technology project seems to end up costing much more than originally estimated and may never actually do the work it was originally required to do, or at best, do it very badly. A report, published in January 2000 by the Public Accounts Committee, drew

> "*lessons from more than 25 cases from the 1990s where the implementation of IT systems has resulted in delay, confusion and inconvenience to the citizen and, in many cases, poor value for money to the taxpayer.*" (http://www.parliament.the-stationery-office.co.uk/pa/cm199900/cmselect/cmpubacc/65/6503.htm 07/01/2004)

The responses to the problem of building good quality, reliable software that can be delivered on time and to budget have been myriad. Hall et al. (1998) note,

> "*The history of software development is, in large part, a history of salvationary devices that have come full of a high promise of deliverance and redemption, and then gone again. At various times, salvation has been sought through structured programming, relational databases, logic programming, knowledge based systems, CASE technology, formal methods, object oriented technology, ... These salvationary devices have been the subject of grand passions and have attracted financial support from national and transnational structures.*" (http://mcs.open.ac.uk/mcs-tech-reports/96-07.pdf 07/01/2004)

Chapter 1 Teaching and learning programming: introduction and background

Pressman (1992) describes the early days of software development: "*Most software was developed and ultimately used by the same person. You wrote it, you got it running, and if it failed, you fixed it.*" (Pressman 1992 p 5) This largely undocumented, personal style of software development obviously could not continue in the face of rising demand for increasingly sophisticated software products. (Pressman 1992 pp 5 – 6)

In the late 1960s the move towards the idea of software engineering began:

> "*...the difficulties of building big software loomed so large that in the autumn of 1968 the NATO Science Committee convened some 50 top programmers, computer scientists and captains of industry to plot a course out of what had come to be known as the software crisis. Although the experts could not contrive a road map to guide the industry toward firmer ground, they did coin a name for that distant goal: software engineering* []. *A quarter of a century later software engineering remains a term of aspiration. The vast majority of computer code is still handcrafted from raw programming languages by artisans using techniques they neither measure nor are able to repeat consistently.*" (Gibbs 1994 p 86)

Software engineering is a broad term that encompasses various approaches, from structured programming, documentation requirements and software life cycle models to review strategies, such as structured walkthroughs of code. There is not enough room here to discuss these elements individually. Pressman (1992) defines software engineering as encompassing

> "*a set of three key elements – methods, tools, and procedures – that enable the manager to control the process of software development and provide the practitioner with a foundation for building high-quality software in a productive manner.*" (Pressman 1992 p 24)

Despite developments in programming practice, in particular the emergence of the view of coding as one element of the discipline of software engineering, as opposed to the craft of programming, the issues around having a sufficiently skilled workforce who can produce reliable software fit for purpose have not gone away.

In a sense, the software crisis is a permanent one but not a crippling one. "*We used to talk of a software crisis. It's been going on so long now that we must now speak of a software malaise.*" (http://www.jdl.co.uk/training/4-siders/softwEng_4s.html 20/112003)

Chapter 1 Teaching and learning programming: introduction and background

If there are still problems with the development of large-scale software systems (as there obviously are), then one response has been to focus on quality issues, to look for the application of TQM (Total Quality Management) techniques, first used in manufacturing, to the software production process and to emphasise the use of SQA (Software Quality Assurance) methods and tools. There is insufficient room here to review, even briefly, the large body of theoretical work done on quality issues within software development. Galin's (2004) book on quality, *Software Quality Assurance*, is a comprehensive overview of the key issues, as well as giving an historical perspective. Given that TQM is well established in the manufacturing industry, why has its use in software development *not* led to the same gains as its use in, say, the car industry or the manufacture of consumer electronics?

Writers conclude that producing software is fundamentally different from building cars or manufacturing chairs. Galin (2004) notes that "*No developer will declare that its software is free of defects...This refusal actually reflects the essential elemental differences between software and other industrial products...*" (Galin 2004 p 4) Galin (2004) summarises those essential differences as being product complexity, product visibility and the product development process. (Galin 2004 pp 4 – 5)

Software of any magnitude presents a challenge in ensuring all the possible paths (which may run to several million combinations) through the software work appropriately, as well as each individual line of code. Software is also a product where faults may lie hidden or dormant for years, before a unique set of conditions triggers a fault that generates a software failure. Detecting problems with the software is limited largely to the product development phase or the software lifecycle. Once the software is written, making copies of it presents a very limited opportunity for correcting problems. (Galin 2004 p 5)

The software development industry therefore requires specialist QA (Quality Assurance) methodologies and standards, such as ISO9000-3. (Galin 2004 p 6) Even given the development of such methodologies, the writing of software continues to be a problematic human activity. As noted earlier, any brief review of non-trivial software projects will reveal a catalogue of missed deadlines, budget problems, poor quality product and, in many cases, no usable product at all. (http://www.scit.wlv.ac.uk/~cm1995/cbr/library.html 26/11/2003)

Given this background, it seems obvious that one strategy is to focus on *how* programmers are taught to be programmers. All the QA in the world cannot militate against a programmer with a flawed understanding of the basic concepts of structured programming. "*The education of software engineers has been a matter of substantial debate and discussion in the UK for some time.*" (McGettrick 1998 p 1)

Pressman (1992) notes "*Each individual approaches the task of "writing programs" with experience derived from past efforts.*" (Pressman 1992 p 19) With this in mind, the author, when she began to teach programming, imagined that there would be large body of work examining the particular difficulties of teaching novice programmers.

However, the research available seemed to focus on the minutiae of programming constructs (Green 1990 b) or the development of programming languages or environments. (Barr et al. 1976, Merriënboer et al. 1992, Mukherjea and Stasko 1994, Martin 1996, Hauswirth et al. 1998, McIver 2000) Other projects looked at activities such as the automated assessment of students' code (Benford et al. 1993)

In contrast, there is a wealth of material on how people learn, from the very earliest theories about teaching and learning to the work of Bruner, Vygotsky and Gardner, to name but a few. Theories about learning and teaching abound, from those centred in schools of psychological thought such as behaviourism to those that are learner-centred, such as Kolb's active learning cycle.

In the author's view, there seems to be a clear divide between those who research issues around programming and those who research learning in general. Those who undertake research in the learning of programming generally seem to focus on pragmatic, technology-based approaches, such as writing new programming languages (GRAIL, Euphoria and Turing), new programming environments or software development tools (MPE, BIP, SDE, DynaLab, BALSA, KEOPS, APSE and DISCOVER).

Those who are interested in learning may examine the teaching of broad programming constructs, but focus on the power of learning to program as a way of fostering thinking or problem solving skills: the obvious example here is the work of Seymour Papert with LOGO. (Papert 1980) What seems to be lacking, in the author's view, is an examination of the issues in teaching programming that focuses on the learning of programming as a prelude to professional work in that area and that also acknowledges at least some of the wide range of theories and ideas about how people learn.

In this thesis, the author attempts to articulate some of the key issues around the teaching of programming to novice programmers, to review potentially fruitful theories about learning and clarify how a particular theory might underpin an approach to the teaching of programming. The author is aware that this is ambitious but feels her work and background enable her to attempt to create an approach that bridges the divide described above, between technology-centred research in programming and the rich variety of theories about learning. The next section describes the author's academic background and the initial impetus for her interest in researching of programming.

The author

The author's first degree is in English literature. After completing a PGCE in English and drama, she taught English in a secondary school for five years. In the academic year 1989 – 1988 it became obvious that progression and promotion to what was then called Scale 2 would be difficult or even impossible during to the shortage of funding for Scale 2 posts. The author made the decision in early 1988 to undertake a conversion MSc in computing, at what was then Sheffield City Polytechnic.

A large element of the introductory material, in the first four weeks of the course, was programming in Pascal. It was a mandatory requirement and the assignment had to be completed successfully. The author found the learning of programming concepts a frustrating and difficult process and the memories of these early attempts at writing pseudo-code have remained vivid. It seemed to the author that many of the concepts embodied in programming were doomed to remain at least partially incomprehensible to her. Concepts such as parameter passing remained opaque and difficult for years, as she freely admits.

The author did pass the Pascal assignment and went on to complete the MSc. The course required work in a range of programming languages including Occam and Prolog. She wrote a range of teaching materials for Prolog and submitted a dissertation on the teaching of Prolog to novice programmers in logic programming. The author traces her interest in the teaching of programming from her own, sometimes miserable, often frustrating, experiences as a novice programmer.

After graduating the author worked in industry for two years, as a technical writer for large-scale software projects. In January 1992, she started work as a lecturer at Humberside Polytechnic, as it was then called, in the Department of Engineering. Because of her background in education and technical authoring she was chosen as second supervisor for a student project that looked at choosing a CAL (Computer Aided Learning) authoring language and developing initial CAL materials for the teaching of network concepts (such as networks in aviation e.g. CAN – Controlled Area Networks).

This project, undertaken in the academic year 1992 – 1993, proved to be the starting point for significant work within the Department. The Department itself underwent several large-scale changes, including moving to a different campus, a new head of department and several re-organisations but the CAL development work continued to flourish.

It is useful to outline the broader background to the CAL effort within the Department. The changes in higher education during the 1980s and 1990s are well summarised by Maier et al. (1998) in the first chapter of *Using Technology in*

Teaching and Learning. Student numbers have more than doubled since the early 1980s and that expansion has not been matched by increases in staff or teaching support resources. Also, the student profile has, for many universities, changed noticeably, although some universities still place emphasis on the traditional points requirement at A Level for potential students. More students are looking to take up part-time study and even full-time students need to work to supplement their income.

There are also subtle and not-so-subtle changes in student expectations of higher education:

> "*Increasing questioning of students of 'what they are getting for their money'- raising the importance of taking on board student feedback about courses and addressing student concerns*." (Maier et al. 1998 p 5)

Students are also concerned about employability and many now look to higher education for the acquisition of skills that will make them employable upon graduation, rather than relying solely on the status of a degree, any degree.

These pressures are not going to go away: indeed, with the UK Higher Education Funding Councils' drive for quality assessment within university departments, these pressures are likely to be increasingly formalised and measured. Part of the pressure on academic staff as well as the answer to the problems of increased student numbers and widely differing academic profiles among students is the development of computer technology. Students may come to a course already computer-literate and have no difficulty using the Internet, on-line CD-ROMs and electronic mail or conferencing as part of their study pattern. The issue of the academic use of these resources is a broad one and beyond the remit of this chapter but there can be few academic disciplines that have not felt the impact of this technology in one form or another.

The Department chose to focus its effort on CAL, developing packages of several hundred screens each to deliver material on a wide range of subjects. The final list included Propagation, Satellites, Radar, Engineering Materials, Health and Safety, Introduction to Programming Ideas, and Resistance: a small selection of the final total of 48 packages, each one consisting of several hundred screens of material.

Chapter 1 Teaching and learning programming: introduction and background

The then Dean of the Faculty of Engineering and IT (Information Technology), as it became, Roland Stokes, had a personal and organisational stake in the work. The initial student project was his idea and the management of the subsequent teams was his particular and personal brief. The CAL developed was, at first, for use with the Faculty, but a company, Feedback Holdings PLC expressed interest in selling the packages through their well-established sales network. By late 1997, the CAL package styles and documentation standards and program templates were well established. More than twenty packages were delivered and work continued on developing, maintaining and upgrading CAL packages.

The project team grew from students undertaking final year dissertation work to initially two placement students, then later, four and then, six students on one year placements from Nottingham and Sheffield Hallam universities. A full time team member was employed for graphics creation in 1996. At the end of 1998, the project was moved out of faculty control, to become a private commercial company. By then, the team had grown to 11 members, with a mix of full time staff and placement students.

Throughout the project, the author was involved in the creation of the packages, undertaking several roles. As a lecturer, she provided material and ideas for several packages including number bases, and introduction to computers: these topics reflected her general teaching requirements. A key part of her work was the teaching of introductory programming, mainly in Pascal, at year one or foundation level. (The foundation year was for students without A-level points and provided a general entry route to the computing degrees offered by the Faculty, which included by then both engineering and computing). Here two areas of interest to the author could be joined: the teaching of introductory programming and the development of CAL.

Another of the author's roles with regard to CAL was as editor of the storyboards. On this capacity, she saw all of the material being turned into CAL and found that she had some concerns about the ad-hoc style of development and the lack of any stringent pedagogical theory for the CAL structure, style and content. Some basic principles of

HCI (Human Computer Interaction) had been applied, such as consistency of button design and placement but an educationally sound rationale for the design and use of the CAL material was wholly absent.

The CAL development project was, towards the end of its life, a well-funded, commercial enterprise, with full-time team members, as well as placement students and lecturer support. However, at the start of the project, the development efforts were small scale and easily comprehensible by one person.

The lack of a theoretical basis for the CAL being developed concerned the author sufficiently for her to begin an investigation of the nature and history of CAL. She felt that some theoretical perspective should form part of the work of the project. Her research convinced her that a purely pragmatic approach to CAL development was not appropriate and that further work needed to be done on constructing a cogent and theory-based argument around the design, development and use of CAL.

The author felt that an introduction to programming concepts was a challenging topic for CAL development and she worked with the team on two versions of the CAL, over about two and a half years. This material proved as difficult to convey coherently and meaningfully in CAL as it did in lectures or seminars. After trialling the original and later versions, the author felt that this avenue had been thoroughly explored and that the results were disappointing. The author came to feel that programming is a deeply constructivist activity and requires students to build their own knowledge is a reflective way. The author concluded CAL was not the whole answer, at least not in the format available to the team: a standard tutorial page-turning CAL, with end of section questions.

The author's work with CAL is described later in this thesis, in Chapter 6. Her experiences with simple tutorial style CAL have led her to reject that approach and to seek, over a number of years, other ways to foster understanding of programming concepts in novice programmers. The author's work on developing on-line help is described and finally, her initial work on the fostering of a pattern community among novice programmers.

The author's early exploration of CAL as a way of supporting the teaching of programming led the author to question how others had approached this topic, without the use of CAL. As noted earlier, her initial thought was that there must be a wealth of material on how to teach topics such as the algorithmic toolkit, parameter passing and modularity. She was surprised to discover that, on the whole the teaching of programming seemed to lack any clearly articulated, genuinely pedagogical foundation. The author's initial work on types and uses of CAL (drill and practice, simulation and modelling and so on) led her to explore theories of learning. If one reads about drill and practice for instance, as in the Stanford project of the 1960s, one will inevitably meet references to behaviourism. From there, the vast range of theories and ideas about how people learn requires at least some exploration.

The author then thought that perhaps research was being conducted in ways to teach programming more effectively (however one defines the adverb 'effectively'). The author found that most research focused on the technical tools available to novice programmers such as languages, compilers, and environments, as described earlier. There was very little pedagogical material available.

The author found that researchers in programming languages concentrated, not surprisingly, on demonstrating the superiority of the programming language they had devised. Researchers in the developments of programming environments focused on the success of their product in a similar way. Few seem to acknowledge any pedagogical element to their work. That lack of theoretical underpinning appeared to be a notable omission to the author.

Where the human aspects of learning to program did seem to be considered was in work around which paradigms, languages or features of languages were easiest to learn. Even here, though, suggested teaching strategies based on ideas of the ways in humans learn, seemed to be missing. Finally, texts and websites that appear to aim to teach a language or introductory concepts seem to contain mainly example-based materials on 'how to' in a particular language or discursive material on programming concepts. The design of the materials on these topics does not, in the author's view, either acknowledge or yield a coherent educational model for the delivery of such material.

It seems to the author as if those who research in the topics of introductory programming focus on one aspect of the topic and rarely (or never) acknowledge all the other elements that must form part of the activity of learning to program or teaching others to program. As noted earlier, this thesis aims to go some way to articulating these sharp divisions. The author has made a conscious decision to range widely over a number of areas that, to the author, seem relevant to the central question: how can we teach programming to novice programmers more effectively?

The questions of interest to the author can be articulated as follows:

- What are the key issues in learning to program?
- How significant is the choice of first programming language?
- Is there a meaningful distinction between learning a programming language and learning programming skills?
- What theories of teaching and learning can be most usefully employed to support the teaching of novice programmers?
- What theories drawn from various areas of psychology are of relevance to the teaching of programming?
- What aspects of CAL can be used to support the learning and teaching of a programming language?
- How can the author's work on the teaching of programming be embedded in an appropriate academic research framework?
- Having used a particular approach to the teaching of programming, what ways are there of gathering and collating meaningful evidence about that approach?
- Having gathered the evidence, what conclusions can be drawn from the author's work?

These questions do not have a specific ordering – all are relevant and all come first, in a sense! For clarity, each chapter's description has noted in following brackets the question or questions above that the chapter addresses. There is some overlap, with some questions being referred to in more than one chapter.

Structure of thesis

This chapter, Chapter 1, offers the context of the author's research and clarifies why she has chosen this area of study.

Chapter 2 outlines previous approaches to the teaching of programming explored by the author, including the writing of a CAL package, as noted above. It describes briefly her work on on-line help. It then moves on to examine in more depth the author's work on beginning to foster a pattern community among novice programmers: an approach that the author feels begins to address some of the pedagogical issues around the teaching of programming, rather than simply using technology for its own sake. [What aspects of CAL can be used to support the learning and teaching of a programming language?]

Chapter 2 also looks at the research methodologies that seem, to the author, appropriate for this work. It touches on the difference between the scientific method and educational research. It focuses on the methodology within educational research that seems most fruitful and identifies tools and approaches within it, that appear both appropriate and useful. [How can the author's work on the teaching of programming be embedded in an appropriate academic research framework?]

Chapter 2 then describes the key concept of a patterns and a pattern community and describes the author's initial approaches to developing a pattern community with a group of first year students at the University of Lincoln.

Chapter 3 examines some of the key issues in teaching programming generally. It looks at what constitutes an appropriate programming syllabus for novice programmers and describes some of the debate around the choice of first programming language. [What are the key issues in learning to program? *and* How significant is the choice of first programming language? *and* Is there a meaningful distinction between learning a programming language and learning programming skills?]

Chapter 4 looks at other research efforts in the teaching of programming. It describes areas of research effort such as the development of programming environments, debugging aids and programming languages designed to be used by novice programmers. It also looks at research that supports, or impacts on, human aspects of learning to program, such as the work on confirmatory bias and inattention blindness. [What theories drawn from various areas of psychology are of relevance to the teaching of programming? *and* What aspects of CAL can be used to support the learning and teaching of a programming language?]

Chapter 5 presents a brief overview of learning theories. The main focus is on learning theories that seem promising for the teaching of programming, with particular emphasis on constructivism. [What theories of teaching and learning can be most usefully employed to support the teaching of novice programmers?]

Chapter 6 looks briefly at whether CAL can be used to support the teaching of programming. It reviews some types of CAL and looks at the Stanford Project of the mid 1960s. It also describes the author's experience of developing CAL to support novice programmers. [What aspects of CAL can be used to support the learning and teaching of a programming language?]

Chapter 7 describes in more detail the work outlined in Chapter 2. It describes the creation of the pattern template and the outcomes of the use of various methods within the author's research. It presents some initial results from the author's work with novice programmers and clarifies some of the difficulties in undertaking this type of research. [Having used a particular approach to the teaching of programming, what ways are there of gathering and collating meaningful evidence about that approach?]

Chapter 8 presents a more personal overview of the thesis and briefly reviews the whole research cycle and the author describes the ways in which the research was, or was not, fruitful. Chapter 9 presents some initial outline conclusions, based on the author's work. [Having gathered the evidence, what conclusions can be drawn from the author's work?]

Chapter 1: Conclusion

This chapter has described the initial impetus and context for the work described in this thesis. The roots of the author's interest in the teaching of programming lie in her own experiences as a novice programmer. Later, her role as a teacher of programming led her to question further why those who meet programming concepts for the first time generally find it so difficult (as she did) and why there are no good, well-tested, pedagogically sound strategies for making the initial learning of concepts (and the use of them) less problematic.

The author argues that, since the selling of software is such a hugely lucrative enterprise (on the whole) and the industry is worth billions of pounds each year, then the issue of writing good quality software is always going to be a critical one. There have been many strategies suggested and tried for improving the quality of software: too many to review in detail in this thesis, but the teaching of novice programmers does not seem to be the burning issue the author believed it would be.

This chapter touches upon some of the issues around the teaching of programming such as the lack of clearly articulated pedagogical theories as foundations for this area and the apparently largely unacknowledged divide between those who research programming concepts and those who research learning.

Finally, this chapter outlines the structure of the thesis and briefly notes the contents of each subsequent chapter.

Introduction to Chapter 2

Firstly, this chapter describes what the author is hoping to achieve with her work on teaching and learning programming. The chapter also describes the methodological context for this work. It outlines some of the key aspects of educational research and looks at some of the methodologies available to educational researchers. Choice of a methodology implies a choice within a sub set of research tools and this chapter presents a discussion around the tools chosen by the author in her research.

This chapter also describes the focus of the author's later research, around the concept of a patterns and a pattern community. It briefly describes previous work by the author in using CAL and the reasons for her decision to move on from tutorial CAL, to using a more constructivist approach. It describes how the idea of a pattern community could be fostered through the use of software made available to those learning to program. It describes the goals and aims of the author's research and identifies some of the main issues around this effort.

Goals of the research effort

As described in Chapter 1, the impetus for the author's interest in the teaching and learning of programming came from her own experiences of learning to program in her late twenties, having previously been a teacher of English for five years. The sense of completely lacking any understanding of the material she was studying (how to write pseudocode) and, more significantly, having no coherent strategy for developing that understanding, is an abiding memory for the author. She considers herself to be moderately intelligent but the first time she was required to write a non-trivial program in Pascal for an assignment, all confidence fled!

The author wrote her dissertation for her MSc on the teaching of Prolog and developed a workbook for the language, teaching it from simplest concepts (facts and rules) to more complex ideas (green cut), with examples. At the time, the answer seemed clear: good teaching material with a plethora of examples preferably developed by the teacher him or herself, is all a teacher of a programming language needs.

The author worked for two years in industry, as a technical author, writing user manuals for large-scale software projects. In 1992, she joined the Humberside College of Higher Education, which became Humberside Polytechnic shortly after. The Department of Engineering had started a BSc course in Business and Technology in 1992 and one of the routes was Software Development. Students on the BSc tended, at least initially, to be those who had hoped to undertake business studies but whose A level results had not gained them a place on their chosen course. The joint degree was an option for those students that gave them a strong business studies element but with a technology aspect, such as manufacturing or software.

The author then encountered a whole range of problems in teaching a first programming language to these students and her belief that good worksheets were all that students needed was severely tested. Examples on work sheets were not enough: the students battled to understand the concepts that *underpinned* the first programming language that they learnt. How can someone program a loop appropriately in any language if he or she does not understand the concept of different types of repetition? The difference between REPEAT...UNTIL and WHILE...DO is a critical one, for the execution of the code.

Surely, then the lectures, which looked in detail at the concepts such as the algorithmic toolkit, addressed the understanding of these ideas? The author found that the lectures did not seem to help all the students: some still struggled. So, the author felt that perhaps CAL would prove to be a fruitful way of delivering this material, moving away from the mass lecture model. Over the next three years the CAL was rewritten and used with several cohorts. It was not notably successful.

The author then focused on the idea of on-line help and developed a help system that would grow as the student entered queries. Again, this was not particularly successful. (These previous efforts will be described in more detail in Chapter 6) The author then read about the work of Christopher Alexander and the concepts of patterns, pattern languages and pattern communities. Considerable work has been done in developing pedagogical patterns and patterns for software development: it seemed to the author that the developing of patterns by and for novice programmers was potentially very promising.

This is the focus of the work done by the author over the academic year 2003 – 2004. The author emphasises that this work is very much in its early stages and that this thesis represents a summation of key ideas, a report on previous work and an initial view of a longer term project, the development of a pattern community among novice programmers, in the first year of their computing degree.

The goal of this ongoing work is to foster an approach to the *learning* of programming that is much more openly constructivist, where the collaborative nature of learning to program is acknowledged by the institution and department at a very early stage and is made explicit in one of the resources offered to the students by the department. That is not to say that traditional lectures and workshops are not needed: the idea of a pattern community represents an addition to these activities rather than a replacement for them.

The question then must be, having attempted to develop a pattern community, what impact does it have on the students' perceptions of the process of learning to program and what affect (if any) does it have upon the success of their programming practice? These are questions easily posed but it is, in the author's opinion, extremely difficult to formulate definitive answers. The author's own reading of research around the teaching of programming languages leads her to believe that many researchers do shy away from defining exactly how they measure the success of their work, in terms of student learning. The question of how we may meaningfully measure the impact of pedagogical strategies and the conclusions we can legitimately draw from such measurements is returned to later in this thesis. There are two key issues with the idea of evaluation and assessment.

The first is the assessment of the students' learning, as embodied most obviously in the code they produce. How do we assess their programs? What makes a piece of code good or bad? This view assumes *a priori* that a student who produces good code has learnt the concepts successfully and applied them appropriately. In the author's experience, a student who submits good code may simply be adept at borrowing code chunks rather than understanding and applying programming concepts. Programs are rarely written under examination conditions and there are many sources for useful snippets (or sizeable chunks) of code. We could argue that it is the programmer's role to re-use code, rather than write everything from first principles, and that a student who borrows successfully *is* a good programmer. This is not an issue that can be settled easily or resolved cleanly.

Secondly, there is the question of how we evaluate the research itself. What can be usefully said about the choice of methodology and tools? What are the limitations of the research? What value does it have in a wider arena? There are many complex issues around evaluating research.

The author makes no claim to have clear answers to any of the questions touched on above but believes there is value in at least attempting to clarify some of the issues and in investigating some of the ways these questions might be addressed. It is in this spirit that the author's work is undertaken.

Educational research methodologies

It seems obvious from the discussion above that the author's work must sit firmly in the area of educational research. The scientific method, with its emphasis on the certainties of the physical world, is not an appropriate framework. (For a discussion of positivism and hermeneutics, see Appendix B.) Bernard (1994) notes, "*The split between the positivistic approach and the interpretive-phenomenological approach pervades the human sciences.*" (Bernard 1994 p 15) Bernard (1994) stresses that the scientific approach should not be confused with quantification. Dealing with aggregates rather than individuals adds a quantification element to interpretive work, such as that of ethnography, but as Bernard (1994) notes "*...all quantification is not science, and all science is not quantification.*" (Bernard 1994 p 16)

The work done by the author focuses on the human activities of learning and learning to program and, as such, is human-centred, much as research in social sciences is. Verma and Mallick (1999) acknowledge that research in the social sciences is "*not perceived as having the same 'scientific' status as the natural sciences...*" (Verma and Mallick 1999 p 8)

Given that we cannot draw conclusions about human behaviour in the same way as we might about the chemical reactions or the physical behaviours of objects, there is still value in exploring the rich and diverse elements that make up human activities.

Having placed the author's work in the arena of educational research, is there anything else that needs to be said? Educational research is not a monolithic activity: it has variety. Verma and Mallick (1999) describe three types of research that have received much attention in the field of educational research: evaluative research, consumer-oriented research and action research. (Verma and Mallick 1999 p 46)

Evaluative research is the collection of data about the effectiveness of an educational experience and is often used to support decision making in areas such as "*course improvement, decisions about individuals and administrative regulation.*" (Verma and Mallick 1999 p 46) Consumer-oriented research is aimed at producing data of direct value to teachers (who, of course, constitute a varied group of consumers of educational research data).

Action research

Cohen and Manion (1994) define action research as "*small-scale intervention in the functioning of the real world and a close examination of the effects of such intervention.*" (Cohen and Manion 1994 p 186) Cohen and Manion (1994) make a useful distinction between action research and applied research. Cohen and Manion (1994) note that both use the scientific method but that applied research is "*concerned* *mainly with establishing relationships and testing theories,* [and] *it is quite rigorous in its application of the conditions of this method.*" (Cohen and Manion 1994 p 187) Action research adheres less closely to the scientific method because it focuses on a "*specific problem in a specific setting.*" (Cohen and Manion 1994 p 187)

Cohen and Manion (1994) describe the purposes of action research as falling into one of five broad categories. The first category we have already noted: that of a specific problem in a particular setting. The second category is that of in-service training: action research as a way of "*equipping teachers with new skills and methods*" (Cohen and Manion 1994 p 189). Thirdly, action research can be a vehicle for introducing change into an educational setting, where change might otherwise be resisted. Fourthly, action research may facilitate better communication between those who teach and those who research. Finally, it is "*a means of providing a preferable alternative to the more subjective, impressionistic approach to problem-solving in the classroom.*" (Cohen and Manion 1994 p 189)

We can see that these categories may well blur together within some action research activities. To solve a problem might require the introduction of notable changes in classroom practice, and the work might lead other teachers to adopt a new approach: to take up the research results, in effect.

The author sees her work as being clearly situated in the field of action research, mainly within the first category noted above, given that the overall aim of this work is to make more effective her own teaching of programming.

Action research is a methodology that has caught the imagination of many researchers in education and the social sciences:

> "*Apart from phenomenography, action research is perhaps the most influential and almost certainly the fastest-growing orientation towards educational and staff development at present.*" (Webb 1996 p139)

What implications does this have for the author's work? As noted, the focus of action research is upon the specific problem or issue in a specific situation, with less emphasis on developing a theory:

> "[Action research] *is about the nature of the learning process, about the link between practice and reflection, about the process of attempting to have new thoughts about familiar experiences, and about the relationship between particular experiences and general ideas.*" (Winter 1996 p14)

Verma and Mallick (1999) describe action research as being more concerned "*with the immediate application rather than the development of theory.*" (Verma and Mallick 1999 p 12)

Kurt Lewin coined the phrase action research in the 1940s. Bryant (1998) describes how the concept of action research (a concept only fifty years old) has its roots in a positivist view, in the work of Lewin (1948). In Lewin's model of action research, "*the distinction* [between researched and researchers] *was preserved.*" (Bryant 1998 p107)

Later developments in action research were to move away from this view, towards a more reflexive stance. Reflexivity implies that the researcher is more than just self-aware, that he or she has an understanding of how his or her practice and experience is both part of self and created by the self. Such an understanding is based in the social context of that practice and research. Winter (1996) writes of the researcher's need to understand fully the nature of his/her interest in the research problem: this nature is itself a complex resource, involving emotion, memory and "*half-glimpsed insights.*" (Winter 1996 p 15)

In a sense, reflexivity argues that all research is ultimately saying something about the researcher and the researcher, in undertaking research, is saying something about him or herself. Reflexivity is discussed in more detail later in this chapter.

Action research can take a variety of forms, but it is possible to identify key elements that mark this methodological framework:

> "*Action research can be described as a family of research methodologies which pursue action (or change) and research (or understanding) at the same time.* [] *It is thus an* ***emergent*** *process which takes shape as understanding increases; it is an* ***iterative*** *process which converges towards a better understanding of what happens.*"

(http://www.scu.edu.au/schools/gcm/ar/whatisar.html 29/01/2004)

> "*Action research is deliberate, solution-oriented investigation that is group or personally owned and conducted. It is characterized by spiraling cycles of problem identification, systematic data collection, reflection, analysis, data-driven action taken, and, finally, problem redefinition.*"
> (http://www.ericfacility.net/ericdigests/ed355205.html 29/01/2004)

We can, therefore, summarise action research as being iterative, in which a problem or issue is addressed a number of times, seeking a closer understanding (but perhaps not a definitive solution) within a particular educational context and environment. This seems to neatly sum up the crux of the author's work, where she has worked with the issues of teaching novice programmers over several cycles of reflection and action!

We can say that action research is the practical tool of the involved, aware professional:

> "*Action research is simply a form of self-reflective enquiry undertaken by participants in social situations in order to improve the rationality and justice of their own practices, their understanding of these practices, and the situations in which the practices are carried out.*" (Carr and Kemmis 1994 p 162)

In action research, even if the researcher does not adopt a reflexive stance, it is important to note that is not possible for a researcher to treat the educational practices he or she studies (his or her own) as independent phenomena, separate entities, subject to "*universal laws*" (Carr and Kemmis 1994 p 180). Action research is criticised for a lack of scientific rigour and its inability to produce generally applicable conclusions or results.

Cohen and Manion (1994) note

> "*That the method should be lacking in scientific rigour, however, is not surprising since the very factors which make it distinctively what it is – and therefore of value in certain contexts – are the antithesis of true experimental research.*" (Cohen and Manion 1994 p 193)

Winter (1996) identifies four practical problems for action research: the problem of collecting data as a practitioner (with other demands on time and energy), the difficulty of identifying a research topic that is not "*too minimal to be valid, or too elaborate to be feasible…*" (Winter 1996 p 17), the problem of methods, and the ways in which these methods can "*contribute a genuine improvement of understanding and skill…*" (Winter 1996 p 17)

Cohen and Manion (1994) summarise the criticisms made about action research. Action research is specific to a situation, rather than general and its sample is usually "*restricted and unrepresentative*" (Cohen and Manion 1994 p193). Independent variables are poorly controlled (or not controlled at all) and the conclusions drawn from such work are not widely applicable. (Cohen and Manion 1994 p 193)

Given the criticisms above, why would the author place her work within such a framework? She has already stated that she wishes to draw some more generally applicable conclusions about the teaching of programming to novice programmers: why select a methodological framework that seems to militate against that aim? The answer lies in the shortcomings of the methodology: the problem may be general but the author's opportunities to study it are local, specific and shifting, from year to year. The scientific method requires a rigour that is not available within the author's situation. Also, the question of how to teach novice programmers is a one that requires a number of caveats.

To talk about 'novice programmers' implies uniformity among all new programmers, which is patently not the case. To talk only of 'teaching' novice programmers implies a behaviourist view that the author is uncomfortable with. There are also ethical issues around more rigorous control of independent variables, as well as the dependent variable. It is tempting to try to adopt the scientific approach and hope to hide the messy non-scientific aspects of such work but the author feels her commitment to her chosen methodology represents her best professional and personal efforts to engage with a significant issue. Overall, in the author's view, action research (or some flavour of it) is the most suitable framework for her research.

Zuber-Skerritt (1996) defines a type of action research, emancipatory action research, and defines such activity as having a cyclical set of stages: strategic planning, implementing the plan, evaluation and reflection. (Zuber-Skerritt 1996 p 3)

The teacher in this paradigm does not take sole responsibility for the interactions in the classroom. This research model embodies the view of research as a joint undertaking, exploring educational practices, the whole educational situation in which those practices are undertaken, and educational theories:

> "*The form of action research which best embodies the values of a critical education science is emancipatory action research. In emancipatory action research, the practitioner group takes joint responsibility for the development of practice, understandings and situations, and sees these as socially-constructed in the interactive processes of educational life.*" (Carr and Kemmis 1994 p 203)

This flavour of action research seemed to best to describe the approach the author wished to take, in contrast to her almost positivist stance when writing and using CAL in her teaching. In her earlier research, the students were required to work through the CAL package developed by the author, Introduction to Programming, without exception. There was no sense in which the students had any real say in the design or use of CAL, except in opting out altogether, in the same way as they might miss lectures or workshops.

Stages in action research

Cohen and Manion (1994) describe eight stages that a programme in action research *may* go through. It should be noted that these stages are potential, not mandatory. The first stage is the "*identification, evaluation and formulation of the problem.*" (Cohen and Manion 1994 p 198) The second may involve negotiation among involved and interested parties: "*teachers, researchers, advisers...*" (Cohen and Manion 1994 p 198) The third stage may be a literature review. The fourth stage may "*involve a* *modification or redefinition of the initial statement of the problem*" (Cohen and Manion 1994 p 199).

The fifth stage may be concerned with the selection of research procedures such as use and allocation of resources and materials and sampling strategies. The sixth stage may look at the way the project is to be evaluated. The seventh stage "*embraces the implementation of the project itself (over varying periods of time).*" (Cohen and Manion 1994 p 199) The final stage involves the overall evaluation of the project and analysis and evaluation of the data gathered during the project.

Having placed the research effort in a methodological context, the next question is, what tools or methods are available within that framework? For example, if a researcher places his or her work firmly in the area of scientific method, we can argue that he or she is likely to generate a theory or measure a hypothesis, using quantitative data, in ways that are largely repeatable by others in the same field. This argues the use of certain approaches based on quantitative data. Since the author has noted that she places her work firmly within the action research framework, what tools are appropriate and manageable for the research effort? In the next section, the author examines the concepts of qualitative and quantitative, within the context of research methods.

Educational research tools and methods

Research tools can also be divided along the qualitative/quantitative axis, in the same way as the positivist approach can be seen as scientific and quantitative and the hermeneutic framework can be seen as qualitative. Strategies such as experimental, correlational and ex post facto are generally seen as quantitative. Methods such as structured interviews and questionnaires are seen as quantitative while other methods such as unstructured interviews, diary keeping and participant observation have been seen as qualitative. (http://trochim.human.cornell.edu/kb/qualdata.htm 07/01/2004) Work has been done by researchers such as Rist (1977), Smith and Meshusius (1986) Bryman (1988) and Hammersley (1992), as cited by Scott (1998 a), to challenge the qualitative/quantitative divide.

Qualitative and quantitative research

Allied with the classification of research tools is the broader qualitative versus quantitative question: Hammersley (1992) notes, "*In one form or another the debate about quantitative versus qualitative research has been taking place since at least the mid-nineteenth century.*" (Hammersley 1992 p159) Do these two approaches genuinely represent two opposing epistemological positions?

Cohen and Manion (1994) describe these two views of research (in social sciences) as if they shade into each other. To view the social world as if it possessed a defined and quantifiable external reality, which can be agreed upon, is to adopt a quantitative approach. To acknowledge that the individual situates him or herself within the social world and to seek to understand that subjective, particular and personal view is to adopt a qualitative approach. (Cohen and Manion 1994 pp 7 – 8) Giarelli and Chambliss (1988) define the aims or concerns of qualitative thinking as clarity, context and consciousness.

Verma and Mallick (1999) say, "*Strictly speaking neither 'quantitative' nor 'qualitative' is a discrete perspective on research.*" (Verma and Mallick 1999 p 26) The authors do go on to note that choice of an approach can influence the type of data gathering and the ways that data is used. Verma and Mallick (1999) define quantitative as "*any approach to data collection where the aim is to gather data that can be quantified; that is to say it can be counted or measured in some form or another.*" (Verma and Mallick 1999 p 26) Qualitative data gathering concentrates on evidence that reflect the "*experiences, feelings or judgments of individuals.* "(Verma and Mallick 1999 p 27)

It is the qualitative focus that seems most appropriate for the author's work but there are difficulties with this approach. It could be argued that a purely qualitative approach generates no broadly applicable ideas. As we have seen, action research focuses on a particular situation but work that is applicable only to the people who participated in it could not be useful in any wider sense. There surely must be a drive (even if it is not explicitly stated) to generate some broader conclusions, that may inform or illuminate the practice of others.

This is a tension not easily resolved. As the author has noted, her methodological framework argues for largely qualitative approach, a harvesting of the student experience but her wish to draw more general conclusions argues for the use of some quantitative tools. One area where the quantitative approach could be appropriate might be in the assessment of students' work.

If using a CAL package (for example) does improve student learning of programming concepts how do we measure that improvement? Are we measuring the improvement on marks for programming assignments against last year's cohort (with all the vagueness that implies) or are we using some other method? What if we claimed that the improvement could be seen perhaps only when students learn their next programming language or take up their first programming post? How do we measure that improvement over a course of two or three years? The question of assessing programming effort is returned to later in this thesis.

In the author's case, she wishes to reflect upon her own strategies for teaching programming and to bring to the teaching effort elements that seem to have been missing from the material in this area, such as ideas about learning in general. The author has already said that the work is ongoing. It is also small scale and specific. However, the author hopes, from her qualitative stance, to draw useful conclusions that may form the basis for further work.

This is not to say that there is no room for quantitative approaches in educational research.

> "*Note that, while these two approaches to research are often presented as if they were in binary opposition to one another, they can also be used to complement one another. Furthermore, some methods* [] *actually fuse elements from both approaches.*" (http://hc.les.dmu.ac.uk/michael/qual_aims.htm 13/01/2004)

LaRose, Gregg and Eastin (1998) present a small-scale study that uses a quantitative approach to explore

> "*...whether a minimalist audiographic approach could still be as educationally effective as classroom instruction and whether it would appeal to learners sufficiently to sustain attendance to online instruction.*" (LaRose, Gregg and Eastin 1998 found at http://www.ascusc.org/jcmc/vol4/issue2/larose.html 13/01/2004)

Willig (2001) notes,

> "*Qualitative researchers tend, ... to be concerned with the quality and texture of experience, rather than with the identification of cause–effect relationships. They do not tend to work with 'variables' that are defined by the researcher before the research process begins. [] Using preconceived 'variables' would lead to the imposition of the researcher's meanings and it would preclude the identification of respondents' own ways of making sense of the phenomenon under investigation.*" (Willig 2001 p 9)

Having placed her work mainly in the qualitative arena, within educational research, what are the implications for the author's research activities? The author's reading around the design of the research activities quickly presented her with a number of problems.

Little q and big Q

Willig (2001) cites the work of Kidder and Fine (1987) in distinguishing between two types of qualitative research, termed little q and big Q. Willig (2001) notes that little q

> "*methods of data collection and analysis do not seek to engage with the data to gain new insights into the ways in which participants construct meaning and/or experience their world; instead, they start with a hypothesis and researcher-defined categories against which the qualitative data are then checked.*" (Willig 2001 p 11)

Big Q places the emphasis firmly on the participants' construction of meanings around their own experiences. It is only big Q that Willig (2001) believes to be genuinely qualitative. Willig's (2001) definition of little q and big Q seem to go some way towards constructing a defence against those who criticise qualitative research as being unscientific: Willig (2001) believes that truly qualitative research has no need for the control of variables: the relevance and value of the work is to be found in "*the exploration of meanings...*" (Willig 2001 p 11) The question is, is the author's work big Q or little q? In retrospect, the author finds she has moved between the two, with her final effort being much more big Q. In contrast, we can examine some of the little q approaches. Scott (1998 a) says that the experimental researcher seeks to discover causal relationships between phenomena.

A key aspect of this type of research is the control of all the variables. For the experimental model, Scott (1998 a) defines three approaches. The simplest version of this is the pre-test and post-test design, with a small or single group. Pre- and post-tests are designed to highlight the effect of the intervention. A control group can be added to this experimental model to offer a basis for comparison between the groups, which should be similar in composition. The third approach is to concentrate on a range of different groups, "*each of which has different characteristics, and each of which subjected to different types of interventions*." (Scott 1998 (a) p 53) This experimental research model is deductive, involving the testing of hypotheses.

Immediately, we can see problems with trying to use this model within the author's qualitative, educational research oriented framework. What tests could the author meaningfully administer to novice programmers *before* they learned to program? How can the myriad of variables involved be controlled? In the author's view, there is no valid way to control the factors that impact on a student's experience of learning to program over weeks and months. It could be argued that perhaps the intervention (CAL, on- line help or pattern community) should be offered to only a random sample of the students. This raises practical and ethical issues.

If the intervention is designed to make more effective the students' learning it is unacceptable to offer it to only some students, even if it were possible to stop the non-selected students using or learning about the element being employed or offered to the other students! Given the very short time scales of semesters it is not possible to design an experiment that allows for a control group: in a ten week teaching period for a single unit, it is not feasible to have a control group that would have to catch up with the material in some way while starting a new semester's worth of study units.

So, there are a number of problems with the experimental method in areas of research that focus on people. People are difficult to control in the sense of managing both the independent and dependent variables. Scott (1998 a) argues that there are four areas of difficulty. First, effects being looked for may be very subtle and difficult to test for. Secondly, the situation being studied may be partly or wholly artificial, so that experiments are ecologically invalid. (Scott 1998 (a) p 54) Thirdly, the experimental approach may sit poorly with an ethnographic approach. Finally, the experimental method may involve the researchers in "*complicated ethical dilemmas*" (Scott 1998 (a) p 54) that do not admit of easy solutions, as the author notes above.

In the case of researching students learning to programming, the first and second points are extremely relevant. What exactly is the author looking for when she studies an independent variable? Is the desired effect to make students feel more positive about learning to program or is it to improve their ability to select appropriate constructs? There are shades of effect between those two poles, also. If an intervention makes students feel better about their programming efforts, they may be prepared to put more time and energy into programming, perhaps creating better (however that is defined) code.

If, as a researcher, the author forces students to work as much in isolation as possible (by setting a formal programming test, for example), there are two key objections. In the real world, programming is not a usually a solitary activity. To make a novice programmer program by him or herself in a test situation gives rise to the unreality that Scott (1998 a) refers to. Such unnaturalness could be said to negate any conclusions that can be drawn.

Scott (1998 a) notes that an experimental approach may not blend well with a more humanistic view such as ethnography. This seems to argue that the researcher needs to declare an allegiance to one or the other: either a researcher is a positivist or he or she is an ethnographer! The author feels that this is a point that could be debated over at length. She felt initially that there is room for some quantitative measurements (some little q) within a broadly humanist approach, although the data needs to be treated with some care and its limitations clearly noted by the researcher. Later, the author's view shifted somewhat and she felt that big Q offered more meaningful engagement with the research issues.

In summary, the author's reading persuaded her that for the purposes of this research, a purely positivist, scientific approach was neither appropriate nor ethical.
The author must then focus on other methods that will answer the needs of the research. Given that the core of this work is the student experience of learning to program, the author feels that she must use methods that enable her to garner that experience, in appropriate and meaningful ways.

Survey methods

This stance argues for the use of survey methods. Survey methods can be said to have two main approaches: correlational research and ex post facto research. Verma and Mallick (1999) note, "*The survey has come to be one of the most widely employed tools in educational research.*" (Verma and Mallick 1999 p 115)

Correlational research uses statistical devices to identify relationships within data "*and a determinant of the probability of those relationships holding firm in other settings adduced.*" (Scott 1998 (a) p 55) We can see that this is a very strong little q approach that may use "*incorporation of non-numerical data collection techniques* into hypothetico-deductive research designs." (Willig 2001 p 11)

In ex post facto research, the researcher attempts to re-construct what happened and find the causal relationships from that reconstruction. Cohen and Manion (1994) notes that is a way of "*teasing out possible antecedents of events that have happened and cannot, therefore, be engineered or manipulated by the investigator.*" (Cohen and Manion 1994 p 146)

This method typically involves structured interviews, questionnaires and attitude inventories.

Verma and Mallick (1999) note:

> "*...the questionnaire is often a vital tool in the collection of data. If it is well-constructed, it can provide data economically and in a form that lends itself perfectly to the purposes of the study.*" (Verma and Mallick 1999 p 117)

Which of these seems most useful in the context of this research? Correlational research might be useful to establish, for instance, a relationship between a student's grades at A level and the final grade for the study unit Introduction to Programming. The author could undertake some very fine-grained work around looking at A level subjects taken, grades achieved and success in the student's first programming efforts. For example, does a student with an A level in philosophy have any advantage in learning to program? Do grades D or E in A level computing generate a positive effect on student achievement in that unit? These are interesting questions, but limited. What is of interest to the author is the student's experience of learning to program, not his or her subject choices at 16.

However, the research the author undertook cannot be said to be ex post facto research, since the author uses independent variables, at various points, rather than seeking to reconstruct the factors that generated student success in programming. Some of the tools of ex post facto research did seem appropriate to the author and questionnaires (in several versions) have been written and used by the author.

Willig (2001) notes four main tools in straightforward qualitative research: semi-structured interviewing, participant observation, diaries and focus groups. The author has used three of those tools, with varying success. Her uses of semi-structured interviewing, focus groups and participant observation were perhaps more tentative than was ideal and the results were mixed.

This section will look at some of the tools that may be appropriate to qualitative research effort, such as semi-structured interviews, participant observation, diaries and focus groups. Willig (2001) notes "*Qualitative data collection techniques need to be participant-led, or bottom-up, in the sense that they allow participant-generated meanings to be heard.*" (Willig 2001 p 15)

This raises different issues for researchers who engage in qualitative research from those who adopt a quantitative approach. Willig (2001) notes that the type of data collected for a qualitative study must be naturalistic, that is, "*the data must not be coded, summarized, categorized or otherwise 'reduced' at the point of collection.*" (Willig 2001 p 16) Obviously, at some point, researchers need to work with the (often voluminous) data they have collected but the shaping and interpretation of data is an extremely critical stage of the qualitative research process.

Willig (2001) notes that good qualitative research questions are process oriented. A qualitative researcher asks a How question rather than a What question. Interestingly, Willig (2001) says "*qualitative research is open to the possibility that the research question may have to change during the research process.*" (Willig 2001 p 19) This is true of the author's own work, where the question has changed quite significantly during her research effort.

Willig (2001) notes that there are a wide range of qualitative data collection techniques, some of which preclude particular types of data analysis. The example given is that of note taking in a semi-structured interview: the notes cannot be the subject of conversation analysis. It behoves the qualitative researcher to match method(s) of data collection with the appropriate data analysis technique(s). (Willig 2001 p 21)

Semi-structured interviews

Willig (2001) notes that the semi-structured interview "*provides an opportunity for the researcher to hear the participant talk about a particular aspect of their life or experience.*" (Willig 2001 p 22)

Also,

> "*Semi-structured interviewing is useful in situations where broad issues may be understood, but the range of respondents' reactions to these issues is not known or suspected to be incomplete.*"
> (http://www.ucc.ie/hfrg/projects/respect/urmethods/interviews.htm 03/02/2004)

The key to gathering data successfully through semi-structured interviews is to prepare thoroughly. The researcher should reflect on how to find and recruit participants, how best to record the interview(s) and how to elicit from participants a full and meaningful response. Willig (2001) says "*A good way to obtain detailed and comprehensive accounts from interviewees is to express ignorance.*" (Willig 2001 p 22)

The interview may flow in ways not predicted by the researcher (part of the challenge of the qualitative effort!) but the researcher will need to keep in mind a general interview agenda, if all sight of the original purpose of the interview is not to be lost.

It may also be useful to classify the questions to be posed according to Spradley's (1979) classification (Spradley 1979 in Willig 2001 p 24). Questions can be classified as descriptive, structural, contrast or evaluative. Descriptive questions ask for general accounts from the interviewee, such as, 'What were you doing on the day war was declared?' Structural questions ask interviewees about the ways in which they perceive the world and make sense of it. Contrast question pose comparisons and evaluative questions focus on the feelings of interviewees.

Interviews will need to be recorded in some way. Note taking can be distracting for both interviewer and interviewee(s) and cannot hope to match the richness of a video or audiotape record. However, it is important that the interviewee(s) are comfortable with both the process of recording the interview and the later use of the transcribed interview material. Finally, Willig (2001) notes, "*An interview transcript can never be the mirror image of the interview.*" Willig (2001 p 25)

Participant observation

Participant observation requires the researcher to be sufficiently involved with the activities around him or her to grasp what is going on but at the same time, to be in some sense apart from the activity so that s/he may observe what is happening, and also, ideally, observe his or her role in the activity also.

For example, when the author sat in on a colleague's programming lab session, she was introduced to the group at first and found it hard later not to become involved in the teaching by responding to the lecturer's questions! The author also found it hard not to admonish a student for writing emails during the teaching session.

> "*Participant observation has a quite distinct history from that of the positivist approach to research. Positivist researchers employing questionnaires and surveys assume that they already know what is important. In contrast, participant observation makes no firm assumptions about what is important. This method encourages researchers to immerse themselves in the day-to-day activities of the people whom they are attempting to understand. In contrast to testing ideas (deductive), they may be developed from observations (inductive).*"
> (http://uk.geocities.com/balihar_sanghera/qrmparticipantobservation.html 02/02/2004)

We can identify three key strands in participant observation. Firstly, we can consider that the complex set of interactions that we identify as the social aspect of being human, is constantly changing and is also shaped by the environment. Therefore, to understand what is happening, a researcher must be in that environment, subject to those same constraints as those he or she is seeking to understand.

> "*...researchers must become part of that environment for only then can they understand the actions of people who occupy and produce cultures. This technique is least likely to lead researchers imposing their own reality on the social world they seek to understand.*"
> (http://uk.geocities.com/balihar_sanghera/qrmparticipantobservation.html 02/02/2004)

This is pragmatism. Secondly, there is formalism, which

> "*argues that while social relationships may differ from each other, they take forms that display similarities. In this way, researchers explore the typicality of relations and events.*"
> (http://uk.geocities.com/balihar_sanghera/qrmparticipantobservation.html 02/02/2004)

Finally, there is naturalism, which "*proposes that, as far as possible, the social world should be studied in its 'natural state' undistributed by the researcher.*" (http://uk.geocities.com/balihar_sanghera/qrmparticipantobservation.html 02/02/2004)

In this view, participant observation stresses the participation aspect. A researcher who joins a group of football fans, for instance, needs to feel the thrill of triumph and the disappointment of defeat, just as the other fans do.

Willig (2001) notes that there are different types of notes that the researcher needs to be making: substantive, methodological and analytical. Substantive note are concerned with the descriptions of conversations and events. Methodological notes describe the challenges and events of the observation process. Analytical notes are the beginnings of theory, of seeing patterns and themes emerge from the observation.

Diaries

Getting participants in a research project is not an easy task. Keeping a diary requires commitment and effort from the diary writer, more perhaps than can be asked of most participants. There may be cases where diaries constitute a truly meaningful way to gather qualitative data. This diary approach has been used by the author for assessment purposes, where keeping a reflective log of work done on the assignment formed part of the assessment. The author has used this at undergraduate and postgraduate level, with varying degrees of success.

Focus groups

These provide an alternative to the use of semi-structured interviews. "*The focus group is really a group interview that uses the interaction among participants as a source of data.*" (Willig 2001 p 29) What makes a focus group different from a semi-structured interview is the opportunity it affords participants to comment upon, and respond to, others' contributions. Willig (2001) notes that focus groups should ideally consist of no more than six people. Such groups may be homogenous, heterogeneous and pre-existing or new. (Willig 2001 pp 29 – 30) The caveat with a focus group setting is that participants may be less willing to be open and honest, where their comments are for general consumption by the group.

In summary, these four data collection methods are not the only ones available but they represent tried and tested ways to gather qualitative data. A researcher may choose to combine two or more methods in order to view the same problem or issue from different perspectives. "*This constitutes a form of triangulation.*" (Willig 2001 p 30)

If we are to gather the student experience in some way, we then need to ask, which students and why? As noted above, questionnaires can be used to gather data from as many of the students involved with the unit as possible. For methods like interviews, it would not be possible to interview every student on a unit, even for a small student intake. The next question would then be, how do select which students to talk with? Whose experience should be gathered and whose left unarticulated? The question is one that brings us back to a much more numerical approach, in defining sampling strategies.

Sampling strategies

There are two general sampling modes: probability and non-probability sampling. A simple, random sample is the commonest type of probability sample. One method is to list all the members of the target population according to a criterion which is independent of the research e.g. by last name.

Researchers can then select every fifth or fifth person. This is known as a systematic sample, and is a very easy means of selecting a random sample, but lists can sometimes contain trends which are difficult to identify, and which create a non-random sample.

A variant of random sampling can be used in which the target population consists of a number of clearly defined groups, which are of interest to the researcher. For example, consider a study in which ethnicity is one of the important factors. If a simple random sample is taken of a mixed-ethnic population, then some of the ethnic groups may not be represented in the final sample. This kind of situation happens by chance in random sampling, and in this case may be counterproductive to the research.

The problem can be avoided by considering each ethnic subgroup of the population as being a separate entity, and drawing a simple random sample from each subgroup. Each subgroup can be considered as a section or stratum of the overall population, and hence this sampling procedure is termed a stratified random sample. In the case of the author's research, in larger cohorts, it may be useful to distinguish between student with some previous programming experience and those with none and to gather data from those two groups separately.

In practice, this has proved to be difficult to do, for a number of reasons. Students may claim to have previous programming experience but when asked, in fact place them selves firmly in the beginner or novice category and the reverse is also true. Students who claim to be novices have backgrounds or experiences that help them to grasp programming concepts with apparent ease. It is an interesting question: why do people claim more or less competence than they may have? How realistic a student's claims of proficiency or understanding may be is a vexed one and not one that admits of an easy solution.

Non-probability sampling includes convenience or accidental sampling, which is building up a collection of the nearest individuals until a sample size is reached. There are also purposive samples where individual are handpicked. A commonly used type of purposive sampling is key-informant sampling. A key informant is someone who has a specialist, insider knowledge of the research issue.

In practice, the author has had to rely on students volunteering to take part in interviews. The author has found that students who are prepared to discuss their experiences of learning to program, whether by email, in a face-to-face interview or in a focus group are those who are articulate, motivated and generally academically very able. They are a self-selecting sample and those who are less successful or struggling with the material do not come forward. This is a difficulty that the author has not yet been able to overcome! This problem of a self-selecting sample also means that very small numbers respond to requests to further information. Out of a cohort of nearly

150 students just three agreed to take part in a discussion about learning to program in December 2003.

The question is what methods are most appropriate to the research being undertaken by the author? The author has used a mix of approaches, with varying success. These will be described both here and in Chapter 6, when the author reviews her work with CAL.

Given that the author is examining the progress of students' understanding of programming and their programming proficiency, is there a case for a longer-term study of a sample group? There are three main types of descriptive or developmental research: longitudinal, cross-sectional and trend or prediction studies:

> "*Because education is primarily concerned with the individual's physical, social, intellectual and emotional growth, developmental studies continue to occupy a central place in the methodologies used by educational researchers.*" (Cohen and Manion p 68)

Longitudinal studies are conducted over a period of time, either weeks or years. Sampling the same people over the time period is a follow up, cohort or panel study. Sampling different people constitutes a cross sectional study. Studying a few selected factors continuously is a trend study.

Cohort and trend studies are longitudinal methods. Retrospective longitudinal studies focus upon people who have reached a particular point or end state e.g. those who have failed GCSE English. In the case of the author's work, one approach might be work with students who do not pass the assignments in the relevant programming unit. However, there are significant ethical issues involved in this approach.

Conversely, the author might work with students who are notably successful in learning to program, in terms of passing the required assessments. This might be easier to do but may not give any insight into why other students do less well. Also, it is a truism that good attendance strongly correlated with good overall results. Students who are less successful in their programming efforts may be so because of lack of attendance (or other factors not within the control of the tutor) rather than because of any element within the delivery of the material.

Given the practical constraints of the academic cycle, the author has largely engaged in small sample studies with one cohort over one semester. This would present significant problems if the author were taking a purely quantitative approach, as student groups even of 180 (which would be very large for a first year student intake on the computing BSc routes at the University of Lincoln) do not constitute a large enough sample size to be statistically significant. Given that the focus is qualitative, the sample size is perhaps less important than the data gathering methods. The question does still need to be asked: how do we measure the validity of the research effort?

Internal and external validity

The scientific method puts emphasis on validity (and repeatability) of experiment, if they are to form foundations for the testing of hypotheses and the formulating of theory. Willig (2001) defines validity as "*the extent to which our research describes, measures or explains what it aims to describe.*" (Willig 2001 p 16)

We need to examine validity criteria, which fall into two categories: internal and external. External validity is where the results obtained would "*apply in the real world…*" (Tuckman 1978 p 4) and can therefore be generalised to populations outside those of the experiment. Internal validity questions whether the "*experimental treatments…make a difference in the specific experiments under scrutiny?*" (Cohen and Manion 1994 p 170)

Cohen and Manion (1994) argue that threats to validity (both internal and external) are of greater significance in educational research than they are to scientific experiments. (Cohen and Manion 1994 p 170) In contrast, Willig (2001) argues that "*As a result of their flexibility and open-endedness, qualitative research methods provide the space for validity issues to be addressed.*" (Willig 2001 p 16) The perceived weakness of the non-scientific approach to research can also be perceived as a strength.

Cohen and Manion (1994) define six threats to external validity. The first is the failure to describe independent variables clearly. The second is the lack of representativeness of the populations studied. Thirdly, there is the Hawthorne effect: participating in research may be enough to make substantial difference to those being researched, whose behaviour is noticeably affected by participation in itself. Fourthly, there is "*inadequate operationalizing of dependent variables*" (Cohen and Manion 1994 p 171) There are the changes brought about in subjects by the use of pre-tests, which effectively sensitise subjects to the factors being tested. Finally, there is the general category of interaction effects, where there are "*interactions of various clouding factors with treatments*." (Cohen and Manion 1994 p 172)

We can see that since the author has clearly stated that she wishes to draw general conclusions about the teaching of programming that may be applicable to other learners of programming, then she cannot ignore these elements and must address the relevant issues outlined above.

For the two main research efforts that the author has undertaken, the independent variable (the use of a CAL package or some form of on-line resource) is clearly identified. What is more difficult is the dependent variable to be studied. The author has looked at various dependent variables including assessment success, student satisfaction with the teaching module, student estimates of resource usefulness and test results. All of these have proved either difficult to measure meaningfully or have provoked disappointingly few responses from the students involved.

The second point that Cohen and Manion (1994) raise is that of representativeness. How far do the students studied by the author represent the general population of novice programmers? This is an extremely difficult question to answer. Probably there are as many types of novice programmers as there are novice programmers! What can be said is that the intake of students onto BSc courses in Business and

Technology (at what was then Humberside Polytechnic) were mainly recruited through clearing. They would generally have had a lower A level points total than students going to what was then the university sector. For BSc computing, students have been recruited with perhaps an E and D at A level. We can say that the students studied by the author *may* be less academically successful at age 18, as measured by their A level results, than their counterparts going to study at Durham, Sheffield or UMIST.

Does this make any conclusions drawn by the author invalid for students with higher A level points? The author would argue that a good strategy for teaching programming is of use to almost everyone learning to program, depending perhaps on his or her preferred learning style. The author has encountered very able students who were happy to help fellow students and conversely, students who went out of their way to avoid helping anyone else. If a resource is made available, such as pattern community, some will contribute and others will not. A pattern community cannot be a compulsorily created artefact.

The Hawthorne effect is also an issue that must be considered significant in educational research. If a tutor is focusing on a group of students, to the extent of creating and using new approaches to the teaching of material, how far does that affect the students' perception of that learning experience? The author would argue some contamination is unavoidable; especially where the tutor is presenting the students with an artefact that is not being used in any other teaching situation. The author's approach to this has been to be open with the students about the fact that she is doing research with and through this artefact.

Cohen and Manion (1994) also point out that the dependent variables examined in a piece of research must have validity outside the experiment. (Cohen and Manion 1994 p 171) As noted earlier, the dependent variables have proved to be extremely difficult to focus upon. What seems most promising is an exploration of student attitudes to their experience of learning to program. However, the difficulty with this is that a positive experience does not necessarily mean that the student has learnt what was hoped for or required! Perhaps some quantifiable dependent variable is needed such as a test of student understanding of programming concepts. This idea throws up problems of its own, however, as discussed earlier. Programmers do not program in isolation in test conditions and an understanding of concepts does not necessarily mean that the person is a good programmer (however we wish to define 'good'). A student may be able to parrot the structure of the constructs available but not be able to make a sensible choice of type of loop within a piece of code.

Sensitisation to "*experimental conditions*" (Cohen and Manion 1994 p 172) is also an issue. The author has used versions of an entry questionnaire a number of times, to garner information about the students and their previous programming experiences. This in itself may have sensitised the students, alerting them to 'something different' about the teaching module. The author cannot see any way around this problem! With human subjects, all data gathering is an opportunity for sensitisation of the subjects. Only entirely covert observation would be an answer but not one that is either ethical or suitable.

Finally, Cohen and Manion identify the problem of extraneous factors in effectively blurring the results of the work. The author feels that this is one of the most significant elements in educational research, on a macro and a micro level. The context in which the unit on introductory programming was delivered changed every year. There would be staff changes, changes to the physical plant (labs were sub-divided or opened up), the network interface was changed or the Department changed its name and organisation!

Some changes were obviously minor, others more significant. The author had no control over these changes. At the start of the first semester there would often be severe timetabling problems: students frequently complained about a very ragged start to the semester which for first year students meant a difficult beginning to their academic life at the University. The network, which provided students with access to the software required for their work was, in some years, unreliable. This generated problems with maintaining progress through the planned work and was frustrating for the students. These organisational problems are just the beginning. As a tutor, the author dealt with a range of personal issues raised by the students, from bereavement and illness to drug use and family disputes.

The author feels it is important to remember this personal dimension when thinking about educational research. As a researcher, it is very easy to be caught up in the vision of what the research may achieve. Ultimately, those taking part in the work are complicated human beings, with secrets and problems that the researcher may never know about. Their problems may have an impact on the student learning experience in ways that the researcher can neither predict nor mitigate. How well can a student learn to program if he or she has personal problems that drain his or her energy? It behoves a sensitive researcher to be aware of those he or she works with as unique individuals, not just 'research subjects' or 'students'.

For internal validity, we can accept the results based on the aspects tested or explored in the study. The threats to internal validity are, as defined by Cohen and Manion (1994), history, maturation, statistical regression, testing, instrumentation, selection and experimental mortality. These factors are worth briefly examining. It should be noted that Cohen and Manion (1994) are assuming a scientific approach for the research.

For the first factor, the timescale of the experiment may mean that other factors intervene and post-test changes are attributable to other events, not those managed by the researcher. For longitudinal studies, maturation is undoubtedly a factor that will affect results considerably. At certain points in a person's educational life, a month or six months will make a considerable difference to their abilities. This is a significant point to bear in mind when discussing the author's work: first year students aged 18 or so are generally adapting to a very different lifestyle, both personally and academically. The first semester or term might be the most difficult for some students. Many academic staff have stories of first year students who spend their entire loan in the first few weeks or fail to turn up to any classes for months. In teaching students, it is sometimes easy for staff to forget that the students may not be feeling settled, confident and motivated to learn.

Statistical regression means that in *"in pretest-post-test situations, there is regression to the mean."* (Cohen and Manion 1994 p 170) Such regression can be mistaken for experimental effects. Testing presents problems of its own in that it may sensitise subjects to the "*true purposes of the experiment…*" (Cohen and Manion 1994 p 171) Sensitisation has been briefly discussed above.

For instrumentation, poorly designed tests and inconsistent observation can skew results. Bias can be introduced inadvertently when subjects and control groups are selected. Bias introduced through choice of groups might also interact with the other previously mentioned factors. Experimental mortality refers to the drop out from a long-running experiment. An initially randomly selected sample may be self-selected at the end of the experiment. A qualitative, big Q approach may not encounter all the type of problems encountered by the hypothetico-deductive approach (such as statistical regression) but there are still issues that may be relevant (such as experimental mortality).

This section offers a very brief discussion and overview of a complex area: there are many detailed qualitative and quantitative tools available to the researcher. The key question is, which seems most appropriate to the task of researching the teaching of programming? The author has placed her work firmly in the realm of educational research. What value can educational research have?

The value of educational research

At its simplest, there are two views about educational research. Sax (1979) claims "*Reliable knowledge about education can be derived from the application of various research methods: analytic, descriptive and experimental.*" (Sax 1979 p17) In contrast, Burroughs (1971) said, "*Education* [] *does not go far in supplying conditions in which the scientific method can operate very powerfully.*" (Burroughs 1971 p 1)

For some, the scientific method is the only one that generates meaningful results. If educational research is not scientific then its work and conclusions are invalid and irrelevant. Such a view seems to regards educational research as a monolithic activity. In fact, there are many different approaches to educational research and many different methods and tools that can be employed within those approaches.

Verma and Mallick (1999) discuss the various definitions of educational research offered by authors from the 1950s onwards and conclude that a single, all-embracing definition of what properly constitutes educational research is not possible. Definitions change with prevailing fashions in research, education and the social sciences, (because there is an inevitable overlap with the field of social science).

Verma and Mallick (1999) do identify a common element in the widely varying definitions they survey: the idea of "*the application of systematic methods to the study of educational problems*." (Verma and Mallick 1999 p 33) Interestingly, Verma and Mallick add, "*It should be emphasized that there is no universal recipe for conducting educational research.* "(Verma and Mallick 1999 p 33)

Educational research can be seen as a matter purely of policy: a way to establish optimum strategies within a permanently embattled public sector: "*Educational policy is usually a matter of establishing the most efficient use to be made of scarce resources - time, building, intelligence, teaching skills and so on.*" (O'Connor 1957 p 76)

In contrast, "*In the late nineteenth and early twentieth centuries, a movement developed based on the conviction that education itself was to be studied as a science.*" (Giarelli and Chambliss 1988 p 30) How legitimate was that belief? Chambers (1992) has some harsh words for those who would elevate education and social science research to the status of the positivist natural sciences.

Chambers (1992) identifies empirical research with:

> "*observational, experimental and quasi-research of all kinds, statistical and correlational research, triangulation, cluster analysis, factor analysis, questionnaires...*" (Chambers 1992 p1)

Chambers (1992) rejects the idea that empirical research is scientific: educational empirical research "*uses exceedingly sophisticated statistical methods, but that sophistication is not enough to make it scientific.*" (Chambers 1992 p 5)

So, a quantitative, statistical approach is not enough to rescue educational research from its exile, outside the scientific paradigm. It does not matter how many questionnaires a researcher in education collates: the data is meaningless. This seems to be a dead end for all educational research: it simply is not worth doing! What distinguishes educational research from say, particle physics, is its emphasis on the human aspect. Ultimately, educational researchers are studying a human activity, within a social context. It is this idea of the significance of the social aspects of what is being studied and the importance of context that sets educational research (along with social science research) largely outside the scientific method but endows it with a different type of validity and meaning.

The scientific method is also distinguished by its emphasis on the neutrality of the researcher, within the research. In chemistry, it does not matter, who is doing the researching or where: the results are the same all over the world, given the same initial experimental conditions. In fact, this is not entirely true: there are many stories in what might be termed pure scientific research of extended and bitter debate among researchers about what it is they have observed. For one example of this, that of work on tumour-causing agents, see Kevles (1997).

What is different about educational research and social science research is that there is a philosophical stance that acknowledges the presence and wholeness of the researcher. This is the concept of reflexivity, discussed in the section below.

The concept of reflexivity

The notion of scientific truth does not embrace reflexivity (the researcher is not only wholly neutral, he or she is not really part of the research in a personal sense) whereas educational research paradigms generally need to engage with the notion of reflexivity. As Carr and Kemmis (1994) note:

> "*Social life is reflexive; that is, it has the capacity to change as our knowledge and thinking changes, thus creating new forms of social life which can, in their turn, be reconstructed. Social and educational theories must cope with this reflexivity: the 'truths' they tell must be seen as located in particular historical circumstances and social contexts, and as answers to particular questions asked in the intellectual context of a particular time.*" (Carr and Kemmis 1994 p 43)

What is reflexivity? One definition centres on the idea that how we see the world and how the world sees us are indivisible. The metaphor of seeing is appropriate and illuminating. If we wear coloured lenses, our view of external reality is coloured by these lenses and others' views of us include those coloured lenses. (http://www.neoscience.org/reflexiv.htm 13/01/2004) "*The observer and the observed are inextricably tied together in a reflection.*" (http://www.neoscience.org/reflexiv.htm 13/01/2004)

The concept of reflexivity is a complex and many layered one. Woolgar (1988) identifies the rise of reflexivity with the emergence of relativism in social sciences, while noting that the relationship of relativism to reflexivity remains "*confused*" (Woolgar 1988 p 18). Relativism is a philosophical stance that questions whether it is "*possible to translate the meanings and reason of one culture into another…*" (Woolgar 1988 p 17) Woolgar notes that the relativistic argument "*implies that human actions or beliefs are* less reliable *(valid, true) because they are specific to particular circumstances...*" (Woolgar 1988 p 18)

Willig (2001) notes,

> "*…it is now generally accepted that observation and description are necessarily selective, and that our perception and understanding of the world is therefore partial at best* []. *What people disagree about is the extent to which our understanding of the world can approach objective knowledge, or even some kind of truth, about the world. The different responses to this question range from naive realism, which is*

> *akin to positivism, to extreme relativism, which rejects concepts such as 'truth' or 'knowledge' altogether.*" (Willig 2001 p 9)

Woolgar (1988) identifies what the schema within which reflexivity operates. There must be both a distinction between what the reality of the object of research and its representation and there must be a similarity between those two things. The description or representation must be faithful but not so faithful "*as to be indistinguishable from it.*" (Woolgar 1988 p 20)

A faithful representation must also therefore embody distinctiveness from the object of research. It is this tension between similarity and distinctiveness that generates a dilemma for researchers. "*If...the means of study and the object of study are not distinct, their interdependence suggests that our research process assumes (at least part of) the answer it sets out to find.*" (Woolgar 1988 p 21)

Willig (2001) describes reflexivity:

> "*Reflexivity requires an awareness of the researcher's contribution to the construction of meanings throughout the research process, and an acknowledgement of the impossibility of remaining 'outside of' one's subject matter while conducting research.*" (Willig 2001 p 10)

Woolgar (1988) identifies a spectrum of types of reflexivity. Woolgar (1988) places radical constitutive reflexivity at one end and benign introspection at the other. (Woolgar 1998 p 21)

Constitutive reflexivity is a "*denial of distinction and an affirmation of similarity.*" (Woolgar 1988 p 22) Woolgar identifies this type of reflexivity as springing from the Garfinkel's 1967 work on the documentary method. (Woolgar 1988 p 21)
Benign introspection can be identified as the 'scientific' form of reflexivity. Science emphasises the differences between the researched object and its representation: they are seen as "*essentially different kinds of entity.*" (Woolgar 1988 p 22)

The kind of reflexivity that does not question this distinction is more about reflection on, or analysis of, the research process by the researcher than it is about a deep reflexivity. "*Far from raising any fundamental problem, this kind of reflexivity sustains and enhances the Scientific axiom of the research effort.*" (Woolgar 1988 p 22)

This is not to say that introspection and constitutive reflexivity are mutually exclusive. Introspection may reveal elements that are deeply reflexive, within the work. However, deep or radical reflexivity seems to have no place in research in the natural sciences. To argue otherwise would require a belief that physical aspects of the world, such as electrons had "*belief systems, their own theories of interaction and so on.*" (Woolgar 1988 p 23)

We can also identify the notion of a disengaged reflexivity that "*defers the application of the research to the researcher.*" (http://www.psy.dmu.ac.uk/michael/reflexivity.htm 17/01/2004) This is a post hoc reflexivity, in which the researcher leaves reflexivity as the task to be done after the research, rather than during it.

Social science research, in Woolgar's (1988) view, sits in the middle of opposing ends of the reflexivity spectrum. Social science research (like educational research) has a clear liking for, and varying adherence to, a scientific approach while still acknowledging the potential for similarities between the research object and its representation. Social science and educational research both can be said to blur the boundary between research methods and research object, in some cases.

Willig (2001) notes that there are two types of reflexivity: personal and epistemological. Personal reflexivity requires researchers to think about how their own belief systems, assumptions and prejudices have affected and/or moulded the research effort. It is a two way process: the act of researching must, by implication, change the researcher, perhaps in subtle ways that require more than benign introspection.

Epistemological reflexivity requires the researcher to contemplate questions about the research effort. Willig (2001) gives examples of such questions:

> "*How has the research question defined and limited what can be 'found'? How has the design of the study and the method of analysis 'constructed' the data and the findings? How could the research question have been investigated differently? To what extent would this have given rise to a different understanding of the phenomenon under investigation?*" (Willig 2001 p 10)

It seems to the author that researchers in education must engage with both and epistemological reflexivity, even if only to acknowledge some of the key issues, rather than making the whole research text a reflexive artefact.

Reflexivity can be seen as springing most strongly from ethnography and anthropology. Woolgar (1988) notes that, in anthropology, there is a tension between the culture being studied and the perceived validity of the anthropologist's report. Woolgar (1988) notes that if the culture being studied can be regarded as exotically other, the observer is then seen as privileged in terms of analytical distance. The anthropologist has access to epistemological tools not available to the observed and the greater the distance between the two elements, the more certain or valid the observer's report.

To perceive a culture as exotically other is to push reflexivity further from the research process. By extension, therefore, the more like the researcher the subject of the research is, the less privileged is the researcher's viewpoint. Woolgar notes that exoticism "*and the minimization of uncertainty are crucial to a wide range of representational practices.*" (Woolgar 1988 p 28) These stances inform the relationship between those who study and those they study.

We could ask, what elements of ethnography inform educational research, even at a remove? The idea of exoticism is one that denies reflexivity and places emphasis on the cool neutrality of the text describing the research. Reflexivity is not a feature of the text, and the potential or actual complexities of the text are neither acknowledged nor elucidated. Without articulating beliefs about the relationship between those studied and the researcher(s) and the role of the text as an artefact of that research, some crucial elements of the research endeavour are, in the author's opinion, missing.

The author perceives a distinct tension here between her 'scientific' stance in trying to formulate the various stages of her research and her description of those stages in the text and her centrality to the research. In describing the impetus for the research, the author identifies herself as a one-time novice programmer. A strongly reflexive view might follow: that what this research is doing is trying to articulate how the author *herself* learnt to program. The researched and the researched are ultimately so alike as to make any idea of difference unworkable, in the sense of a scientific method.

The idea of the role of language is central to reflexivity. Willig (2001) describes this as "*Critical language awareness*" (Willig 2001 p 10). We can identify a range of views about the potential transparency and neutrality of research writing, from those who claim it is possible to describe an agreed upon reality, that it is possible to say 'what really happened. At the other end are those who see the use of language as being of critical concern to researchers.

In this view

> "*language plays a central role in the construction of meaning and that it is the task of researchers to study the ways in which such constructions are produced, how they change across cultures and history, and how they shape people's experiences.*"
> (Willig 2001 p 11)

What this text is doing could be seen as sleight of hand – the author describes her experiences of novice programmers' experiences and accounts of their experiences, which give a realist illusion, while actually unfolding and delineating the author's own struggles with programming concepts. The layering of possible interpretation of this material becomes potentially dizzying.

The author feels it incumbent upon her to articulate a clear position with in the axes of exoticism/uncertainty and constitutive reflexivity/benign introspection. The question of the ways in which language is used to describe both the research effort and author's experiences (personal and within the research context) cannot be ignored. The text must be held up for inspection as a research object itself.

This presents problems for the author and her work. Her stance, within her chosen methodological framework of action research, is that of benign introspection rather than any more 'pure' mode of reflexivity. The author cannot exempt herself from scrutiny, in the way a physicist might: what the author believes about teaching and the teaching of programming, what she feels and thinks about the students she works with, all have a bearing, covert or otherwise, subtle or gross, upon the research process. Indeed, those elements are inextricably part of the results. What the author sees in her work may not be what another observer might see. What she concludes from (and about) her endeavours is intensely personal, however coolly presented.

Outline of the author's research

With such uncertainty is there anything to be gained from such research? Placing her work in an action research context means that the author is making clear an allegiance to a way of perceiving and thinking about her chosen area of interest. Her focus is the tension between what she believes or hopes to be the appropriate teaching of introductory programming concepts and the students' repeated problems with those concepts. "*Teachers' action research questions emerge from areas they consider problematic, from discrepancies between what is intended and what actually occurs.*" (http://www.ericfacility.net/ericdigests/ed355205.html 29/01/2004)

As noted earlier, the impetus for this work comes from the author's own experiences of firstly, learning to program and secondly, teaching programming to novice programmers. Her earliest work focused on using CAL to support the learning of programming concepts and this work is described in more detail in Chapter 6. The iterative cycle of carrying out some change and reflecting upon that activity, has led the author a long way from the tutorial CAL approach. This work is, in some sense, a description of that shift, from a positivist stance to one that attempts to take account of a range of factors, from the author herself to issues of individual learning. This thesis is an account of the ways in which the author has sought to engage with the problems of teaching programming to novice programmers, seeking ways to support the students' learning, addressing the problem in a number of ways.

This account, overall, constitutes one centred on benign introspection rather than the more dramatic constitutive reflexivity. The account also perhaps gives a retrospective 'shape' to the research effort. At the time, it seemed to the author that each effort ended in failure but in hindsight, this is a harsh judgment. If the author accepts the premises of action research, than each cycle of doing and reflecting represents a development of the author's understanding, a moving closer to what is most promising. The sense of frustration often felt by the author can either be regarded as a natural result of her engagement with the research or as proof of the intractability of the problem. In action research, the first view is surely the most appropriate.

The earliest research effort involved the writing of a piece of tutorial CAL, called Introduction to Programming or Introduction to Programming Concepts. (The CAL went by both titles during its life!) This covered programming concepts, in the style of the CAL being written by the CAL development team. In fact, the package was completely rewritten and extended so two quite different versions were created. The CAL was trialled with foundation level students and some feedback gathered on their perceptions of the package.

The author then felt that the possibilities of this type of CAL for supporting novice programmers were fairly limited. Her work in teaching Pascal as a first programming language lead her to believe that the Pascal help system was a significant stumbling block for new programmers and she decided to develop a more focused and specific approach to providing a help system. This help system used a key word input to trigger very basic help text, written by the author.

The author hoped that students would use the help system and that as they developed their programming practice. She envisaged the help system growing, as the program stored queries that had no results. If a student entered a completely new query, the author envisaged adding the help text within a very short time so that at the end of the first semester's use of it, a fairly broad set of query results had been created. Again, students were asked to rate the use of this resource. In fact, no student used it.

These first two efforts are described in more detail in Chapter 6. Having tried these two approaches with some little q data gathering, the author felt it was time to take stock of her approaches. Her stated aim was to carry out action research using a constructivist theory of educational practice. Her first CAL effort had been anything but constructivist. Her help system was a closer approximation to the spirit of constructivist effort but no matter how good the fit between artefact and theory, if students do not like or employ the system, it is of no use, either to students or to the researcher.

In summary, the first two major stages of her research proved to be dead ends. Tutorial CAL did not address the key issues in any way that the author could perceive as meaningful. The on-line help strategy moved closer to a constructivist approach bit did not model a truly emancipatory style of action research, because the filling in of the help text would be done by the author rather then the students/participants.

This caused the author to review both her approach to research generally and the kind of intervention she was attempting. It seemed to the author that the tensions between the teaching and learning strategies she had undertaken and the goal of a constructivist approach required her to re-think what she was doing. There is no disgrace in this: action research requires action and reflection upon that action but a more definite change of strategy was required.

With the discovery of the idea of patterns and pattern communities, the author felt that she had found a genuinely promising approach. Here was a structure that was coherent but flexible, and an approach that had been used in related areas, such as software development by experienced programmers and in general pedagogical way.

The author also did not find any examples of a pattern community being fostered among novice programmers, to support the learning of a first programming language.

That is not to say that such work has not or is not being done but that the best efforts of the author did not find any published material around this specific area.
For the latest iteration of her research on teaching novice programmers, the author decide to use the concept of patterns with a new cohort of first year students. The author commissioned a small software structure that would hold simple patterns and be updatable and searchable by the students. These students were taking the Introduction to Programming unit but for the first time in a number of years the author would not be teaching the unit. This presented a number of problems, discussed later in this thesis. The overall aim of presenting students with the opportunity to develop patterns of their own to be shared with their programmer community was to make more overt the knowledge sharing, to make more explicit the constructivist aspects of learning to program. The author's work around the use of patterns with these students is described in Chapter 7.

The next section introduces and describes the concepts of patterns and pattern communities and their history and outlines some of the work done on using patterns in software development and in pedagogical practice.

The concept of patterns and pattern languages

Patterns come from the work of Christopher Alexander, an architect who created the concept of patterns and pattern languages and described patterns for creating buildings and living spaces.

> "*Alexander extracted patterns from building and town structures of numerous cultures. He believed that by documenting these patterns, he could help people shape the buildings of their community to support life to its fullest.*" (Coplien 2000 p 3)

Alexander describes patterns as having three parts, which together describe the relationship between a problem, its context and a solution. This is the Alexanderian (or Alexandrian) pattern: other pattern styles have evolved as other disciplines have taken up the idea of patterns and pattern languages.

Pattern languages are collections of patterns that address asset of related issues. Alexander (1979) describes pattern languages as being like ordinary languages. Both are "*finite combinatory systems which allows us to create an infinite variety of unique combinations, appropriate to different circumstances, at will.*" (Alexander 1979 p 187)

It is important to note that Alexander's (1979) concept of patterns is much more than a problem-solving approach or clever packaging of a mental trick. Central to the idea of patterns are the concepts of the quality, the gate and the way. The quality is as Alexander notes, hard to define but not vague, difficult to describe but instantly recognisable, in more than just places or buildings: "*There is a central quality which is the root criterion of life and spirit in a man, a town, a building or a wilderness.*" (Alexander 1979 p ix) It is the quality that is the focus and the driving force in patterns.

The gate is the generating of that quality through a vital, shared pattern language. The gate must be created to give access to what Alexander terms the "*timeless way*" (Alexander 1979 p xii) He defines the timeless way as "*a process which brings order out of nothing but ourselves; it cannot be attained, but will happen of its own accord, if we will only let it.*" (Alexander 1979 p 2)

As we can see, Alexander's concept of patterns is much more than a framework for solving problems. There is insufficient room to discuss in detail Alexander's work: see *The Timeless Way of Building* (1979) for a readable and interesting exposition of his own ideas by Alexander. Alexander (1979) notes that pattern languages are universal:

> "*For the use of pattern languages is not merely something that happens in traditional societies. It is a fundamental fact about our human nature, as fundamental as the fact of speech.*" (Alexander 1979 p 198)

Both software development and pedagogy practitioners have adopted the idea of patterns, pattern languages and pattern communities with enthusiasm. It would be fair, in the author's opinion, to say that proponents of pattern languages in those arenas have adopted the structural, mechanical aspects of patterns without necessarily subscribing to the more philosophical, even mystical, elements of the idea. At its simplest we can say that a pattern language is the set of ideas that someone has in their head about how to do something, whether than thing is to build a barn or write software. Alexander notes that an "*act of design, whether humble, or gigantically complex, is governed entirely by the patterns* [the person] *has in his mind at that moment, and his ability to combine these patterns to form a new design.*" (Alexander 1979 p 203)

It is these patterns that enable and release, not constrict, a person's creativity. It is the pattern language for a particular endeavour that ensures that meaningless or unproductive approaches are rejected. Alexander uses the example of the rules of English: it is the pattern language for sentences in English that save the speaker or writer from struggling with incoherence.

For Alexander (1979) the fact that patterns can be expressed very simply does not rob them of their power. For Alexander, the ordinary and simple are what creates the most ineffable beauty: the example he gives is of old Turkish prayer rugs. (Alexander 1979 p 220) He sounds a note of caution: the fact that the rules are in themselves simple does not mean that "*they are easy to observe, or easy to invent.*" (Alexander 1979 p 222)

For Alexander, the critical test of a pattern's rightness, its quality is the way it makes someone *feel.* A pattern generated by intellect alone is likely to be dead. Alexander (1979) cites the example of a student's pattern about balconies in mental hospitals. This pattern, he says, is laughable, even though it conforms to the pattern structure. (Alexander 1979 p 283)

Patterns must pass the test of feeling. Patterns that possess the quality without a name "*make us feel good, because they help to make us whole, and we feel more at one with ourselves in their presence: but it is still the quality itself which matters most...*" (Alexander 1979 p 297)

A pattern language is not a static thing: it evolves, of its own accord. Good patterns flourish, by being widely shared and bad patterns die out. Each pattern stands alone so can develop or be rejected without affecting the stability and validity of the pattern language overall. Alexander says it is this fact that "*guarantees the evolution of pattern languages will be cumulative.*" (Alexander 1979 p 345)

Others have sought to define patterns and pattern languages. For example, Bergin defines a pattern language as a "*set of patterns that work together to generate complex behaviour and complex artifacts, while each patterns is itself simple.*" (http://csis.pace.edu/~bergin/patterns/fewpedpats.html 09/03/2003) Bergin also states "*A pattern is supposed to capture best practice in some domain.*" (http://csis.pace.edu/~bergin/patterns/fewpedpats.html 09/03/2003)

What distinguishes a pattern from a technique? Patterns seek to identify the essence of a problem and to produce an answer that captures the solution to a problem that may occur in many different guises and contexts. "*The essence of a pattern is that the situation to which it applies recurs in different circumstances.*" (Eckstein and Voelter 2001 p 2)

Coplien (2000) says "*The pattern foreshadows the product: it is the rule for making the thing, but it is also, in many respects, the thing itself.*" (Coplien 2000 p 3)
The specific structure and elements of a pattern vary, depending on the problems they address. Patterns created by educators (pedagogical patterns) seem to have a simpler form that those for say, software development.

Patterns, software development and teaching

The arrival of patterns in software development (particularly the object–oriented approach) can be attributed to the work of two people, Ward Cunningham and Kent Beck, who worked on five simple patterns for object-oriented design in Smalltalk in 1987.

Cunningham and Beck found that Smalltalk users produced a usable interface design, employing the five patterns to solve design problems. Cunningham and Beck presented their findings at OOPSLA 87 in Orlando, America. Although there was enthusiasm for the idea of patterns in software development, there was not much practical development until 1991when Richard Helm and Erich Gamma worked on a set of patterns, which they presented at conference in 1991.

During 1991 and subsequently, a group of software developers with an interest in patterns worked in close collaboration. These were known as the Gang of Four, co-authors of the book Design Patterns but by 1993 there was widespread interest in patterns and many others in industry and academia were developing, researching and publishing patterns. In 1994, the first PloP (Pattern Languages in Programming) conference was held. For a detailed history see http://c2.com/cgi-bin/wiki?HistoryOfPatterns.

Patterns have also moved out into pedagogy. There is strong interest in patterns to teach the object oriented approach but more general patterns are emerging such as patterns for teaching seminars, delivering lectures and testing students. See Bergin's work on pedagogical patterns at http://csis.pace.edu/~bergin/patterns/fewpedpats.html. For a pattern language on the effective teaching of seminars see the work of Fricke and Voelter (2000).

Eckstein and Voelter (2001) note that,

> "*Pedagogical patterns attempt to capture the knowledge of experienced teachers in a way that can be applied by novice teachers. Patterns can also capture expert knowledge of teaching techniques in a certain domain. This expert knowledge can be valuable to experienced teachers who are unfamiliar with a specific domain that they have to teach.*" (Eckstein and Voelter 2001 p 2)

Bergin describes the Design Pattern community (those involved in PloP in its various forms) and those working with pedagogical patterns as "*two movements, with different goals*" (http://csis.pace.edu/~bergin/papers/SimpleDesignPatterns.html. 07/08/2003).

Bergin describes the Design Pattern community as looking at patterns for what should be taught and the Pedagogical Patterns community works on patterns for how to teach. (http://csis.pace.edu/~bergin/papers/SimpleDesignPatterns.html. 07/08/2003)

Bergin has also worked on patterns for beginners in Java. He has written some of his own and pulled others into the collection at http://csis.pace.edu/~bergin/patterns/codingpatterns.html. Bergin notes that the collection contains different patterns that address different areas of coding difficulties or requirements such as readability and maintainability. (http://csis.pace.edu/~bergin/patterns/codingpatterns.html 09/03/2003)

Pattern syntax varies from pattern language to pattern language. Some pedagogical patterns are very simple: a pattern name, a short problem statement and the solution flagged with a bold face **Therefore** to introduce the solution. Bergin's patterns have a confidence level asterisk attached ranging from 0 for no confidence in the pattern (it could be substantially improved) to 2 asterisks for a pattern that contains what seems to be a fundamental answer. Bergin identifies sixteen elements of a pattern that make up the pattern syntax: Name, Author, Intent, Other Names, Problem, Motivation, Context, Example, Required Elements, Structure, Variations, Applicability, Consequences, Known Uses, Related Patterns and References. It should be noted that Alexander et al. (1977) in A Pattern Language do not have this rigid structure for pattern.

The pattern elements (Alexander et al. 1977) encompass discursion, description, anecdote, sketches, photographs and statistics. The patterns have sense of meaningful fluidity about them. Each pattern conveys complex meaning in the way that seems most appropriate to that pattern. Perhaps as the idea of patterns has moved out into other subjects, the richness of the idea has been somewhat lost. Ironically, in widening the application of pattern languages, the sense of the essence of patterns - the quality - seems to have been sacrificed for syntax and a fixed structure.

Interestingly, Bergin offers examples of patterns that address programming design issues such as outputting columns of values, dealing with nested IF statements and handling linked lists. (http://csis.pace.edu/~bergin/papers/SimpleDesignPatterns.html 07/08/2003)

In summary, the idea of patterns as it was first put forward by Alexander et al. (1977) has a richness and a quality of elegance that seems missing from its later incarnations. Perhaps there is a case for developing a much richer collection of patterns to support novice programmers: patterns that incorporate commentary, video and images. Before attempting this, the author felt it appropriate to 'start small' with a more limited but manageable pattern structure. See Appendix C for the pattern input screen. The research done by the author around the students' responses to this artefact are described in Chapter 7.

Chapter 2: Conclusion

This chapter has placed the author's research in a methodological context. Firstly, the author identifies her work as being within the humanistic rather than positivist arena. Within that, she notes her work as being within the remit of educational research, and her preferred methodology is that of action research. From a choice of methodology come further implications, for the type of data gathering/analysis methods and tools that are appropriate. The author noted that her approach was qualitative rather than purely quantitative. The chapter described a small sub set of qualitative data gathering methods and went on to discuss the concepts of internal and external validity, in the context of educational research. The author also presented a brief discussion of the value of educational research.

The chapter also described a key concept in research that adopts a non-positivist stance, that of reflexivity, from benign introspection to constitutive reflexivity.

Finally, the chapter outlined very briefly the author's earlier research efforts and focused on the idea of developing a pattern community among novice programmers, concluding with an outline of some key ideas in patterns and the use of patterns.

Introduction to Chapter 3

This chapter discusses a range of issues associated with the teaching of programming to novice programmers. It outlines some of the key issues in teaching structured programming concepts to non-specialist or novice programmers at a higher education level. An initial distinction is made between learning to program as a goal-directed activity (learning to become a programmer) and learning to program as a developmental activity (learning to think algorithmically and/or develop problem solving skills). The chapter concentrates on the first category but the work of Papert, which falls into the second category, is also described in this chapter.

The main elements of structured programming, as embodied in languages such as Pascal, are outlined. It should be noted that newer programming paradigms such as object-oriented design (and languages such as Java) are mainly excluded from the discussion. Given an agreed subset of concepts to be conveyed to novice programmers, other issues then arise, such as the choice of first programming language, which is discussed in this chapter.

Finally, the chapter examines some of the human aspects of learning to program, such as mental models, confirmatory bias and attention blindness.

Learning to program versus programming to learn

A distinction needs to be made at this early stage between two desired outcomes in teaching programming. Teaching programming, at various levels, can be seen as a way of fostering and developing transferable skills, including the learner's meta-cognitive skills. One goal of educational practice may be to develop the learner's ability to reflect upon his or her learning strategies, in order to enable him/her employ such strategies more effectively and in a more discriminating manner. The task of making the learner's thinking available to them for examination and analysis is complex.

Some authors see programming as one way to foster such meta-cognitive skills. As Mendelsohn et al. (1990) note, "*Perhaps there really is such a mental exercise, and it is just possible programming is it.*" (Mendelsohn et al. 1990 p 194) We could describe this as the strong 'programming to learn' position. Guzdial (1994) notes "*...we are even more interested in having students learn through programming because we recognize that programming is a good lever for understanding many domains.*" (Guzdial 1994 p 1)

The other outcome is the result of teaching students to program successfully, in a specific language, so that they become, ideally, skilled programmers who can continue to develop their programming skills, and apply their higher-level understanding to almost any programming language. Along with the syntactical and semantic features of the language, students usually learn programming concepts such as the algorithmic toolkit, good program design (however that is defined!) and aspects of the software lifecycle.

Another way to express this is to talk about the cognitive *effects* of learning to program and the cognitive *demands* of programming. In the second case, students learn to use the programming language appropriately in a range of (ideally) non-trivial tasks. It is on this second role of programming that this chapter concentrates but the first view is also discussed when examining certain languages such as LOGO.

Having said that it is learning to program that is the focus of this chapter, we could then ask, what aspects of computing and programming do we need to teach to a novice programmer? Should programmers learn computing rather than just programming in a particular language?

Computing and programming curricula

It is worth noting that there has been debate about what properly constitutes a study of computing. Computing is more than learning to program, although programming (or at least an appreciation of programming concepts) is a central aspect of computing. Denning et al. (1989) describe as strong the view that computer science is mainly about programming but their report for the ACM makes it clear that the field of computer science has "*intellectual substance*" (Denning et al. 1989 p 9) that encompasses far more than programming.

For Denning et al. (1989) the phrase 'discipline of computing' covers both computer science and computer engineering. Programming is seen as part of the core curriculum and competence in programming is a key goal of any curriculum. Introductory courses in computing would introduce "*programming, algorithms and data structures (and) materials from all the other sub disciplines as well.*" (Denning et al. 1989 p 13)

For programming languages, Denning et al. (1989) place under the heading of theory elements such as formal languages, Turing machines and formal semantics. Under the heading of abstraction are: classification of languages (by syntax and application), methods for compiling and abstract implementation models. Under design, Denning et al. (1989) list procedural functional and object oriented languages, programming environments run-time models computers and software debugging and tracing. (Denning et al. 1989 p 18) A programming curriculum for the novice will probably encompass only a small subset of the broader collection defined by Denning et al. (1989).

The balance between theoretical, abstract and design elements, is, as we shall see, a difficult one to achieve. There are questions of what subject areas are most appropriate to beginners in programming and issues around the order and emphasis of those subject areas. It could be argued that students need a broad view of the subject before they begin to program in any language or conversely, that students should be familiar with a small collection of programming ideas before approaching concepts such as Turing machines or formal semantics. As DePasquale notes, "*Novice programmers generally are introduced to simple, stand-alone concepts that can be easily digested and understood.*" (http://people.cs.vt.edu/~edwards/RESOLVE2002/proceedings/DePasquale.html 04/02/2003)

A course that taught both programming concepts and wider software development concepts (such as software lifecycle models) might be termed a software engineering course. The term 'software engineering' implies a coherent, clearly defined, well-understood set of rules that can be rigorously applied: programming as a science rather than a craft.

Shaw (1990) describes how the phrase 'software engineering', coined in 1968, has stood more as a goal than a definitive description. Shaw (1990) defines how an engineering discipline evolves, from a haphazard enterprise through a craft-based approach to a professional theory-based set of practices and guidelines. The idea of software engineering is a response to the software crisis of the 1960s that was briefly discussed in Chapter 1.

Shaw (1990) identifies a critical lack of models and theories that directly inform and support practice in the teaching of computing generally: "*Computer science has a few models and theories that are ready to support practice, but the packaging of these results for operational use is lacking.*" (Shaw 1990 p 21)

In summary, then, deciding what to teach novice programmers is not necessarily clear cut or beyond debate. What needs articulating is a position on what might constitute a useful and relevant syllabus for teaching programming to novice programmers.

A programming syllabus

Teaching programming is about more than teaching the syntax and semantics of a specific language, although a programming language is generally regarded as an essential element. The vexed (and often debated) question of *which* programming language to teach first is discussed later in this chapter. Allied to the use of a language must be a set of programming concepts. It is possible to identify a small set of concepts that most teachers of programming would regard as indispensable. These concepts could be summed up as the core of the structured programming approach and these are outlined in more detail in the next section.

Around these concepts might be wrapped a wide variety of related materials, from the history of programming to a detailed examination of the software lifecycle.

Sample introductory programming syllabuses may make reference to the practical elements of language use as well as concepts such as constructs (control structures) and data types. (http://www2.sis.pitt.edu/~peterb/0012-031/syllabus.html 13/02/2003, and www.bc.edu/bc_org/avp/eve/02-03mt35001fall.pdf 13/02/2003) Others seem to emphasise more relaxed, creative aspects of introductory programming. (http://www.cs.iastate.edu/~leavens/ComS227.html#Overview 13/02/2003)
Others focus on language-independent, structured programming concepts. (http://pizza.cs.ucl.ac.uk/teaching/html/Programming-I.xml.html 13/02/2003)

There are different flavours of approach that can be distinguished. One is to teach novice programmers about the machinery of computing, both physical (storage and buses) and logical (XOR, AND) and embed the programming elements within a (hopefully) broader understanding of how a program relates to the device upon which it runs. Another is to teach the detailed mechanics of a language and embody general principles in both extensive use of examples and student practice.

The converse approach is to teach structured programming principles that just happen to find expression in a specific language but with much less emphasis on learning fine or low-level details of the language. An example of this is error handling: Pascal does not have the sophisticated in-built error handling of Java. Students who want to trap input errors can do so by building the appropriate constructs or using some of Pascal's features but the time and effort spent on developing detailed error handling code can be better spent understanding the concepts of error handling, rather than seeking to implement a fine-grained solution.

Which approach a novice programmer encounters will depend on a wide variety of factors. Institutions do model, either explicitly or implicitly, their beliefs about what computing or programming is, within that institution, in their syllabuses. Some introductory programming syllabuses will embody a department's or faculty's belief in the broader approach, because computing machinery (what could be called computer science) is the focus of the department's interest or research. Other syllabuses will focus on the principles of structured programming as a basis for exposure to other languages. Others may take this more general approach, because of what they perceive as the initial levels of interest, skill and ability of the students.

Within this diverse collection, questions about the order in which concepts are introduced arise. What do you teach first? Next? After that? Allied to this, would always be the issue of how those ideas are introduced and how the students' acquisition of those concepts is supported and developed. Finally, there is the question of assessing the learning that has taken place, which usually means grading student programming effort.

Having said that there are many ways to structure and present a programming syllabus, the author believes that there is a core set of concepts which must be covered, whatever the emphasis of the curriculum. In summary, most programming students need to know about the algorithmic toolkit and this is outlined below. All *imperative* languages rely upon the concepts of the algorithmic toolkit: languages such as Prolog (which is a declarative language) do not. Imperative languages require the programmer to define clearly every instruction and the associated flow of control within a program.

Elements of structured programming

It is worth noting that programming has not always been structured! There is insufficient room here to review the history of coding and programming languages but the earliest coding efforts of the second half of the twentieth century owed more to individual, even idiosyncratic, decisions than to a codified set of guidelines or clearly articulated practices.

The impetus to move programming towards a much more formalised activity came from what is usually termed the software crisis of the 1960s, as noted in Chapter One. The response to the range of issues and challenges presented by the crisis was to examine the ways in which programmers developed code. Structured programming represented the first steps at formalising the stages and activities of programming.

One of the critical starting points for the debate about structured programming is generally held to be Edsger Dijkstra's article *Go to statement considered harmful* which was first published in the Communications of the ACM Volume 11 November 3 March 1968.

Edsger Dijkstra's article on goto

Dijkstra opened his article on the use of goto with the statement:

> "*For a number of years I have been familiar with the observation that the quality of programmes is a decreasing function of the density of go to statements in the programmes they produce.*" (http://www.acm.org/classics/oct95/ 20/10/2000)

Dijkstra marshals his arguments against the use of goto in code. Goto is a construct that allows a programmer to specify a jump to another part of the program, depending on a condition. Pascal supports goto and labels. The programmer can label a section of code and then write code to jump to that label.

For example

```
if reply = 'y' goto posreply
.
.
.
posreply writeln('You entered y');
```

The use of goto, Dijkstra says, effectively hands over the control of the process created by the program to the machine. This delegation is not appropriate: the programmer should have control of the process through the correct and predictable execution of the code.

Secondly, human beings are poor at grasping processes changing over a period of time. The program code (static, relatively easily grasped) should be as close to the (dynamic) processes it generates as possible. As Dijkstra says, "*the correspondence between the program ... and the process ...* [should be] *as trivial as possible*" (http://www.acm.org/classics/oct95 20/10/2000)

It should be noted that, even some years after Dijkstra's work, goto could be found in code presented as an exemplar. Adams' (1973) program example, presented as a legitimate piece of code for study, contains the dreaded goto, twice!

```
SUM: = 0;
     I: = 1;
LOOP IF I > N THEN GO TO STOP;
     SUM: = SUM + × (I);
     I: = I + 1;
     GO TO LOOP;
STOP SUM/N
```

(Adams 1973 p 5)

The use of the construct goto generally means that monitoring the progress of a program becomes very difficult for a human being. It is possible to define the program's progress by specifying the number of statements executed (what Dijkstra calls a kind of normalized clock) but meaningful description of where the program is in terms of what it has done is very difficult. Dijkstra describes the go to as "*just too primitive; it is too much an invitation to make a mess of one's program.*" (http:/www.acm.org/classics/oct95/ 20/10/2000)

The analogy that is frequently used is that of spaghetti code versus lasagne code: "*A program that is chock-full of unconditional GOTOs is often called spaghetti code, because tracing through it is like trying to separate strands of interwoven pasta.*" (http://www.chisp.net/~dminer/c64/gazette/8801/8801090.html 04/02/2003)

Structured programming aims to produce lasagne code: code with clearly defined, differentiated layers, that can be taken apart cleanly!

The algorithmic toolkit

Structured programming relies on the algorithmic toolkit, which encapsulates the elements that allow the specification of the program's algorithm. A program is sometimes described as being made up of the algorithm and one or more data structures.

The algorithm is the series of steps or instructions that define what the program has to do. A data structure is the way in which the data is stored by the program. All algorithms are made up of a combination of four basic ideas, modelled as one or more constructs.

Constructs

SEQUENCE

The simplest type of algorithm is one where there is a sequence of steps to be carried out one after the other. Once the last step is done, the program is finished.

SELECTION

Selection means that one action is carried out rather than another depending on a test built into the program. The key constructs for selection are:

```
IF condition THEN
        action
```

and

```
IF condition THEN
        action
ELSE
        different action
```

The other selection construct is CASE. Most structured programming languages support some implementation of the CASE statement although the same multiway branch can be built using nested or separate IFs.

REPETITION

Some programs need to carry out one or more actions several times. Repetition allows the program to loop many times, according to the program instructions. Loop constructs include WHILE condition DO action and FOR COUNT DO action. In the WHILE construct, as long as the condition is true, the statement (action) or block of statements will executed repeatedly until the condition finally evaluates to false.

The FOR loop allows a loop to execute a specified number of times. This is useful when the number of repetitions is known in advance. Different languages support slightly different loop constructs. Pascal does not support DO action WHILE condition but does have a REPEAT action UNTIL condition construct. The condition is tested AFTER the statement in the loop is carried out. This means that the statement is executed at least once, whatever the initial value of the condition.

PROCEDURE

Larger programs can be broken down into a collection of smaller chunks of code called procedures. (Different programming languages have different names for these chunks of code.) The idea of modularisation of code is central concept in structured programming and requires an understanding of the scope of variables and parameter passing.

Algorithms

The algorithmic toolkit is used to build algorithms. The essential characteristics of algorithms are that when each step has been executed, the next step is to be executed is clear and the algorithm has a clearly defined starting point and one or more clearly defined stopping points.

In all instances the algorithm will terminate after a finite number of steps and the algorithm is composed of primitive operations whose meaning is clear and unambiguous to the person or machine executing it.

Algorithmic primitives provide an unambiguous way of expressing instructions that can be easily described in a specific programming language. Primitives are verbs such as read and display.

Variables

The scope of variables is an important concept and closely linked to the idea of parameter passing. Global variables are those declared at the top of a program and are accessible to all procedures and the main program. This can generate unexpected side-effects as procedures alter in turn the values of the global variables. Local variable are those declared for the main program and within procedures. One way to avoid global variables is to use parameters.

Modularity and parameters

No teaching of structured programming would be complete without an introduction to modularity.

> "*A modular system is one that consists of a number of small, coherent, chunks (or modules) each of which is self-contained and has a well-defined interface.* [] *Modularity helps to reduce complexity. A modular system can be understood in terms of the individual modules, without having to grasp all the details at once.*"
> (http://www.louisepryor.com/showTopic.do?topic=6 13/02/2003)

Students need to understand the principles of modularity for several reasons. Firstly, any non-trivial project will require a dividing of the program's activities into chunks or modules of code. This may represent a team effort, with modules being written by different team members or it may still be an individual piece of work that requires the modules to exhibit appropriate levels of coupling and cohesion. Coupling is a measure of the interactions between modules, which should, ideally be tightly controlled and kept to a minimum. Cohesion is the grouping of logically cohesive tasks within a module: a module should perform one coherent task or a very few activities which are logically grouped.

With modularity comes the issue of moving data between modules and the main program. It does represent a new level of complexity and challenge for the novice programmer but I do not believe any one can claim to understand (or undertake) structured programming without understanding parameter passing.

Parameters are used to pass data between the main program and the procedures. Formal parameters are specified when the procedure is declared. Actual parameters are passed to the procedure when it is called. In Pascal the simpler form of parameters are the value parameters which do not affect the values of the variables passed in to the procedure. Value paramters use copies of the variables passed to them.

Var parameters in Pascal pass back the changes made to the variables inside the procedure. The difference beween value and var parameters is significant but is not an easy concept to grasp. Parameters are implemented in slightly different ways in other imperative languages but the concept remains the same.

It should not be assumed this is an exhaustive or fully explicated list. There would have to be other material that would support and develop these core ideas. Other concepts related to structured programming include the modelling of a discrete set of stages in program development, referred to as the software lifecycle or the software development model. As noted earlier, what other elements are taught will depend on the tutor, the first language chosen and the scope and goals of the course.

This is not to say that the concepts noted above are sufficient only and in themselves. Students will need, for the purposes of developing good coding habits, exposure to clearly articulated elements such as appropriate commenting styles and program layout, writing pseudocode (or some other specification approach) and use of a programming language editor and compiler.

Some comments on programming

Programming is both a vast industry (the US government will spend an estimated 63 billion dollars a year on IT by 2007 (http://news.com.com/2100-1017-913223.html 04/03/2003) and the UK government 12.3 billion pounds in 2003 (http://www.kablenet.com/kd.nsf/Frontpage/ 03/02/2003)) and a possible hobby. A programmer may be writing one element of a safety-critical system, coded in several million lines, or a twenty-line program at home on a PC.

There are undeniable qualitative and environmental differences between these activities but the essence of the challenge remains similar: to write stable, maintainable code that does, reliably and predictably, what it is supposed to! Of course defining terms such as 'stable' or 'maintainable' is problematic and the question of specifying program activity presents difficulties and challenges in itself.

There is a huge and diverse population of programmers, from academics, industry practitioners and those who program for fun: all these groups are represented in postings about programming on the Web. An Internet search for material on teaching and learning programming throws up some serious work around the subject as well as personal material. "*Programming is like riding a bicycle – a few days of grazed knees, and then puzzlement as to why it looked so difficult.*" (http://www.csc.liv.ac.uk/~ken/cpp/prog1.html 03/05/2000)

Some are less sanguine:

> "*Unfortunately, experience shows again and again that learning programming in Higher Education is far from easy. Students continue to struggle, and many academics will have found themselves faced with final year students who profess a complete inability to undertake even the simplest programming task. Something is clearly wrong.*" (http://www.ics.ltsn.ac.uk/pub/conf2000/Papers/jenkins.htm 13/02/2003)

We might ask ourselves why programming is so popular? What is it about this activity, however it is constituted, that it appeals so strongly to some (not all!) people?

Blum (1996) says:

> "*Programming... has a very large proportion of fun. That is why it is a hobby. It offers the perceptions of creating a product; when the product performs in a predictable fashion, there also is a sense of having accomplished something useful.*" (Blum 1996 p 249)

Johnston (1985) says, "*Learning to program is bit like learning to play football; anyone can do it badly but doing it well is a lot more difficult.*" (Johnston 1985 p ix)

Others such as Hawksley (1998) and Rosin (1973) use the musical instrument analogy: "*Teaching someone <u>how</u> to program is similar to teaching him to play a musical instrument: neither skill can be taught, they must be learned.*" (Rosin 1973 p 83)

Other writers take a more academic view:
"*Why is programming hard? Getting a computer to do what we want, programming it, means building a very special kind of machine - an immaterial machine.*" (Rawlings 1997 p 63)

Shneiderman (1980) says "*Programming is neither simple nor unidimensional; there is a rich and varied texture of tasks which fit under the term programming.*" (Shneiderman 1980 p 41)

Rawlings (1997) goes on to say:

> "*... programming is much more like writing, sculpting or painting than it is like building a house... no other engineers have machines they can turn into helicopters or toasters on a whim.*" (Rawlings 1997 p 67)

Johnston (1985) says, to be a good programmer, "*Three factors are involved: natural ability, coaching and practice.*" (Johnston 1985 p ix) Reynolds (1981) produced a book called *The Craft of Programming.*

This could lead us neatly into the debate about programming as an art or craft versus programming as engineering. There is no room to review the arguments here. The author takes the stance that structured programming and its concepts are sufficiently well established to allow for the treatment of programming as a technical subject, capable of being taught in an academic context, rather than passed on in some more diffuse, apprentice-style manner.

What is of interest here is that there does seem to be a view (not always explicitly acknowledged) of programming as a craft, something that requires both learning and a modicum, at least, of talent, a feel for programming. It is this view of programming as an art, the province of the talented few, that needs some elucidation: how far is it embedded within the mind- sets of novice programmers and what effect does that view have upon their learning?

Having earlier defined what we would wish to teach as a minimum to novice programmers, we now need to widen the discussion. One of the critical questions is, what language should we use to first embody and animate those structured programming concepts?

Which language should be the first programming language?

Shneiderman notes that in the mid to late 1970s there were about 166 programming languages in use. The Language List (http://cuiwww.unige.ch/OSG/info/Langlist 04/12/2000) lists over 2 300 published languages. That does include proprietary versions of languages (such as Pascal and Turbo Pascal) and variations or dialects but it can be seen that the growth in the number of languages of all types has been considerable over the last 20 to 25 years.

The proliferation of languages available and the diversity of choice are best summed up by Bowman:

> "...*there seems to be very little consensus as to the most suitable first programming language to teach. At a Workshop on the topic organized by the Open University ...at least 16 different first languages and programming notations were advocated. Included amongst these were traditional imperative languages, such as Turbo*

> *Pascal...; more modern imperative languages, such as Modula-2..., Modula-3 ...Ada ...and Extended Pascal...fully fledged object-oriented languages, such as Eiffel..., C++ ...Omega ...and Cool...and more formally oriented languages and notations, such as ML ...and VDM....* " (http://www.ulst.ac.uk/cticomp/bowman.html 28/11/2000)

Cunniff, Taylor and Black (1989) describe the creation and use of FPL (First Programming Language). The authors describe some of the vital work done to compare its effectiveness a first language with Pascal. They conclude:

> "*It would be impossible to make any definitive judgements on whether FPL or Pascal is a more appropriate language for beginning programmers based on the small number of programs analysed for this study. However the differences that apparently exist suggest that there may be some important advantages to the use of FPL. The avoidance of common syntax-related bugs involving BEGINS and ENDS.*" (Cunniff, Taylor and Black 1989 p 428)

In the 1970s and early 1980s, BASIC was extremely popular among home computer users. Stansifer (1995) notes "*BASIC spread widely, aided by the personal computer revolution. The simplicity of BASIC appeals to the many people who want to use the computer, but do not wish to learn programming.*" (Stansifer 1995 p 25) BASIC is unlikely to be recommended as a first programming language but for many computing enthusiasts, it served its purpose. We could add to the list of the available and popular languages regularly: in the last few years Java has come strongly to the forefront, for a variety of reasons. There are also the markup languages such as HTML and more recently, XML.

Bowman offers a summary of the key issues in the choice of a first programming language. Is the aim to teach a specific paradigm such as that of structured programming or to teach an approach such as the object-oriented approach? Bowman identifies a third approach, which is to teach a semi-formal theory of programming. Within this, Bowman distinguishes three different areas of interest: functional programming, formal specification and a theory of programming using pre and post conditions, with an emphasis on the theory, not a specific language.

The idea of concentrating on higher levels of programming knowledge was one presented by Shneiderman (1980). He distinguished between a higher level of semantic knowledge and language-specific (i.e. syntactical), lower levels of knowledge.

Holt (1972) noted that there were only three real programming languages: FORTRAN, COBOL and PL/I. PL/I had major disadvantages at the time, due to the huge size of the language and the difficulty of writing a compiler for PL/I. Holt (1972) goes on to say:

> "*We must teach structured programming. The structured programming approach is just plain superior to encouraging students to spend their time inventing and glorying in microtricks to save microseconds. Students must be taught how to break large programming problems clearly into smaller ones.*"
> (http://plg.uwaterloo.ca/~holt/papers/fatal_disease.html 19/10/2000)

McKeown and Farrell echo Holt's (1972) point:

> "*Students are seldom taught specific design techniques. Novices possess an incomplete understanding of program design and are unable to analyze a problem, dissect it, and create a working program.*"
> (http://homepages.dsu.edu/mckeownj/CPCCCSCpaper.html 04/12/2000)

This lack of understanding of broader concepts e.g. what an array is or the ways in which a loop might be used makes the design of a program in a structured language very difficult.

Novice programmers also need, when learning a programming language for the first time, to understand the syntactical constraints of that language e.g. what punctuation is required, the use of terminators like the semi-colon (in Pascal and Java) and the use of brackets. Shneiderman (1980) proposes a model of cognition and program development that rests heavily on the use of high to low levels of programming knowledge, pressed into service in that order. (Shneiderman 1980 p 51) Once the general semantic structures have been chosen and collated by the programmer, Shneiderman suggests, the actual coding is a comparatively trivial task.

The debate about programming languages can be traced back several decades. Sometimes, authors declared the debate (and the dilemma) to be no longer relevant. In 1984, an article in Creative Computing stated:

> "*There is very little need for most microcomputer users to learn to program in traditional computer languages. The main "programming languages" of today are spreadsheets, word processing programs and database managers.*" (Creative Computing – August 1984 pp 5 - 8)

In fact, programming languages continue to proliferate (as noted above) and continue to find passionate practitioners and equally passionate critics.

Just six years later, Green (1990 b) could declare:

> "*We shall all be programmers soon. That, at least, is the impression one receives from the vast spread of programming possibilities, creating new environments such as... knowledge-based and visual systems...; new models of programming such as logic-based and constraint-based programming; and new applications, to science, learning, and education, to the office world, the domestic world, and the leisure world.*" (Green 1990 (b) p 36)

Organick (1973) notes that the idea of teaching concepts such as data structures and relating use of various structures to algorithms started in the early to mid 1960s and that textbooks embodying this approach appeared in the late 1960s. These texts and improved interfaces made the teaching of storage concepts and algorithms a much easier proposition. (Organick 1973 p 140)

Mendelsohn et al. (1990) describe the tension between the needs of disparate groups of learners and the demands of learning a programming language. Languages like LOGO and Prolog seem to offer teachers a route into programming for new audiences without some of the cognitive burden associated with structured programming.

The choice of language as a first programming language is a choice not only of a paradigm but also effectively of a culture. (Mendelsohn et al. 1990 p 177)

The culture of a language includes the kind of mistakes and misconceptions that are fostered by the design and structures of that language:

> "*The unfortunate novice is prey to misconceptions when meeting any programming language. There have been excellent studies of misconceptions about even so 'small' a topic as assignment.*" (Mendelsohn et al. 1990 p 186)

Green (1990 b) offers a solution that seems to cut across the paradigm debate:

> "*Many problems seem to be peculiarly intractable to any single programming paradigm. If approached procedurally, it becomes clear that part of the problem is best approached declaratively, and vice versa. Why not use a mixed paradigm, and treat each aspect of the problem on its merits?* " (Green 1990 (b) p 35)

Another approach, taken by Traxler at the University of Wolverhampton, is to present students with a range of paradigms and ask them to write a program in each of three paradigms, tackling the same problem. The three paradigms used were functional programming, with the students using Miranda, concurrent programming in Modula 2 and object-oriented programming, using what the authors term a 'pure' object language, SmallTalk. The students were asked to reflect upon their experiences of each language and paradigm:

> "*Many of the students have implicitly questioned whether procedural programming is the best introduction to programming and have suggested functional programming in Miranda as a viable alternative. Unfortunately, the parts of Miranda they find most difficult are the parts of Pascal that they find most reassuring. Nevertheless they raise interesting issues for the choice of the base teaching language and their work shows how significantly any given base language influences subsequent language acquisition.*" (http://www.ulst.ac.uk/cticomp/papers/traxler.html 09/07/1998)

This last point was also made earlier by Shneiderman (1980) who says that learning a second programming language is much easier if the underlying semantics are the same as the first language acquired. Learning an entirely new set of semantics is as hard as learning a first programming language. (Shneiderman 1980)

McKeown and Farrell's discussion of the problem of teaching novice programmers does not mention any specific language. The problems they identify as being at the heart of programmer education have nothing to do with the choice of language or even of paradigm. The problems are those of curriculum design at the most detailed level: students are not taught the abstract thinking strategies and the problem solving skills need to be at least moderately successful programmers and those who teach programming lack insight into the relevant pedagogical processes. (http://homepages.dsu.edu/mckeownj/CPCCCSCpaper.html 04/12/2000)

Lewis (1981) notes that two ingredients are missing from the structured programming syllabus: principles of formal logic and an awareness of the psychology of problem solving. (Lewis 1981 Preface)

It is interesting to note that the kind of debates that focused on the type of language with which to learn programming (and the most appropriate environment with which to work) now begins to focus on the requirements around an object-oriented first language. Kolling, Koch and Rosenberg note, "*none of the existing languages is appropriate for teaching object-oriented principles*." (Kolling et al. 1995 http://www.sd.monash.edu.au/blue/papers.html 03/05/2000)

Eisenbach and Sadler (1981) note, "*Most of us learned our first language because it was the one available at the time*." (Eisenbach and Sadler 1981 p1) The choice is much wider now: compilers are available easily over the Internet. JDK (Java Development Kit). for instance, can be downloaded from its own site.

McIver (2000) notes that a programming language GRAIL (Genuinely Readable and Intuitive Language) was developed to "*minimise unnecessary errors*." (McIver 2000 p 181) using the design principles, of maximising readability and minimising unproductive errors.

However, McIver notes in the article that powerful concepts have been omitted from GRAIL, in order to make it an easily learned language. For example, GRAIL contains only one precision number type, static arrays and a small subset of control structures.

McIver (2000) describes the seven rules applied to the new of GRAIL's features. The first rule is, start where the novice is. Different concepts should be modelled using a different syntax. The syntax should be readable and consistent in the eyes of the novice programmers. The fourth rule: "*provide a small number of powerful non-overlapping features.*" (McIver 2000 p 182) Also, be cautious about fine design of input/output features. In addition, the language should provide better error diagnosis. Finally, choose the appropriate level of abstraction.

McIver (2000) describes the experiment using GRAIL comparing two groups of students, one learning LOGO and one learning GRAIL. McIver (2000) concludes that, given the small number of students, the difference, in error rates for the two groups is interesting but not statistically completely valid. The GRAIL students did have fewer errors in their code but the experiences of the two groups were not identical in terms of student interest (three students in the GRAIL group did much more than required).

Pane and Myers (2000) observe, "*New languages are usually spawned from technical demands and innovations, and gain their critical mass from adoption by the technical community.*" (Pane and Myers 2000 p 194) What language designers mostly seek to do is embody some technical consideration in a language: The human computer interaction elements and the constraints suggested by the psychology of programming are largely ignored. Pane and Myers (2000) note that for the psychology of programming and human computer interaction a dimension, answering a demand in one area militates against meeting an objective in another. "*Improving a system in one measure may result in a reduced performance in another.*" (Pane and Myers 2000 p 195)

Pane and Myers (2000) identify three key problem areas in language for novice programmers: visibility, closeness of mapping and the jargon used. Visibility refers to the problems novice programmers have in holding in memory all the elements they have to deal with successfully with the code they are writing. Pane and Myers (2000) suggest, "*The programming system should make information visible or readily accessible any time it is relevant.*" (Pane and Myers 2000 p 195)

Closeness of mapping refers to the translating of a mental plan into code that the computer can run. This requires the language to provide the operators that will model the user's mental plan. Finally, words and symbols in the programming language should not contradict their use elsewhere. The problem is that minimal, easily grasped languages present problems again when greater flexibility or sophistication is required.

Mendelsohn et al. (1990) note the difficulties associated with learning a logic language, in this case, Prolog:

> "*which originally promised to be easy to learn since it reduces the amount of program control that the programmer needs to define, but which turned out to be very prone to serious misconceptions. Recent work suggests that Prolog difficulties may be caused by an inability to see the program working.*" (Mendelsohn et al. 1990 p 175)

In summary, we can see that are no easy answers to the question of what language to teach first. It could be argued that in fact this is a trivial question, given the lack of theoretical underpinning for the teaching of programming. Perhaps it would be better to teach programming, at least initially, without recourse to coding: the focus should be on algorithmic thinking and problem solving (an echo of the syllabus model discussed earlier). Then the language chosen may embody imperatives other than purely educational ones: students who understand structured programming and can think in a modular, algorithmic fashion should be able to tackle any imperative language, whether it is C, C++, Java or Pascal.

We have so far assumed that the novice programmers under discussion are higher education students but this is not necessarily the case. Students often have exposure to at least some programming by the time they reach the first year of university. The question then has to be asked, is that appropriate or useful? Is previous work in some programming style beneficial, either as an underpinning for more formal learning or as a framework for problem-solving skills or algorithmic thinking?

At what age should students learn to program?

Clements (1985) describes three justifications that are generally put forward for teaching programming before students are 18. Programming is seen as a required skill, another form of literacy. Programming is also seen as fostering general computer literacy and finally, learning to program can develop problem-solving skills. (A reiteration of the programming to learn view.) Clements notes that the first two reasons are important but the third reason "*is the most powerful and it is in this area that programming may have the most impact on education.*" (Clements 1985 p 128) Clements goes on to describe ways in which children can be prepared for the activity of programming by undertaking structured problem-solving exercises. Such exercises might include playing robot or sorting items by various categories.

VanLengen and Maddux (1990) argue:

> "*The rationale for teaching computer programming is that it aids in the development of critical thinking, problem solving, and decision making skills. This contention is not supported by empirical data. An experimental study was conducted to ascertain if instruction in computer programming improved problem solving ability. The results of the study did not show support of improved problem solving ability from instruction in computer programming.*" (http://www.gise.org/JISE/Vol1-5/DOESINST.htm 24/04/2001)

For children, LOGO is often presented as the appropriate language for their first programming language. Proponents of LOGO for children often adopt very strong positions on its worth in the classroom. They argue that children can become confident in using the computer and can develop problem-solving strategies. Learning programming can be scaffolded in very definite ways (Clements 1985 pp 148 - 149) LOGO can be offered to children in various modes: the simplest being where only single keystrokes are needed to move the turtle. At a higher level children can write procedures and programs while the turtle carries them out simultaneously. At another level children are provided with re-usable procedures to develop programs.

Mendelsohn et al. (1990) provide a useful description of LOGO:

> "*LOGO is the language that is predominantly used in general education in France and is also widely used in Britain and North America. It has immediate appeal to children because of its 'turtle graphics', which originally were instructions driving a small path and had sensors to record obstructions. [] LOGO is a procedural language with provision for procedures (using parameters called by value), variables, conditionals, recursion and graphics. Data structures include numbers, strings and lists.*" (Mendelsohn et al. 1990 p 178)

Authors see Papert as a strongly biased advocate of LOGO:

> "*Papert* [] *maintains that other languages such as BASIC cannot provide or environment that facilitates the kind of epistemological reflections that LOGO can. Others believe that any language, taught correctly, can enhance logical thinking.*" (Clements 1985 p 170)

There is however, some doubt about the claims of enhanced cognitive ability fostered by the use of LOGO. Pea et al. (1987) concluded, "*...results suggest that a school year of LOGO programming did not have a measurable influence on the planning abilities of these students.*" (Pea et al. 1987 p 187) See also Clements (1985) pp 171 - 172

Similarly, there seems to be only a weak link between students reasoning abilities and their capacity to meet the cognitive demands of programming. Students took a 6-week LOGO programming course, designed to improve their mathematical understanding and build their confidence in this subject. The authors conclude:

> "*Our results indicate that certain reasoning abilities are linked to higher levels of achievement in learning to program, but that most students opt for a programming style that negates the need for engaging in high-level thinking or planful, systematic programming.*" (Kurland et al. 1987 p 126)

Mendelsohn et al. (1990) note that they used LOGO to research what they termed the 'transfer of competence' idea: "*This hypothesis has received limited support in a detailed sense, but not in its original more grandiose conception of programming as a 'mental gymnasium'.*" (Mendelsohn et al. 1990 p 175)

The culture of a language can be extremely powerful: LOGO is a good example of a language that has had expectation heaped high upon its use in the classroom. LOGO is the language of choice for those who adopt the strong 'programming to learn' view. Research in the use of LOGO must, to some extent, partake of that culture. Radical questions can be asked in relation to the use of a language as a first programming language but this does not happen as often as might be hoped for. Mendelsohn et al. (1990) go on to say:

> "*research on LOGO tends to be dominated by the same set of questions. In such a way does a culture perpetuate itself. And more: the LOGO culture has always*

> *emphasized interaction, either with real devices ('turtles') or else with virtual devices ('screen turtles'); these crawling and drawing devices add amusement but also present powerful challenges. LOGO systems without graphics are unthinkable.*" (Mendelsohn et al. 1990 p 177)

A variation of LOGO can be found in the work of Dalbey and Linn, published in 1984. As the abstract notes "*SpiderWorld is a interactive program designed to help individuals with no previous computer experience to learn the fundamentals of programming.*" (http://www.csa1.co.uk/htbin/ids51/txtdisp.cgi. 03/08/2000) SpiderWorld encouraged the use of constructs such as REPEAT...UNTIL, DEFINE...END, IF...GOTO (notice the use of goto!) and the authors studied the different cognitive effects of using other software/languages such as BASIC. They concluded that students do appear to have acquired the SpiderWorld templates, which they do employ in related situations gaining these skills in contrast to students learning BASIC who appeared to make fewer gains. (http://www.csa1.co.uk/htbin/ids51/txtdisp.cgi 03/08/2000)

In summary, the arguments for teaching programming at younger ages seem to rely on the strong 'programming to learn' view. There seems to be a lack of research and evidence about how such work impacts on the teaching of programming in more formal, assessed and examined way at the ages of 16 and 18. Do students who have worked with LOGO at the age of 8 or 10 move more easily into a procedural, imperative language such as Pascal? There is insufficient room here to debate this question but it would bear further research.

The choice of first programming language has been discussed previously in this chapter. We can accept that there is no perfect first language and that the choice of first language may be a matter of personal (tutor) choice, an institutional decision or an historical fact (what the department or institution has always used). Given that novice programmers do have to encounter a first language at some point, then Pascal, a language designed to act as a vehicle for teaching programming concepts, is probably no worse (and no better) than many other candidate languages. The next section looks more closely at Pascal.

Pascal

The language Pascal is closely associated with the idea of structured programming.

Farmer (1984) says:

> "... *Pascal first saw the light of day in a preliminary draft in 1968. It followed the spirit of Algol 60 and had the following goal:*
> *to encourage good programming practices,*
> *to communicate algorithms among people,*
> *to be efficiently executable on a variety of existing computers and*
> *to be useful for teaching students programming.*"

(Farmer 1984 p 1)

For standard Pascal (not the proprietary Turbo Pascal) the language is an end in itself, designed to teach structured programming concepts, not write applications. Versions of Pascal like Turbo Pascal remedy application development shortfalls such as file input/output and string handling. These versions of Pascal can give portability problems for the application developer but such issues should not affect the novice programmer, writing short programs that do not attempt any complicated file handling.

Pascal was designed to insulate novice programmers from machine internals such as memory registers. This is in contrast to C that has features that allow the programmer to manipulate individual registers:

> "*Mistakes are detected at compile-time whenever possible, and further checks are made at runtime. The syntax was designed to be unambiguous and was defined precisely by a context-free grammar.*" (Green 1990 (b) p 22)

Martin (1996) notes:

> "*Pascal is often used in schools, colleges and universities because it is highly structured and strongly typed, thus providing a 'safe' programming environment for a relatively novice programmer. Pascal has however, been criticised for certain 'overheads' which require explicit teaching prior to use. These are the program header, begin/end pairs and statement separators (;).*"

(http://www.holtsoft.com/turing/essay.html 31/10/2000)

Green (1990 b) classifies languages as neat or scruffy. Pascal was designed as neat language: that is a clearly defined language with little or no variation and one that creates programs that run predictably. Examining code can be done section by section without reference to other code sections.

Pascal can be described as a pedagogically-inclined programming language: "*Pascal is a great first-step language to learn that allows you to explore the concepts common to other programming languages."* (http://xrs.net/resources/Pascal/ 13/02/2003)

Bishop (1993) describes Pascal as a language that is "*modern, easy to understand and has good protection against mistakes.*" (Bishop 1993 p 10)

Pascal is a strongly typed language: that is, the programmer must make definitive decisions at the stage of defining variables and their datatypes, about the permitted range of values for a variable. If the programmer defines a variable as a char it takes only one character and no more. If a variable is an integer it takes a value in the range of -32768 to 32767 (signed 16 bit integer). Other integer types include longint and shortint, allowing the programmer to choose exactly the required datatype for a variable.

Arblaster (1982) notes "*BEGIN and END seem to be a poor choice as scope markers, since they are so easily mismatched and are hard to pick out in the program.*" (Arblaster 1982 p 217) which perhaps militates against the use of Pascal in teaching novice programmers! On the other hand, Arblaster (1982) says that it is easier to write structured programs in that the structure provides "*cues which help to make the semantics and underlying logic of the program more obvious …*" (Arblaster 1982 p 220)

Arblaster cites the linguist Whorf's belief that natural (human) language shapes the speakers view of the world and is, in turn, shaped by the external world. This Whorfian view would lead us to conclude that the choice of first programming language is critical in creating the language space in which programmers can 'say' what they understand and embody in code what they wish to 'speak of'.

Green (1990 b) describes the two opposite ends of the neat-scruffy continuum as Pascal and C. Green classifies Occam, Eiffel and Algol 68 as neat languages while Lisp, Fortran and Forth are scruffy languages. Green points out that these categories are not absolute for the programmer – a programmer may program neatly in a scruffy language (Green 1990 (b) p 23) and presumably vice-versa.

Pascal has advantages for the novice programmer in its closeness to natural language, its strong datatyping and its high-level language approach. Its disadvantages could be summed up as almost the same features. Those who dislike Pascal as a first language argue for a choice (such as C++) that is a 'real' language i.e. one that is widely used in industry. There can be no definitive resolution of the argument. Perhaps the most useful approach to the issue is to identify teaching strategies that focus and rely on non-language elements. What is critical, perhaps, is not the choice of language, but a host of other factors and issues, some of which are sometimes made overt, others which are ignored or discounted.

Appendix D looks at a small selection of programming texts, as novice programmers are frequently directed to (or seek out for themselves) a definitive book on how to program in their new language. Given the age of Pascal, there is a vast range of texts on how to program in various versions of the language. The sample discussed in Appendix D is a random sample of texts and inclusion does not constitute endorsement.

The teaching of programming

In reading about learning programming, what the reader most often encounters is material on programming, not teaching. The teaching of programming is somehow regarded as a given or as something not of interest to programmers (which indeed it may not be). Pedagogical considerations seem to be set aside when looking at the programming, as if teaching is some kind of monolithic endeavour that does not have variation, something that is not subject to a wide range of influences. This can be seen when one asks what models of learning underpin the teaching of a language. The focus is on the language and its technical demands and not on the pedagogical issues around the learning of that language. The assumption often appears to be that being a skilled programmer is an appropriate pre-requisite for being a good teacher of programming:

> "*People assume that if the person is good in this programming language, she will be good at teaching it. But knowing the subject matter is very different than knowing how to teach it.*"
> (http://csis.pace.edu/~bergin/patterns/ExperientialLearning.html 09/03/2003)

In treating the teaching of programming as an activity that does not require close scrutiny or analysis, the author feels that those who focus on the programming issues are ignoring critical areas of debate.

> "*Excellence in teaching is complex and difficult to achieve. It is about content and methodological technique, as well as about participants in the educational enterprise valuing and achieving quality outcomes. Excellent teaching takes a holistic perspective of the educational enterprise and ensures an integration of process and outcome.*" (Andrews et al. 1996 p101)

As described in Chapter 2, one approach that may be interest here is the concept of patterns, and, in the light of the above quotation, pedagogical patterns. "*A pattern is supposed to capture best practice in some domain. Pedagogical patterns try to capture expert knowledge of the practice of teaching.*"
(http://csis.pace.edu/~bergin/patterns/fewpedpats.html 09/03/2003)

Pedagogical patterns are clearly defined strategies that address a particular teaching problem or issue. Coplien (2000) says that "*there are many things called patterns that share a small core of techniques, principles, and values…*" (Coplien 2000 p 1) A pattern language can be defined as a collection of patterns. Patterns rely on what are termed pattern languages, which can be described as a collection of patterns. Coplien (2000) describes a pattern language as building a system: "*It is through pattern languages that patterns achieve their fullest power.*" (Coplien 2000 p 17)

Pattern languages give each pattern a name, with a clear problem statement followed by key advice or points to note. There may be variations on this, such as one pattern language flagging the solution with the word Therefore (as in Bergin's work). Most pattern languages follow Alexander's convention of noting patterns with one, two or no asterisks. One asterisk denotes a pattern that seems appropriate but may admit of further improvement or development. Two asterisks denote a pattern that seems to be optimal or that states invariants. (Fricke and Völter 2000 p 5)

The idea of patterns comes from the work of Christopher Alexander whose key work on patterns, in architecture, as published in 1977. Fricke and Völter (2000) note "*patterns provide hints along which lines a proven solution can be found to the problem.*" (Fricke and Völter 2000 p 2) Initially, Alexander offered 253 patterns in architecture and in the 1990s software developers picked up the idea of patterns and worked on patterns for software development. A group of authors (Gamma, Helm, Johnson and Glissades) called the Gang of Four presented a catalogue of 23 patterns for designing software systems. From this, the concept of patterns has moved out into areas such as group working and the teaching of computing and IT concepts.

Appleton (2000) notes

> "*The goal of patterns within the software community is to create a body of literature to help software developers resolve recurring problems encountered throughout all of software development. Patterns help create a shared language for communicating insight and experience about these problems and their solutions.*" (Appleton 2000 p 1)

Bergin offers a range of pedagogical patterns at http://csis.pace.edu/~bergin/patterns/fewpedpats.html. Most of the sample patterns, as defined by Bergin (http://csis.pace.edu/~bergin/patterns/fewpedpats.html 09/03/2003) could be used for almost any academic topic but some are specific to the teaching of programming such as the pattern called Read Before Write:

> *"You are teaching an elementary course that has a strong programming or design component. You want to help them learn to eventually create large and complex programs. Creating anything is hard work, even for the skilled. Novices, on the other hand lack these skills. However, as with natural language, students have an ability to read and understand larger artifacts than they can be expected to create.* []
> *Therefore give your students large programs or designs to read and evaluate before you ask them to build even small artifacts. These readings can and should be far more difficult than anything you would expect them to create. The artifacts you give for reading must be elegant in design and execution. These will be the exemplars that you want students to learn from. "*

(http://csis.pace.edu/~bergin/patterns/fewpedpats.html 09/03/2003)

For Bergin, textbooks are not an appropriate teaching device. Bergin describes the kind of approach seen above, in the section on Pascal texts, which is the dividing up of concepts into a linear structure: always IF statements before loop constructs and so on. Bergin's answer to this is a pattern he terms Abandon Systems All Ye Who Enter Here, which abandons the systems approach (rigid, linear, assumes the student knows nothing) and emphasises a constructivist approach (assumes the student can create and develop understanding and meaning from his or her own experience):

> *"**Therefore** choose materials and methodologies that let students progress on multiple fronts, individually, and which enable new and meaningful problems to be solved. Do not force them to follow a particular, rigidly defined, path. Instead, set them to tasks they will find meaningful and provide minimalist aids for solving those problems and avoiding common difficulties."*

(http://csis.pace.edu/~bergin/patterns/fewpedpats.html 09/03/2003)

Fricke and Völter (2000) offer a pattern language that addresses the effective teaching of seminars. In what is termed Quick Access table they offer typical problems and suggest patterns to deal with each problem or issue. For example, "*After the seminar, people say they did not understand the topics.*" (Fricke and Völter 2000 p 8) patterns offered for this problem include DIGESTIBLE PACKETS, DIFFERENT APPROACHES and REVIEW AFTER BREAKS.

There are pattern languages for experiential learning, found at http://csis.pace.edu/~bergin/patterns/ExperientialLearning.html and initial approaches to a pattern language for computer science course development, found at http://csis.pace.edu/~bergin/PedPat1.3.html.

However, it should be noted that the author's interest lies in the development of a pattern language by and for novice programmers, rather than the development and use of pedagogical patterns by the author herself.

In general, the material on the teaching of programming tends to concentrate on the technicalities of the language chosen (or, one stage further back, which language to choose) or the order in which concepts should be presented. For the author, the concept of pattern languages for both software development and the teaching of software development seem most promising, as noted in Chapter 2. The research the author undertook to use the idea of patterns in teaching novice programmers is described in Chapter 7.

The next section broaden the discussion about programming, to include material often not addressed in material about the teaching and learning of programming: that of the ways simply being human may affect the experience of programming. This covers topics such as mental models, confirmatory bias and attention blindness.

Human aspects of learning to program

We can contrast approaches to teaching programming that are based on ideas that focus on the learner, with those that emphasise the programming language or environment. In focusing on the learner, we are engaging with the subtle nature of what it means to think and learn. This is not a trivial undertaking. One American neuroscientist expressed this lack of knowledge about the human brain rather pungently: "*...despite zillions of us slaving away the subject, we still know squat about how the brain works.*" (Sapolsky 2003 p 71) There is insufficient room in this chapter and thesis to explore the wide-ranging and complex area of neuroscience but some elements of thought about how the brain works are touched on in this chapter.

What knowledge we have about learning to program tends to be situated in specific instances of language use, rather than offering more broadly applicable, language independent theories. Green (1990 b) says: "*...we know something about the psychology of learning to program, of understanding programs, etc, but only in a very restricted range of languages and environments.*" (Green 1990 (b) p 42)

The psychology of programming

We can then ask what broader cognitive or psychological approaches, which move away from a particular language or environment, may have to offer. Are there such theories readily available? Hoc (1983) notes: "*It is true that theoretical frameworks of* *cognitive psychology do not permit us to derive very precise models of the psychological mechanisms underlying programming.*" (Hoc 1983 p143)

Nevertheless, Hoc (1983) concludes, such models are necessary and better than unarticulated, general frameworks for experiment and discussion.

Hoc's thinking concurs with that of Green (1980). Green notes that psychology cannot offer "*a general Theory of Thinking*" (Green 1980 p 272) and will not be able to for some time, if ever. Green (1980) does have strong views what psychology, or more particularly, applied psychology has to offer in the field of programming. Applied psychology's value lies in offering ideas about how to evaluate expert programmers' notions about the psychology of programming and in providing general insights into how people think about specific aspects of programming practice. The example Green (1980) offers for the second point is the work of Sime, Arblaster and Green (1977) on the form and use of the IF… THEN … ELSE statement.

Shneiderman (1980) described programming as an "*intensely human experience whose esthetics cannot be imitated or appreciated by mere machines.*" (Shneiderman 1980 p 2) Authors generally agree that programming is not a homogenous activity, but a collection of sometimes widely differing tasks that place different cognitive demands on the programmer. For example, Shneiderman (1980) argues that flowcharts, being a visual representation, are better handled by those who are right hemisphere dominant whereas coding, a verbal activity in most programming environments, is more comfortable for those who are left hemisphere dominant. (Shneiderman 1980 pp 82 – 83)

If there are specific but small scale aspects of psychology that offer some help to learners and teachers, but no general, overall theory of how novice programmers learn, is it then fruitful to explore the psychology of expert programmers? It may be possible to identify skills and behaviours in expert programmers that can be fostered in novice programmers. Work has been done on the understanding demonstrated by experienced programmers in learning an entirely new paradigm. Are experienced programmers given an advantage by virtue of their experience?

Some research suggests not. Rogalski and Samurçay (1993) state that, for novice programmers or experienced programmers learning a new paradigm, there are persistent errors of understanding or "*epistemological obstacles*." (Rogalski and Samurçay 1993 p 15)

If, for both novice and experienced programmers, there is no broad theory about how to overcome the obstacles to understanding and using a programming language, are there aspects of programming that are amenable to cognitive research, as described by Green (1980) above? One such aspect could be the design of programming languages.

Myers (1998) notes:

> "*There are substantial gaps in the knowledge about how to make programming languages effective for people, and how to apply this knowledge to the design of new programming languages.*"
> (http://reports-archive.adm.cs.cmu.edu/anon/usr/anon/home/ftp/1998/CMU-CS-98-101.html 24/04/2001)

This situation has obtained, says Myers (1998), for decades: money is poured into developing new languages and compilers but very little is spent on the human aspects of programming. Myers (1998) goes onto to articulate some of the questions he sees as being immediately relevant to the teaching of programming to non-programmers, such as the choice of language, the use of notably difficult constructs, like loops, and the use of abstract concepts such as variables.

Myers (1998) notes:

> "*It is somewhat surprising that the designs of new programming languages have generally not taken advantage of most of the human-factors results. For example, the newly popular Java and JavaScript languages use the same mechanisms for looping, conditionals and assignments that have been shown to cause many errors for both beginning and expert programmers in the C language.*"

(http://reports-archive.adm.cs.cmu.edu/anon/usr/anon/home/ftp/1998/CMU-CS-98-101.html 24/04/2001)

Applied psychology can offer useful insights into how people make sense of text/written material using patterns as clues or guides. Green (1980) offers the example of learning a nonsense language and assembling a sentence in the language. Signals embedded in the language enabled subjects to create sentences. As the number and meaningfulness of the signals was degraded, performance on the test "*dropped dramatically*" (Green 1980 p 282).

Another view of how cognitive psychology might offer insight into programming is to look at two different aspects of programming: programming as problem solving and program comprehension. Rogalski and Samurçay (1993) phrase this concept slightly differently: "*Cognitive activities involved in design tasks can be analysed along two dimensions: representation and processing.*" (Rogalski and Samurçay 1993 p 9)

If we accept that there is no broad, psychologically-based theory of how to teach programming and no definitive ideas about what makes a language easy to learn and use, then perhaps the focus needs to be on those learning to program. We can ask, are there some people more suited to being programmers than others? This immediately presents difficulties.

Personality and aptitude tests

Weinberg (1971) stated that "*the programmer's personality – his individuality and identity – are far more important factors in his success than is usually recognized.*" (Weinberg 1971 p 158)

Shneiderman (1980) echoed this point and noted, "*Unfortunately, it is difficult to say what aspects of a person's background influence competency in programming.*" (Shneiderman 1980 p 34) In other words, we can accept that some people will make better programmers than others but which people is not easy to define or predict! Focusing on the whole person does not necessarily generate any more definitive answers than the debate about which programming language to learn.

Personality tests

One way that we could identify those who are more suited to be programmers (whatever we decide we mean by more suited) is to garner psychological profiles of potential programmers. Of course, the question then arises, what are the psychological traits of good or expert programmers? Is there an expert programmer type?

If there is an expert programmer type then can a test or tests be developed that will reliably identify those with the appropriate profile for the tasks? Using such tests involves assuming that the tests have some genuine merit. We could argue that such tests are indicative, not definitive, but the author feels that caution is needed here. We should resist the urge to pigeonhole people on the basis of psychological tests.

Two well-known tests are the Myers-Briggs test and the Keirsey Temperament and Character tests, which are similar:

> "*The Keirsey Temperament and Character tests [] can be taken on line and scored immediately. The temperament and character types are explained. There is also a short list of careers aligned with the character types. Architect, for example, is aligned with rational, teacher is aligned with idealist, etc.*"
> (http://www.hcc.hawaii.edu/intranet/committees/FacDevCom/guidebk/teachtip/keirsey.htm 08/12/2000)

The Myers-Briggs Type Indicator typifies a person according to their placing on the poles for four opposed styles or archetypes (based on Jung's work.) The four archetypes are:
Introversion- Extroversion
Thinking – Feeling
Intuition – Sensing
Judging - Perceiving

A Keirsey profile generates a type based on the test taker's place in each of the four types: NFs, NTs, SJs and SPs. Each type has four subtypes. NFs are Idealists with the subtypes: healers, teachers, counsellors and champions. For example, the test taker might be characterised as iNFj, which is described as the Counselor Idealist. (http://keirsey.com/personality/nfij.html 28/09/2002)

Such insights might prove interesting, as people are always curious about themselves but it is a considerable leap from such general characterisation to defining those most suitable (or not) to become programmers.

Other tests based on the MBTI have been created, such as the Personal Style Inventory devised by R.Craig Hogan and David W. Champagne and the Grasha-Riechmann Student Learning Style Scales developed by Anthony Grasha and Sheryl Riechmann. The second test looks at social interaction styles in university students, with regards to tutors and peers, in the learning environment.

For a comparison of several different systems, including Myers Briggs, True Colours and De Bono Six Thinking hats, see
http://www.insightsvancouver.com/comparisons.htm

Other approaches include CBCA (Cross-Battery Cognitive Assessment) used in by school psychologists in North America. Watkins et al. (2002) note that *"CBCA posits the existence of around 70 narrow abilities that are hierachically subsumed by 10 broader abilities.*" (Watkins et al. 2002 p 1) Watkins et al. (2002) express serious concerns about the design, use and approaches of CBCA and "*advise caution*" (Watkins et al. 2002 p 3) until further independent evidence validates CBCA's benefits and use.

It is hard to define what positive input a personality test might give to the teaching of novice programmers. Perhaps in very small groups, some discussion of such tests might be useful but the author has deep reservations about such inventories. Are there more specific ways of identifying those suited to programming? One approach to selecting programmers (or potential programmers) for employment or training is to use aptitude tests. These are discussed in the next section.

Aptitude tests

Pea and Kurland (1983) noted that here might be five factors in predicting programming skill: mathematical skills (although there may be no correlation), memory capacity, analogical reasoning skills, conditional reasoning skills and procedural thinking skills. They note, "*These cognitive abilities are presumed to have an impact on or mediate computer programming learning in a number of ways.*" (Pea and Kurland 1983 p 35) However, general intelligence as shown in intelligence test also correlated strongly with programming ability. (Pea and Kurland 1983 p 42) We can conclude that clever people make good programmers!

Green (1990 b) notes:

> "*... individual style is still detectable in programming. What do we know about it? Next to nothing. Aptitude tests, widely investigated as a personnel selection tool, tell us nothing about programming style, and personality factors seem to have been unhelpful in investigations. One of the few successful investigations into personality factors in this area, perhaps even the only one, is the demonstration that risk aversion plays a part in success with certain types of programming environment...*" (Green 1990 (b) p 24)

So according to Green (1990 b) the only personality factor that seems to be relevant is cautiousness!

There are many aptitude tests for programmers and those who administer them make strong claims for them:

> " *...the "Aptitude Assessment Battery: Programming" (AABP) test,* [] *was designed in the late 1960s by Dr. Jack Wolfe (a pioneer of early data processing and computer education) to evaluate computer programming potential. The five hour test is comprised of five very challenging problems. It has been, and continues to be, a successful evaluation tool. It not only answers the question, "Should you work as a programmer?" but also "How proficient will you be?" in a definitive manner.*"

(http://www.developercareers.com/ddj/articles/1999/9914/9914b/9914b.htm 08/12/2000)

Some tests are much shorter:

> "*The B-SYS is a 24 question multiple-choice test. It measures an examinee's potential to be trained as a systems programmer. The test was designed for candidates who*

> *have no significant systems programming experience. The test can be administered in 1 hour."*
>
> (http://psy-test.com/Bsys.html 08/12/2000)

In reading about the aptitude tests offered by various commercial companies, one point is very clear. Strong claims are made for the diagnostic capabilities of these tests and much is made of their long history of use. What is not discussed (for obvious reasons) is whether those who fail such tests go on to become good programmers in some other environment and conversely whether those who are working as programmers would pass such tests.

In summary, it seems that, despite strong claims being made for some of these tests, they do not constitute a truly meaningful approach to assessing potential programmers.

Are there general concepts in psychology that might have something to offer in the field of learning to program? One of the broader ideas that can be applied to programming is that of mental models. The next section examines the idea of mental models and looks at ways they might be used to support the teaching of programming.

Mental models

It is useful define exactly what we mean by mental models. Mental models can be explored in many contexts, not just computing and programming but we need a focused definition that takes account of the interplay between the mind of the programmer/user and the system he or she is working with. "*Mental models are the conceptual and operational representations that humans develop while interacting with complex systems.*" (http://www.cica.indiana.edu/cscl95/jonassen.html 25/04/2001) van der Veer (1992) defines a mental model as "*some kind of internal representation that is individually developed by human beings in order to cope with complexity in the environment.*" (van der Veer 1992 p 21)

A mental model of programming might be developed as a standard, a representation that would embody all the understanding required to program. This is not the same necessarily as the mental models developed by individual expert programmers. Research seems to have largely concentrated on the second aspect of mental models: the particular models developed and used by individual programmers. It is important to distinguish between different types of models such as cognitive models, conceptual models and mental models.

> "*Cognitive models are typically developed by cognitive psychologists, using information processing conceptions of skills and propositions, to describe the processes that humans use to perform some tasks such as solving problems, using a computer system, programming computers, etc.*"
> (http://www.cica.indiana.edu/cscl95/jonassen.html 25/04/2001)

Conceptual models of a system are created by those who designed or programmed the system. It is their version of how the system can be perceived by those who use it. A conceptual model is a tool for the teaching of a physical artefact, such as a computer. One type of mental model can be described as the users' models of the system with which they are engaged. Ideally, the conceptual model and a user's mental model should be coherently and closely related. Norman (1983) notes, "*All too often however, this is not the case.*" (Norman 1983 p 12)

van de Veer (1993) defines a mental model as " *some kind of internal representation that is individually developed by human beings in order to cope with complexity in the environment.*" (van de Veer 1993 p 21)

Another definition states that "*Mental models are representations of reality that people use to understand specific phenomena.*" (http://www-hcs.derby.ac.uk/tip/models.html 13/03/2003) Studies of mental models generally focus on a small and well-defined area of understanding, such as particular construct's role in a programming language, or a specific concept such as light, electricity or gravity.

More broadly, lines of research in mental models can be classified falling into one of two areas: cognitive psychology (which can include philosophy, linguistics and anthropology) and AI (artificial intelligence). This second line of research, AI, "*...has* *provided powerful formalisms in which to explicitly notate theories of human knowledge representation and processing.*" (Gentner and Stevens 1983 (Eds) p 3)

AI has done much to explore how it is human beings think about the world around them and how they handle ambiguity and complexity but AI has not provided, any more than cognitive psychology has, a general theory of thinking. Modelling the ways self-aware people behave or make decisions while interacting with an unpredictable and changing environment has presented AI researchers with many decades' worth of challenges but the artificial brain is perhaps no nearer than it was in the 1960s. As the AI laboratory at MIT notes in its web page:

> "*The Artificial Intelligence Laboratory has been an active entity at MIT in one form or another since at least 1959. Our goal is to understand the nature of intelligence and to engineer systems that exhibit intelligence. [] Our intellectual goal is to understand how the human mind works. We believe that vision, robotics, and language are the keys to understanding intelligence...*" (http://www.ai.mit.edu/ 12/03/2003)

What AI has not given us is definitive answers on how people think: there is no easy, single way to access and embody the ideas that, consciously or unconsciously, inform or influence people's thinking.

This point is reinforced by Norman (1983), who makes some general observations about mental models. Norman (1983) characterises the models held by people as incomplete, not wholly accessible to the person who 'thinks' them, unstable because "*...people forget details of the system they are using...*" (Norman 1983 p 8), lacking in firm boundaries (models get confused together) and both unscientific and parsimonious (people seek to avoid elaborate mental models, even if this lack of sophisticated models ultimately involves greater physical effort).

Interestingly, Norman (1983) notes that verbal protocols (for accessing mental models) do not give a complete or accurate view of the model. People may say one thing but do another. Norman (1983) states:

> "*All the people I observed had particular beliefs about their machines and their own limitations, and as a result developed behaviour patterns that made them feel more*

secure in their actions, even if they know what they were doing was not always necessary." (Norman 1983 p 10)

Models, as noted above, are often incomplete (for various reasons). This incompleteness is manifested as simplicity or sparseness of detail. Models are always simpler than the artefacts or phenomena they represent.

So, if mental models are neither a full representation of the external world (and may even run counter to it) and may not admit of full articulation by the person who 'owns' them, what use are they? Examining mental models may still offer useful insights into why people behave as they do. Where the model is lacking in depth or detail, there may be scope for deliberately re-drawing the model to improve understanding of the real world system. In programming, one area that seems initially promising is the comparison of the mental models of experts and novices.

It would seems obvious that if, experts' mental models consistently exhibit some qualities or features that are absent in novices' models, then offering novices short cuts to those aspects must improve their understanding of programming. This, however, is not as simple as it would seem.

Much work has been done on trying to capture the essence of expert programmers' mental models versus those of novice programmers'. Capturing mental models can be seen as nomothetic (seeking to capture a *general* model of internal representation, valid across different situations and people) or as one that is highly context-bound or idiographic, where generalisation of the models is not appropriate. In other words, is one person's mental model of how a computer works likely to be valid for a range of people? Both views can be taken of research of mental models in programming. (van de Veer 1993 p 21)

Mental models are often referred in the context of programming to denote users' own conceptual models of how a computer works. The emphasis is often on enabling users to develop a semi-realistic model of the computer, the virtual machine that is more transparent in its operation than the physical object.

Blackwell (1996) notes:

> "*The activity of programming involves planning the behaviour of a virtual machine that is defined by the programming language. Much of the programming task consists of maintaining a mental model of that machine.*"
> (http://cui.unige.ch/Visual/local/Blackwell96c.html 25/04/2001)

In describing an instructional theory aimed at teaching non-computing students computer science at an introductory level, Urban-Lurain (1997) noted:

> "*The lessons are designed to help students construct mental models of how computers and software work to enhance retention and facilitate transfer to learning new software to solve new problems.*"
> (http://aral.cps.msu.edu/CPS101SS99/CPS101Visitor/InstructionalTheory.htm 25/04/2001)

In describing the programming needs of specialist computing students versus non-specialist students, Urban-Lurain (1997) noted that specialist students had some background in programming, whether in FORTRAN or Pascal.

Non-specialist students lacked that background so any teaching materials had to take account of those lacunae:

> "*The goal is for the students to construct mental models or schemata of the computing systems they are using without learning the minutiae of computer programming. To accomplish this goal, the course [adopted] a spiral curriculum in which students* [were] *presented with increasingly more challenging problems to solve using a variety of software packages.*"
> (http://aral.cps.msu.edu/CPS101SS99/CPS101Visitor/InstructionalTheory.htm 25/04/2001)

If the goal is to equip the students with accessible, relevant mental models that underpin their programming practice, then the question must be, when we have given the students that model, how do we check that the model the students now hold in their heads (potentially at least) is the one we wished to convey to them? van de Veer (1993) states "*Collecting knowledge of mental models and mental representations is a problem.*" (van de Veer 1993 p 24) Green (1980) notes "*One of the hardest questions in cognitive psychology is what we have in our heads when we think about problems.*" (Green 1980 p 302)

Rogalski and Samurçay (1993) broaden the concept of a mental model by describing what they term the KEOPS schema, a representation of the operational knowledge required by a programmer. KEOPS is an acronym for Knowledge, Experience, Operative (tools) and Problem Solving. Operative tools include programming environments and programming methods. The authors note, "*Developing individual competences in programming is the result of multiple interactions between the components.*" (Rogalski and Samurçay 1993 p 13)

For programming, then, the question is not just that of a mental model of how the computer works, but models of various elements and an overarching model of how those models relate to each other and interact. A good example of this is the activity of a compiler, when it generates object files and executable files. Novice students seem to have difficulty with this and, when using Pascal, often overwrite editor files with system generated files, by using Save As and naming an editor file wrongly.

van de Veer (1993) describes the idea of mental models as a "*core concept*" (van de Veer 1993 p 21) in work on designing programming languages and investigating how people program or work with computers. Given that capturing mental models is difficult, van de Veer (1993) presents the idea of teach-back tasks, where users are asked to describe the ways they imagine the internal workings of a system. The example van de Veer gives is that of using email, including deleting or forwarding specific messages.

The users are given no constraints on how they structure or present their answer apart from requiring use of pen and paper. The styles of responses were then interpreted and scored. van de Veer (1993) identifies four levels of description: task, semantics (concentrating on functionality), syntax and key stroke. The styles of representation can be classified as verbal, visual-spatial, based on production rules (a declarative representation) and program structure. It is perhaps possible to classify closely the pen and paper representations of tasks but we would have to ask how we might use that classification. How does it advance our understanding of mental models to identify the most widely used styles and descriptions?

In summary, mental models appears to be an initially promising line of enquiry for researchers wishing to clarify the differences in mental models that are formed or used by novice and expert programmers but, in practice, both the nature of mental models and the difficulties of accessing and meaningfully articulating those models leads to less than productive conclusions.

The idea of comparing the approaches, habits and mental models used by novice programmers versus those of expert programmers is one that seems to offer the possibility of useful insights. The ways in which novice and expert programmers might be compared and contrasted is discussed in Appendix E.

The next section examines an aspect of human physiology that might not at first, be obviously related to the idea of teaching and learning programming. However, the author feels that in her work with novice programming the metaphor of 'seeing the problem' embodies a literal, physical issue around visual processing of code on screen or paper.

Visual cognition

Visual cognition is a complex area of psychology and research. This section can give only a brief outline of some the interesting and possibly relevant topics that can be grouped under the broad heading of visual cognition. What relevance can vision have for programming? Programming is, despite its mathematical foundations, a literate activity: it requires sophisticated reading skills, a function of both eyes and brain. Reading for comprehension is about more than being able to see. There is the question of the processing of visual stimuli by the brain and the issue of visual attention. There are two key topics here, which can be separated to some extent: change blindness and inattention blindness.

Human beings are creatures of the savannah and our vision system is designed for that. We are attuned to reacting to movement. If human beings react to movement, how do we react to perfect stillness of an image? If the image isn't changing or moving and the retina is still, then the image on the retina can be said to be perfectly still and the image disappears.

Glynn (1999) identifies four approaches to the question of 'What is seeing?' The two historic approaches are psychophysical (a term dating back to 1860) and neurological (the study of effects on vision of injury or disease). The two newer approaches, identified by Glynn (1999) are neuropsychological and computational (concerned with the information processing done in the nervous system).

Change and inattention blindness

Austen and Enns (2000) note:
"There has been a recent explosion of interest within the psychophysical community in the role of attention in perception (e.g., change blindness, inattentional blindness, repetition blindness, the attentional blink, masking by object substitution, amnesic visual search). A central theme in this research is that perception of the visual world is not as rich as our subjective experience gives us to believe." (Austen and Enns 2000 http://psyche.cs.monash.edu.au/v6/psyche-6-11-austen.html 22/02/2001)

Change blindness and inattention blindness are related to the physiological facts of human vision. Noë et al. (2000) state:

> "*Theorists have long been impressed by the fact that visual experience is underdetermined by the input to the visual system. The eye is in nearly constant motion; the resolving power (spatial and chromatic) of the retina is limited and non-uniform; passage to the retina is blocked by blood vessels and nerve fibers; there is a large 'blind spot' on the retina where there are no photosensitive receptors; there are two retinal images each of which is inverted.*" (Noë, Pessoa and Thompson 2000 http://www2.ucsc.edu/people/anoe/GrandIllusion.html 22/02/2001)

Together with the physical/physiological restrictions on visual input are the changes in the input that we cannot avoid.

Henderson notes:

> "*The image projected onto the retina changes dramatically three to four times per second due to saccadic eye movements, yet we do not experience the world as a series*

> *of temporally discrete visual snapshots."*
> (http://eyelab.psy.msu.edu/people/henderson/research.html 22/02/2001)

Change blindness can be defined as the non-registering of significant changes in a scene, when the change occurs during saccades (eye movements) or when the scene change is masked by flicker or the insertion of a blank screen between the changes.

Inattentional blindness is when observers do not report an odd event or unexpected item in the scene:

> "*...studies of change blindness have shown that striking changes to objects and scenes can go undetected when they coincide with an eye movement, a flashed blank screen, a blink, or an occlusion event. These studies suggest that relatively little visual information about objects and scenes is combined across views. Despite these failures of change detection, observers somehow manage to experience a stable, continuous visual environment.*"
> (http://taylorandfrancis.com/psypress/BKFILES/0863776124.htm 22/02/2001)

An example of this is the Harvard Cognition Laboratory's video of a group of people throwing basketballs to each other. The observer is asked to count the number of times the ball is thrown. The video shows the group of people moving rapidly, with many changes of ball possession. Through this scene strolls a person in a gorilla suit. Most observers do not see the gorilla!

Rensink (2000) noted:

> "*Mack and Rock (1998) showed that IB vanishes once observers suspect they will be tested on the item that is introduced: this suggest that divided visual attention is necessary (or at least sufficient) to see a stimulus.*" (Rensink 2000 http://cogprints.soton.ac.uk/documents/disk0/2000/2000/10/50/index.html 22/02/2001)

For Rensink (2000) change blindness and inattentional blindness are related but separate phenomena. The perception of change requires a sequence of operations. The image needs to be loaded into visual short-term memory, held across any blank interval(s), compared to the changed/ newly displayed image and the new image loaded into short-term visual memory.

Inattentional blindness is a first-order phenomena, that does not require the second-order activity of change blindness. Rensink (2000) reports, "*...all experimental results to date support the assertion that focused attention is needed to visually experience change.*" (Rensink 2000 http://cogprints.soton.ac.uk/documents/disk0/2000/2000/10/50/index.html 22/02/2001)

Noë et al. (2000) state:

> "*Taken together the change blindness literature suggests the following hypothesis: subjects notice only changes to features that have been encoded by the visual system. Thus change blindness suggests that under normal viewing conditions only a minor part of the environment is encoded in detail. Although the factors that determine which features of a scene are encoded remain unknown, it seems likely that attention plays a major role...*" (Noë et al. 2000 http://www2.ucsc.edu/people/anoe/GrandIllusion.html 22/02/2001)

What implications do these aspects of vision processing have for programmers? Reading is generally regarded as being a process that can become automatic for the reader:

> "*Automatization is interesting because it is an important part of daily life. We perform a variety of automatized behaviors quickly and effortlessly. In some cases people report that they do not consciously know* how *the behavior is performed, they just will it to happen, and it does happen.*" (http://coglab.psych.purdue.edu/coglab/Labs/StroopEffect.html 22/02/2001)

One way to examine the automated skill is to place the exercise of that skill under strain: one example of this is the experiment that demonstrates what has become known as the Stroop effect:

> "*Stroop (1935) noted that observers were slower to properly identify the color of ink when the ink was used to produce color names different from the ink. That is, observers were slower to identify red ink when it spelled the word* blue. *This is an interesting finding because observers are told to not pay any attention to the word names and simply report the color of the ink. However, this seems to be a nearly impossible task, as the name of the word seems to interfere with the observer's ability to report the color of the ink.*" (http://coglab.psych.purdue.edu/coglab/Labs/StroopEffect.html 22/02/2001)

Durgin (2000) says, "*The Stroop Effect is one of the easiest and most powerful effects to demonstrate in a classroom, but not the easiest to explain.*" (http://www.swarthmore.edu/SocSci/fdurgin1/publications/ReverseStroop/PBRStroop.html 22/02/2001)

What implications do the concepts of change blindness and inattention blindness have for programmers? Programming relies heavily on reading code, either on screen or on paper and seeing where the problem lies, either in terms of the text on screen (missing semi-colon or misspelled variable name) or in the logic of the program (as in the loop example below). In the author's experience, seeing where the problem is a visual skill that requires practice. Perhaps it is more accurate to say that debugging is a cognitive skill that requires a subtle interplay of vision (seeing the code) and understanding (knowing what it is you are seeing).

Reading is a skill that perhaps needs further examination for programmers. We can assume a certain level of literacy but reading code present particular challenges, in terms of syntax and structure. A missing semi-colon in Pascal generates an error, which the compiler flags, but it notes the error on the previous line. This requires the novice programmer to develop reading skills that acknowledge the peculiarities of the language and the compiler he/she is using. Reading code for meaning also present particular challenges: the condition at the top of the loop, in a WHILE statement, affects the statement within the loop that ensures the loop finally terminates. For example,

```
WHILE (number > 10) DO
    BEGIN
        Statement
        Statement
        Statement
        number := number – 1
    END
```

The decrement statement at the bottom of the loop may be separated from the initial condition by several statement lines. Students need to read at both the level of each line (a what it does approach) and also at a construct level (what the loop does, what the IF...THEN does).

This accretion of meaning at a line and construct level can be very demanding. Perhaps there is scope for making explicit the kind of reading skills needed by a novice programmer. We could ask whether reading other people's code and re-reading their own code present the same kind of challenges or subtly different ones?

Is it easier or harder to re-read your own code? Anecdotal evidence from the author's own teaching suggest that student find it hard to re-read their own code, perhaps because it is their own and they are anxious or worried about it. Is it any good? Is it right? Have they made a stupid mistake that will mean they have to re-write large sections of the work? It is less threatening to inspect their code in a cursory fashion than it is to examine it closely. Here are issues that move into the affective domain: how people feel about their own code. Linked to this question of re-reading and evaluating code is the idea of confirmatory bias.

Confirmatory bias

The concept of confirmatory bias has implications for the work of programmers at all levels: "*Confirmatory bias is the tendency to emphasize and believe experiences that support one's views and to ignore or discredit those that do not.*" (Mahoney 1977 p 161 http://www.mang.canterbury.ac.nz/courseinfo/AcademicWriting/Prejud.htm 25/04/2001) Human beings are reluctant to abandon their assumptions and prejudices, despite of evidence that contradicts what they 'know to be true'.

Blum (1996) describes the work of Wason (1960):

> "...*Wason introduced a very simple reasoning task in 1960 in which subjects were asked to discover the rule to which the triple "2 4 6" belonged. The subjects would produce test triples, and the experimenter would tell them if the test sample conformed to the rule or not; periodically, the subjects also would offer a tentative hypothesis, and the experimenter would tell them if it correct or not. The process*

> *continued until the rule (which was "any ascending sequence") was discovered or the subject gave up.*" (Blum 1996 p 162)

Blum (1996) goes on to state that test subjects largely looked for confirmation of their theories. Only the Protestant minister test subjects used disconfirmation. Psychologist and physical scientists preferred to look for confirmation of their theories. Scientists, typically seen as rigorous in their testing, are as prone to confirmatory bias as almost any other group. Blum says: *"There is a high cognitive cost for attempting to disconfirm what we believe, and we tend to avoid this cost except when confronted by a breakdown.*" (Blum 1996 p 164)

We could conclude that novice programmers, unsure of their work, will be even more likely to adopt this strategy. They will believe, until forced otherwise by circumstance, that their code is solid, functional and meaningful. 'Circumstance' might be the compiler flagging errors or it might be a fail grade for an assessed piece of code. Again, anecdotal evidence leads the author to believe that students focus on what their code can do and ignore elements that do not work or do not work correctly.

Students often submit code that does not compile or does not begin to offer the required functionality. What leads them to hope or believe that the work will be acceptable? The answer seems to be the blind optimism of confirmatory bias! It seems to the author to be useful to present the idea of confirmatory bias to novice programmers, to alert them to its possibly deleterious effects.

Metaphor and analogy

Can metaphor and analogy provide any useful tools for the teaching of programming? Are there identifiable benefits in describing, say, a computerised database as being like an electronic filing cabinet (a simile that is often used)? Is explaining the concept of a bi-state device aided by references to light bulbs? (A bi-state device is like a light bulb – it can be either off or on. There are no in-between states) Or are metaphors and analogies confusing to novice programmers? The use of figures of speech and analogies appears to be a somewhat under-researched aspect of the teaching of programming. Blackwell (1996) says, "*Metaphor and analogy have always* [] *provided an important perspective in the study of how we acquire mental models of programming languages.*" (http://cui.unige.ch/Visual/local/Blackwell96c.html 25/04/2001) In some cases "[o]*nly the simile can give the necessary perspective, the necessary vision, and can communicate the whole idea intended.*" (Leatherdale 1974 p 94)

Leatherdale (1974) notes that writers such as Buchanan believe that:

> "*analogy in the sense of resemblance of relations is fundamental to knowledge and understanding and to the growth of knowledge and understanding, whether this be in literature or science. It is by way of analogy that we have metaphor, simile, allegory, proportion, mathematics, poetry and symbolism ...*" (Leatherdale 1974 p 127)

It appears to the author that there is a potentially fruitful line of enquiry here. Can the use of analogies convey some of the abstract computing concepts that are needed for programming? A good example is the distinction between actual and formal parameters. This is a difficult concept to explain, even using sample code, and the author usually falls back on an analogy about mealtimes such as breakfast, lunch and supper versus the types of food consumed at each meal. The formal parameter is the mealtime and the actual parameter is the type of food offered at each meal.

The author has done no work on the effectiveness of what are fairly ad hoc analogies, offered in lectures, but it may be useful to attempt to unpick the effects such analogies have on the understanding of novice programmers. Until such investigation is done, the author feels that arguments for and against the use of analogies can be advanced. Some students fasten on the analogy and ignore the system that is being described. There are no definitive answers at the moment.

Learning styles

One focus of research in education and psychology has been the idea of learning styles: the notion that people have preferred ways of engaging with the tasks involved in learning. Taylor (1997) states, *"A learning style is a way in which a learner begins to concentrate on, process, and retain new and difficult information..."* (Taylor 1997 http://www.br.cc.va.us/vcca/i11tayl.html 09/12/2000)

Learning style thinking draws heavily on psychology and other areas of research:

> *"Learning styles research is drawn out of studies about the psychological, social, and physiological dimensions of the educational process. It has yet to be precisely (or singularly) defined. Still, the scholarly literature provides a range of working models that can help us deal with some of the mysterious terrain between teacher & learner."* (http://www-isu.indstate.edu/ctl/styles/learning.html 08/12/2000)

There are many ways to measure or quantify a learning style since a learning style is a model that attempts to capture a particular approach to learning taken by an individual. There could be as many models as there are researchers about learning. *"The aim of learning style research is to find clusters of people who use similar patterns for perceiving and interpreting situations."* (http://www-isu.indstate.edu/ctl/styles/learning.html 08/12/2000)

If we can quantify, even broadly, the ways that people handle new information then we can draw some conclusions about how best to support and foster learning for individual learners. In learning styles, the emphasis is on the individual but also on placing that individual within a pre-defined point on the spectrum presented by the learning style model, much as with the personality tests. Personality tests form one of the learning styles categories discussed below.

A useful classification of various learning style models can be found at http://www-isu.indstate.edu/ctl/styles/articles.html. This puts work on learning styles into four categories, according to the type of learning style model: instructional preferences, social interaction models, information processing and personality.

Under instructional preferences are cited the work of Canfield, (the Learning Styles Inventory Manual) and Dunn and Dunn. Under social interaction models are cited the work of Grasha and Reichmann (mentioned earlier in this chapter), and the works of three separate authors, Perry, Mann, and Belenky. Under information processing are listed three of the most well known learning styles models: the work of Gardner, Kolb and Gregorc. Each of these will be discussed briefly. Under personality are cited the MBTI and the Keirsey tests. It should be noted that the distinction between personality tests and learning styles is blurred. Someone's Keirsey type (SP, NF, SJ, NJ) can be argued as a predictor of how they will approach, evaluate and integrate new information or skills. (http://www-isu.indstate.edu/ctl/styles/articles.html 08/12/2000)

The work of Howard Gardner

In the 1980s Howard Gardner at Harvard developed a model of intelligence that moved away from the monolithic definition of intelligence, used particularly by proponents of the intelligence test as embodied in the classic (and somewhat discredited) IQ tests by Alfred Binet.

Gardner identified seven discrete intelligences (in 1996 he added an eighth). These are:

- Linguistic
- Logico-mathematical
- Visio-spatial
- Musical
- Bodily - kinaesthetic
- Interpersonal
- Intrapersonal
- Naturalistic

It should be noted that intelligences are not in themselves learning styles but tend to influence strongly the preferred or most productive ways of learning for a learner. Often lecturers and teachers design material that fits in with their own style of intelligence style and pupils or students with a different intelligence style find the tasks unproductive or meaningless. Gardner suggested ways in which each intelligence might best be served in a classroom environment. For example, linguistic intelligence might draw on storytelling, debate, or creating television advertisements. Logico-mathematical intelligence might be most productive in creating a time-line or analysing similarities and differences.

The work of Kolb

Kolb's model was of a four stage, cyclical model of learning, reflecting the stages the learner moves through. The model can be entered at almost any point in the cycle but the stages follow each other. "*Kolb refers to these four stages as: concrete experience (CE), reflective observation (RO), abstract conceptualization (AC) and active experimentation (AE).*" (http://www.chelt.ac.uk/el/philg/gdn/discuss/kolb1.htm 09/12/2000)

A key feature of Kolb's model is that the cycle should be iterative. Learners should engage in meaningful reflection and preparation and receive appropriate feedback at relevant stages in the learning task. The model, ideally, loops, giving the learner many chances to reflect on previous learning as part of preparation and development.

Another critical aspect of this model is the founding of learning upon concrete experience. This does not mean the learner has to have done something related to the topic but that there is some experience of the topic to be learned: "*The cycle begins with a concrete experience, (doing) which can be* [] *the experience of reading about someone else's experience.*" (http://panizzi.shef.ac.uk/medtl/lrnjrnl.html 09/12/2000)

The work of Gregorc

Gregorc created the Mind Styles Model with the four style types: Concrete Sequential (CS), Abstract Sequential (AS), Abstract Random (AR) and Concrete Random (CR).

Gregorc's work focuses on two aspects of learning: perception and ordering. Ordering has two dimensions: sequential and random. Perception has two dimensions: abstractness and awareness. Other aspects not measured by this learning style evaluation method are, say, the learner's preference for inductive or deductive reasoning or their personality profile as regards extroversion/introversion.
CS learners tend to be direct, practical learners who thrive on order. AR people are sensitive, intuitive and prefer unstructured learning environments. AS learners have excellent reading, writing and verbal abilities. CR people prefer trial-and-error approaches, they thrive in a competitive environment and are often instigators of change.

Gregorc is cited as having made strong claims for his Mind Styles (which has an associated diagnostic tool, the Style Delineator): "*Gregorc contends that strong correlations exist between the individual's disposition, the media, and teaching strategies...*" (Taylor 1997 http://www.br.cc.va.us/vcca/i11tayl.html 09/12/2000)
It should be noted that Gregorc has strong views on the appropriate use of the Delineator and application of Mind Styles. "*My position is that analyzing styles for children and youth may not only be unproductive, it can be unethical and immoral...*" (Gregorc 2000 in a personal communication to the author)

In Gregorc's view, to place learning style labels on a young person may harm or hinder that person's development. He concludes, "*That is SERIOUS accountability!*" (Gregorc 2000 in a personal communication to the author) Gregorc has a well-developed philosophy surrounding the use of his work: for more information see the FAQ section at http://www.gregorc.com/

In conclusion, we can see that the field of aptitude tests, personality tests or profiles and learning styles all have something to offer the educational process but there are no definitive answers on the use of such models or tools within the teaching of programming.

We might feel, as tutors, that some of the models described above are more immediately accessible or relevant to the teaching of programming. We could then look for ways to use those models in the teaching of programming, in a transparent and ethically sound way. For example, we could ask students to take the Keirsey temperament test and reflect on their type as a basis for their learning experiences.

This approach would contrast with most of the research on novice programmers done to date. McKeown and Farrell note:

> "*Much of the research into novice computer programmer pedagogy has focused on very specific programming constructs and the errors associated with them* [] *or on the educational value of teaching programming...*"
> (http://homepages.dsu.edu/mckeownj/CPCCCSCpaper.html 04/12/2000)

We could model, in an explicit fashion, the learning cycle of Kolb. Any of these could, with care, form a foundation for a clearly articulated pedagogy within the teaching of programming. We would have moved away from a language or system oriented approach and begun to focus on the learner within a specific context. This is a point the author returns to in a later chapter.

Chapter 3: Conclusion

In summary, then, we can see that the teaching of introductory programming presents a number of significant issues, including the language chosen, the computing concepts used to underpin programming, the programming concepts and the ways in which those concepts are linked to, or embodied in, the programming language. It is hard to meaningfully separate the technical from the pedagogical. For example, the choice of first language should be more than a decision about what language is popular or fashionable, or what the staff are well-versed in: it should be founded on a coherent educational rationale.

This chapter has also looked at some of the human aspects learning to program. The question is, What is it about programming and programming languages that people find so hard? What is it about the way people think and perceive the external world, which makes programming such a potentially error-filled undertaking?

This chapter has presented a heterogeneous collection of ideas, perhaps best described as being linked only by virtue of the fact they are not focused on a technology, such as sophisticated programming environments, intelligent tutoring and software visualisation approaches but on a human centred approach. Psychology has perhaps some useful things to say about how people organise ideas inside their heads and how those ideas are stored and accessed, on the level of mental models (rather than a neurophysical level, which is not discussed in this chapter.) Psychology also has useful insights into the limits and quirks of human memory.

In summary, than, there are no simple answers. Problems, on various scales, can be identified but definitive, easily-implemented human-centred solutions are not be found.

The next chapter will examine some of the approaches to, and research in, the teaching of programming. It will examine concepts such as debugging, software visualisation, programming environments and industry-based approaches such as extreme programming.

Chapter 4 Some research efforts in the teaching of programming

Introduction to Chapter 4

This chapter will look at a diverse set of some of the various aspects of teaching programming that have been explored by researchers, including debugging, software visualisation, programming environments and extreme programming. These elements are linked only by the fact that researchers in the teaching of programming have found them worthy of attention. What are not addressed here are the pedagogical aspects of teaching programming: research efforts seem to be largely focused on technical solutions to perceived teaching and learning issues.

Issues in debugging

The biggest problem for most novice programmers is that their code does not compile. It is not a question of eliminating errors of logic and ensuring the elegant design of code that executes; it is simply to get the program to compile at all. Beginners often struggle to get programs past the compilation stage: their effort is not directed at ensuring any sensible functionality but at achieving the magic result of the program actually compiling!

The key issue for novice programmers is knowing what to look for when a program does not compile. McKeown and Farrell note "*Error detection and correction is recognized as a central problem among novice programmers.*" (http://homepages.dsu.edu/mckeownj/CPCCCSCpaper.html 04/12/2000) and Shneiderman (1980) says, "*Debugging is a challenging task...*" (Shneiderman 1980 p 53) Leventhal (1993) says, "*Among novice programmers, testing and debugging appear to be particularly problematic.*" (Leventhal 1993 p 94)

It should not be assumed that errors at the compilation and execution stages are a problem only for novices. Krems (1995) notes:

> *"One of the most common, time-consuming, and also frustrating tasks in programming is the identification, location and correction of errors. It is estimated that half the costs which accrue during software development have to be spent for the testing and de-bugging of programs."* (Krems 1995 p 241)

Debugging is more than the correction of syntactical errors. Syntactical errors include (but are not limited to!) missing or misplaced semi-colons, missing or misplaced commas and missing single quotation marks around output strings. Programs may also have errors of logic and semantic (and algorithmic) faults, such as endless loops or IF..THENs that generate odd results. Unsurprisingly, Krems (1995) notes that expert programmers are "*superior to novice programmers especially in the identification of logical or algorithmic bugs*." (Krems 1995 p 242) Novice programmers can learn to identify missing commas, brackets and so on but errors of logic that generate odd or unexpected results are more difficult to identify and correct.

Eisenstadt (1997) notes:

> *"Despite the availability of industrial-strength debuggers and integrated program development environments, professional programmers still have to engage in far more detective work than they ought to."*
> (http://lieber.www.media.mit.edu/people/lieber/Lieberary/Softviz/CACM-Debugging/Hairiest.html 18/04/2001)

A key area of research into learning programming is the examination of the expert programmer. Putting aside questions about how one decides who is an expert programmer (is the number of lines of code produced in a programming lifetime or is it the ability to produce closely-textured, highly-wrought code, or does it span the whole spectrum of coding, from coding, commenting to maintenance?) the focus on how expert programmers do what they do has generated some interesting work on programmers' mental models.

Wiedenbeck and Fix (1993) describe the critical role of the internal mental representation of the program in code debugging and modification. This internal representation is the programmers (or readers) understanding of the code. The authors describe how such representations can be examined for novice and expert programmers and a clearer grasp of what makes the experts representations different can be gained. For Wiedenbeck and Fix (1993) the expert's representations are:

> *"...hierarchical and multi-layered;* [] *it contains explicit mappings between the different layers;* [] *it is founded on the recognition of basic patterns;* [] *it is well grounded internally;* [] *it is a well-grounded in the program text."* (Wiedenbeck and Fix 1993 p 794)

The authors go on to hypothesise that novice programmers' representations will lack most of the above characteristics. Their method was to give novice and expert programmers a Pascal program of 135 lines and then ask questions about it. The questions were designed to elicit information about the representations' characteristics.

Their conclusions support their initial hypothesis that expert programmers' mental representations of a program differ qualitatively from those of novice programmers. The authors' suggestions for improving the internal representations of code by novice programmers are fairly general in their tone:

> "*...good mental representations are likely to be developed as students carry out programming tasks such as debugging and modifications of numerous programs, and learn from a distillations of these experiences what characteristics are important to support programming tasks.*" (Wiedenbeck and Fix 1993 p 809)

Krems (1995) sums up the activity of debugging as "*a goal-guided search through a program text.*" (Krems 1995 p 242) Activities involved in this search include the evaluation of error messages (difficult for new programmers), observation of what the program actually does (versus what its writer expects it to do) and "*mental evaluation of code segments...*" (Krems 1995 p 242)

Efficient debugging of code, especially long code segments, requires the programmer to choose a segment for inspection and decide which of the debugging strategies to employ. Krems (1995) notes that "*experts do not generate better initial hypotheses than novices but are able to modify their error related assumptions in a much more flexible manner.*" (Krems 1995 p 243) Experts are better at analysing the program at an algorithmic level, whereas novices concentrate on surface features. Experts are also better at identifying the crucial code segments.

Experts and novices differ in the way they write programs too. Ormerod (1990) says the suggestion is that "*novices work forwards, writing a program line by line, whereas experts work backwards, breaking the program into modular units.*" (Ormerod 1990 p 72)

McKeown and Farrell note:

> *"Students often invoke one of several strategies when debugging a program. These strategies are hampered by a 'fragile' knowledge of the language... a 'trial and error' approach... and poor problem-solving strategies."*
> (http://homepages.dsu.edu/mckeownj/CPCCCSCpaper.html 04/12/2000)

Forcing novice programmers to write structured code improved their programs (reduced the number of bugs) because they were not using what Green (1983) terms a "*jump-happy*" language such as BASIC (Green 1983 p 82) but still debugging remained a problem. What Green (1983) proposed was a notation that allowed for what he termed circumstantial information to be displayed. Green (1983) notes, "*The circumstantial information needed for debugging must be also be built in.*" (Green 1983 p 83)

Draper (1996) says:

> "*'Programming' in fact requires a diverse set of skills, reflecting the many component tasks of the overall activity of "doing programming". The environment modifies what the tasks and skills are, for instance by relieving the programmer of some jobs. In a teaching context, this can mean they fail to learn some skills (possibly the most transferable and important skills).*"
> (http://wildcat.psy.gla.ac.uk/~steve/PPIG96.html 06/12/2000)

One example of this de-skilling is the way in which students do not learn or develop debugging strategies. Draper identifies the gaps in students' knowledge and understanding. Students work at the lowest level of debugging, that of identifying syntax errors and fail to look at the possible semantic or logical possibilities. Students rely on the compiler to spot bugs and do not read their code carefully. Students also fail to use debugging tools available to them. When code compiles they do not test the code, even at a basic input/output level. (Draper 1996)

Draper (1996) goes on to say:

> "*students often seem helpless, and unaware of even basic debugging ideas such as inserting tracing code to show how far the program gets before failing or what the values of variables are during a run, or how to go about discovering what an obscure error message really means.*" (http://wildcat.psy.gla.ac.uk/~steve/PPIG96.html 06/12/2000)

Bonar and Soloway (1989) argue that novice programmers generate errors in their code because they apply natural language-based strategies to their code:

"*Our key idea is that many programming bugs can be explained by novices inappropriately using their knowledge of step-by-step procedural specifications in a natural language.*" (Bonar and Soloway 1989 p 325)

This patching process is used when novice programmers reach what Bonar and Soloway (1989) call an impasse. The programming knowledge a student has is insufficient to solve or break the impasse so the student falls back upon what Bonar and Soloway (1989) refer to as SSK (step-by-step natural language programming knowledge). Bonar and Soloway (1989) argue that their study shows that, the more a student uses SSK, the more bug-ridden his or her program will be. (Bonar and Soloway 1989 p 347)

The affective aspect of debugging

One of the issues around debugging is the question of how people feel about their code being examined (critically or not) by someone else. Few authors seem to acknowledge an affective component to programming and to debugging in particular. There is, the author feels, a strong emotional element to programming: either of pride or fearfulness, for the novice programmer. A new programmer might be very proud of his/her initial effort and to have the code severely critiqued is uncomfortable.

Conversely, the student may perceive his or her efforts as poor and fear any feedback on the material, believing it to be inferior. It can be a relief to the latter type of student to discover that the code is not too bad after all but any criticism or dissection of any new programmer's code may lead them to conclude that they are just no good and programming and never will be.

Cooper (1985) in *Teaching Introductory Programming* has pragmatic advice that takes account of the personal reactions of the student to assessment.

Cooper (1985) says:

> "*avoid putting the student on the defensive ... every program, no matter how atrocious it is, has something good about it. You must find a feature worthy of congratulation.*" (Cooper 1985 p 113)

Solomon (1986) discusses the role of bugs in learning to program. For Solomon, bugs are to be welcomed:

> "*In this culture collecting bugs is an important activity. The essential idea is that we can classify and talk about bugs; they are not just unpleasant things that are embarrassing. In this culture, bugs are beneficial; they help us to learn. [] The message that emerges is that debugging is an enriching experience.*" (Solomon 1986 p 127)

What Solomon (1986) is partially addressing is the affective component of programming: making bug detection and correction an explicitly supported and productive activity, rather than an emotion-fraught search for mistakes.

Learning about bugs, correcting them and discussing them may give users a metalanguage, a language for talking about what it is to program. Can the development of such skills affect other areas of learning? And if it does, how can such an effect be measured? Solomon (1986) notes that evaluation seems to fall into two categories: either a project is closely evaluated using instruments such as test scores on standardized tests or it is evaluated entirely subjectively: what Solomon (1986) terms "*influence without evaluation...*" (Solomon 1986 p 128) Solomon argues that the key to evaluating a programming language (such as LOGO) and its impact on children lies in asking the right questions. Formulating the right questions is not a trivial matter.

In programming, the emphasis should be on a community of users, some of whom will be expert: usually but not solely the teachers, in a classroom situation. For example, LOGO is seen by Papert as a culture, where the use of LOGO is a shared experience, rather than a solitary one:

> "*Often Piagetian learning a la Papert is interpreted to mean that children do not need help from experts, that exploring LOGO without human intervention is sufficient. This was not Papert's intention. LOGO provides an environment and*

culture in which expert and novice can find common ground to discuss their research and the bugs they encounter." (Solomon 1986 p 131)

This argument shades neatly into the concept of ego-less programming – the move away from the proprietorial attitude of the programmer to his/her program and towards an open forum where debugging is a communal enterprise. Whether novice programmers would be comfortable with such an approach is debatable. The author's own experience leads her to believe that such an open discussion could only be carried out where code was submitted anonymously: new programmers often do not have the confidence to offer their work to colleagues for discussion and analysis.

Allied to the question of the affective component is the question of the individual learner, which seems to be rarely addressed in research on teaching programming: many projects seem to assume that the learner audience is homogenous. An exception to this is the work of Jenkins and Davy (2000). Jenkins and Davy (2000) took a cohort of undergraduate, first year programming students and divided them into four broad sub-groups, according to their skills and proficiencies:

"*Rocket Scientists – who are already highly proficient programmers, and who would learn little new from the module.*
Strugglers – students who will find the course challenging, but who would be expected to pass reasonably well.
Serious Strugglers – students who will find the course extremely difficult, and who will not pass without significant additional support and encouragement.
Averages – those who remain; they will pass the module well, and will need only an occasional word of advice or help with debugging."
(http://www.ics.ltsn.ac.uk/pub/conf2000/Papers/jenkins.htm 13/02/200)

The different groups of students had differently structured teaching. For example strugglers had supervised lab sessions with intensive tutor support (one member of staff to 10 students). This group received the most tutorial support. Rocket scientists, in contrast, received less tutorial help but were given the option of undertaking a programming project that would challenge and engage them. Jenkins and Davy (2000) concluded,

"*The attempt to classify students has been a success. The module results showed fewer students failing than in previous sessions, and certainly fewer students appeared during the semester in hopeless situations. It seems that the various routes*

> *were effective in helping all the students learn as well as they could.*" (http://www.ics.ltsn.ac.uk/pub/conf2000/Papers/jenkins.htm 13/02/2003)

This is a blend of Solomon's (1986) two evaluation approaches described earlier: some subjective reporting of 'how the project went' and some more formal, number-based results. It does require some careful unpacking of the results elements: how many is fewer failing students? Is the cohort larger or smaller that last year's? What is meant by 'hopeless situations'? What also is not discussed is the question of whether students were told clearly their group's classification and how students felt about being classed as average or strugglers.

Debugging and large scale programming

Another question is that of scale. Novice programmers write short programs of 50 lines or less. Writing larger programs is not a matter of hacking out more lines in the same way as for a 10-line program. McConnell notes:

> "*Non-professional programmers ... people who do some programming but whose primary training and expertise lies elsewhere ... can usually muddle along quite well on small projects. They learn enough about tools along the way to get the job done. However, people who have written a few small programs sometimes think that writing large, professional programs is the same kind of work ... only on a larger scale. It is not.*" (http://www.computer.muni.cz/cse/cs1996/c2062abs.htm 03/05/2000)

Along with the question of scale is the question of programmer productivity. Rawlings (1997) says:

> "*Many companies in the software industry expected the average programmer to generate no more than about twenty blemish-free lines of computer instructions a day – around five thousand a year – regardless of the computer language being used.*" (Rawlings 1997 p 69)

Commercial software has reached staggering levels of complexity:

> "*Today, even the software for a small telephone exchange would be around two kilometres long [when printed out]; the space shuttle's on board controller's about four kilometres; the Hubble space telescope's about eight kilometres ...*" (Rawlings 1997 p 78)

It is important to remember the people writing this software all were once novice programmers. What is taught when a students is still writing only 10 line programs may have significance when s/he is writing 100 line modules as part of code 100 000 lines long.

One idea that seems to have much promise but that seems to elude researchers is the automated debugging of code:

> "*Over the past decade automated debugging has seen major achievements. However, as debugging is by necessity attached to particular programming paradigms, the results are scattered.*"
> (http://www.irisa.fr/manifestations/1995/AADEBUG95/welcome.html 12/11/2000)

Green (1980) notes that two approaches to the problem of programming are first, to educate everyone differently "*preferably starting 20 years ago*" (Green 1980 p 272) or await the arrival of "*technical wizardry that's said to be just around the corner, such as Jones's Automatic Bug Detector.*" (Green 1980 p 272)

The issue of debugging is not a simple one. In a sense, it underpins all of what it means to program. The author would argue that it is a vital skill that becomes increasingly important as the complexity of code increases, even taking into account all the strategies for handling large scale programming projects. It is also an aspect of programming that is generally not made explicit in teaching texts.

Approaches to teaching programming using software

One approach to teaching programming is to support the creation and debugging of programs with software tools. This category, on investigation, proves to encompass a wide range of tools, ideas and approaches, from fairly simple to very complex. For example, a syntax editor (allowing the user to edit the lines and chunks of code) can be seen as a simple idea, although it may be complex in its implementation. Software visualisation tools range from graphic design approaches to very complex, dynamic representations of programs.

Editors

An early example of a syntax directed editor is Brad Templeton's ALICE, developed in the mid 1980s to run on the Atari ST and the IBM PC. A syntax directed editor works at the construct level rather than the text level. "*The units you work with are* [] *terms, expressions, statements and blocks*." (http://www.templetons.com/brad/alice.html 26/10/2000) The idea of ALICE, Templeton says, can be seen in the visual, integrated programming environments available today such as Microsoft's Visual Basic. One ALICE's key features was debug mode – being able to step through a program sequentially, to track bugs.

The syntax approach can be contrasted with the low-level text editors that can be seen as working at a line-by-line level. Over and above the syntax editors could be placed semantics editors. These increasing levels of complexity all present different cognitive challenges to the novice programmer. Green (1990 b) stated:

> "*Low level text based environments are still commonest but structure-based editors, usually built on syntactic structure, are becoming more common. More recently, editors based on semantics have been investigated.*" (Green 1990 (b) p 21)

Programming semantics is about the meaning of instructions within a language. Syntax is about the format of those instructions. Misspelling a command is a syntax error: typing in a command that exists in the language but is not appropriate in its current context it is a semantics error.

Hauswirth et al. (1998) note that there are several approaches to the study of programming semantics. "*Existing approaches to semantic description may be categorized as axiomatic, denotational and operational*." (http://www.infosys.tuwien.ac.at/Staff/pooh/papers/Simplesem/1190-HTML.html 03/05/2000)

Axiomatic semantics are

> "*A set of assertions about properties of a system and how they are effected by program execution. The axiomatic semantics of a program could include pre- and post-conditions for operations.*"

(http://foldoc.doc.ic.ac.uk/foldoc/foldoc.cgi?query=axiomatic+semantics 19/02/2003)

Denotational semantics are "*A technique for describing the meaning of programs in terms of mathematical functions on programs and program components.*" (http://foldoc.doc.ic.ac.uk/foldoc/foldoc.cgi?query=axiomatic+semantics 19/02/2003)

The operational approach to semantics involves examining how the state of a computing machine changes during program execution. Elements examined include the stacks and registers.

Hauswirth et al. (1998) describe the environment the authors call SDE, which allows students a view of the system's operational semantics. Hauswirth et al. (1998) describe the software that forms a dynamic abstract machine, allowing the user control of the contents of memory and flow of execution:

> "*SDE is a graphical java-based environment that allows the student to edit, run and debug SIMPLESEM programs. SDE animates SIMPLESEM executions by visualizing the code and data memories of the SIMPLESEM processor.*" (http://www.infosys.tuwien.ac.at/Staff/pooh/papers/Simplesem/1190-HTML.html 03/05/2000)

The SDE supports a graphical view of SIMPLESEM programs, using a tabular format. The student types in code in the line-oriented editor. Either side of the code window are breakpoints and data windows. The SDE can be accessed over the Internet, with no installation needed on the computer the user is working on. SDE can also be downloaded. Java's security features handle security. The authors note, "*the basic idea of SDE is to visualize abstract concepts that have no physical counterparts.*" (http://www.infosys.tuwien.ac.at/Staff/pooh/papers/Simplesem/1190-HTML.html 03/05/2000) The authors go on to describe how the SDE has been adopted by a number of programming teachers but offer no formal evaluation.

SDE is a system that uses both operational semantics and graphical elements. The graphical elements are an aspect of software visualisation, which is discussed in a later section.

Intelligent Tutoring Systems

Another approach is to adopt elements of AI (artificial intelligence) and create an ITS (Intelligent Tutoring System). An example of an ITS was MENO-II which was:

> "*...a computer-based tutor intended to help novices learning to program in Pascal. The BUG-FINDING component attempts to find non-syntactic bugs in a students program. It draws on a database of 18 common types, represented as templates, and attempts to match these templates against its analysis of the student's program. The TUTORing component then attempts to infer the misconception that might underline the bug and present the student with remedial instruction.*"
> (http://www.csa2.co.uk/htbin/ids51/procskel.cgi 02/08/2000)

Work on MENO-II was done in the early 1980s and the article about MENO-II by Soloway et al. was published in 1983.

Stanford's BIP Project used AI to provide a "*computer-based programming laboratory...*" (Barr et al. 1976 p 567) BIP's features included a BASIC interpreter, a bank of 100 programming problems, a HINT system to give textual and graphical help to students, a student manual and a mechanism for individualized task selection. The authors describe the initial use of BIP as demanding of staff time but "*effective*" (Barr et al. 1976 p 567) although the authors identify elements for further development such as improved task selection.

An intelligent-processing approach is also described by van Merriënboer et al. (1992). The system the authors created was centred on:

> "*An instructional strategy for teaching introductory computer programming which focuses on the completion of increasingly larger parts of well-designed, well-readable but incomplete computer programs.*" (Merriënboer et al. 1992 p 62)

Merriënboer et al. (1992) wrote a system to automatically generate incomplete assignments for students to finish. The system CASCO (Completion ASsignment COnstructor) consisted of three knowledge bases: a model of the knowledge to be taught, a student profile consisting of four sets of plans and a database of problems to be drawn upon. Each assignment offered to the students contained a programming problem, an example program, task instructions, explanations on new features and questions about the structure of the program.

The student profile was not a full model of student learning. The sets of plans were a declarative description, partitioned into four categories. Set N was material not yet presented, Set P was material presented but not applied (by the student), Set E was material used by the student but not yet fully learned and Set A was material assumed to be learned to the point of automatic use by the students. The system chose a subset of sets N, P and E at each stage, to present to a student.

Merriënboer et al. (1992) sum up their view of the strengths of CASCO:

> "*CASCO may be argued to fulfil the three requirements [] for the selection and sequencing of problems and examples in procedural tutors: Manageability, structural transparency, and individualisation.*" (Merriënboer et al. 1992 p 74)

Interestingly, the authors also note that their focus is automatic task generation, rather than teaching programming, and propose other subjects areas to which the system might be applied, such as circuit design or planning production processes.

Designing new programming languages

Another approach to the teaching of programming is to add to the plethora of programming languages available by designing a language that addresses some of the problems identified with older, more widely used (and criticised) languages. The idea has already been touched upon in Chapter 3, with reference to GRAIL.

As Spooner (1986) noted, the challenge for language designers is to create a language that is as easy to use as a natural language "*and so adaptable that for all one's dealings with the computer one need never use a different language.*" (Spooner 1986 p 215) Spooner (1986) describes the creation of ML, the language that combined precision of meaning (and levels of meaning) with excellent readability. To this end, the designers used general constructs and a "*small orthogonal set of devices*" (Spooner 1986 p 221). (Orthogonal refers to the mutual independence or separateness of the language's constructs.) The richness of the language lay in the combination of its elements. Spooner (1986) concludes that mistakes were made in the early design of the language and that some definitions were clumsy. Nonetheless, Spooner (1986) sees ML in a developed form as being "*most promising*" (Spooner 1986 p 236)

Other languages have been designed with similar goals and their creators make similar claims:

> "*Euphoria is a powerful yet simple programming language, developed by Robert Craig at Rapid Deployment Software in 1993. It is very easy to use, and it has good support, which makes it an excellent language for novice programmers. Euphoria is an interpreted language, just like AWK or QBasic.*" (http://www.engin.umd.umich.edu/CIS/course.des/cis400/euphoria/euphoria.html 04/12/2000)

Another programming language developed to teach programming is Turing:
"The Turing language was developed in 1982 at the Dept of Computer Science at Toronto University, Canada. [] Fourteen years later, Turing is used in 65% of Ontario High Schools and provides many Computer Science students with their first exposure to computer programming. In 1993, Turing was adopted by Imperial College, London for use in their introductory MSc and Undergraduate Computing Courses."
(http://www.holtsoft.com/turing/essay.html 31/10/2000)

Martin (1996) goes on to describe one of the key features of Turing as its detailed diagnosis of errors, to aid the student in debugging. Martin (1996) concludes:

> "*Turing students spent less time and needed less assistance with syntax correcting than Pascal students. They were also more likely to resolve syntax errors on the first attempt and less likely to require assistance in doing so. The Pascal students were more likely to be misled by syntax errors and to alter the logic of the program in the syntax correcting process.*" (http://www.holtsoft.com/turing/essay.html 31/10/2000)

Other approaches were more sweeping still:

> "*This effort intends to improve the state of the art of computer use, not by introducing, nor even (primarily) through new software, but simply by empowering*

> *all users to be computer programmers.*" (http://www.python.org/doc/essays/cp4e.html 09/03/2003)

The CP4E project (Computer Programming for All) was based on the use of the Python language, with emphasis on developing aspects of the language for ease of use and the careful development of teaching materials. The website noted: "...*we want to explore the notion that virtually everybody can obtain some level of computer programming skills in school, just as they can learn to read and write.*" (http://www.python.org/doc/essays/cp4e.html 09/03/2003)

In fact, the CP4E project was under funded (according to its participants) and was left incomplete:

> "*Unfortunately, the move of the Python development team to another employer* [] *meant that we didn't get to complete the CP4E project at CNRI. This move was motivated in a large part by the disappointingly small amount of funding that DARPA committed to CP4E.The project is now in limbo...*" (http://www.python.org/cp4e/ 06/05/2003)

Software and program visualisation

A richly featured and complex area of research has sprung up around the concept of software visualisation. SV (software visualisation) is a broad field, encompassing any graphical display of the behaviour and nature of software, from design to program execution. It is not necessarily a new field of research: " ... *the history of SV dates back to the days of von Neumann in the 1940's.*" (Price et al. 1998 p 3)

A distinction needs to be made between software visualisation and visual programming. Visual programming is the creation of new programs using graphical programming environment, such as Visual Basic. Visual Basic uses graphical items such as scroll bars, command buttons and text boxes which are dragged and dropped into a form or forms. Software visualisation is the manipulating of conventional text based code so that some aspect of the code such as its activities, flow of control or constructs are graphically represented.

By 1994, the SV approach had matured sufficiently for two researchers in that field to claim, "*Much of the recent research in software visualization has been polarized towards two opposite domains.* " (Mukherjea and Stasko 1994 p 215) The authors describe these two areas of research as being at the data structure level: (low-level views, generated automatically that are useful for debugging) and "*algorithm animation*" (Mukherjea and Stasko 1994 p 215)

As Mukherjea and Stasko (1994) note the primary use of algorithm animation was for teaching programming. The authors go on to note "*algorithm animation could offer key benefits to program de-bugging and testing.*" (Mukherjea and Stasko 1994 p 217) The animation of a program's algorithm can be a critical tool for understanding what the program is doing (or not doing correctly!) Mukherjea and Stasko briefly review systems such as Incense (which is data structure display system) Movie (which is an algorithm annotation system) and the Gestural, Dance and Illustration systems.

Mukherjea and Stasko (1994) characterise their own graphical system, called Lens, as being suitable for high level debugging, and one that allows iterative testing and refinement. The goal of Lens' authors was to create "*a system that can provide application specific animation news for de-bugging purposes.*" (Mukherjea and Stasko 1994 p 218)

The Lens system was designed to allow programmers to design their own innovations "*without having to learn a graphics toolkit and write code using it.*" (Mukherjea and Stasko 1994 p 218) Other graphical systems include Dynalab and Jeliot.

DynaLab was a project at the Montana State University from 1991 to 1994:

> "*DynaLab, an acronym for DYNAmic LABoratory, is an ambitious software project conceived as the infrastructure for supporting formal laboratory experiments in the introductory computer science curriculum.* [] *In its current incarnation, DynaLab provides a dynamic,interactive, hands-on software environment for studying virtually all aspects of programming, as well as time and space complexity issues. The current version of DynaLab supports program animation of Pascal programs, with unbounded reverse execution for repeated study of puzzling program constructs.*"
> (http://www.cs.montana.edu/~dynalab/description/index.html 08/12/2000)

Jeliot, developed in Finland, concentrates on animating Java programs over the Web:

> *"The algorithm animation environment Jeliot allows a web user to visualize his own algorithms, written in Java(tm), over the Internet. Jeliot is based on self-animating data types: the user selects the visualized data objects of the source code, and Jeliot produces the animation."*

(http://www.cs.helsinki.fi/research/aaps/Jeliot/ 08/12/2000)

Domingue and Mulholland claim that their software visualisation approach, for Prolog, supports both synchronous and asynchronous support for student programmers as students can create live animations of their code, using broadcast mode, or record the animation for future viewing. The researchers are working on strategies to allow students to retrieve animations of algorithms similar to the ones on which they are working, using two different search approaches. (http://kmi.open.ac.uk/people/paulm/aied/ 04/12/2000)

The work of Domingue and Mulholland makes it clear that algorithm animation is a significant area of research:

> "*Recent work within the Algorithm Animation community has lead to the creation of a number of systems which allow pre-written animations to be viewed remotely. These systems primarily concentrate on delivering animations synchronously as part of a classroom style tutorial.*" (http://kmi.open.ac.uk/people/paulm/aied/ 04/12/2000)

It is useful to distinguish between software visualisation and program visualisation. SV (software visualisation) is the use of a range of features such as typography, design of graphics and HCI (Human Computer Interaction) principles to enable people to use/understand software. Program visualisation is the "*visualization of actual program code or data structures in either static or dynamic form.*" (Price et al. 1998 p 4)

The static and dynamic distinction is significant. Static visualisation might be a well-formatted program, where a dynamic one might show changes to data as the code runs.

Baecker (1998) notes that only animation is truly appropriate for drawing the activity of a program, with its flow of control, use of memory, allocation of values to variable sand so on. Static drawings are often less than adequate. He notes "*Animation is a compelling medium for the display of program behaviour.*" (Baecker 1998 p 369) Baecker goes on to describe one of the earliest software visualisation projects, a 30-minute film on "*nine different internal sorting methods.*" (Baecker 1998 p 371)

Baecker claims that the film explains the sorting algorithms so effectively that a student can program some of those algorithms after watching the file. He says, "*The movie has been used successfully with computer science students at various levels.*" (Baecker 1998 p 371) His claim is that the film works, "*even today, 15 years later.*" (Baecker 1998 p 378)

Kimelman et al. (1998) widen the idea of program visualisation. For them programs in the real world interact with many other elements such as libraries (to handle memory and input/output) hardware. To understand and debug the system, the programmer must "*consider behaviour at numerous layers of a system concurrently. As this behaviour unfolds over time.*" (Kimelman et al. 1998 p 293) A system that aims to reflect this multi-layered complexity is IBM's prototype system PV:

> "*With PV users watch for trends, anomalies and interesting correlations, in order to track down pressing problems. Behavioural phenomena, which one might never have suspected, or thought to pursue, are often dramatically revealed.* [] *Resolution of the problems thus discovered often leads to significant improvements in the performance of one application.*" (Kimelman et al. 1998 p 294)

Brown and Sedgewick (1998) reinforce the distinction between program animation and algorithm animation in stating that most program visualizations can be created automatically whereas most algorithm animations cannot. Brown and Sedgwick say:

> "*most algorithm animations displays cannot be created automatically because they are essentially renderings of the algorithm's fundamental operations; an algorithm's operations cannot be deduced from an arbitrary algorithm automatically but must be denoted by a person with knowledge of the operations performed by the algorithm.*" (Brown and Sedgwick 1998 p 156)

Price et al. (1998) create a taxonomy for SV(software visualisation) systems, which uses six top-level categories: scope, content, form, method, interaction and effectiveness. Each of these categories can have sub-categories, which may in turn have sub-sub-categories. Each category can be described using yes/no questions (Does the SV code handle concurrent code?), a range of functionality (to what degree does the SV code ...?) or a set of attributes. See Price et al. (1998) pp 3 - 27

Price et al. (1998) conclude:

> "*over one hundred fifty software visualization prototypes and systems have been built in the last twenty years, yet very few of these systems were systematically evaluated to ascertain their effectiveness.* [] *The most disturbing observation is the lack of proper empirical evaluation of SV systems, for if the systems are not evaluated and shown to be effective what is the point of building them?*" (Price et al. 1998 p 27)

Algorithm animation

With the arrival in the 1980s of high resolution, colour monitors the way was open for the development of graphical views of code:

> "*The most important and well known system of the new era was BALSA ..., followed by BALSA II, which allowed students to interact with high level dynamic visualizations of Pascal programs.*" (Baecker and Price 1998 p 34)

BALSA animated algorithms and produced graphical representations of the program's activity:

> "*An impressive piece of development, it uses both topological and metric conventions in representing program activity. Many of the outputs of the system are histogram-like in nature, intending to demonstrate the differences in effectiveness between alternate algorithms. Brown claims the system has helped computer scientists discover flaws in widely used algorithms.*"
> (http://www.nickerson.to/visprog/ch2/progvis24.htm 20/02/2003)

North (1998) says, "*Graphs are suitable models for software.* [] *graphs are well-understood models in many domains for representing relationships between abstract objects.*" (North 1998 p 63) For North the crucial element in employing graphs to describe code and its activities is the readability of the graph. There are always conflicts between optimising readability and the computation or processing required for effective visualisation (North 1998 p 64)

What emerges, says North (1998), is often in collection of heuristics (general, practical rules) rather than exact solutions to the graph creation problems. For more detail see North (1998) pp 64 - 66. North goes on to describe the dot system, which consists of several programs allowing users to create software visualisation systems. North (1998) reviews several software visualisation systems, including CIAO, Improvise and VPM.

What need to be noted here that algorithm animation or program animation are potentially complex solutions to the question of making a program or programming activity more understandable. The author questions whether adding a layer of complexity, whatever its purpose, will truly aid the novice programmer, at least in the earliest stages. Green (1990 b) notes that, at the time of writing:

> "*Text-based editors remain the favourite choice, despite all subsequent developments. Present-day editors for use in programming environments provide some help with formatting programs [but] text-based editors have changed little. Their users like them partly because they are simple to understand and do not add much of an extra learning load. More importantly, they do not constrain their users in any way at all. Programs can be built in any order – top down, bottom-up, middle-out; pieces of code can be moved around the design at will...*" (Green 1990 (b) pp 27 – 28)

Having said that, there is a case for moving beyond text editors and making the code during execution more transparent to the user, a pared down animation approach.. Algorithm animation at a high level of complexity is, perhaps, a tool more suited to expert programmers than to novices. What other graphical features, less technically demanding and more accessible, might aid the novice programmer? One focus is the appearance of code, how it looks on screen and on paper.

Code layout

Baecker and Price (1998) review the changes and developments in programming including structured programming techniques, the creation of more expressive and clearer languages (the authors cite Modula as an example) and human factors ideas such as chief programmer teams. The authors also mention the impact of better computing technology and the use of CASE (Computer Aided Software Engineering).

Baecker and Price (1998) conclude that all these developments (they term them 'advances') do not deal with the question of the *appearance* of code, how it looks on the screen. Baecker and Price (1998) also say that the code's appearance does not "*reflect the history of a program*" (Baecker and Price 1998 p 31) or make the structures of the code as visible as it could, nor does it "*deal ... with the fundamental problem of software comprehensibility, that of software complexity.*" (Baecker and Price 1998 p 31)

Baecker and Price (1998) go on to review the key developments in software visualisation, beginning with the layout of code on screen and paper. Flow charts were developed to make the visualisation of the program's execution easier. Baecker and Price (1998) note "*the 1970's saw the first of many alternatives to flowcharting the development of Nassi-Schneiderman diagrams ... to counter the unstructured nature of standard flowcharts.*" (Baecker and Price 1998 p 33)

Baecker and Marcus (1998) argue that two under-utilised elements of computer hardware, in terms of supporting program understanding, are the printer and monitor: "*Programs can now easily be represented using those elements therefore omitted, such as multiple typefaces; variable weights, slants, point sizes, word spacing, and line spacing; and rules, gray scale and pictures.*" (Baecker and Marcus 1998 p 45)

Baecker and Marcus (1998) go on to give detailed example of what Knuth calls 'literate programming' where the programmer aims to layout code that is human comprehensible. The literate programmer can be supported by software that automates much of the program presentation.

Baecker and Marcus (1990) claim that principles of program layout and style have altered little since the most basic of output media were available:

> "*The impoverished appearance of programs today is an artefact of the technologies available for composing and printing them in the early days of computing namely, the keypunch, the teletype and the line printer.*" (Baecker and Marcus 1990 p viii)

The C program example offered by Baecker and Marcus uses a number of layout features and design elements such as highlighting of key words, use of white space and serif font. (See Baecker and Marcus 1998 pp 45 - 49)

The authors conclude that their work on "*programs as technical publications*" (Baecker and Marcus 1998 p 60) goes far beyond the pretty printing of code: "*We have identified basic graphic design principles for program visualization and developed a framework for applying them to programming languages.*" (Baecker and Marcus 1998 p 60) Their work also involved the development of SEE visual computer, which automatically enhances C source code. The authors conclude that their work "*helps to establish programming as a literary form deserving a mature graphic appearance.*" (Baecker and Marcus 1998 p 61)

For Baecker and Marcus (1990), the use of graphic design principles is the key to their argument. The authors do not argue that the principles they derive are the only ones: others may be equally valid. They do argue that the designs derived for C (the language they use as an example) could be equally well applied to other languages such as Pascal. The design principles put forward by Baecker and Marcus (1990) allow automated visual compiling. The goal of this work is to create an "*intelligible, communicative, and attractive*" interface to program code. (Baecker and Marcus 1990 p ix)

Baecker and Marcus (1990) note: *"Teachers of programming may find that improved appearance of code and documentation helps make the literature and examples more interesting, compelling and understandable."* (Baecker and Marcus 1990 p xi) What elements of graphic design are relevant? A basic list includes, but is not limited to, typefaces, fonts, point sizes, character widths, word and line spacing, colour and tints and page layout.

For Baecker and Marcus (1990) literate programming is a completion of, and extension to, the software engineering approach. They describe it as the seventh software engineering approach, with structured programming being the first, the development of expressive programming languages (e.g. Modula) being the second, psychological/management based theories such as chief programmer teams being the third, the fourth, the high performance workstations, the fifth, CASE (Computer Aided Software Engineering and the sixth, software engineering tools and environment.

Baecker and Marcus (1990) describe the early attempts at improving program appearance, including a presentation version of the language ALGOL60, and pretty printing. Baecker and Marcus (1990) note, "*the earliest work was done on LISP, so that program readers would not drown in a sea of parentheses.*" (Baecker and Marcus 1990 p18) Other approaches include the Vgrind UNIX utility and Xerox Cedar's typesetting and formatting screen.

Baecker and Marcus (1990) put forward ten fundamental principles of program appearance with seven secondary principles, derived from the first ten. The ten fundamental principles are: legibility, readability, clarity, simplicity, economy, consistency, relationships, distinctiveness, emphasis and focus and navigability. Most are self-explanatory. 'Relationships' refers to the use of visual elements to emphasise elements of the program that have a relevant relationship. By definition, where this visual emphasis is omitted, there is no relationship to be grasped. Distinctiveness refers to the use of visual elements to show vital properties of code.

The seven secondary principles are: page characteristics, page composition and layout, typographic vocabulary, typesetting, symbolism, colour and texture and metatext. Metatext refers to the supplementing of the original code with text. Baecker and Marcus (1990) refer to "*mechanically generated supplementary text.*" (Baecker and Marcus 1990 p 45)

The focus of literate programming is making code *human* readable, an area generally neglected in coding: "*The activity of reading programs has always received far less attention than that of writing programs.*" (Baecker and Marcus 1990 p vii)

Related to this idea of graphic communication is the question of how people read code:

> "*The field of reading research illustrates the problem. Most reading research [] deals only with the readability and comprehensibility of prose. When investigating in the field study the role of format they typically consider only simple variations such as the role of typeface, case, line length and leading...*" (Baecker and Marcus 1990 p 263)

Chapter 4 Some research efforts in the teaching of programming

As touched upon in the previous chapter, we can ask if psychology can aid the researcher in supporting and fostering programming comprehension? Baecker and Marcus (1990) are not optimistic:

> "*The service of psychology* [] *especially the fields of perception, cognition, and memory deals with issues at far too low a level to tell us how programmers read and understand. Even when developed from an engineering-oriented, systems and information processing perspective,* [] *psychological theories are still too primitive for the task of modelling program reading and understanding.*" (Baecker and Marcus 1990 p 262)

Software visualisation and teaching programming

Stasko et al. (1998) note that the "*original impetus*" (Stasko et al. 1998 p 367) for research in SV was the idea of supporting novice programmers in their exploration and understanding of algorithms and code.

The authors note that SV began with students seeing animations through the BALSA system (referred to earlier) in 1983. They note:

> "*to our knowledge, no conclusive study has ever established the educational value of software visualization techniques. But the evidence of this value is overwhelming. Year after year students report, directly and an anonymous questionnaires, that algorithm animation and program visualization tools help then understand the concepts they are learning.*" (Bazik et al. 1998 p 383)

The authors then go on to describe what they see as some of the obstacles to teaching with software visualisation tools. The list is familiar to anyone who has looked at the adoption and use of CAL. The authors argue that the use of the SV software must be integrated with the curriculum and the supporting materials for that curriculum. Any tool must be "*easy to use in order to convince teachers to invest the time to learn its features and interface and adapt their courses to it.*" (Bazik et al. 1998 p 387)

The cost and effort involved in developing powerful flexible and easy to use systems is considerable. Bazik et al. (1998) discuss the development effort required for the MEADOW systems (an extension of FIELD, a UNIX-based set of tools for programming in that environment) the Piper system (graphical display of code execution) and Passé "*a self-contained Pascal development environment*" (Bazik et al. 1998 p 389).

Bazik et al. (1998) conclude:

> "*Integrating software visualization tools into the educational process is difficult for many reasons, the most crippling in the academic environment being the need for ongoing software maintenance to keep the tools working and up to date.*" (Bazik et al. 1998 p 397)

As Mulholland and Eisenstadt (1998) note:

> "*Over the past twenty years, we have undergone a change of perspective in the way we teach programming. Having begun by worrying in detail about the needs of novices and trying to understand their problems and misconceptions we developed a range of environments to help them...*" (Mulholland and Eisenstadt 1998 p 399)

Mulholland and Eisenstadt (1998) pose two questions about SV: how should SV be used in teaching programming? Can SV be used to develop novice programmers into expert programmers? They conclude that:

> "[the] *introduction of rich interactive environments will not guarantee a fruitful learning experience. Rather, the nature of the environment and the level of free exploration must be appropriate to the characteristics of the learner.*" (Mulholland and Eisenstadt 1998 p 408)

Some reservations have been expressed about the value of algorithm animation by Stasko and Lawrence (1998). They note that a passive display of an algorithm's activity has "*minimal impact on learning.*" (Stasko and Lawrence 1998 p 436)

They balance this assertion by adding: *"Algorithm animations can serve as valuable learning aids when used in the proper ways. Fundamentally, it is critical that students actively interact with the animations."* (Stasko and Lawrence 1998 p 437)

Petre et al. (1998) adopt a more cognition-based approach to assess the nature of SV. SV projects are undertaken by enthusiasts, who are, by definition, enthusiastic programmers and designers who relish the challenge of tricky animation coding and difficult interface creation. The authors step back to ask is SV "*the powerful cognitive tool it is often assumed to be?*" (Petre et al. 1998 p 454)

Petre et al. (1998) ask, is it appropriate to talk of one field for SV when the goals, purposes, audiences and subject of the visualisation can vary so widely? They cite the visualisation of large collections of data through to visualisation a virtual machine. Visualisation is also broad in its remit: from complex animations to highlighted indented source code.

A great deal of work has been done on algorithm evaluation and visualisation: there is insufficient space to describe that work here. It is worth noting that while effort is being put into graphical representation (including 3D) work is also going on in auralization of code. This is described in Brown and Hershberger (1998) pp137-143. There will be as many ways to represent what a program is and does as there are researchers looking to develop ways of presenting the nature and activity of code.

Programming Environments

Another approach to the question of making programming easier to learn and easier to do, is the idea of a coherent, integrated programming environment where all the elements such as debugger and editor work together to support not only the programmer but others associated with the development and creation of the software.

Shneiderman (1985) describes in some detail how such an environment might be constituted and what it might best do. Shneiderman (1985) notes "*No programming environment can be optimal for all programmers in all programming situations.*" (Shneiderman 1985 p 106) His chapter describes an environment to support three different work styles (individual, team and project) and two "*styles of access*" (Shneiderman 1985 p 106): command line based and menu driven.

Shneiderman's MPE (Model Programming Environment) would hold parts for the work. A part may be a code fragment, a references list or any documentation associated with software production and software production management. (Shneiderman presents a list of twelve sample types of part.) These parts can be combined into what Shneiderman (1985) terms an assembly, which is larger scale, perhaps more formal document.

Rogalski and Samurçay (1993) describe a similar concept as part of the KEOPS schema (Rogalski and Samurçay 1993 p 13) They note that the tools and environment provided can decrease the cognitive load on programmers and provide structures for handling tasks and activities. (Rogalski and Samurçay 1993 p 16)

Shneiderman notes:

> "*The MPE is designed to facilitate large project team coordination and management. The uniform user interface to multiple software tools should reduce learning time, speed performance, reduce error, increase user satisfaction and enhance human retention of commands over time.*" (Shneiderman 1985 p 107)

The central feature of the MPE is the editor. Shneiderman estimated that programmers spend 90% of their time using an editor and a more powerful editor should provide all the standard features of insertion and deletion but also provide "*cosmetic enhancement*" (Shneiderman 1985 p 121) such as line compression, highlighting and selective display, a approach that has echoes of literate programming. Shneiderman also offers the idea of syntax editing, allowing the programmer to insert a structure which requires the action condition clauses replaced with code. More dynamic features would include control structure transformations such as replacing an IF... THEN... ELSE with a CASE statement, or splitting an arithmetic expression into two assignment statements with a variable to hold a temporary value.

In fact, some of the ideas put forward by Shneiderman had been attempted by a project to develop a full support environment for Ada programming. Arblaster (1983) describes the APSE (Ada Programming Support Environment), produced by several software houses for the Department of Industry and the Ministry of Defense in the early 1980s.

The minimal APSE elements were the CLI (Command Language Interpreter), the linker, a database utility, a test and debug utility and an editor. Most people would argue that theses are the key elements of any programming language environment. What the UK APSE project considered, in some depth, were human factors influencing the design of such tools. The project evaluated these tools at the design stage as well as undertaking extensive prototyping. Arblaster's 1983 article did not report long-term outcomes from the implemented environment.

Brusilovsky (1993) claims that the following teacher activities can all be supported by various ITSs (intelligent tutoring systems): instructional planning, task sequencing, mastery learning, scaffolding, selection of program example, strategic instruction and intelligent debugging (Brusilovsky 1993 p 115). Brusilovsky (1993) describes what he calls the intelligent programming environment, which bridges the divide between ITSs and programming environments. "*Intelligent programming environments are able to offer the flexibility of micro-world like environments accompanied by an artificial intelligent coach or tutor.*" (Brusilovsky 1993 p 114)

A programming environment must include tools to enable the student to input code, use previously developed code, compile, debug, run and test code and should include a help system. An intelligent tutor should, as a minimum, handle knowledge sequencing and instructional planning, generate tasks and provide feedback to the learner. An intelligent programming environment will have tools and capabilities from both systems. This is a development of the idea of automated or intelligent debugging.

Brusilovsky (1993) describes the ITEM/IP system (Intelligent Tutor, Environment and Manual for Introductory Programming) for learning and programming in a mini-language, Turingal. Brusilovsky (1993) concludes that one of the key strengths in ITEM/IP was the help given to students in debugging: "*In about four fifths of all cases to identify the bug it was quite enough for a student to run the wrong program visually with a test data suggested by the system.*" (Brusilovsky 1993 p 122)

A slightly different approach is embodied in DISCOVER, an intelligent programming environment for novices that "*synthesizes domain visualisation and microworlds with automatic, active program diagnosis and tutoring.*" (Ramadhan and du Boulay 1993 p 125) Students can build any programs they wish in what the authors call the 'free phase'. During the guide phase, the system sets the problems and monitors the user's code. DISCOVER's language is a pseudocode style language. The authors conclude that the system "*appears to help novices in solving programming problems more accurately and more quickly.*" (Ramadhan and du Boulay 1993 p 132)

With all these projects, it is sometimes difficult not to feel that the technical challenges of building the system take precedence in the researchers'/developers' minds. How far it genuinely supports the programmer is a question that is not directly addressed. It seems to have been enough, in some cases, to have solved the programming problems inherent in creating such sophisticated systems.

Other approaches

This section looks at small range of more human-centred approaches to solving the problem of programming for new programmers: remove most of the cognitive load by using previous examples (programming by demonstration or example) or move the emphasis from individual programming effort to a shared programming style (pair programming, part of the extreme programming approach).

Programming by Demonstration

Canfield Smith et al. (2000) state baldly:

> "*since the late 1960s, program language designers have been trying to develop approaches to programming computers that succeed with novices. None has gained widespread acceptance.*" (Canfield Smith et al. 2000 p 75)

The authors go on to describe what they see as a powerful paradigm for teaching users how to get their computers to do just what the users want: PBD (Programming by Demonstration) and 'visual before-after rules' (Canfield Smith et al. 2000 p 75) The authors go on to discuss an application that models PBD, called Creator. Creator allows users to create simulations of their own and was designed for educational use.

The authors describe how they rejected any idea of a required syntax, feeling that no programming language was ever easy for a novice to learn. They decided upon the PBD approach where the software can be modified without the use of a programming language. Creator uses beginning and end states and a graphical (drag and drop) method to define and develop a rule. Canfield Smith et al. (2000) cite games developed by 11 and 12 year olds. A game where a spaceship beams up cows required only 13 rules to create. Another, more complex game required 57 rules.

Canfield Smith et al. (2000) notes, perhaps rather wearily:

> "*With all of the bright CHI* [Computer Human Interaction] *researchers in the world, it is disappointing to note that the problem of enabling novices to program computers hasn't been solved. People deserve better.*" (Canfield Smith 2000 p 92)

Canfield Smith et al. (2000) describe optimistically the potential of programming by example as embodied in Creator. The graphical approach of software such as Creator appeals, say the authors, to novices. Their argument is that this before-after rule-based approach can be expanded to any domain that is visual, capable of direct manipulation and interactive.

A related idea is that of PBE (Programming By Example). St Amant et al. (2000) note that PBE is where "*the system records actions performed by the users in the interface and produces a generalized program that can be used later in analogous examples.*" (St Amant et al. 2000 p 107)

Visualisation and PBE would use the visual features or properties of the screen elements to note user actions or inactions. St Amant et al. (2000) make it clear that there is no full-blown PBE visualisation system but regard the concept as one well worth elaborating. St Amant et al. (2000) make clear in discussing aspects of such a system that building this kind of software is not a trivial task.

Lieberman (2000) notes optimistically "*PBE is one of the few technologies with the potential for breaking down the Berlin Wall that has always separated programmers from users.*" (Lieberman 2000 p74)

It is Repenning and Perrone's (2000) description of the arena in which PBE is most appropriate that sums up the topic:

> "*End-user programming is emerging as a crucial instrument in the daily information processing struggle.* [] *Such programming allows customized personal information processing. But few end users have the background, motivation or time to use traditional programming approaches or the means to hire professional programmers to create programs for them.*" (Repenning and Perrone 2000 p 90)

Repenning and Perrone's (2000) work on programming by analogous example has concentrated mainly on the development of simulations: how far can new simulations be developed by re-using previous simulations? There are problems, as the authors note. Just changing superficial features is not enough: any software that creates new simulations would almost certainly require users to input quite detailed semantics, which possibly brings users back to more programming language-like requirement.

Extreme programming

Another approach examines a different but relevant and related issue: that of software production in a commercial environment. The approach, called XP or extreme programming is built around what Ambler (2000) describes as "*four core values: communication, simplicity, feedback and courage.*" (Ambler 2000 p 24)

The abstract defines extreme programming as a: "*lightweight software process that is aimed at small-sized teams operating in an environment of ill-understood and/or rapidly changing requirements.*" (Ambler 2000)

Linked to the idea of extreme programming is the idea of pair programming: "*Pair programming is a practice in which two programmers work side by side at one computer, continuously collaborating on the same design, algorithm, code or test.*" (Williams and Kessler 2000 p 108)

XP uses pair programming and its proponents claim it can be used with reasonable benefits by novice and expert programmers alike. Williams and Kessler cite the payroll system developed using XP. The payroll code has 30,000 methods and 2,000 classes (object oriented code) and was delivered almost on schedule: a notable achievement in large scale software development.

Williams and Kessler (2000) conclude that: "*both anecdotal and statistical evidence indicate the pair programming is a powerful technique for generating high quality software products*." (Williams and Kessler 2000 p 114)

Chapter 4: Conclusion

This review of some of the projects that have looked at the teaching of programming argues that researchers seem much more likely to look for a technical solution, whether that lies in elegant animation of algorithms or complex tools for developing and debugging code. Perhaps this should not be surprising: researchers interested in programming are quite likely to enjoy programming and the challenge of programming a system to support programming is not an easy one to resist. The systems described in this chapter all seem to have elements in common. They all appear, at least in the articles and papers that describe them, to be very complicated. Screen shots of the systems, often with multiple windows, do nothing to dispel that idea. They all seem to be very specific to the institution or team that created them. There is no indication that these systems would be easy to adopt or use in other institutions.

Work on evaluating these highly personal systems varies but generally appears to be largely empirical, with anecdotal evidence about the effectiveness of such systems in teaching and learning. There is no little or no evidence about how students perceive such systems. Are student users comfortable with these systems? How much effort did students users have to put into learning to use the system? On the whole, these questions are neither posed nor answered by researchers in the teaching of programming. These are questions that should be addressed before claims of success are made for languages or environments.

The next chapter looks at an element frequently not addressed (or even acknowledged) in discussions around the teaching of programming, that of pedagogical theories that may or may not directly support certain approaches to the teaching of programming. The author feels most strongly that this is a significant shortcoming in much of the research around this area: researchers cheerfully engage with difficult technical problems but rarely review or discuss a pedagogical theory that could be said to underpin the artefact they have created and employed. Anyone who is seeking to improve student learning must have some engagement with, and understanding of, theories of learning. The next chapter reviews, very briefly, what the author feels to be some key theories.

Introduction to Chapter 5

The two previous chapters have looked at the teaching of programming, focusing either on the concepts of structured programming or the technology that supports that learning (languages and development or debugging environments). In order to develop a coherent approach to the teaching of programming, we need an overview of some key ideas in learning.

This chapter will examine some of those theories: the narrow focus of this chapter needs to be stressed. In the space available, the author can attempt only an outline review of some of the ideas that seem relevant to the teaching of programming. The theory of behaviourism, and the work of Thorndike and Skinner, is discussed. The rise of cognitive psychology will be outlined. Key theories advanced by Vygotsky and Bruner are also discussed. Finally the chapter will briefly describe the constructivist and constructionist approaches to learning.

Roots of learning theories

It is useful to define some initial terms. In discussing learning theories we can claim we are discussing both issues of epistemology and ontology. Epistemology is the study of knowledge, the asking of questions about *how* it is we know what it is we know. Ontology is the study of the external world, which forms the basis, in part at least, of our knowledge. Human beings usually agree that there is a coherent world external to ourselves that can be studied and learnt about, although there may be disagreement about the exact nature of that external world. Philosophy has, from the first, posited questions about that external world and the ways in which we know what we know about it: "*Philosophers have traditionally asked about the relation between perception and knowledge.*" (Bird 1972 p 80)

These questions may at first seem remote from the psychologically-based theories about individual learning that are debated currently but for millennia, questions about learning were questions about human knowledge.

Magee points out that:

> "*for well over two thousand years, no distinction was made between what we now call philosophy and what we now call science. People were simply trying to understand the world - and learning in the process about their process of learning.*" (Magee 1997 p 245)

Directly or indirectly, learning theories owe some debt to earlier philosophical approaches, as can be seen with behaviourism.

Behaviourism

Behaviourism can trace its roots back to Aristotle, with a strong empiricist tradition providing a platform for the ideas of behaviourism. Empiricism emphasises the evidence of the senses. For empiricists, the mind is a blank slate and the mind's knowledge can be founded only on direct experience of the world.

Early behaviourist thought can be more properly termed associationism and can be found, for example, in the work of Locke in the seventeenth century and James Mill in the nineteenth. It can be argued that the flowering of associationist beliefs needed the fertile soil of experimental psychology, available for the first time in the latter half of the nineteenth century. Behaviourism can be regarded as the flowering of associationist thought, although within behaviourism there are widely or subtly differing viewpoints.

The work of Thorndike

E L Thorndike (1874 - 1949) is regarded as one of the early founders of behaviourism. His work was deeply influential, as Miller (1991) notes:

> "*Both of Thorndike's laws are excellent descriptive statements.* [] *A large part of scientific psychology in America in the twentieth century has been based on, or directed at, these two laws.*" (Miller 1991 p 227)

The question of how human beings perceive and interact with the world was one that had much occupied philosophy. Different philosophers had arrived at various broad explanations of why human beings behaved in certain ways or how they perceived the external world and reacted to painful stimuli, for example. Thorndike's work concentrated on how sense impressions of the external world can become linked to certain behaviours. Generally, for behaviourists, sense impressions of the external world generate actions so there is an identifiable, quantifiable connection between impressions and response. There is no need to invoke the concept of mind or features of the mind such as personality.

Trial and error learning, argued Thorndike, showed that learning (at least as done by animals) was not a product of thought but of simple actions that were either successful or unsuccessful, the successful actions gradually being favoured over the unsuccessful, producing a learning curve. From his work, Thorndike observed that animal responses to a situation are strengthened by the quality of response to the stimulus. An unpleasant response means that the animal is less likely to attempt the action next time. Thorndike saw a direct correlation between the nature of the response and the likelihood of the animal attempting the action again.

Thorndike formulated the law of effect, arguing that if all other things are equal, the response that gains the animal the food (or some other satisfaction) will be favoured or more closely connected with the situation. This leads to the idea of rewards and non-rewards for the strengthening or weakening of behaviours.

For Thorndike, human learning and animal learning shared some characteristics: complex learning behaviours in humans could be separated into simpler learning mechanisms:

> "*Although always aware of the greater subtlety and range of human learning, he showed a strong preference for understanding more complex learning in terms of simpler learning principles, and for identifying the simpler forms of human learning with that of animals.*" (Bower and Hilgard 1981 p 25)

Thorndike's work is called connectionism because of the importance of the idea of connections, strengthened or weakened in some way.

Thorndike also examined the concept of motivation:

> "*Other things being equal, connections grow stronger if they issue in satisfying states of affairs. Learning, whether by use or effect is facilitated by the identifiability of the situation and by availability of the response.*" (Thorndike 1931 p 101)

Motivation then could be seen not as mysterious, internal given but as a biological drive. Depriving animals of food increased their drive to eat, a basic biological need. Their hunger made food a powerful reinforcer. However, this concept of biological drive cannot explain all animal learning or human learning but nonetheless, Thorndike rejected fiercely any explanation of learning that required the invocation of mind or personality.

After 1930, Thorndike revised his central ideas, in the light of extensive experiments based on the laws of exercise and effect. Notably, experiments showed that the law of effect had a specific corollary: under some circumstances, reward was a more powerful reinforcer than punishment. Other revisions or developments to connectionism included the spread of effect, belongingness (where a connection between two elements is made more easily if they are perceived as belonging together), associative polarity, stimulus availability and response availability. (Bower and Hilgard 1981 pp 29 - 37)

In summary, it is important to note that Thorndike was deeply concerned with educational issues and sought to improve pedagogical theory and practice. Much of his work was designed to generate empirically founded principles through which education might improve but Thorndike's emphasis on the law of effect, in its several, increasingly fine-tuned versions, meant that there was little room for the role of insight and understanding in education.

The work of Pavlov

Pavlov did not carry out his work on conditioned reflexes until he was in his fifties. The idea of conditioned reflexes did not originate with Pavlov but he was the first to study the phenomena in depth and to quantify relationships precisely.

The central concept is that of the unconditioned stimulus (US) such as food, which provokes the unconditioned reflex (UR) of salivation. The food is that combined with another stimulus such as light or a sound. Eventually the light or sound will be the conditioned stimulus (CS) provoking salivation as a conditioned reflex or response (CR).

Pavlov also identified the phenomena of reinforcement, extinction and spontaneous recovery. Extinction is the gradual falling away of the conditioned response where the conditioned stimulus is presented without reinforcement. This extinction is reversed after a period of time by the reappearance of the conditioned response (spontaneous recovery).

Pavlov went on to look at the production of neuroses in a dog by presenting it with increasingly difficult discrimination tasks. The creature's performances became erratic and the animal showed signs of distress. Again, animal experimentation results were taken to be a pointer to similar states or activities in humans. Pavlov believed that human beings developed various psychiatric problems in broadly similar ways, although he invoked neurological elements (cortical cell excitability or inertia) to support his ideas.

Pavlov also explored second-order conditioning, where the conditioned stimulus is paired with another stimulus (e.g. a light and a sound) and the conditioned response is shown, after a number of reinforcements, to the new conditioned stimulus.

Pavlov's contribution to learning theory is immense. His work has passed into popular culture as well as informing much of the work on learning done in the twentieth century. We need to ask whether the work of behaviourists such as Pavlov have anything to offer education in the twenty-first century? More specifically is there anything of use to teachers of programming? We shall return to this point later.

The work of Hull, Mowrer and Tolman

Hull (1884 - 1952) was a behaviourist much impressed by Pavlov 's work. His explanation of learning was based on conditioning but emphasised intervening variables. Intervening variables are factors such as previous behaviour or the current physiological state of the subject (hungry, thirsty and so on) Hull did extensive work on quantifying intervening variable and their effects on behaviour, presenting his theory most fully in his 1943 work *Principles of Behaviour*:

> "*Hull's system had many points of superiority over other contemporary psychological systems. It was at once comprehensive and detailed, theoretical yet empirically quantitative.*" (Bower and Hilgard 1981 p 130)

Mowrer did extensive work on conditioned anxiety: what happens when a conditioned response is associated with punishment? Mowrer identified six types of reinforcers that could follow a response by the subject, in two types: negative and positive. Positive reinforcers include hope and relief; negative reinforcers include fear and disappointment.

Edward Tolman studied learning from a behaviourist viewpoint but his acknowledgement of the cognitive aspects of behaviour set him apart from what might be termed 'pure' behaviourists. His opposition to the strongly expressed tenets of behaviourism took the form of analysing behaviour with an awareness of ways in which concepts like volition, knowledge and planning might be part of that behaviour. Tolman's work forced the advocates of strong behaviourism (particularly Hull) to re-state or develop their theories to reflect the impact of such mentalist constructs as purpose or inference. Tolman believed "*that behavior should be analyzed at the level of actions, not movements; that behavior was goal directed, or purposive...*" (Bower and Hilgard 1981 p 326)

For detailed discussion of the work of Hull and Mowrer, see Bower and Hilgard 1981 pp 74 - 113. For discussion of schools of thought such as structuralism, functionalism and associationism see Sternberg 1996 pp 7 - 10 and Richardson 1988 pp 41 - 63.

The work of B F Skinner

B F Skinner is a central figure in the field of animal conditioning, a behaviourist who rejects the idea of a mind as the mediator of behaviours. There is no need to invoke the nebulous or untestable concept of the learner's mind, in order to understand behaviour!

Skinner used animal subjects for his work but crisply rejected the notion that his work was relevant only to animal behaviour:

> "*It is often pointed out that I have specialized in the behavior of rats and pigeons, and it is usually implied that as a result my judgment about people has been warped, but at least sixty per cent of what I have published has been about human behavior.*" (Skinner 1978 p 16)

Skinner departed from what could be termed classical conditioning by identifying a distinction between respondent and operant behaviour. Respondents are behaviours that occur in response to known stimuli such as pupils reacting to light. Operants are emitted responses where the stimulus is not present or not recognised. A discriminated operant is where the behaviour is associated with a stimulus but is not a product of that stimulus in the same way as a respondent.

Skinner's work on operant conditioning was an attempt to produce laws for a science of behaviour that could be applied as well to humans as to pigeons:

> "*We must expect to discover that what a man does is the result of specifiable conditions and that once these conditions have been discovered, we can anticipate and to some extent determine his actions.*" (Skinner 1953 p 6)

Skinner (1953) agreed that humans resist, strenuously, any school of thought that removes them from what they perceive as pre-eminence or advantage. He cites the cases of the Copernican theory of the solar system and Darwin's work: "*We have not wholly abandoned the traditional philosophy of human nature; at the same time we are far from adopting a scientific point of view without reservation.*" (Skinner 1953 p 9)

Skinner (1953) examines possible objections to a science of behaviour. It is said that we can never wholly comprehend ourselves because it is a reflexive act from which we cannot disengage entirely. Skinner refutes this: "*We have no reason to suppose that the human intellect is incapable of formulating the basic principles of human behaviour.*" (Skinner 1953 p 18)

Skinner (1953) also deals with the objection raised by the undeniable existence of human uniqueness. This argument, says Skinner, will be weakened as the laws of behaviour are developed, first as unique individual illnesses fell under the remit of general advances in medicine.

Skinner (1953) points out that control of human behaviour is extensive in society and a science of behaviour can, as it develops, inform such controls and increase their effectiveness: "*Any condition or event which can be shown to have an effect on behaviour must be taken into account. By discovering and analysing these causes we can predict behaviour....*" (Skinner 1953 p 23)

Skinner places his emphasis on the understanding of variables. Once we know the variables that produce certain behaviour, we can ensure that behaviour is exhibited by creating those variables. The example Skinner gives, early in his work, is the drinking of a glass of water. Variables such as giving the subject salty food will increase the certainty of the subject drinking the water. The external variables that give rise to behaviour enable the behaviourist to undertake about Skinner calls caused or functional analysis. The independent variables (or the cause of behaviour) related to the dependent variable (the behaviour exhibited). The synthesis of these relations gives the behaviourist an overall view of the subject as a behaving system.

The central concept in Skinner's operant conditioning is reinforcement. Reinforcement is applied in a shaping procedure. If the subject is a rat, then the rat receives a pellet of food firstly when it approaches that lever, then when it touches the lever and finally receives the food only when it presses the lever this procedure may take only a short time, less than an hour.

Skinner explored the concept of schedules of reinforcement. He discovered that, if the required behaviour was reinforced only intermittently, the required action was performed more frequently. Skinner identifies the process of operant extinction. There is a measurable orderliness in the rate at which the operant behaviour declines. Skinner describes two kinds of reinforcers: negative and positive. Negative reinforcement consists of removing something from a situation. Behaviourist experimenters may also present a negative reinforcer or remove a positive reinforcer.

For humans, the reinforcing element is not necessarily obvious to the person who is reinforced. It is not enough to enquire of human subjects what reinforces them. Humans also can tolerate a long gap between behaviour and a primary reinforcer. Skinner identifies the presence of conditioned reinforcers as a possible explanation. Conditioned reinforcers are events that are linked to a reinforcer, such as turning on a light (CR) when giving food (R) to a hungry pigeon. The use of delay or intervening acts between an act and the primary reinforcement is a method of control that Skinner identifies as widely used in institutional situations. Skinner identifies educational successes as tokens that may be exchanged for a primary reinforcement such as esteem or status.

People, argues Skinner, behave in certain ways, not because of the consequences that will follow their actions, but because of the consequences which have occurred in the past, dependent on that behaviour. "*This is, of course, the Law of Effect or operant conditioning.*" (Skinner 1953 p 87)

For more on classical and operant conditioning, see Kagan and Havemann (1980) pp 89 - 123, Atkinson, Atkinson and Hilgard (1983) pp 193 - 219.

In Skinner's view people are not to be praised or blamed for their actions: the actions, and the people, are products of conditioning. Skinner's work gave rise to two main applications. It was used to create systems of negative and positive reinforcement in situations where control was required: prisons, schools and other institutions:

> "*Skinner in 1954 announced and embarked on a series of investigations and inventions designed to increase the efficiency and of teaching arithmetic, reading, spelling, and other school subjects, by using a mechanical device expected to do some things better than the usual teacher can do them, while saving the teacher for tasks that the teacher can do better.*" (Bower and Hilgard 1981 p 191)

Skinner went on to identify, in a 1958 paper, the principles of programmed instruction, principles that formed the basis for rapidly expanding educational effort. Skinner's principles can be briefly summarised as:

- reach a detailed and clear understanding of the knowledge to be learned
- formulate a list of questions and responses
- engages the learner so that he/she is active
- provide feedback
- arrange the question so the right answers are more likely to occur
- let the learner proceed at his/her own pace
- provide reinforcement for the learner's work

(Bower and Hilgard 1981 pp 191 - 192)

In the 1960s, programmed instruction became one of the areas identified with the emerging power of computing and the scene was set for the alliance of behaviourism and computing, an alliance that continues, in various forms, to this day. For more on programmed instruction, see Atkinson, Atkinson and Hilgard 1983 p 217. See also Skinner (1962).

For Skinner, behaviourism and the careful application of its tools, techniques and approaches are appropriate to human learning and behaviour and even required, given some of society's problems. However, this strongly behaviourist view has not gone unchallenged, notably the in the field of verbal behaviour, an activity generally considered uniquely human. As Hayes (1994) says "*By the late 1950s and early 1960s, academic psychologists were increasingly coming to view the behaviourist approach useful but limited.*" (Hayes 1994 p 8) In the 1950s and 1960s, cognitive psychology was also on the rise. This approach to understanding people's behaviours can be neatly (if somewhat superficially) contrasted with behaviourism.

Cognitive psychology

In the early decades of the twentieth century, while behaviourism began its rise, the field of cognitive psychology was also being marked out. As Sewell (1990) notes:

> "*...three sets of influences can be seen as relevant to the present context of educational computing, namely educational philosophies and practice, cognitive and developmental psychology, and computer sciences.*" (Sewell 1990 p 3)

Cognitive psychology can be seen as a reaction to the strong behaviourism that held sway in the first half of the twentieth century:

> "*Cognitivism was seen as breaking behaviourism's hold on experimental Psychology, supplanting it as the most productive theoretical orientation.* [] *More genuinely revolutionary was the adoption of a new set of theoretical concepts, a new technical language. Cognitive psychologists were thinking about thinking in quite a novel way.*" (Richards 1996 p 67)

Cognitive psychology "*deals with how people perceive, learn, remember and think about information.*" (Sternberg 1996 p 2) Cognitive psychology is, as Stillings et al. (1995) point out, a "*remarkably diverse*" field of research (Stillings et al. 1995 p 1) and includes studies of language acquisition and use, concept formation, mental imaging, deductive reasoning and problem solving. (Thagard 1996 p 8)

Gross (1985) notes that, in the middle of the twentieth century, psychologists began to study the information-processing aspects of human behaviour. This shift, along with developments in linguistic and child development theory gave rise to what Gross identifies as the American version of cognitive psychology, "*an amalgamation of ideas stemming from experimental psychology, verbal learning and psycholinguistic research.*" (Gross, 1985 p 9)

Part of the cognitive psychology stream is the information-processing view of cognition. See also Meadows (1994) pp 212 - 235, Stillings et al. (1995) pp 1 - 11 and Roth (Editor) (1990) pp 586 - 587.

Other theories about learning sprang from work done with children, notably the work of Piaget. For researchers such as Piaget, research on the child learning is also the study of general rules of mental development. For summaries of Piaget's work see Meadows (1995) pp 33 - 40, Shaffer (1985) pp 57 - 62, Munsinger (1975) pp 16 - 17 and 99 - 117 and Meadows (1994) pp 198 - 212.

Researchers such as Vygotsky studied the ways in which children learn and produced general theories of learning. The work of Vygotsky is discussed in the next section.

The work of Vygotsky

Vygotsky's research is firmly situated in the political, educational and ideological environment in which Vygotsky worked. As Richardson notes "*First and foremost, Vygotsky traces his ideas to Karl Marx.*" (Richardson 1988 p 73)

Sacks (1989) says that Vygotsky's work was so ahead of its time in the 1930s that a contemporary described him as a visitor from the future. The nature and focus of Vygotsky's research has provided the basis for extensive work on language, learning and child development and influenced figure such as Jerome Bruner, whose work is discussed later in this chapter. (Sacks 1989 p 50)

For Vygotsky, there are three ways to view the relationship between development (particularly child development) and learning. Vygotsky describes as Piagetian the view that development precedes learning:

> "*Because this approach is based on the premise that learning trails behind development, that development always outruns learning, it precludes the notion that learning may play a role in the course of development or maturation of those functions activated in the course of learning. Development or maturation is viewed as a precondition of learning but never the result of it.*" (Vygotsky 1978 edition p 80)

The second view, says Vygotsky, sees learning as development. Learning drives development and without learning there is no development.

The third theoretical view of these two aspects is that they must be combined in some way. Vygotsky sees the analysis of these three viewpoints as generating two relevant questions: "*...first, the general relation between learning and development; and* *second, the specific features of this relationship when children reach school age.*" (Vygotsky 1978 edition p 84)

Vygotsky formulated the important and influential concept of ZPD (Zone of Proximal Development). This is particularly relevant for formal schooling. Vygotsky says that a child has at least two developmental levels (actual and potential) that must be examined to determine the relationship between the developmental process and learning capabilities. "*The first level can be called the actual developmental level...*" (Vygotsky 1978 edition p 85) The ZPD is the:

> "*...distance between the actual developmental level as determined by independent problem solving and the level of potential development as determined through problem solving under adult guidance or in collaboration with more capable peers.*" (Vygotsky 1978 edition p 86)

For Vygotsky, this concept was central to a full understanding of a child's developmental stage. It is not sufficient to examine only what the child can do unaided: the researcher or teacher must look at what the learner could do with some competent support.

This theory, for Vygotsky, could be used to define 'good learning' which is learning that is in advance of development. Learning also creates the ZPD (for example, interacting with a social group begins the long process of developing an internal sense of morality, a conscience, through the adapting of external rules of fairness and good behaviour). For Vygotsky:

> "*...properly organized learning results in mental development and sets in motion a variety of developmental processes that would be impossible apart from learning. Thus learning is a necessary and universal aspect of the process of developing culturally organized, specifically human, psychological functions.*" (Vygotsky 1978 edition p 90)

The concept of ZPD has proved central to thinking about learning and pedagogical practice. It informs a number of views about teaching and learning, notably constructivism, discussed later.

Another key aspect of Vygotsky's work is its emphasis on the social aspects of learning development. As Meadows (1994) notes, Vygotsky argues that any feature of higher mental functioning appears first in a social context and is then internalised, moving from the interpsychological to the intrapsychological: "*This emphasis on the primacy of the social world in cognitive development is very different from the emphases of the Piagetian and information-processing approaches.*" (Meadows 1994 p 237)

This belief that development and learning can be seen as a product of effective social interaction and partnerships means that the question of what is innate (in terms of cognitive structures) is no longer an issue. Vygotsky's view is that children do not develop sophisticated concepts and understandings by themselves but:

> "*... rather the child is helped by the adult in the 'guided reinvention' of accumulation of knowledge and ways of thinking which preceding generations have constructed. The skills required of the child are of observation and imitation, and of generalisation and decontextualisation, but even these fundamental skills develop under the fostering support of social interaction.*" (Meadows 1994 p 239)

For Vygotsky, internal mental processes are a product of socialisation and cannot be divorced from it. The most private thoughts still have, in some essence, a social nature. This internalisation of the external, social world means that the cultural tools for cognition are also absorbed and become part of the child's mental repertoire.

Vygotsky work continues to be relevant: the concept of ZPD is one that can be modelled perhaps quite productively, in the organisation of materials for the teaching of programming. The work of Bruner is also of interest and is discussed in the next section.

The work of Bruner

Like Piaget, Jerome Bruner, in his initial work in the early 1960s, proposed clear stages in the development of thought but unlike Piaget, Bruner made language a central feature of those stages: a Vygotskyian view.

Bruner described three modes of representation: enactive, iconic and symbolic. Enactive focuses on actions, like Piaget's sensorimotor period. Iconic is the child's representation of the world around him or her in fixed images (like the preoperational stage) and the symbolic stage is that of using language and symbols (as with the operational stage).

The progression of stages is from enactive to symbolic but a key element of this progression is the use of what Bruner terms 'cultural amplifiers'. Cultural amplifiers are elements of society and civilisation that extend human abilities in some way. Such amplifiers might be of our physical capacities (machinery and tools) or of our thinking capacity (stories, theories and languages).

Richardson (1988) notes that by the 1970s Bruner had adapted his original ideas and was no longer "*a stage theorist.*" (Richardson 1988 p 72) In *Child's Talk* (1985)

Bruner instead talks of cognitive endowments that support the child's language development:

> "*These four cognitive "endowments" - means-end readiness, transactionality, systematicity, and abstractness - provide foundation processes that aid the child's language acquisition.*" (Bruner 1985 p 30)

Bruner than goes on to take an explicitly Vygotskyian view of how children continue to develop language. The LAD (Language Acquisition Device) must, in Bruner's view be accompanied by a LASS - a Language Acquisition Support System, which involves adult and child in subtle and continuing interaction, a negotiated interplay of conversation and responses.

Like Vygotsky, Bruner believed that the expert adult could with thoughtful intervention foster learning. Bruner and his colleagues called such intervention 'scaffolding'. This concept has strong affiliations with Vygotsky's concept of the ZPD.

Malim and Birch (1998) sum up the features of scaffolding: recruitment, reduction of degrees of freedom, direction maintenance, marking critical features and demonstration. (Malim and Birch 1998 p 472) Recruitment is the gaining of the child's interest. Reduction of degrees of freedom refers to the breaking down of tasks into smaller steps, "*reducing the number of constituent acts required to reach solution.*" (Wood, Bruner and Ross 1976 p 98)

The adult is also responsible for direction maintenance, the ensuring of motivation in the child. Marking critical features means that the child needs to know what elements of his or her efforts are most appropriate to the task in hand. Demonstration is the adult showing the child a correct or improved solution, giving them a model of success, to improve their subsequent efforts. (Wood, Bruner and Ross 1976)

Work done has shown, say Malim and Birch (1998), that "Peer tutoring and cooperative group work have been found to be appropriate means of providing scaffolding behaviour within a classroom." (Malim and Birch 1998 p 485)

The work of Bruner on scaffolding springs out of his work on child development but has implications for many teaching or learning situations, in a more general from. Like ZPD, we can use the concept in teaching adults as well as children. *How* we use it is a more complex and demanding question.

In learning theories, we can see a move from the individual's behaviour (behaviourism in various guises) to the whole individual (cognitive psychology) to the individual within a social situation. One theory or view that places its main emphasis on the social nature of learning is constructivism. This is examined in the next section.

Constructivism

Constructivism can be defined, broadly, as the alternative to a positivist view of an external reality, a reality that can be objectively agreed upon and examined. In such a worldview, there are appropriate truths, which are absolute and can be absolutely agreed upon and transmitted from teacher to learner.

In contrast, constructivists argue that "*Concepts cannot simply be transferred from teachers to students - they have to be conceived.*" (Steffe and Gale (Editors) 1994 p 5) Constructivists also reject the behaviourist approach:

> "*... learning is not a stimulus-response phenomenon. It requires self-regulation and the building of conceptual structures through reflection and abstraction.*" (Steffe and Gale (Editors) 1994 p 14)

Constructivism's hypotheses can be summarised as: (i) Learning (the application of relevant schemata) is a flexible process that is able to take account of the random variety of the real world. (ii) the development of schemata is more than the simple listing of attributes or the construction of a direct model of the experience: it is a distilling of the experience into abstraction. (iii) Behaviour takes account of the subtleties of the developed schemata, rather than being no more than a response to stimuli that is as predictable as the stimuli itself.

It would be inaccurate to identify the constructivist approach solely as a late-twentieth century reaction to the work of the behaviourists in earlier decades.

Dewey (1933) states explicitly a constructivist approach:

> "*The more a teacher is aware of the past experiences of students, of their hopes, desires, chief interests, the better will he understand the forces at work that need to be directed and utilised for the formation of reflective habits.*" (Dewey 1933 p 36)

Dewey (1933) also sees the role of the teacher as a constructivist one:

> "*his province is rather to provide the materials and conditions by which organic curiosity will be directed into investigations which have an own and that produce results in the way of increase of knowledge...*" (Dewey 1933 p 40)

The work of Bruner and Vygotysky, spanning the 1930s to the present day, discussed earlier in this chapter, can also be seen as constructivist in spirit, with the emphasis being on individual development within a social environment: learning fostered by, and through, interaction with others.

One key element of constructivism is the initial exploration of what the learner understands or believes, before some kind of learning is attempted. The idea of 'starting from where the learner is' is not new. Practical exploration of the concepts/understandings already held by a person (before new constructs are required of him/her) has been a feature of much educational research:

> "*As early as the beginning of this century, Stanley Hall started a program in the United States to investigate children's ideas of natural phenomenon such as heat, frost and fire.*" (Duit 1994 p 272)

Specific examples discussed by Duit (1994) include studies of students' conceptions of concepts in physics such as light. If students are asked how can you see the objects around you? What part does light play in seeing? the range of replies will be interesting and inventive. Duit looks at samples of responses to these questions and analyses them. Duit describes the alternative frameworks revealed by such questions. These frameworks are often entirely unscientific but have a rightness about them (for the person holding framework) that makes them strong structures, not easily dismantled by the provision of the scientific (or 'correct') explanation.

Constructivism, says Duit (1994), acknowledges the power of such pre-existing frameworks and works with those structures rather than seeking only to destroy them. Constructivists argue that educators can adopt an approach that allows the scientific frameworks to be built on and around the less formal frameworks that serve adequately in everyday life.

Duit (1994) goes on to discuss the idea of the learner's resistance to constructivist methods. Learners, in their anxiety to know the right answer and move on (either away from the topic altogether or onto something more interesting/less threatening) may not be happy to explore their current concepts as a basis for further understanding.

What Duit (1994) identifies is a requirement for a repeated exploration and echoing of the students' ideas as they form and are shaped.

For Duit, part of the constructivist approach to learning is a questioning of the process of learning development: "*...another important aspect* [is] *the significance of teachers' and students' conceptions of the learning process.*" (Duit 1994 p 283)

Wood (1994) argues, "*As an emerging epistemology, constructivism not only differs a different view of cognition in classrooms, but also has strong implications for pedagogy.*" (Wood 1994 p 333) The problem with this approach, laudable though it is, is that the information transfer process (teacher to learner) is not done away with. The constructivism is apparent not real, as there is no time or resources for the learner to construct individual understandings from the environment.

For Wood (1994), a genuine constructivism can be identified only when the students are provided "*with an opportunity to individually construct their own ... meanings and to rely on their own ways of thinking to make sense of the world.*" (Wood 1994 p 339)

For Strommen (1992):

> "*Constructivism emphasizes the careful study of the processes by which children create and develop their ideas. Its educational applications lie in creating curricula that match (but also challenge) children's understanding, fostering further growth and development of the mind.*"
> (http://www.ilt.columbia.edu/K12/livetext/docs/construct.html 30/07/99)

Constructivism as a foundation for educational practice is firmly established, particularly in America (See particularly http://www.inform.umd.edu:8080/UMS+State/UMD-Projects/MCTP/WWW/Essays.html 30/07/99)

Much effort has been concentrated on constructivist practice in teaching those below the age of 16 but Jonassen et al. (1993) argue that learning environments based on constructivist principles are not suited to all learning situations and are most appropriate for a university setting. Perhaps in a university setting, the opportunities to establish genuinely constructivist ways of learning are both more available and manageable, for a variety of practical reasons. Such reasons certainly include the ages of the learners, class sizes and the structures available within the university environment (lab sessions, tutorial help and so on).

Constructionism

There are variations on the theme of constructivist learning. Constructionism can be seen as development of the constructivist view. Shaw (1996) states:

> "*Constructionist thinking adds to the constructivist viewpoint. Where constructivism casts the subject as an active builder and argues against passive models of learning and development, constructionism places a critical emphasis on particular constructions of the subject which are external and shared.*"
> (http://el.www.media.mit.edu/people/acs/chapter1.html 30/07/99)

Gergen (1994) identifies as exogenic the teacher to learner information transfer approach and contrasts this with the endogenic approach, where the emphasis is on the rational capacities of the individual. With this, Gergen (1994) identifies three key problems. Firstly, for a learner to learn from the external world, there must be some framework for understanding what is happening and why, even at an elementary level. "*To draw conceptual conclusions from observation requires a set of categories to be in place.*" (Gergen 1994 p 21)

Secondly, as learners or teachers we can never experience what another person experiences. We have, at best, an experience of their experience. We cannot, fundamentally, know someone else's mind. Finally, Gergen describes the problem of meaning, which, it could be argued, does not reside externally in the text but is constructed (or de-constructed) in an number of ways by the reader.

For Gergen (1994) these three problems lead to a social <u>constructionist</u> orientation to knowledge. He elaborates his arguments around the concepts of meaning and language. For Gergen, language has a vital <u>communal</u> function. Meaning in language is context-dependent. These elements dictate that language is given its validity by a process of social approval or agreement.

For Gergen (1994), there are no pedagogical imperatives that arise as givens from constructionist theories. Constructionism has implications for beliefs about knowledge but does not dictate educational practice (in sharp contrast to the educational pronouncements of, say, behaviourism). Knowledge, for Gergen (1994), seems to reside in the unfolding, through time, of a dialogue: "... *What we count as knowledge are temporary locations in dialogic space.*" (Gergen 1994 p 30) For constructionism the key concept is that of dialogue.

For constructionists, the dialogue about what is truly out there in the external world for both teacher and learner is meaningful and a relevant educational activity:

> "*Constructionist approaches to inquiry... have centred on the idea of worlds being constructed, or even autonomously invented, by inquirers who are simultaneously participants in those same worlds.*" (Steier 1994 p 70)

In a sense, the philosophical wheel has turned full circle, with the certainties about the external world and the humans in it being replaced by a belief that what is 'out there' depends on who you are.

If we identify two views, constructionism and constructivism, can we identify similarities? Both share scepticism about the ability of behavioural sciences to tell humans how humans learn and why they behave as they do and a questioning attitude to the empiricist view of what legitimately constitutes knowledge. Constructionism has a more radical approach to that which is studied: the very stuff of education, the external world, is no more than the basis of dialogue.

The contrast between constructivism and constructionism lies in the ways in which the ideas associated with each might be used, within an educational context. Constructivism has something to offer in terms of creating or designing educational experiences. Constructionism deliberately moves away from offering guidelines and focuses on the unfolding of dialogue, which is the learning itself.

Chapter 5: Conclusion

We can see that asking questions about what it is to think, to interact with the world and what that world that we seem to perceive truly is, are questions first of philosophy and then of psychology.

Psychology could be said to date back to the nineteenth century, with behaviourism being a central aspect of the intersection between psychology and questions of learning. In the twentieth century, behaviourism became one school of thought among many, some of which owed a debt to it, others which sought to move away entirely from its theories.

The work of Piaget, Vygotsky and Bruner saw the learner in very different terms to that of behaviourism. For Piaget, the learner's mind was present, in its own unfolding complexity. For Vygotsky, the learner's mind was present, with its complexity fostered by the social environment around that learner. For Bruner, the learner's mind is present and its cognition is amplified by the various means made available to that learner by his/her culture. These are, of course, simplifications of an immense body of work.

What we can see from this chapter is that there is no definitive, all-encompassing theory of learning. There is no single theory that will meaningfully and productively underpin all types of educational software and all possible uses of that software. There are potentially fruitful approaches to learning that may have resonance for some models of educational software, such as constructivism for more exploratory CAL such as simulations or models. What this chapter has identified is some of range of learning theories available (and the chapter does not pretend to be exhaustive) to those who teach and use software in their teaching.

The next chapter discusses two main issues: it examines the role and nature of CAL in general and looks at the two earlier uses of software to support the teaching of programming, undertaken by the author, including the use of tutorial CAL, written by the author.

Introduction to Chapter 6

This chapter briefly examines the various forms of CAL (Computer Aided Learning) software, from the earliest drill and practice approaches to artificial intelligence and intelligent tutoring systems. This leads onto the work done by the author in writing tutorial CAL and using it with students.

After using tutorial CAL, the author then moved away from the traditional style of page turning CAL and developed a small on-line help program. The chapter presents an overview of these two main approaches and describes some of the reactions of participants to these different styles of CAL. At the end of the chapter, the author draws some general conclusions about these two research efforts.

A note on terminology

There are many acronyms available for those wishing to describe the use of software to support teaching and learning activities. Different acronyms have come into fashion, only to be succeeded by a newly popular version. CAL is generally held to stand for Computer Aided Learning or Computer Assisted Learning. Other versions include CAI (Computer Aided Instruction), CMI (Computer Managed Instruction) and CBT (Computer Based Training). In some cases the acronym may more accurately denote the type of software and its pedagogic purpose. The author has chosen to adopt the acronym CAL and will use it throughout this and other chapters to denote the general educational software.

Classifying types of CAL

During its life in the second half of the twentieth century, CAL has been put to many educational uses, over a wide range of subject areas for a diverse range of student and pupil audiences. There is not room here to attempt even a partial review of the rich collection of CAL that has been produced by Government-funded projects and educational initiatives, as well as by individuals, schools, colleges and universities, in America and the UK. For a discussion of some of the key projects in the US, see

Appendix F. For a discussion of the work of the NCET (National Council for Educational Technology), and other Government initiatives in the UK, such as the NCET and TLTP, see Appendix G.

Barker (1989) says:

> "*the phrase 'Computer Assisted Learning' (CAL) has come to be regarded as something of an umbrella term that is used to describe, collectively, the many different teaching, training and learning applications of computers.*" (Barker 1989 p 7)

Watson (1987) says:

> "*Any attempt to define Computer Assisted Learning (CAL) is fraught with interesting problems. A perusal of the literature in the seventies finds much confusion between Computer Aided Instruction (CAI) and Computer Assisted Learning (CAL).* [] *Computer Managed Learning (CML) is clearly a separate and distinct area, that of using the computer to manage the learning sequence.*" (Watson 1987 pp 10-11)

Payne et al. (1980) state "*It is possible to group applications under a series of umbrella headings.*" (Payne et al. 1980 p 1) These umbrella headings identify the roles of a computer in education as: presenting simulations, acting as a "*calculating aid*" (Payne et al. 1980 p 1) and as a medium for "*instruction, drill and demonstration...*" (Payne et al. 1980 p 2) They also acknowledge the computer's role in managing the learning environment and performing administrative tasks.

Bradshaw (1985) says, "*The two variables of teaching style and variety of computer program design make the description and classification of CAL difficult.*" (Bradshaw 1985 p 13) Bradshaw goes on to identify a model of teaching and learning in which CAL is described by its function within a particular aspect of phase of the model, rather than classified as a single entity.

O'Shea and Self (1983) classify CAL programs by approach and distinguishing features. For O'Shea and Self, what they term linear programs represent the behaviourist model of CAL. Branching programs use "*Corrective feedback* [and] *tutorial dialogues...*" (O'Shea and Self 1983 p 68) Generative computer-assisted learning is the drill and practice model, using "*task difficulty measures* [and] *answering student questions...*" (O'Shea and Self 1983 p 68)

O'Shea and Self list simulation, games and problem solving as examples of CAL but make more precise distinctions, listing also mathematical models of learning, the TICCIT and PLATO projects, emancipatory modes, (where the computer is an information source and a "*labour-saving device*" (O'Shea and Self 1983 p 69) and dialogue systems that use "*tutorial strategies* [and] *natural language...*" (O'Shea and Self 1983 p 69)

Rushby (1979) defines the terms CAL and CML very closely:

> "*While CAL concentrates on the student and his learning, CML is concerned with helping the teacher and the student by relieving them of the some of the routine, time-confirming management processes.*" (Rushby 1979 p 11)

In terms of learning, Rushby (1979) describes CAL as the presentation of learning materials on the computer whereas in CML: "*The computer is used to direct the student from one part of the course to another and the learning materials themselves are not kept in the machine.*" (Rushby 1979 p 15)

A further distinction is made in terms of the material: CAL is concerned with the presentation of "*rapidly changing, highly detailed information*" (Rushby 1979 p 17) where CML "*systems operate with less detailed information.*" (Rushby 1979 p 17) For Rushby (1979), CAI implies "*an application in which the computer is used to administer drill and practice examples or programmed instruction.*" (Rushby 1979 p 18)

Rushby also describes CAL as being embedded in a framework of four educational paradigms: instructional, revelatory, emancipatory and conjectural. The instructional paradigm, says Rushby (1979), can trace its antecedents to the work of Skinner and other behaviourists. One example of this paradigm in action is instructional dialogue and Rushby (1979) gives a sample of interactive, instructional dialogue. (Rushby

1979 p 24) Rushby (1979) describes drill and practice as falling within the instructional paradigm.

The revelatory paradigm includes, for Rushby (1979), the use of simulations and software that embodies underlying models:

> "*As the student interacts with the model hidden within the computer he develops a feeling for its behaviour under various circumstances and so is led to discover the rules which govern it.*" (Rushby 1979 p 28)

The emancipatory paradigm allows that some of the work of learning may not be wholly relevant or contribute meaningfully to the overall experience. The computer can then be used to free the student from tasks that do not contribute to the learning process. In the emancipatory paradigm, what Rushby (1979) terms "*serendipity learning*" (Rushby 1979 p 34) is also supported.

The conjectural paradigm is where the computer is used to "*assist the student in his manipulation and testing of ideas and hypotheses…*" (Rushby 1979 p 31) Part of this actually is what Rushby (1979) calls model building, noting that "*There are many similarities between the process of simulation used in revelatory CAL and the modelling found in conjectural CAL.*" (Rushby 1979 p 32)

Kemmis et al. (1977) discuss in detail the educational paradigms noted above and examine the "*shortfalls in achievement of the potential of CAL …in relation to each of the paradigms.*" (Kemmis et al. 1977 p 23) For each of the paradigms, Kemmis et al. (1977) describe the key concept, the role of the computer, the assumptions that can be said to underpin the CAL design and subject and describe an example of specific CAL closest to that paradigm.

Kemmis et al. (1977) describe the emancipatory paradigm as the one not fully articulated. This paradigm concerns itself with the reducing of what the authors term the "*inauthenticity of student labour.*" (Kemmis et al. 1977 p 29) For the full discussion, see Kemmis et al. 1977 pp 23 - 33.

Straker (1989) classified CAL for primary school children by the activities potentially linked to the classroom use of software. She noted:

> "[Programs] *can stimulate a wide variety of tasks: craft activities, scientific experiments, creative writing, mathematical investigations, historical research, drama, music or even PE.*" (Straker 1989 p 12)

Opacic and Roberts (1985) described the results of a survey done on CAL usage in secondary schools. They discovered that 24% of CAL use was in science subjects and 6.4% in non-science subjects. They note that the applications being used were *"of the simplest kind*" (Opacic and Roberts 1985 p 67) such as drill and practice in geography and revision exercises in physics but go on to state:

> "*More optimistically, the small sample did in fact manage to exemplify all the paradigms* []*: instructional - drill and practice; revelatory - simulation; conjectural. - data processing; and emancipatory - computation of fieldwork data.*" (Opacic and Roberts 1985 p 67)

Barker and Yeates (1985) distinguished between CAL functions such as management of learning, testing and dissemination of material, among others, and CAL modes. They identified seven CAL modes: "*...Problem solving... Drill & practice... Inquiry mode... Simulation... Gaming... Tutorial mode... Dialogue mode...*" (Barker and Yeates 1985 p 28) Coburn et al. (1982) note that the general CAL "*diet consists of drill and practice, tutorials and demonstrations.*" (Coburn et al. 1982 p 21)

Barker (1989) identifies "*three important models of CAL* (Barker 1989 p 14) He describes these as informatory, exploratory and instructional. Barker classes the first type as the simplest, where the CAL presents information. The second model draws on the concept of the microworld such as that of the LOGO programming language. Barker notes, "*essentially, the methodology involved in this realization of CAL is based upon the 'microworld' approach advocated by Papert.*" (Barker 1989 p 16) The third approach to CAL is to us the computer to deliver courseware.

Barker distinguishes courseware from programmed instruction: courseware is "*far more sophisticated.*" (Barker 1989 p 16)

In fact what Barker goes on to describe is in fact more like an intelligent tutoring system:

> "*First, the courseware involved must be adaptable...* [] *Second, in order to be adaptable the courseware must attempt to assess how well the student is (a) acquiring the skills, and (b) assimilating the knowledge that constitute the domain of instruction.*" (Barker 1989 p 16)

Barker and Yeates (1985) draw up a list of what they term instructional modes, giving a range of broad CAL types: " *...tutorial packages... drill and practice packages... inquiry based systems... dialogue based packages... simulation packages... gaming packages... testing, monitoring and reporting packages... integrated packages.*" (Barker and Yeates 1985 pp 75 - 76) They note that this classification may be less useful than others because one package might combine a number of the functions listed above.

In summary, we can see that are many ways to classify CAL. We can describe CAL by the subject area it covers. We can describe it in terms of the educational paradigm it embodies (instructional, revelatory, conjectural or emancipatory). We can classify broadly by the task it accomplishes: does it manage the learning environment, present material for drill and practice, create and run models or simulations or carry out basic administrative tasks (keeping track of scores and collating results, for example)?

What the above discussion does not address is the rise of network technology. Depending on how one views hardware and software developments, the Internet and all its rich and variable diversity, is 40, 20 or 10 years old. (http://www.peak.org/~sechrest/talks/internet.howto.html 10/02/2004) Also, the ability of institutions to create, manage and use their own network opens up new possibilities for CAL.

The new possibilities can be seen in the CTI's (Computers in Teaching Initiative) view of CAL. The CTI identifies CAL as:

> "*lecture notes and other resource material posted on the Web to help students prepare for tutorials...information retrieval exercises using online reports, journals and databases, electronic archives, hypertext and hypermedia documents....interactive CAL courseware – developed in-house, bought off the peg,*

> *or tailored to your own requirements ...* []*...reinforcement of important ideas with drill and practice packages or self-assessment exercises ...* [] *...collaborative projects with other institutions using email, videoconferencing, a shared Web site or a metropolitan area network ...* [] *...students acquiring a range of IT skills – in email, word processing, presentations, spreadsheets, databases, and different types of CAL – as part of their coursework ...* [] *...simulation software to model real-world situations or run experiments which would be impractical in the laboratory ...introducing new concepts with a computer microworld or problem-solving environment, which allows students to structure their own learning ...* [] ”
> (http://www.cti.ac.uk/info/handbook.html 19/08/1999)

(For more information on the CTI and its work see Appendix G.) This broad definition of CAL is worth quoting at length as it illustrates how the categories of CAL have expanded. The discrete categories seen in earlier definitions remain, and some of the elements identified previously, such as simulations, still have a place but the range of technologies that can be seen as CAL has grown and become much more sophisticated.

The impact of the Internet means that some educational activities can be supported in ways not previously feasible. Videoconferencing between partner schools in different countries is one example. As Kent and McNergney (1999) note, “*Technologically speaking, there are increasing numbers of ways for teachers to react to students in on-line models of instruction.*” (Kent and McNergney 1999 p 38)

Classifying CAL is not necessarily an easy task: variations in terminology may obscure both differences and similarities. The categories become blurred on close examination but, as stated earlier, the modes proposed by Barker and Yeates (1985) offer a coherent basis for discussion. A description of each category of CAL is offered in this chapter but due to lack of space, only the briefest of outlines can be offered. The next sections review drill and practice CAL, games, simulations and modelling, tutorial CAL, intelligent CAL, microworlds and CAL and collaborative learning.

Drill and practice CAL

Drill and practice software is designed to give students the opportunity to work through examples or problems, to reinforce a concept or concepts taught elsewhere. The earliest drill and practice was in arithmetic. (Suppes et al. 1968) This type of software can be seen as automating the role of the teacher in setting (or presenting) and marking practice problems for students.

Scaife and Wellington (1993) note: "*This mode of learning has a long history associated with words such repetition, rote learning, drill, reinforcement and programmed learning.*" (Scaife and Wellington 1993 p 37)

As Coburn et al. (1982) note "*Drill and practice programs are probably the most common, best known and most disparaged educational application of computers.*" (Coburn et al. 1982 p 21) Underwood and Underwood (1990) say, "*Drill-and-practice programs abound.*" (Underwood and Underwood 1990 p 29)

Nash and Ball (1983) describe this type of CAL as CAL "*designed to modify a student's knowledge or behaviour in a manner completely prescribed by the teacher.* " (Nash and Ball 1983 p 99) CAL dating from the 1960s was mainly of the drill and practice type, although proponents and pioneers of CAL had great ambitions for the further developments. (Coburn et al. 1982 pp 170 - 171) and Suppes et al. (1968):

> "*... we feel that the potentialities and prospects for computer-assisted instruction are great. The probability that this approach will bring about a major revolution in education are high.*" (Suppes et al. 1968 p 7)

Barker and Yeates (1985) stated that "*virtually any subject*" (Barker and Yeates 1985 p 327) could be taught using CAL, although they did add a caveat: "*We do not claim that CAL can be used to teach all aspects of all subject disciplines.*" (Barker and Yeates 1985 p 327)

As Coburn et al. (1982) note, the CAL available in the 1960s: "*could not do as much as had been claimed for it. Mostly, it was used as a way of helping students learn rote skills by providing reasonably flexible drill and practice.*" (Coburn et al. 1982 p 171)

Schuell (1990) notes, "*Much of the early instructional software was criticised for being little more than electronic page turners.*" (Schuell 1990 p 39)

The Stanford Project, while significant, was only part of the boom in CAL during the early 1960s. Other projects included the Yale University's Talking Typewriter designed to teach reading and writing to 3 year olds and the Individually Prescribed Instruction projects, based in a Pittsburgh school. (Coburn et al. 1982 p 170) Inglis et al. (1999) note:

> "*The PLATO Project at the University of Illinois at Urbana was one of the best-known examples of a computer-assisted instruction system. The success of the PLATO Project was largely due to its innovative use of technology.*" (Inglis et al. 1999 p 8)

PLATO was also an example of CAL that crossed CAL categories. It was a system which was designed to: "*support many instructional strategies such as gaming, simulations, testing, drill-and-practice, and self-paced programmed instruction.*" (Hofstetter 1979 p 123) For further discussion of PLATO, see Appendix F.

Drill and practice CAL has its roots in programmed instruction. In the 1960s, programmed instruction became one of the areas identified with the emerging power of computing and the scene was set for the alliance of behaviourism and computing, an alliance that continues, in various forms, to this day. For more on programmed instruction, see Atkinson, Atkinson and Hilgard (1983) p 217. See also Skinner (1962).

Any review of material on programmed instruction published during the 1960s shows the huge investment being made, both financially and educationally by governments (notably the American government) and educational establishments. The drill and practice CAL had some success but the uptake of CAL lagged behind the enthusiastic rhetoric of CAL researchers and early adopters, for a number of reasons. For a brief discussion of these reasons, based on a survey by the Educational Testing Service in 1972, see Coburn et al. (1982) pp 170 - 171.

Crook (1994) argues that there are educational justifications for using the drill and practice software within a teaching context. Such software is popular with teachers and can be seen as supporting learners in specific ways: presenting worked examples, allowing students to repeat material, in ways that would not be possible for a teacher who must teach a large class. (Crook 1994 p 14)

In summary, drill and practice software is where CAL began. Such software may have some relevance to the learning process today but it rapidly became apparent that the "*super teaching machine, the teaching computer.*" (Calvin 1969 p 15) had much more to offer than even the scoring of student achievement on a series of problems or tasks.

Simulations, modelling and games

Games and simulations need not be computer-based, although the processing power of the computer undoubtedly lends itself to the creation and running of complex simulations that could not be realistically attempted in the classroom or lecture theatre, given time and resource constraints. Taylor and Walford (1978) identify three aspects of simulation (not necessarily computer-based) that they regard as central to educational effort:

> "*simulations are active and informal: simulations are problem-based; simulations are dynamic, requiring the development of flexibility and responsiveness in participants.*" (Taylor and Walford 1978 p 27)

The authors then go on to discuss the advantages of educational simulation. (Taylor and Walford 1978 pp 29 - 33)

Barker and Yeates (1985) state:

> "*Although direct experience is usually the best form of instruction, reality in education must often be sacrificed to factors of time, cost, safety and equipment available. Simulation programs provide the student with artificial experience of a dynamic real-world environment.*" (Barker and Yeates 1985 p 77)

Scaife and Wellington (1993) distinguish between types of simulations. Some may be a straightforward replay of laboratory experiments. Others may represent industrial processes such as "*the manufacture of sulphuric acid...*" (Scaife and Wellington 1993 p 45) Some simulations may be of processes that are too slow, dangerous or fast to be safely undertaken in the real classroom or lecture theatre. Others may be of "*non-existent entities, e.g. ideal gases, frictionless surfaces, perfectly elastic objects.*" (Scaife and Wellington 1993 p 45) Simulations may also be of theories, such as the wave theory of light.

Scaife and Wellington (1993) go on to discuss the advantages of using simulations in a teaching environment, including cost and time savings, the control of variables in the simulation in ways not possible in a real situation. There are also advantages in the management of the learning experience: it is easier to re-run a program than it is to set up and collect equipment in a busy classroom. (Scaife and Wellington 1993 pp 45 - 46)

Scaife and Wellington also go on to list the dangers of using simulations. "*The main dangers of using simulations lie* [] *in the hidden messages they convey.* " (Scaife and Wellington 1993 p 46) Simulations may give students the idea that all the variables in a physical experiment can be equally easily manipulated and that all variables have an equal weighting in an experiment. Simulations are also, in the last analysis, a model of reality, created by a person or people. Simulations can never be wholly neutral. Assumptions and sources of data underlie all simulations and users may never know (or question) what those assumptions are or where the original data came from.

Simulations are partially idealisations, partly simplifications, of the real worlds and users need to be aware of this

The distinction Tawney (1979 c) makes between models and simulations is as follows:

> "*In a simulation, the model is not under suspicion; the student is learning about it by manipulating it and seeing what happens. In modelling, the model is under suspicion and a simulation is used to test its adequacy.*" (Tawney 1979 (c) p 120)

Scaife and Wellington define modelling as "*the representation of systems and processes by a computer.*" (Scaife and Wellington 1993 p 51) The difference between modelling and simulation is that, in modelling, the user provides much more input. Computer modelling software can be as general purpose as a spreadsheet. A spreadsheet can be used to model all kinds of data and processes, using the dynamic power of 'what if?' Other modelling systems might be specific to a subject area. Modelling systems are one example of the emancipatory paradigm. A good modelling system will free students from the inauthentic labour of calculation and allow them to concentrate on the activity and response of the system they have modelled.

Games could be seen as overlapping the two types of CAL described above. A model or a simulation can be presented as a fun activity, perhaps with a specific goal that makes the first user or team to achieve it the winner.

There is insufficient room here to discuss the phenomenal rise of the games industry from the 1970s and the first computer game, Pong, but the expectations of CAL users in education will almost certainly be influenced in some measure by the sophistication of the interfaces and features available in computer games today.

Tutorial CAL

Barker and Yeates (1985) describe tutorial CAL as

> "*a linear series of factual statements interspersed with pre-determined questions and responses. Regardless of ability, performance or prior knowledge, each student is required to proceed through the same material.*" (Barker and Yeates 1985 p 76)

There is also no attempt to diagnose the errors made by the student and provide tailored feedback or additional material, based on the errors generated by that student in answering questions on the material presented.

This type of essentially simple CAL can be contrasted with the much more ambitious goals of the intelligent CAL, as set by those who believe that a flexible, responsive and individualised CAL is achievable. When teachers or lecturers set out to write

CAL, tutorial CAL is often the design paradigm they consciously or unconsciously work within, perhaps because they are turning paper-based materials already written into CAL. Paper-based materials tend to have the kind of linear organisation that fits neatly into tutorial CAL. At its most basic it becomes a one-page-per-screen mapping.

Much of the work of the NDPCAL (National Development Programme in CAL) and the TLTP (Teaching and Learning Technology Programme) supported the production of tutorial CAL, although not exclusively. In the TLTP Phase 1 Catalogue 1995, out of approximately 117 packages 100 were described as being designed for self-study. The figures are approximate, as some projects do not specify the use of each individual package. For a discussion of the work of the NDPCAL and the TLTP in phases 1 and 2 see Appendix G.

Intelligent CAL

In this area of CAL, acronyms once more abound. CAL is frequently described as ICAI (Intelligent Computer Assisted Instruction) but other acronyms are used such as ITS (Intelligent Tutoring Systems). ITS are an attempt to model human knowledge and expertise in such a way as to make it accessible to the learner, someone new to that area or domain of knowledge or new to that particular aspect of the domain. To develop an ITS is to explore thoroughly that knowledge domain and model it robustly. In this aspect, then, it has strong links to the concept of an expert system. Ridgway (1991) notes:

> "*In the domain of ICAI, researchers have offered a rich variety of accounts in these domains and so no simple description can be offered of the approaches taken within this energetic field.*" (Ridgway 1991 p 124)

The history of AI research is a complex and fascinating one. Early work seemed to offer much potential but proved to be less substantial as the projects came to fruition.

> "*The programs (of early AI) lacked the commonsense of a four year old, and no-one knew how to give them the background knowledge for understanding even the simplest stories.*" (Dreyfus 1992 p x)

Dreyfus (1992) argues that an exhaustive formulation of common sense is not possible. Part of our understanding is situated in our physical present in the world and our presence in the society or culture in which we live. Our use of analogy also requires examination. Some AI researches (working in what Dreyfus calls GOFAI, Good old fashioned AI) such as Lenat claim that all analogies bottom out in the physical or primitives such as pain, up, down, cold, seeing and so on.

In ICAI, the impetus is towards producing software that much more closely models the behaviour of a human teacher: asking questions and adapting tutorial strategy and material in the light of the student's responses. This is a more sophisticated view of the computer, where the software is acting as a tutor, modelling the IRE (Initiation, Response, Evaluation) to be found in classrooms. The teacher initiates, through a question or other prompting, the student responds and the teacher evaluates. This model can be represented with varying degrees of sophistication. The software might be designed to tailor questions, within limits, to the particular learner. The feedback or evaluation may be detailed and flexible.

Romiszowski (1987) declared:

> "*It should not be long before commercially developed expert systems will be available in large numbers to assist in all manner of problem solving tasks, including many of interest and relevance in the schoolroom.*" (Romiszowski 1987 http://cleo.murdoch.edu.au/gen/aset/ajet/ajet3/wi87p6.html 12/09/2000)

An expert system is software that models an area of human knowledge. It can be used to train novices in that subject area or to act as additional expertise in areas of complex decision-making. Expert systems are discussed below. We can see that Lesgold's and Romiszowski's enthusiasm lead them to be over-optimistic about what research in ICAI could deliver to the classroom.

What are the critical differences between CAI-CAL and ICAI? In ICAI the computer is acting as tutor or teacher to the student but perhaps the key difference is the focus of interest of the developer. In ICAI it is probably fair to say the interest of the developer(s) is-are creating software that behaves intelligently: in CAL, the developer (usually a teacher) is looking to teach a particular topic or subject more effectively or efficiently. This is a broad generalisation but one supported by Romiszowski (1987). He describes the developers of ICAI as being mainly computer scientists whose goals is to explore the potentials of AI (Romiszowski 1987 http://cleo.murdoch.edu.au/gen/aset/ajet/ajet3/wi87p6.html 12/09/2000)

What is the difference between ICAI and ITS? ITS have their roots in generative CAL and later in adaptive CAL whose "*sophistication lay in... selection algorithms.*" (Sleeman and Brown 1982 p 1) Sleeman and Brown (1982) go on to describe four types of ITS: "*problem-solving monitors... coaches... laboratory instructors... and consultants.*" (Sleeman and Brown 1982 p 2) They do note that most examples of these exist in *experimental* form.

McCalla (1990) stated that: "*Intelligent tutoring systems can be roughly divided into three categories: discovery learning systems, coaching or helping systems and one-on-one tutoring systems.*" (McCalla 1990 p 93)

Discovery systems include microworlds like LOGO. Here we see how CAL definitions and boundaries are fluid and one example of CAL may fit into several categories, depending on who is doing the categorising!

For Anderson (1988) the modelling of student learning is more than the creation of simple pedagogical imperatives:

> "*There are two key places for intelligence in an ITS. One is in the knowledge the system has of its subject domain. The second is in the principles by which it tutors and in the methods by which it applies these principles.*" (Anderson 1988 p 21)

McCalla (1990) concurs: "*One of the central features of an intelligent tutoring system is that it understands its subject domain.*" (McCalla 1990 p 94) The simplest ITS is one where the knowledge modelled is complex but well-defined: an example of this would be an ITS to teach equation solving. Such an ITS need not have no underlying model of human understanding of how to solve equations: it would be a black box approach.

How can the student's understanding be adequately modelled (or modelled at all) within the software? At its broadest, a raw count of incorrect responses can give some indication of areas of weakness. A more sophisticated approach is to create a representation of the student's knowledge held in a data structure.

However, students do not necessarily learn in a neatly incremental fashion and a mathematical modelling of the student's current state of understanding is conceptually flawed. There is a gap between the student's understanding and understanding as modelled within the software. What appears to be more productive (both for the student and as an approach to this problem) is the comparison of a student's response with the pre-decided optional response. The difference between the two forms the basis of tutorial intervention by the software.

Here the modelling is not a representation of the student but an attempt to develop the student's understanding as compared to a 'best', an ideal. This tutorial intervention-suggestion approach may be much closer to the realities of teaching that the idea of representing what an individual student knows or appears to know.

Another element identified by O'Shea and Self (1983) is tutorial strategies. How does the program 'decide' what to do next? This is not a trivial consideration. If there are multiple branches at each decision point (Should the next problem be harder? Easier? Should the next two problems be omitted?) then the number of possible paths through the program will be enormous.

Hardware, despite the development of increasingly powerful desktop computers, still does not seem to provide the appropriate platform for ICAI or ITS or, alternatively, software that does sit comfortably on these platforms is not truly ICAI or ITS.

Microworlds

Yazdani notes:

> *"CAL and ITSs are rather effective for the teaching of narrow domains but are at a loss when dealing with the teaching of general problem solving skills. An alternative*

> *to CAL and ITSs is presented in the form of 'learning environments' which provide a student with powerful computing tools. The student engages in an open-ended learning-by-discovery process by programming the computer to carry out interesting tasks.* [] *Central to this thesis is the notion of a 'powerful idea'. A powerful idea is one which, when learned in one domain, can be generalised to guide one's actions in a variety of different domains.*"
> (http://www.media.uwe.ac.uk/masoud/author/ideas.htm 24/08/)

An example of this powerful idea is the microworld. The initial concept of the microworld belongs to Seymour Papert, a committed advocate of the role of the computer in education. A microworld is a computer-based environment that students can explore, generating their own understanding of this reality, testing it against what happens and developing their understanding. Papert makes strong claims for the impact of microworlds (such as the use of LOGO and the LOGO turtle) upon students' learning and their cognitive skills generally.

Dunn and Morgan (1987) say:

> "*Papert is perhaps the only writer and thinker to have developed a complete theory about the role of the computer in education, a theory that involves a well-developed and supported argument about such fundamental issues as epistemology, learning and thinking.*" (Dunn and Morgan 1987 p 121)

Papert's enthusiasm for the learner as discoverer in the world of LOGO is well known. If the student is to discover and to learn through that discovery, what is the role of the teacher? The identification of the teacher as providing social interaction does not significantly alter the impetus of Papert's belief in constructionism (the learner as explorer and constructor of understanding within microworlds).

Crook (1994) is cautious about the nature of the teacher's role:

> "*However, if teacher intervention within pupils' computer based learning is to be one way forward we need a greater appreciation of the forms that it must take.* [] *I doubt that constructivist environments of computer-based learning can be made more complex simply by appending human support, as if it were just another resource to call upon.*" (Crook 1994 p 62)

The case for teaching programming as way to improve the learner's cognitive skills was taken up with enthusiasm, at least initially, and not just with LOGO. As Solomon (1986) notes:

> *"At all levels level education, including college, BASIC and CAI were greeted warmly. But by the mid-1970s, the educational promise of BASIC and CAI began to fade."* (Solomon 1986 p 6)

One claim that Papert makes for LOGO is its power to equip children with tools for formal thinking: a way of describing concepts that matches the requirements of those concepts. Some concepts for skills may never be amenable to formalism, such as pure motor skills: Papert cites juggling as an example. Yet he goes on to describe two different types of juggling (showers and cascade) and break the monolithic description into simpler actions in considerable detail.

Can we argue therefore that all activities can legitimately be mapped onto structured programming concepts such as step-wise refinement or the use of procedures?

> "*The fact that computational procedures enhance learning does not mean that all repetitive processes can be magically removed from learning or that the time needed to learn juggling can be reduced to almost nothing. It always takes time to trap and eliminate bugs. It always takes time to learn necessary component skills.*" (Papert 1980 p 113)

Papert says that even quite young pupils can be given, via microworlds, access to more advanced concepts and the example he uses is Newton's laws of physics. The language of Newtonian physics is neither simple nor accessible but the world of the Turtle can be both. It is possible, argues Papert, to design microworlds that allow pupils to explore concepts without the intimidating qualities of formal statements, whether those are equations or difficult sentences. There could be, for example, a Newtonian microworld, with velocity and acceleration Turtles that could help pupils to understand the relevant concepts.

In summary, the work of Papert looms large in the CAL arena. His ideas may not have been taken up in the wholesale, transforming fashion he hoped for but LOGO and the concept of the microworld provide a rich strand of debate and development in the history of CAL.

CAL for collaborative and on-line learning

Again, this is a huge area of computer use in education and one that has expanded very rapidly in the 1990s. Just how far we have come can be gauged by noting that in Saettler's work of 1968, *A History of Educational Technology*, there is no index entry for Computer or Computer Aided Learning. Today, any history would have to dedicate many pages just to the rise of computer use in education and many also to the impact of network technologies on teaching and learning activities.

The idea of collaborative learning supported in many different ways by computers has obviously been given a huge boost by the advent of the Internet and associated technologies such as video transmission across the network. The history of the Internet and the ways in which it has changed many areas of human interaction and activities is a fascinating but complex one. It is not possible here to discuss in detail the development of the Internet but Chapter 1 of Inglis et al. (1999) covers the rise of the Internet, discuses hypermedia and other network technologies. (Inglis et al. 1999 pp 1 - 12) See also, for a history of the Internet beginning in the nineteenth century, http://www.internetvalley.com/intval.html 14/09/2000. All this section can do is give some idea of the kinds of tools that are available to those wishing to use network and other computing technologies in teaching and learning.

At the start of the 1990s there was acknowledgement that the technology of networks was going to change education, perhaps dramatically. Harasim (1990) says:

> "*Online education introduces unprecedented options for teaching, learning and knowledge building. Today, access to a microcomputer, modem telephone line and communication program offers learners and teachers the possibility of transactions that transcend the boundaries of time and space.*" (Harasim 1990 p xvii)

Chapter 6 Supporting learning to program using CAL

It could be said that the arrival of the Internet and its associated features such as chat rooms, bulletin boards and e-mail changed education in the way that had always been envisaged for CAL and CAI over the preceding decades:

> "*Advances in technology are making an increasing impact on educational curricula, learning materials and instructional practices. Multimedia allows the exploitation of multimodal representations of knowledge; Interface designs assist usability and interactivity; Local and Wide Area Networking release the Internet as a learning resource with software tools that enable communication through e-mail, bulletin boards and conferencing systems, whiteboards and chat rooms, and videoconferencing.*"

(http://waltoncollege.uark.edu/disted/effective_pedagogies_for_managin.htm 14/09/2000)

The idea of computer conferencing may seem very new but it has a history dating back to the early 1970s. Woolley lists some "*first-generation conferencing systems that emerged in the early to mid-1970's*" such as EMISARI (1971), which is regarded as the first computer system to facilitate conferencing, PLANET, Confer (1975) and EIES (1976). (http://thinkofit.com/drwool/dwconf.htm 14/09/2000)

What distinctions of terminology need to be made in this area? We can define collaborative and on-line learning separately, beginning with collaborative learning.

Kaye (1991) notes:

> "*It is easier to describe what does not count as collaborative learning than it is to produce a universally acceptable definition.* [] *It is important to distinguish collaboration from communication.* []*...collaborative learning* [is] *any learning that takes place as a result of people working together...*" (Kaye 1991 p 2)

On-line learning can be defined as the use of the Internet to deliver Web-based material. On-line learning need not be collaborative, obviously: it may be individual but computer networks offer ways to augment a student's learning by giving the individual learner access to a wide range of facilities, not least of which is the sheer quantity of information available (unmediated and unedited in many cases) on the Internet.

Alternatively, a distinction can be made between on-line learning and local delivery of material. If a tutorial CAL package is available on CD-ROM for a student to take home, is that on-line learning? What if the CAL is available on a university's or school's own internal network (an intranet)? There are now many ways to present all kinds of CAL to students, from zip drives, CD-ROMS and remote access over the Internet. (The boom in home connections to the Internet through commercial service providers is a notable phenomenon in the first years of the new century).

As Hall notes, the Internet has revolutionised the availability of archive materials, giving people access to digital archives:

> "*In some respects the greatest value of the Internet is the incredible amount of primary materials now being made available free and online. While proprietary learning programs and CD-ROMs once dominated teaching technology, the advent of the Web has inspired a popular revolution. Nothing illustrates this better than American Memory, the Library of Congress' National Digital Library project.*"

(http://www.ala.org/aasl/kqweb/28_5_webprofilefulltext.html 14/09/2000)

Computer-based collaborative technologies can be subsumed in the acronym CMC (Computer Mediated Communication). Mason and Kaye (1990) describe the implications of using CMC in education: there will be a breaking down of the distinctions between distance learning and "*place based education...*" (Mason and Kaye 1990 p 23) There will be changes in staff roles across all aspects of educational institutions, not just the academic and finally there will be:

> "*an opportunity, which never existed before, to create a network of scholars, "space" for collective thinking, and access to peers for socializing and serendipitious exchange.*" (Mason and Kaye 1990 p 23)

So, CMC is not CAL but, as Inglis et al. (1999) note, the distinctions between the technologies of the Internet, CMC and interactive multimedia are increasingly blurred: "... *to speak of these as if they represent separate domains is to maintain a distinction that is becoming increasingly false.*" (Inglis et al. 1999 p 11)

Asynchronous computer conferencing is where participants are separated by time: an example of this might be a message board where a group of people post general messages, requests for information or points for discussion. Synchronous conferencing is where participants are interacting at the same time: almost a real-time conversation discussion. Video-conferencing is obviously synchronous and email can be either.

Helm (1997) concludes that videoconferencing has much to offer but that staff and students need to be much more conversant with the use of the technology and what it can most effectively deliver.

This point is echoed by Hall in Teaching with Electronic Technology:

> "*In the best of classrooms, technology (electronic or otherwise) should support the curriculum, not determine it. This seems obvious, but the new technologies are seductive, and it's easy to be caught up in all the talk about "revolutions" without considering meaningful classroom objectives or purposes.*"
> (http://www.ala.org/aasl/kqweb/28_5_webprofilefulltext.html 14/09/2000)

McConnell (2000) notes that computer systems designed to support group work and co-operative learning can be classified according to varying criteria. One approach that McConnell cites is a classification based on the form of interaction between students-tutors and the geographical location of the system's users. These two aspects then give a matrix of four dimensions. (McConnell 2000 pp 28 - 29) McConnell also considers the difference between what he terms structured and unstructured groupware.

Welsh (1999) identifies another use of CMC in what he calls distributed learning, where cause events may be separated in time and space and electronic media "*supplement face-to-face interaction in real world settings.*"(Welsh 1999 p 41)

The question of setting and assessing work using the various technologies available is a complex one. For material on computer assisted assessment see the CTI website at http://www.cti.ac.uk 14-09-2000. As Nelson states:

> "*Student assessment is a huge issue in Internet delivery of classes. College instructors are worried about how to assess student learning in traditional classes, and the problem appears bigger in on-line classes.*"
> (http://wind.cc.whecn.edu/~gnelson/paper/TCC-98.html 14/09/2000)

For an interesting glossary of terms on assessment see http://spicy.atd.depaul.edu/jumpstart/assess/gloss/glossary.html 14/09/2000.

Wilson (1999) identifies three key paradigms for "*Thinking about support for learning and performance...*" (Wilson 1999 p 32) These are: instructional design, performance support and network systems. Network systems, says Wilson, are an emerging technology whose tools include the Internet, Web browsers, search engines, filtering technologies (push-pull technologies), e-mail and multimedia presentation technologies. Wilson concludes that network technologies are the wave of the future. (Wilson 1999 p 35)

It is also true to say that these network technologies available to teachers and lecturers widen dramatically not just the possibilities of access but also the ways in which tasks and activities can be undertaken. As Dennen (2000) notes:

> "*Problem based learning (PBL), collaborative learning, and online learning tools are all popular topics in education today. Each holds the possibility of promoting active, authentic learning situations.*" (Dennen 2000 p 329)

There is also the question of what pedagogical view underpins the employment of CMC and associated tools. Is the role of the network just to provide a new forum for delivering material in what is essentially a traditional way, one that focuses on the teacher as expert and source of knowledge? Or is the process of communication the focus of the group and individual activity? There is a third view, one that sees the use of CMC as a way of focusing on the group and individual creation of knowledge. The third view would throw into sharp relief the role of the tutor. Is the tutor a facilitator of online, collaborative learning or does the tutor's expertise have a place in the knowledge building approach?

Owen (2000) notes:

> "*Thinking that a high quality computer mediated conference-collaborative learning system is sufficient to generate good collaborative dialogue is clearly an inadequate model.*" (Owen 2000 p 188)

There is clearly a need for pedagogical strategies to underpin the use of CMC and associated technologies in a wide range of subject areas. Some concerns will be common across subject area: increasing the confidence of participants could be one common theme but others will be peculiar to a discipline or even an area of a discipline.

In summary, even the basic terminology requires dedicated effort to untangle and the history of CAL development, with its rich diversity of projects, approaches, ideas, aims and theories is a complex one. Each decade has brought new enthusiasms to the field: in the 1960s it was drill and practice software, in the 1970s it was new hardware (more memory, better displays) and in the 1980s it was the rise of the CD-ROM. In the 1990s it was CMC. Almost every new development in computing has either sparked new life into existing ideas or generated new ideas. However, the durability of much of the software (and the ideas behind it) has proved disappointing. Many millions of pounds and dollars have been spent on equipping schools with computers and the software to teach or manage the learning process: almost all of it no longer in use. Books, papers, reports and articles about the new revolutionary CAL or ICAI lie unread on library shelves.

The second part of this chapter looks at the work done by the author around firstly, a piece of tutorial CAL, called Introduction to Programming and secondly, what could be termed a piece of emancipatory CAL, a small scale on-line help system for Pascal.

Background to the tutorial CAL development

CAL development began in the School of Engineering at Humberside University in the academic year 1992-1993. There were a number of re-organisations of departments and schools within the University, in the 1990s, including a name change for the University itself, when the Lincoln campus was opened in the late 1990s. The University became the University of Lincolnshire and Humberside. The School became the Department of Engineering in the Faculty of Arts and Technology in 1998. The last two years of CAL development were undertaken commercially in a joint venture company, called TekniCAL, founded with an electronics company called Feedback Instruments PLC.

The evolutionary process of adopting CAL within the School of Engineering began with an acknowledgement that changes in teaching methods were inevitable, due to a number of factors:

> "*Four factors combined to provide the impetus and necessity for change. A reduction in staff-student contact hours... Increased lecturing hours... Planned new courses and student numbers... Staffing and money constraints.*" (Stokes 1994 p 5)

The work of investigating CAL and developing CAL materials within the Department was the result of one person's enthusiasm and commitment to this area of teaching and learning: Roland Stokes. His energy and belief in the value of CAL drove the development and use of CAL within Engineering for almost the whole decade. Without him, the author does not believe that the work done would have even been contemplated, much less successfully carried out.

One response to the changes in student profile and numbers was an investigation into the use of commercially produced CAL materials, in the academic year 1991-1992. Initially, eleven packages for the teaching of telecommunications were purchased and trialled with over one hundred students. These students were studying the dual subject degree of Business and Technology. The technology elements of the course presented some difficulty to these students who often would have chosen 'pure' business courses were it not for the A level points required for such courses at the University. Telecommunications, with its mathematical and formulaic elements (the loss of signal in fibre optic cable, for example) often caused students difficulty. The packages were purchased to support the teaching of basic and more advanced telecommunications material.

Of the eleven packages purchased, it is interesting to note that only one package was developed in the UK. The cost of these packages represented considerable capital investment. As with any capital purchase, the life of the asset is an important factor. Other considerations were practical: some packages restricted the use of the floppy disk containing the material to specific computers. The fact that the material was on disk meant that disks had to be signed in and out by a member of staff, severely limiting student access to the CAL .

One of the key advantages of CAL, allowing students to manage their study time, was largely lost due to these restrictions. It was felt that the packages should be available anywhere on the network and not require staff time or presence to enable students to access the material.

The material purchased was used over two years, in a number of courses. The CAL replaced lectures or seminars or formed the basis of assignment work. Student response (gathered via a questionnaire) was analysed. A number of factors emerged: students generally liked the idea of CAL but found it irritating to use a range of packages that had different interfaces and navigation paths.

There was also a mismatch between lecturer and syllabus requirements and the material in the package. Much of the material could not be tailored except by missing specific sections. As the packages were used extensively, technical or design flaws became apparent: one package required fifty keystrokes to return to the top-level menu. The CAL generally lacked supporting documentation, forcing the students to transcribe from the screen if they were less confident and wanted a copy of the material. The interfaces on the packages tended to be rather DOS-looking (limited number of colours, large text and very basic graphics) and lack a sense of design. These flaws generated much student comment and formed the basis of a requirements list.

The next step in the use of CAL was to design and produce in-house materials, bearing in mind the requirements generated by the experience of using bought-in CAL. It was decided to give a final year group of students the remit of developing sample CAL and, equally importantly, a robust and well-documented method of producing CAL. The team consisted of four final year students who were industry-sponsored part-time students. One of the team had worked in the computer-based training department at British Aerospace. The team had a range of skills, including one student with considerable software skills. The project was to be undertaken in the academic year 1992 - 1993.

The students investigated various authoring languages available at the time (the summer of 1992) and chose IconAuthor, a flowchart-based language that allows the programmer to builds a package by combining icons into a flow of control. Constructs such as loops and IF... THEN are pre-designed. The flow of control describes the order in which pages appear and each page has a screen of text and-or graphics associated with it. Another advantage of this language was that once a package template has been set up, the template could be re-used by changing the page files.

Because package templates were re-usable it was appropriate to spend considerable time developing a standard navigation pattern (toolbar at the foot of the screen and standard buttons on the toolbar) as this part of the template would be used for every package, whatever the topic covered.

The group were also asked to consider the issue of monitoring student use of the package. A simple student-time-in-package measure was considered sufficient but it was decided to build in multiple-choice questions foe every section, with the results stored in a database that could be accessed by the tutor but not by the student. There was, therefore, a record of the time a student had spent working through a package and the number of correct/incorrect answers. Students were required to logon to the package and were set up as users only at the request of an academic member of staff. This meant that students could not browse or work through packages before they were required to complete them as part of their coursework.

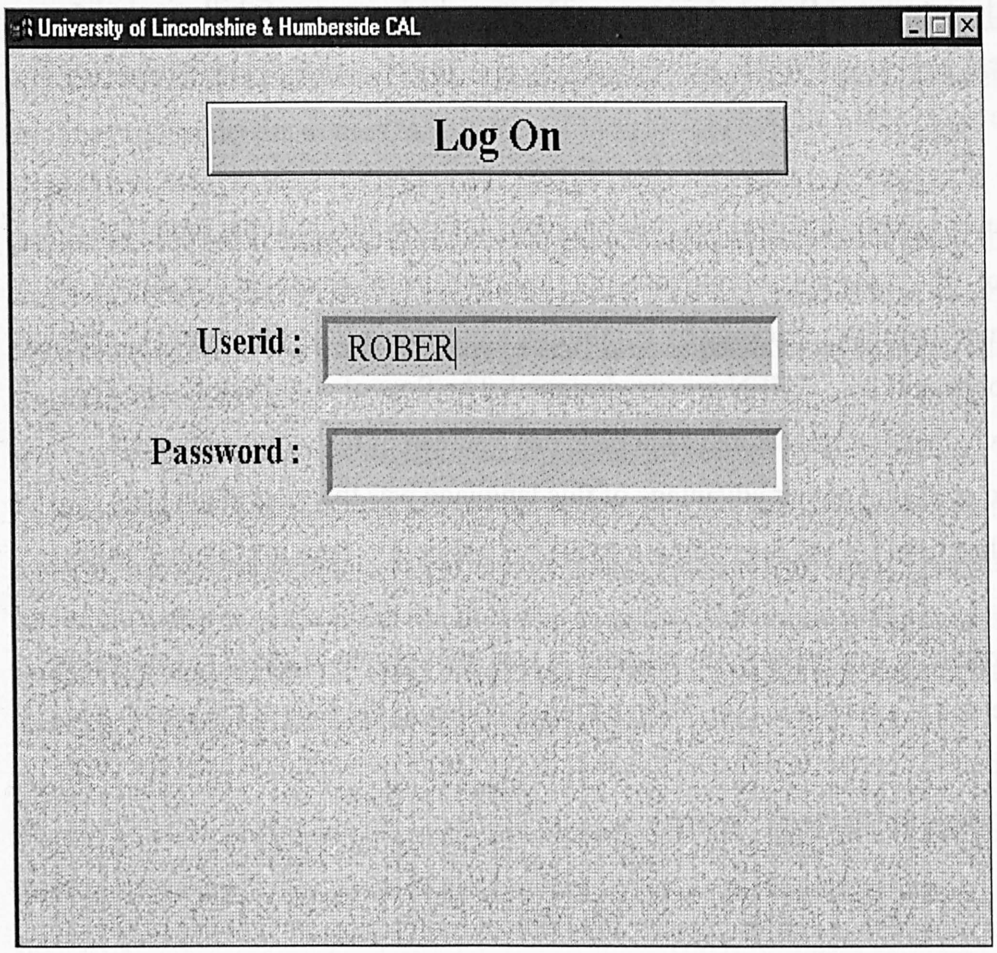


Figure 1: The logon screen

The project also aimed to produce a CAL life cycle that would cut the production time of CAL to fifty developer hours per mean learner hour. The effort rate for CAL development is a difficult concept that has worried CAL researchers and developers since the 1960s. Figures quoted vary wildly, up to 1000 hours of developer time to 1 hour time of CAL usage time. A consensus appears to have arisen around the figure of 200 developer hours per learner hour. Reducing this was not a trivial consideration. This developer effort (however it is measured!) is what makes most CAL packages so expensive. If the developer effort could be reduced, it would be feasible to produce CAL in-house.

This initial project, lasting an academic year (1992-1993) developed a full CAL package (on Controlled Area Networks, an aspect of avionics), re-usable program templates and a carefully-controlled CAL methodology. At the end of the project, the team had delivered a package and associated project documentation as required in the initial brief. The School of Engineering had a good basis for further CAL production. "*With the successful completion of the design project imminent, it was decided to bid for additional staff to produce* [CAL] *materials.*" (Stokes 1994 p 8)

Part of the funding for the next stage came from BP (British Petroleum) and in August 1993 a team was set up to produce 80 learner hours of CAL. The team consisted of four students on their placement year together with two members of academic staff who took on the roles of part time team supervisor and editor. Lecturing staff whose material was to be turned into CAL were given some remission so that they could liaise with the team. At the same time a research assistant, Robert Carter, was appointed to look at ways CAL could be broadened, so that students could undertake investigative work, using CAL.

The CAL package focused on Fourier transforms. It was designed to allow the student to investigate relationship between time and frequency. The analysis program represented waveforms graphically in both the time and frequency domain. It was designed to be easy to use, thereby allowing the student to learn important concepts of time and frequency analysis, unhindered by complex mathematics.

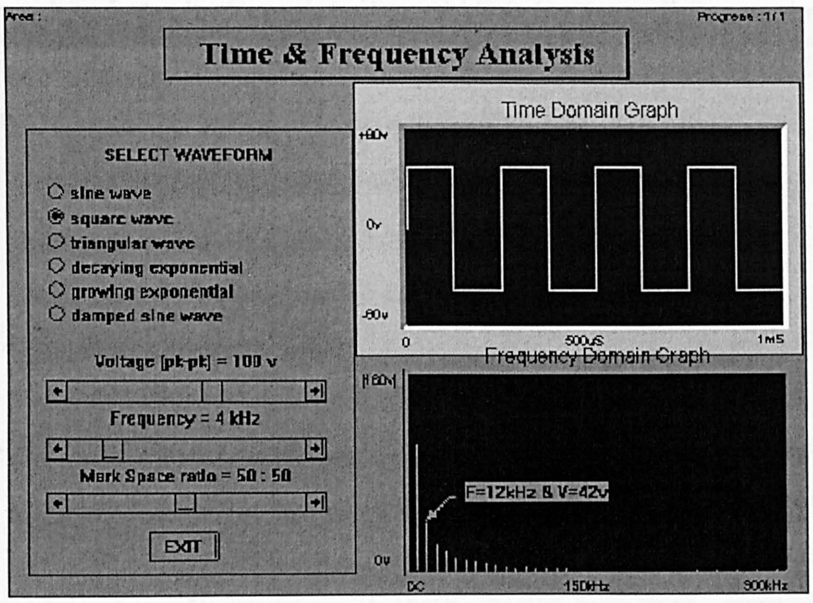


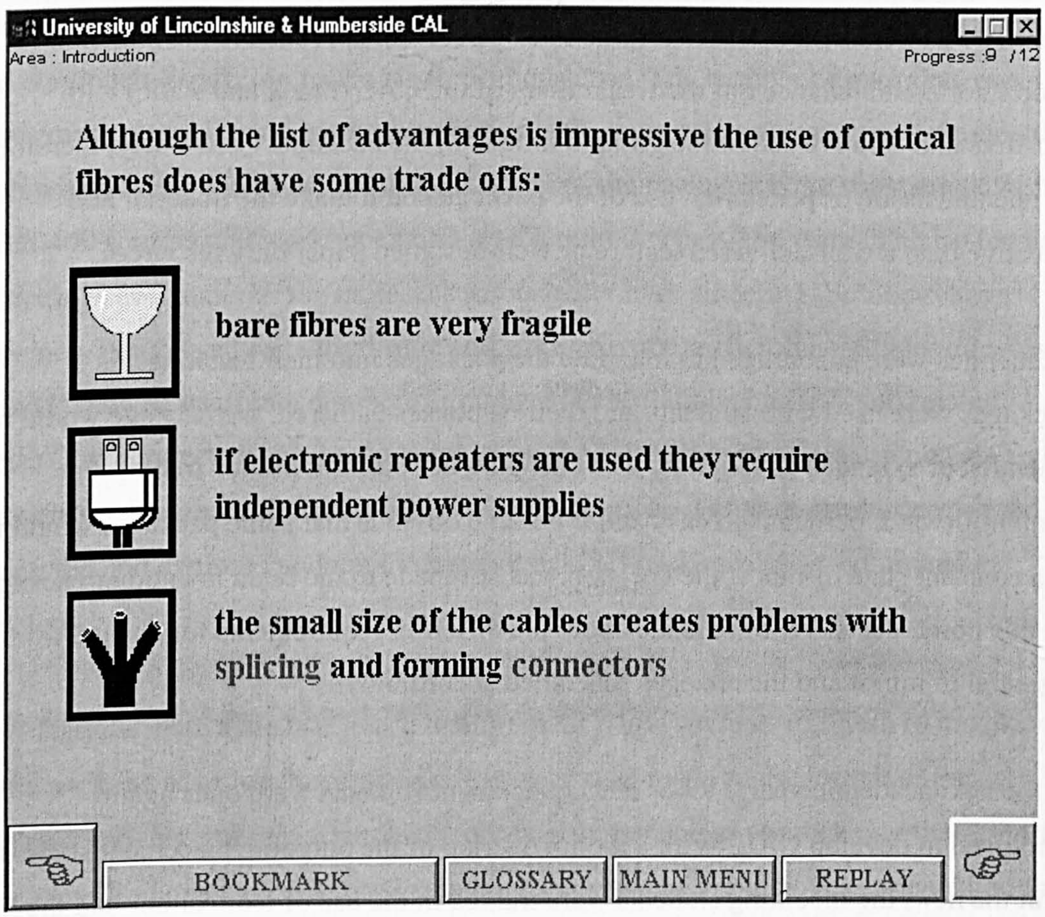
Figure 2: The time domain

The Fourier package demonstrated how the CAL could be developed to include skill-based learning. This avenue was promising but the programming requirements of such work were considerable. Most of the code for the dynamic display was written in C and integrated into the IconAuthor program: a non-trivial task.

The template-based, tutorial or page-turning approach was decided upon for other subjects. The initial technology subject chosen was propagation. After the CAL was written, it was trialled with 50 students for 3 weeks. The CAL replaced the lectures normally scheduled. The students had one seminar class a week, to allow the tutor to monitor their progress.

A questionnaire was sent out to the 50 students at the end of the trial period. The response rate was 70%. The percentages below are based on the sample that returned the questionnaire. 96% of the sample enjoyed using the package, 94% of the sample felt it was at the correct academic level and 75% of the sample felt it could replace the lecture.

This positive response gave further impetus to the development of CAL within the School. At the end of the year, the team had produced packages on propagation, computers, satellites, fibre optics and electronic principles for communications. Other packages were in development. The use of templates, the clear delineation of production responsibilities and the carefully managed involvement of academics all proved to be key elements on the efficient production of required packages. The team had demonstrated that it was possible to produce one learner hour of finished CAL for approximately fifty developer hours of effort.


University of Lincolnshire & Humberside CAL

Area : Introduction

Progress :9 /12

Although the list of advantages is impressive the use of optical fibres does have some trade offs:

bare fibres are very fragile

if electronic repeaters are used they require independent power supplies

the small size of the cables creates problems with splicing and forming connectors

Figure 3: A sample CAL screen

What was not adequately explored were the deeper implications of the tutorial, page-turning CAL approach that was adopted. Little effort was made to justify, in pedagogical terms, the adoption of linear, text-based CAL. The behaviourist model undoubtedly underpinned the idea of the end of section multiple-choice questions. Students were given some feedback (a cross or a tick beside their answer) but not the correct answer. However, the broader context of the CAL could be seen as more constructivist and possibly more useful in terms of genuine student learning.

The author witnessed a package on introductory electronics being used by a colleague, with a small group, in a very flexible and sensitive fashion. Students were encouraged to confer, to ask questions of each other and the tutor and the tutor frequently brought the group together to elaborate or explain a particular point or concept. Here we can identify a key element of CAL use: the tutor's personal approach to the material. A good tutor could balance out the weaknesses of the CAL (linear and static) with sensitive and creative use of the group's time with the package. Conversely, therefore, an unenthusiastic or pedestrian use of the package could make the material less effective than a well-delivered lecture or well-designed paper only resources.

Academics were encouraged to integrate the packages into their teaching. As packages were used with students and their responses gathered, the package would be amended or extended as required. The changing of packages proved to be straightforward, perhaps too easy and it became obvious that some packages would be in a constant state of flux if the decision was not made to move on to another package. At this point, decisions were made about the differing requirements for change from essential to minor and the changes scheduled accordingly.

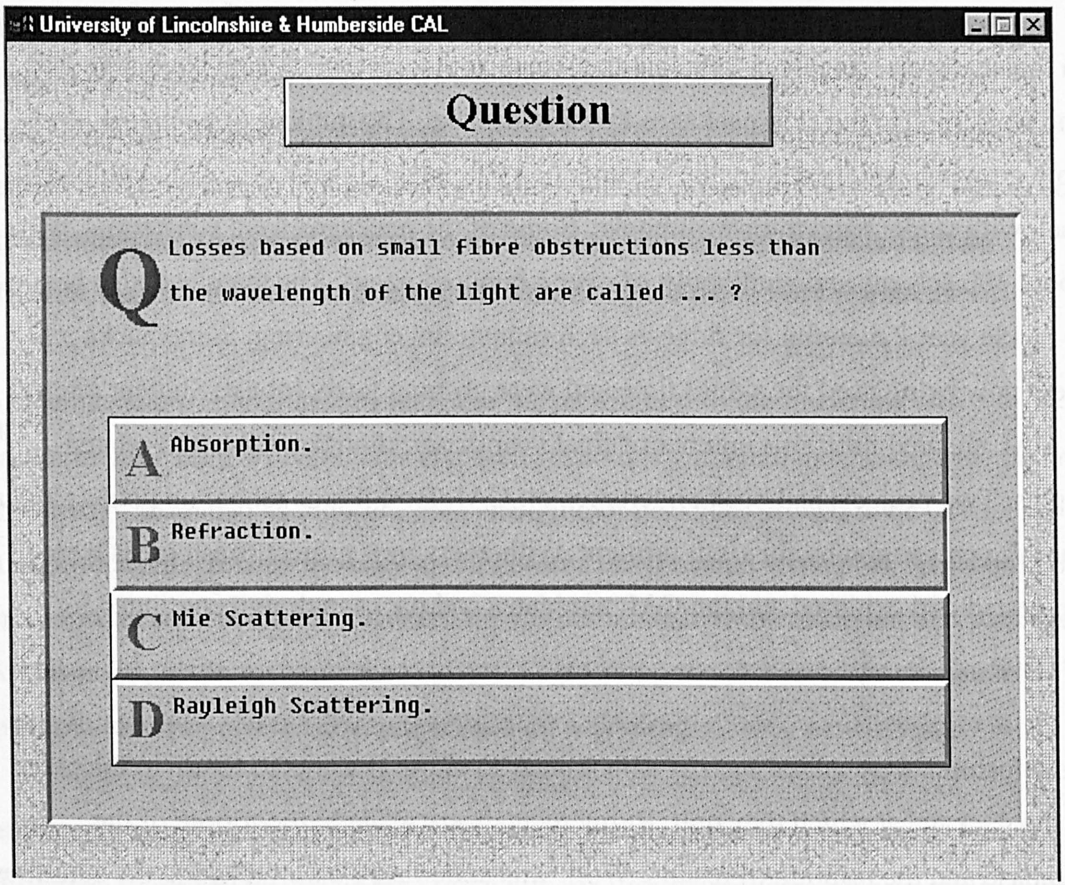
In the next academic year, 1994 - 1995, placement students were once again employed to develop CAL. With a month overlap of the outgoing placement students with the new, the incoming students were trained in the use of IconAuthor and the templates. Student projects in IconAuthor were also set up, to investigate particular aspects of CAL. One project looked at the integration of video into a package on materials technology. The package was successfully completed and trialled with metallurgy students but at the time, the network at the University did not have the bandwidth to carry video.

The use of moving images could be seen as a key element in effective CAL but instead of video clips, the packages incorporated animations done by a full-time graphics specialist. Again, this specialisation proved effective: requirements for a graphic were outlined to the team member and produced to order. This also avoided any copyright problems with images. The animations tended to be broadly illustrative of the material rather than fully informative. In some cases, the illustrations were humorous and not truly relevant to the material in the text. For sample pages with graphics, from the Introduction to Programming CAL, see Appendix H.

In the academic year 1995-1996 another team of placement students were recruited and continued to develop packages for in-house use. The number of packages was now growing rapidly and by the summer of 1996, over 20 packages were completed, each with student and tutor workbooks. Some of the packages were substantial, with about 400 screens and workbooks of over a hundred pages. The team had also begun to tackle year 2 material for subjects such as material engineering. Some packages were now being used for a third time and were proving fairly robust. Student interest in CAL showed itself in a number of final year dissertations on CAL and related topics. Two projects had been undertaken on CAL for a junior school on planets and a package on geography for sixth formers. With every new project, fresh ideas were suggested and explored and the CAL templates and package design continued to develop.

New features were added such as a bookmark to allow students to return to the place in the package at which they quit. Refinements were made to the format of the database and the printing of results. Work was undertaken on broadening the activities on screen, for a package that used equations extensively. Input boxes appeared on the screen for students to enter the number answer for a calculation. Tolerance was built in: answers of 3.45 and 3.4 and 3.41 would be acceptable. Students were allowed two attempts at an answer before the correct answer was displayed and the next screen or question displayed.

Extensive work also went on in looking at the end of section questions: the team explored the problem of free-text input, a goal much sought after by CAL developers. After much experimentation, it was decided to keep the multiple-choice format of questions but to return to the idea of different question formats in the future. New versions of IconAuthor have broadened the test options: version 7 supported drag and drop questions and answers. However, most of the packages continued to have the standard end-of-section questions in multiple-choice format. (See Figure 4)



University of Lincolnshire & Humberside CAL

Question

Q Losses based on small fibre obstructions less than the wavelength of the light are called ... ?

A Absorption.

B Refraction.

C Mie Scattering.

D Rayleigh Scattering.

Figure 4: Sample multiple choice question screen

The team also kept in touch with developments in TLTP and spent time looking at the Phase 1 material. Looking at the packages described in the TLTP Catalogue (Phase 1 - Spring 1995) there are just over 380 packages listed. (Some projects did not give a detailed list of individual elements or packages so the figure is approximate). A survey of the science, mathematics and computing packages shows that the bulk of the packages are described as being designed for students to use at their own pace, as self-study or independent learning materials (the phraseology varies). Other packages were designed to act as support environments or to be called from within other packages.

Out of approximately 117 packages 100 were described as being designed for self-study. The figures are approximate as some projects do not list the pedagogical approach of each individual package. This contrasted strongly with the School's focus on tightly managed, week-by-week monitoring of student use of CAL packages.

The academic years 1996 - 1997 and 1997 - 1998 saw new placement students join a now permanent and full time CAL Production Team Leader and a graphics specialist. The number of packages grew to over 50, including Introduction to Programming Ideas, Introductory Maths, First Aid, Thermofluids and Oscilloscopes. Student projects looked at the teaching of skills, such as using CAL to introduce students to key concepts before they move on to work with the full UNIX environment.

The academic year 1998 - 1999 meant major changes to the production of CAL. The CAL production team grew to 11 full-time staff (some on placement, some permanently employed) and the formation of a joint venture company called TekniCAL, in conjunction with Feedback Instruments Limited. Feedback Instruments Limited provided the sales and publicity infrastructure, selling the CAL through their existing Feedback staff, across the world. This proved successful and in the summer of 1999, the team moved out of University premises, into commercial office space. Effectively, CAL development moved out of what was by now the Department of Engineering entirely. Copyright of the material stayed with TekniCAL, although the Department of Computing and Communications and the Department of Engineering could use the packages, developed from 1992 to 1999, within the University.

In fact, use of the CAL packages ceased entirely within a year. Without a team to maintain and support the packages, staff no longer used them. All the effort that went into the packages was not entirely wasted but the long-term benefits envisaged when the project first began never materialised. There was no large-scale replacement of lectures with CAL and the CAL certainly did not mean that fewer lecturers were needed. Economies in staffing were generally found by increasing seminar group sizes and by increasing staff teaching hours to the maximum allowed. If the development of CAL did not benefit the institution as hoped, did the students benefit from its introduction and use across their subjects? The next section looks at the ways student responses to CAL were gathered and the general tone of those responses.

Gathering feedback on CAL use within the Department

As noted earlier, work began on the use of what was then termed CBT (Computer Based Training) in the academic year 1993-1994. Questionnaires were distributed to students who used the initial CBT bought in to support mainly the teaching of telecommunications material. What kind of comments were students making on the commercially produced, communication principles and telecommunications CBT? It should be noted that there was never any coherent, long-term design for the gathering of student feedback. Questionnaires were produced and employed but there was no review of the methodological considerations involved in such data gathering.

The earliest questionnaires were administered in February 1994 to students using a commercial CBT package on propagation. There were 30 replies: 28 out of 30 enjoyed using the package. 28 out of 30 felt it to be at the right academic level. 29 out of 30 felt it could replace the lecture. Feedback from students tended to focus on problems with the package:

> *"The questions sections are not correct."*
> *"The amount of annotation in the package could be increased."*
> *"There aren't enough figures and diagrams to illustrate points."*
> *"I think larger page numbers in the top right hand corner would make this clearer and less confusing."*

(Student questionnaire replies 24-02-1994)

There were some comments about the learning required:

> *"Bit parrot fashion, not really made to think."*

"The copying of paragraphs straight from the screen to the sheet became repetitive, perhaps more interaction between user and package?"

(Student questionnaires 24-02-1994)

At the end of the use of the package, the questionnaires were administered again on the 10th March 1994. We could ask, did this filling in of questionnaires become wearying for the students? Statistics were gathered on the length of time students used the package and the number of correctly answered questions. The average time of use of the package was 6 hours. Total percentage of correct answers varied from 0% to 93%. One comment on using the package was: "*I prefer using CBT because I can work at my own pace and go back if I don't understand. I find that after half an hour of a lecture I tend to start fidgeting...*" (Student questionnaire replies 10-3-1994) This kind of questionnaire and feedback is generally common among CAL trials. The generally positive tone of the responses was taken as a strong vote by the students for more use of CAL as a teaching vehicle.

In April of the same academic year (1993 - 1994), a general questionnaire about what was then called CBT was administered to second year students on the BSc Business and Technology course.

Question 2 asked, "How do you prefer to learn a new topic or subject? Lectures, tutorials, private reading, discussion with friends, through assignments, seminar or lab work, through CBT? Try to describe how you tackle a completely new subject."

Student comments included:

"*I personally prefer learning a new topic through an introductory lecture and then finding out more for myself.* [] *I do not mind using CBT but prolonged periods tend to lead to severe boredom.*"

"*I think lectures to introduce a totally new topic are good to outline the basic ideas. A discussion in class also helps, especially if you can relate it to every day life. With CBT it is easy to totally mis-read something and therefore your first impression of a new topic could be totally wrong so I don't think CBT in this instance is a good idea.*"

(Student questionnaire replies April 1994)

The students were generally in favour of a traditional lecture approach, and a few were strongly against the extensive use of CBT.

Question 3 asked, "How do you feel about the CBT you have done up to now? If you could identify clear advantages and-or disadvantages, what would they be?" Replies included:

> "*If we miss something, we can take it whenever we want to.*"
>
> "*You can work on CBT at your own pace. Problems may arise if you do not understand a topic but most I have done are straightforward and easy to understand.*"
>
> "*I have mixed feelings about the recent CBTs. How I accept them depends on my mood at the time. If the subject is completely new then I tend to be more interested than if I have had a lecture on it or have come across it before.*"
>
> "*I don't like working with CBT, especially not for 2 hour long seminars. It's easy to forget work, and takes along time to write notes on everything. The only advantage of CBT is it allows you to work through it at your own pace.*"

(Student questionnaire replies April 1994)

Students were very clear about the advantages and disadvantages of using CBT. They were able to argue both for and against its use and to identify situations where they felt it to be effective. The key point that seems to emerge from this is the need for flexibility in employing CAL. Students, even early in their degree, were able to articulate the need for varied approaches in delivering material. The kind of blanket consensus implied by some who report on student feedback for this kind of project was not evident: students displayed a more discriminating response to the software.

Question 4 asked, "How do you see CBT fitting into the learning environment for your course-subject? Should there be more or less CBT? Is CBT better than any other method?" Replies to this question included:

> "*I think CBT is a good, efficient idea and programme but it should not dominate classes as it will cut off class interaction and communication.*"
>
> "*...I do not think it should take over lectures seminars as these are normally more beneficial and raise questions. I would prefer to read a book than work on a CBT package. May be 1hr per subject on CBT a week could be done in addition to the 12 hrs already allocated.*"
>
> "*I think there should be less CBT and more group discussions in seminar. At the moment were doing loads of CBT and in the end it just gets boring – so you don't read it all and miss loads out.*"

> *"I find CBT is the best method of being introduced to a new topic-subject and makes the learning process easier than using traditional teaching methods such as lectures."*

(Student questionnaire replies April 1994)

Again we can see the fine-grained response of the students. The emphasis is definitely on interactivity, either with peers or with the software (which would, of course, be a very different type of interactivity!)

Question 5 asked, "What could be added to CBT to improve its effectiveness? Any ideas welcome!" Student responses included:

> *"More interactivity*
> *Less writing*
> *More practical examples*
> *Things explained better."*
> *"It could be made more interactive, like more multiple choice questions to break up the text. I think it should have less writing on a screen as it just overwhelms you. It would be good if you could ask the computer questions about things you're not sure about."*

(Student questionnaire replies April 1994)

Students again identify some key weaknesses of a blanket approach to teaching a subject through CAL: it becomes tedious quickly and the lack of interaction frustrating. Also the copying of information from screens is time-consuming and presents the same sort of problem as extensive note taking in lectures the students are so busy writing they are not really paying attention to the material in any meaningful way.

To question 6 "Should CBT replace the bulk of mass lectures? Does CBT even need scheduled classes?" replies included:

> *"No lectures should continue, CBT should not dominate classes any more than it already does."*
> *"No. Human to human interaction is usually more interesting, although at times not necessarily as informative."*
> *"NO It should have separate scheduled classes but with 1 lecture and 1 seminar per subject."*

> "*No, definitely not. A well presented lecture at a reasonable speed, will always be better than a CBT I think it may be better if CBT did have scheduled classes, as well as a formal group seminar to support the work on the CBT.*"
>
> "*I don't think CBT should replace the bulk of mass lectures. I think you need lecturers as often they can explain things better then a computer.*"
>
> "*I think it would be worth while having more CBT. Even though it entails more work, you definitely have to put in the time = reap the benefit i.e. concise notes which you have had to understand in order to complete the tests.*"

(Student questionnaire replies April 1994)

The overall tone of the student responses was thoughtful and their replies displayed a genuine willingness to engage in a discussion about CAL. It seems to the author that there were key issues raised by the students (such as interactivity and the importance of peer and lecturer interaction) that were not addressed in any fashion by further project developments. The general willingness of the students to use CAL, with some reservations, was seized upon and made much of, without a concomitant willingness to make much more flexible and subtle the academic structures in which the use of CAL was embedded.

In November 1994, 63 BSc Business and Technology students used a package on electronic principles. A questionnaire was distributed and 49 replies were received. Average age of the student was 18.8 years.

Comments included:

> "*The idea of working independently, without lectures works for me, as there are no limitations on the speed I wish to work.*"
>
> "*I feel I have learned and more importantly understood all the sections in the CBT programme more so than I would in a lecture.*"

(Student questionnaire replies 18-11-1994)

In January 1995, with 73 HND students using the package in electronic principles, 45 questionnaire replies were received. Interestingly, all of the respondents preferred the CBT to lectures.

It is also interesting to compare the BSc and HND students over 1994 and 1995. The pass rate in the package for the degree students was 46.3% and for the HND students, 74%. This variation could have a number of causes. The student cohorts must have been different, by virtue of the fact that they were degree and HND students. The academic backgrounds between the two groups must vary. Students in the HND cohort being vocationally oriented rather than traditionally academic could account for the variation. Some of the HND students may already have worked in or studied electronics. The BSc students were generally post A level entry with no requirement for a maths or science A level.

At this point, the Head of School, Roland Stokes, felt that the case for using and developing CBT or CAL in-house was largely proved. The focus of effort moved to development rather than exploring student attitudes to CBT or CAL. This created what the author felt to be a serious weakness in the project. Little or no effort was made to critically evaluate the impact of the use of CAL on learning outcomes. The assumption, generally unquestioned, was the use of in-house CAL was academically sound in its conception and application. The author felt that a full study of all the CAL packages developed, which was nearly 60 large packages by the spring of 1996, was too great a task for one research project. The author decided to focus on the teaching of programming and the potential support for this offered by CAL.

The author's own CAL

A first version of CAL on the basic concepts of structured programming was created in 1995 but the author felt it to be unsatisfactory and it was put aside. It was revisited in the academic year 1996-1997. It was completely re-written and extended considerably, to become the version shown in the sample pages from the student and tutor workbooks in Appendix H, called Introduction to Programming Ideas.

The CAL covered basic elements of hardware, including memory, and software. It looked at the history of programming languages and introduced concepts of structured program design, including arguments against GOTO and the use of the algorithmic toolkit. It then explored key aspects of the software lifecycle, covering problem solving, specification, top-down design, modularisation, implementation and debugging. The CAL also surveyed briefly some programming languages from

FORTRAN to Java. Finally, it offered some sample programs in Pascal, C++ and Visual Basic.

From this description, it can be seen that the CAL written by the author attempted to cover a great deal of material. It was useful, perhaps, from that point of view, but as page turning, tutorial CAL, it lacked interactivity (apart from the questions at the end of each section) and made no pretence of being able to support a student's individual programming effort.

Developing the material to the right level of detail was also challenging: too much depth and the CAL would be (and feel) much too large. Not enough depth and critical aspects of a topic or topics might be omitted. The final CAL represented a number of compromises in the scope and technical complexity of the material covered. One might question, how useful can just two sample programs be, in any programming language? Is it truly meaningful to present such small examples?

The author taught introductory programming at year 1 of the BSc computing degree and also at Foundation level. The Foundation year was designed for students who had not achieved any post-18 qualifications. Passing the Foundation year gave student access to year 1 of the computing degree but also to other degree first years, with the agreement of the course leader. Some students took the Foundation year with the intention of switching degree subjects: others intended to study computing or media technology, which was also underpinned by the Foundation year.

In January 2000, a questionnaire was administered to Foundation students. These students had used the author's own Introduction to Programming CAL in the December of 1999. Twenty questionnaires were given out. Eleven responded.

Comments about the learning of programming included:

> "*Very difficult if you have never used it before. Programming is easy if tutors go through it with us first, step by step.*"
> "*I have limited experience of Pascal but the bit I have done has been fairly enjoyable. Not all the help files are helpful.*"
> "*If you have never used Pascal before it is not easy to use the first time. Once you get the hang of it Pascal is no harder to use than any other package.*"

(Student questionnaire replies January 2000)

At the same time, the author examined the Foundation students' work on the CAL workbooks. Students, in using the workbooks appear happy to fill in the blanks and attempt the more demanding activities. Out of 18 workbooks examined, 11 had attempted some of the activities. The remaining 7 had not attempted any of the activities. This may be for a number of reasons. The activities were purposely designed to require the students to use sources other than CAL. Students who wanted just to finish could give themselves a sense of progress by skipping all the questions that needed research and reformulation of material, away from the CAL. Student responses where the activities were attempted varied. Some were very detailed while others were more perfunctory but still relevant. The question would be, what contribution did the undertaking of the activities make to the students understanding? Did the students who ignored the activities lose out in terms of material covered?

It seems to the author that no definitive conclusions could be drawn from this small sample of qualitative responses. The students, new to programming, had found it hard, but that was expected. The CAL did not appear to address any of the problems presented by the students and the background to structured programming material contained in the CAL did not seems to connect with the <u>use</u> of a structured programming language. The use of tutorial CAL to support the learning of programming seemed to be a dead-end. There was nothing in the (admittedly small range of) student responses to argue that they had seen the CAL as relevant to their learning of, and coding in, Pascal.

One problem the author did identify was the reluctance of students to work on their code outside scheduled tutor contact times in the lab. There were comments about the help files contained in the Borland Pascal compiler. Students found the help text provided by the compiler generally confusing and unclear. Was the lack of focused feedback, couched in simple terms, hindering the students? Students undoubtedly need help in debugging programs and generally they relied heavily on tutor intervention to deal with fairly simple debugging problems. .

Other research on novice programmers

Perkins et al. (1989) identify two types of novice programmers, what they term 'stoppers' and 'movers'. Stoppers stop dead when they encounter a problem and have no idea of how to progress and no idea of generating new approaches. Movers try new approaches but often abandon potential solutions because the fix does not work first time and often they re-try failed and unpromising approaches just for something to do. (Perkins et al. 1989 p 266) Movers do not appear to learn from previous attempts at problem solving.

Perkins et al. (1989) note that the affective element is very important in novice programmers "*Computer work can be a challenging and stimulating experience, but it can also become a threat to self-esteem and one's standing with peers and teachers.*" (Perkins et al. 1989 p 267) Linked to this affective component is the question of making mistakes. Students can either tackle mistakes they make without too much frustration, seeing it as part of the learning process, or they can see every mistake as reflecting badly on their knowledge and self-worth. Perkins et al. (1989) go on to state, "*Such attitudes almost inevitably breed stoppers. Naturally, some stoppers become so disengaged that they learn very little.*" (Perkins et al. 1989 p267) In such cases, tutor intervention is needed at critical junctures to move the students on. Interestingly, the example Perkins et al. (1989) give is that of a student who ignored an error message ("Subscript out of range") because he did not understand it, until the tutor posed a number of helpful questions.

These points echo the author's own experiences of teaching programming. Students tend to regards their programs as direct reflections of their own worth and a program that does not compile or does not function as expected, directly devalues them as learners, in their own view. Students struggle with compiler error messages and on-line help, which adds to their sense of frustration. Some refuse to engage with the task of programming unless forced to via marked assignments and then only in a last minute hurry where lack of achievement can be blamed on lack of time rather than lack of knowledge or understanding.

Another area in which students learning programming often struggle is that of tracking their code so they can identify and solve problems in the software. This, again, says Perkins et al. (1989) can be traced to a confidence issue: students may not be confident they understand their code or the elements of the code and not at all confident that they can fix it. Close reading of code is like proof reading, say Perkins et al. (1989) and would therefore, in this author's opinion, subject to all the cognitive and issues problems proof reading presents. Perkins et al. (1989) identify strategies that movers use to solve code problems, such as tinkering: amending or developing one or two lines at a time.

Tinkering can be positive and effective but it can also be just 'busy work' that will never solve the problem. The difference is in the close attention to code required for effective tinkering. Again this chimes with the author's own experiences as a programmer (writing a program revisited a year later and having no idea what the uncommented code is supposed to do!) and as a teacher of programming. One year, a student had written code with about twenty levels of nesting: it was beyond her ability to debug to that level and there were serious errors in the code that could not be fixed. The student would not or could not explain how he had managed to write code that was effectively un-debuggable!

The author wondered if the most productive area to concentrate on was help with compiler error messages. It seemed to be an area that offered scope for supporting the students outside the scheduled laboratory times. By this point, the author had concluded that the Introduction to Programming Ideas CAL did not address the practical problems of teaching programming. As background material it had some value but it did not do what the author had originally hoped. It did not serve as a platform for the development of confident, competent programmers and with the benefit of hindsight the author acknowledges that it was never very likely so to do!

Spohrer and Soloway (1989) argue that novice programmer misconceptions about constructs are not "*as widespread and troublesome as is generally believed.*" (Spohrer and Soloway 1989 p 401) The problem lies with the putting together of constructs and instructions in a program. Spohrer and Soloway (1989) explored the types of bugs and frequencies of those types found in novice programmers. They concluded that for three programming problems set for students, 20 percent of bug types account for 55 percent of bug instances. To support their assertion that construct misunderstanding is not a large factor in flawed code, Spohrer and Soloway (1989) cite figures from their study showing that more than half of the bugs identified "*had plausible accounts that were definitely not construct based. Less than 10 percent of the bugs had plausible accounts that were definitely construct based.*" (Spohrer and Soloway 1989 pp 409-410) Spohrer and Soloway (1989) conclude that the performance of programming novices may be improved by teaching them about the most common bugs and by teaching them strategies to piece together code effectively.

The author's own Pascal Help system

When the compiler generates error messages, those are not context-sensitive and are often difficult for the new programmer to understand. Sample error messages for Borland Pascal include (the original numbering has been preserved):

Error #	Error Message
2	Identifier expected
3	Unknown identifier
4	Duplicate identifier
5	Syntax error
8	String constant exceeds line
10	Unexpected end of file
12	Type identifier expected
20	Variable identifier expected
21	Error in type

25	Invalid string length
26	Type mismatch
29	Ordinal type expected
30	Integer constant expected
31	Constant expected
32	Integer or real constant expected
36	BEGIN expected
37	END expected
38	Integer expression expected
39	Ordinal expression expected
40	Boolean expression expected
41	Operand types do not match operator
42	Error in expression
43	Illegal assignment
44	Field identifier expected
50	DO expected
54	OF expected
58	TO or DOWNTO expected
66	String variable expected
67	String expression expected"

(Borland Pascal 7.0 Help files 12-08-99)

[The original list goes up to 170]

Choosing an error message e.g. number 3 gives

> *"Compiler error 3: Unknown identifier*
> *This identifier has not been declared, or it may not be visible within the current scope."* (Borland Pascal 7.0 12-08-99)

Some of the error messages may never be generated by students working with small programs. Others are very likely to be invoked by missing variable declarations, missing semi-colons, misspelt variable names, missing BEGINs or ENDs or a missing full stop. What the author found was that students did not or could not use the error messages to make sense of the problem.

It seemed to the author that this was a major problem for the students and that this is where an appropriately written set of help notes might be most useful, rather that the tutorial CAL approach. In August 1999, the author used the software developed by the Department's Linux technician, to develop a small version of on-line help for Pascal. The author chose a subset of error that she encountered in student's early work and wrote short but hopefully more readable help for those common errors. For example, for the error generated by a missing semi-colon the author wrote the following feedback/help message:

Pascal lines are finished by a semi-colon which looks like this ;

There are rules about the placing of semi-colons but for this error, check the end of the program lines and make sure they have semi-colons. Each missing semi-colon generates an error.

The software was then made available to students using Pascal. For a listing of the software see Appendix I.

Use of the help system

The question was, was the on-line help developed by the author likely to be any more used than the Pascal compiler help? For the Foundation students, for the questionnaire administered at the end of the unit on software development (taught using Pascal) 17 replies were returned, out of a possible 24. Students were asked to rank the usefulness of resources available to them, from the list given in Figure 5. If a student felt that books were the most significant resource in their learning of Pascal, they were asked to give it 1 and the second most important resource a 2 and so on. Respondents were advised to leave blank any resources not used.

The author noted the ranking given to a resource and the number of students who allocated it a ranking. If seven students gave a resource a ranking of 3, 5, 7, 3, 2, 3, and 4, the total for the rankings would be 27 and the average ranking would be 3.85. If a resource ranked highly and gained all 1s and 2s (1, 2, 1, 2, 1, 2, 2) the average ranking would be 1.57. The lower the average, the higher the ranking of the resource. As a foray into quantitative data, the author acknowledges that this is weak but it can be seen as giving some indication of what these students perceived as most relevant to them at that time, in terms of resources for learning programming.

The most highly ranked resource was lecture notes. Second were sample programs, closely followed by students on the same course. Individual programming practice and lab sheets (provided by the tutor) were ranked at 4.3 and 4.5 respectively. Interestingly, the CAL (Introduction to Programming Ideas) and the Pascal help system received low rankings of 7.6 and 6.1.

This confirmed the author's impression of the value of her CAL to students learning to program: it was of little practical help. The low rating of the help system provided by the compiler also matched the author's general impression of students' responses to error and help messages. Generally, these messages served only to frustrate and baffle the students, rather than illuminate the problems they were encountering.

	Average Ranking
Language's Help System	6.1
Books from library	4.5
Internet sources	4
Myton house help desk	7.3
Friends-colleagues not at the University	6.7
Lecture notes	2.5
CAL	7.6
Individual programming practice	4.3
Sample programs	3.7
Online help set up by the University	6.7
Students on the same course	3.8
Magazines	12
Lab sheets (provided by tutor)	4.5

Figure 5: Resource rankings

The academic year 2000 - 2001

In the following academic year, 2000-2001 the author taught first year computing students on an introductory programming module, lasting one semester. At the start of the semester, the author distributed a questionnaire that profiled the students and attempted some exploration of each student's attitudes to learning. Questionnaires were administered in September 2000 to the incoming group of first year students.

88 questionnaires were filled in by students, out of a cohort of approximately 164. Seven students were female. 72 were male. The remainder did not indicate their gender.

Four students had five GCSEs, nineteen had nine GCSEs. Two students had 11 GCSE's and two 6 GCSEs. Ten students had ten GCSEs but the largest number had eight GCSEs. Nine did not note their GCSE results. Two students declared their education at 16 to be "junior high". 21 students had either GCSE computing or GCSE IT. For education after age 16, there was a fairly broad mix. 21 students had three A levels and 13 students 2 A levels. 12 students had a GNVQ in IT. Students also had NVQs and GNVQs in tourism, electrical installation, art and design, construction and business. A total of 6 stunts had BTEC qualifications in IT or software engineering. From this brief summary, it can be seen that this cohort of novice programmers presented a wide variety of experiences and academic backgrounds, as did most intakes onto first year computing and the foundation year.

The questionnaire also asked students to specify their programming knowledge, especially the languages they knew. Students listed more than one language so the totals do not add up to the number of students. Programming languages previously studied by the respondents broke down as follows:

Pascal	32
Basic	19 (Including Atari Basic 8 bit)
Visual Basic	27
Unix - Linux	2
HTML	12
CNC Work	2
C-C ++	16
Machine code	1
Assembler	1
Cobol	5
ASM	1
A86	1
Delphi	5
Clipper	1
Q Basic	6
SQL	2
Java/Javascript	3/1
Prolog	1
Smalltalk	1

25 students declared that they had no programming experience or knowledge. Students were also asked to respond, in the same questionnaire, to a series of statements about learning e.g. 'Learning is easier in small groups', 'I find it difficult to grasp a new topic when I cannot relate to it', 'Correct lecture notes should be given for reference' and 'Learning is a process of being told what to do and then going away to practise it.'

They were asked to rate their response to each statement on a scale of 1 to 5 for Strongly Disagree to Strongly Agree (Likert Scale). Below are some of the responses shown graphically.

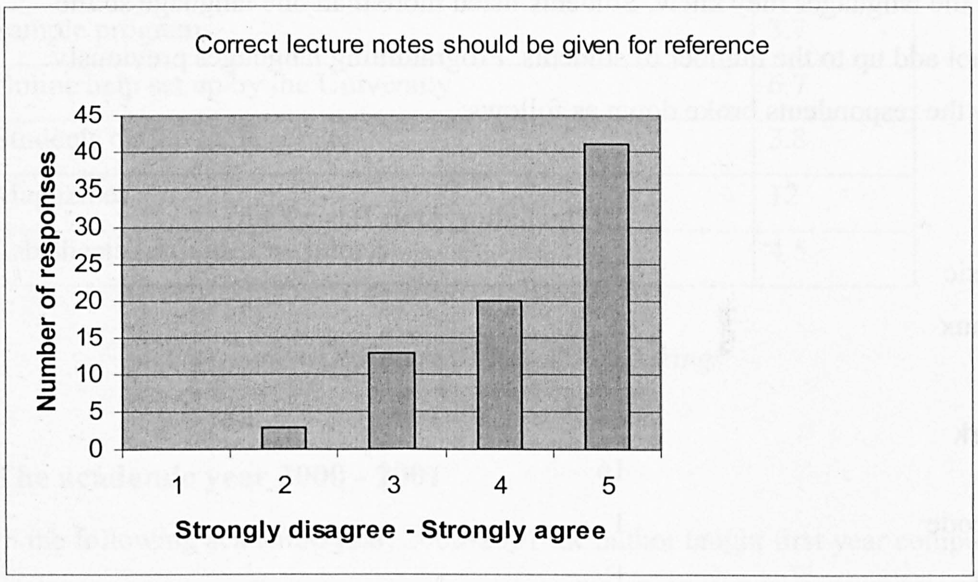


Figure 6: Correct lecture notes should be given for reference

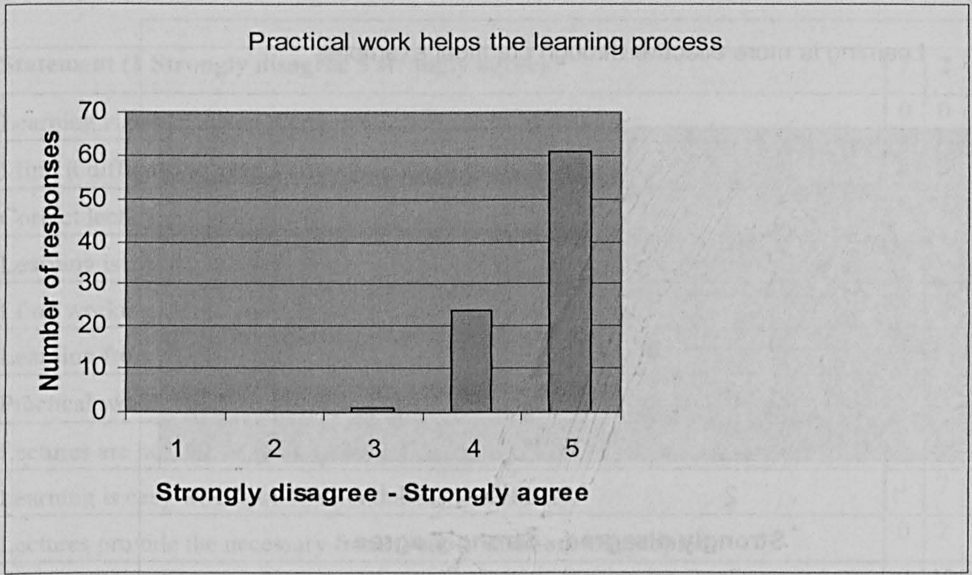


Figure 7: Practical work helps the learning process

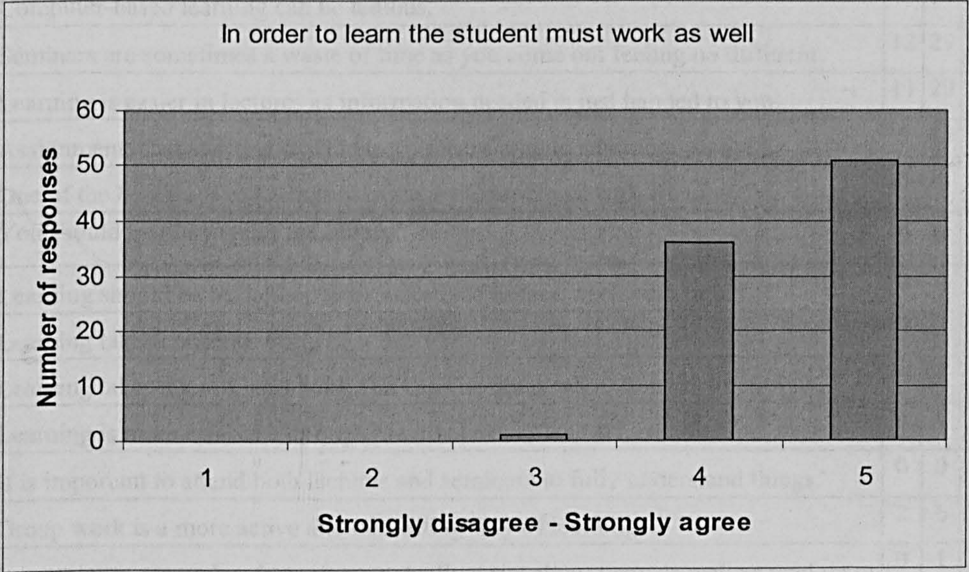


Figure 8: In order to learn the student must work as well

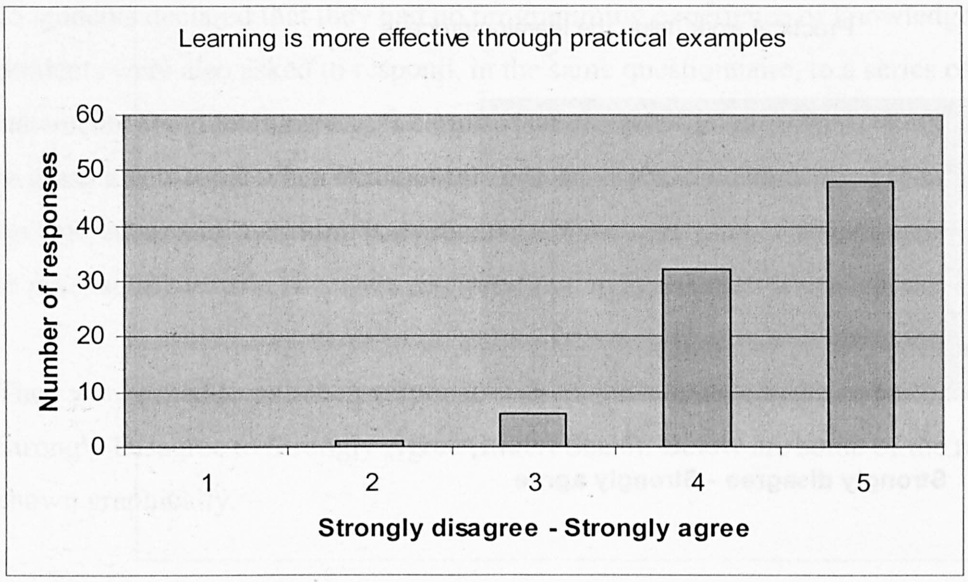


Figure 9: Learning is more effective through practical examples

Statement (1 Strongly disagree 5 strongly agree)	**1**	**2**	**3**	**4**	**5**
Learning is easier in small groups.	0	0	17	28	29
I find it difficult to grasp a new topic when I cannot relate to it.	2	14	25	27	9
Correct lecture notes should be given for reference.	0	3	13	20	41
Learning is a process of being told what to do and then going away to practise it.	4	21	23	30	8
I find working in groups difficult because people learn differently.	7	30	30	13	8
Learning from a booklet of printed lecture notes is difficult.	1	23	34	19	10
Practical work helps the learning process.	0	0	1	24	61
Lectures are not that easy to follow all the time.	1	13	23	40	11
Learning is easy when you have enthusiastic lecturers.	1	7	13	36	29
Lectures provide the necessary framework to understanding topics.	0	2	20	49	16
I like to learn in my own time away from the university.	0	10	30	33	14
There is some useful material to learn from on the Internet .	0	1	14	45	27
In order to learn the student must work as well.	0	0	1	36	51
I find it easier to learn and absorb information in silence	5	17	31	24	10
Computer-based learning can be tedious.	8	27	31	18	4
Seminars are sometimes a waste of time as you come out feeling no different.	12	29	28	14	4
Learning is easier in lectures as information needed is just handed to you.	11	29	37	8	2
Assignments that are long don't help you learn a topic any more easily.	5	14	38	23	7
One of the best ways to learn is to discuss the problems with friends.	0	4	14	46	24
You should learn by using the library.	0	5	26	39	18
Learning should be backed up with more face to face work with tutors.	0	4	19	43	18
Learning takes concentration.	0	3	6	42	37
Learning takes a lot of hard work and can't be done solely from lectures.	0	2	9	30	46
Learning is more effective through practical examples.	0	1	6	32	48
It is important to attend both lectures and seminars to fully understand things.	0	0	7	30	49
Group work is a more active and interesting way of learning.	2	6	27	30	22
It is easier to remember facts if you actually write about them as well as read them.	0	1	22	32	31

Figure 10: Responses to statements about learning

The cohort's responses to the questionnaire showed them to be strongly practical in their approach to learning, at least in theory. Their responses seemed to argue that they saw learning as a joint undertaking, rather than as a passive activity. The author's experience in teaching this cohort threw up a distinct dichotomy between the student's stated views on the learning process and their attitudes towards undertaking self-directed learning, spending time coding without tutor help and sourcing ideas for themselves. Their preferences were for a much more passive style of learning, with reliance on tutor generated materials, as can be seen from their responses to resources in the January 2001 questionnaire.

At the end of the unit, in January 2001, the return of twenty questionnaires, when the author marked 158 assignments for a year 1 cohort of 164 was disappointing but not entirely unexpected. The questionnaire was kept to a single side of A4 for the ease of answering but an open section was left at the bottom for general comment.

The first question asked what were the hardest and easiest aspects of the programming language studied in the unit. The hardest elements to grasp were variously listed as:

> "*passing- sharing the variables between procedures- modules,*
> *learning the syntax, using array of records,*
> *understanding whether or not variables should be local or global,*
> *the whole concept of algorithmic thinking and maths,*
> *procedures,*
> *relating arrays,*
> *Pascal syntax, arrays and records.*"

(Student questionnaire replies January 2001)

Eight responses mentioned parameter passing and four responses mentioned procedures only.

The easiest elements of programming were listed as:

> "*mathematic calculations within the program,*
> *debugging,*
> *'hello world' program,*
> *constructing useable pseudo codes,*
> *work for the first assignment,*
> *Syntax Stuff.*
> *Basic data structures*
> *Setting up variables and calling them variables and procedures.*"

(Student questionnaire replies January 2001)

Interestingly, one student did not fill in the easiest element response but noted under suggestions, "*Do the easiest stuff first be sympathetic to struggling students.*" It would have been useful to know what s/he considered to be the easiest stuff!

As before, students were asked to rank the usefulness of resources available to them. The author noted the ranking given to a resource and the number of students who allocated it a ranking. One questionnaire was spoiled (all responses were 1's) and one unreadable. The students had access to the help system written by the author using a code framework developed by the Department's Linux technician, later the System Administrator. For this questionnaire, the help system provided by the author was added to the list of resources to be ranked.

Resource	**Responses**	**Average Ranking**
Language's Help System	8	8.1
Books from library	12	19.6
Internet sources	12	6.8
Myton house help desk	6	8.3
Friends-colleagues not at the University	9	8.1
Lecture notes	18	3.3
CAL	8	7.5
Individual programming practice	11	2.5
Sample programs	18	2.4
Online help set up by the University	7	9
Students on the same course	15	4.26
Magazines	5	11.6
Lab sheets (provided by tutor)	18	3.3

Figure 11: Resource rankings

The most highly ranked resources were sample programs (2.4), individual programming practice (2.5) and, equally ranked at 3.3, lecture notes and lab sheets. The results are very similar to the previous results that used the resource rankings approach. The addition of a new approach, the tutor developed on-line help seemed to have little impact on the students' practice or learning and was ranked low by them (9).

Students seem to prefer tutor-generated materials, which are a combination of paper and electronic resources. The students were given programs to run: they did not have to retype code. Lecture notes were given out on paper at the start of the lecture and posted on the author's web page.

What can we conclude from this? That, despite all efforts by the tutor to move the style of learning to a more constructivist approach, students continue to rely heavily on tutor-generated materials. They do, in some senses, see each other as resources but anecdotal evidence suggests that a small selection of students are perceived by their peers as knowledgeable about a particular subject and it is a select few who are asked for help and input. Another factor cutting across the social context aspect of learning programming is the question of plagiarism. Students are warned about it in a number of situations, from formal student regulations to informal discussions of the acceptable boundaries of acceptable boundaries during presentation of assignment briefs.

Overview of work up to 2001

The use of the author's own help system was disappointing. Few students used it and few submitted serious queries. Part of the problem was almost certainly the lack of time available to update and develop the system so that it covered more error messages. Part of the problem was also the requirement to have another window open on the desktop and to type in a query number, key word or syntax item such as semi-colon.

Despite the poor response to the help written by the author, she still felt it to be a more promising avenue of exploration than the tutorial CAL originally developed by the author. The tutorial CAL has major problems, in that it addresses a very limited aspect of programming, that of concepts rather than practice. There is certainly room for supporting material on concepts and terminology but the real need is for a flexible CAL approach that is closer to the programming environments described in some research without the complexity of very sophisticated pieces of software. Anything that purports to help the naïve or beginning programmer must not impose too much extra learning upon them: an environment that is too complicated or detailed will begin to present the same kind of cognitive demands as the programming language itself!

The role of the tutor, and how she is perceived by the students, does not appear as a formally separated out factor in the work described above, but at an anecdotal level, teachers know that the perceptions of the relationship between teacher, learners and subject is a large element of the learning process.

Alexander and Bond (2001) note:

> "*A teacher's personal enthusiasm for subject can be transmitted through non-verbal behaviours such as eye contact with students, voice projection, body language and storytelling. Students can be stimulated by seeing and hearing a person talking about what excites him or her...*" (Alexander and Bond 2001 p 6)

The author would add to this the students' perceptions of the teacher's competence in that subject colour their responses to the presentation of the subject.

Other researchers have noted the student emphasis on tutor-generated materials. Linn and Dalbey (1989) discuss the results of studies undertaken for the ACCCEL Project (Assessing the Cognitive Consequences of Computer Environments for Learning) funded by the National Institute of Education. The studies looked at a number of schools (in the U.S.) where programming was taught, for at least 12 weeks, with sufficient classroom resources (8 computers) and experienced teachers. The studies looked at student ability, previous interest in computers and programming, gender, out of school access to computers and the quality of instruction at the school. Schools that explicitly taught design skills in their programming classes were flagged as exemplary.

Linn and Dalbey (1989) note that the untangling of various factors that affect a student's final measured performance in programming is not an easy task. They note, "*The major finding of this investigation is the influence of explicit instruction in the design of computer programs or the ultimate success of student.*" (Linn and Dalbey 1989 p 74) Interestingly, students were usually taught BASIC and the authors do note the drawbacks interest in the use of this language.

What is needed, Linn and Dalbey (1989) conclude is good curriculum materials, presumably for a more appropriate language their BASIC and experienced teachers of programming.

Are there models of teaching programming that might usefully inform a computer based approach to teaching programming, one that is more than the use of the compiler? Can an appropriate model transform the pedestrian approach of Introduction to Programming Ideas?

The author felt that she has explored two approaches well enough to be able to articulate their shortcomings, if not to detail them exhaustively. The next stage was to review the pedagogical model underpinning the approach to teaching programming.

One possible model was suggested by Linn and Dalbey (1989). Linn and Dalbey (1989) note, "*Because programming does involve solving problems, such instruction,* [in programming] *at least superficially, teaches about problem-solving.*" (Linn and Dalbey 1989 p 58)

Linn and Dalbey describe what they term an "*ideal chain of cognitive accomplishments*". (Linn and Dalbey 1989 p 58) They state, "*The chain has three main links: (a) single language features, (b) design skills, and (c) general problem solving skills.*" (Linn and Dalbey 1989 pp 58 - 59)

The first item in the list is the comprehension of particular language features such as loops. Students can be asked to develop or change programs to assess their understanding of the features used in those programs. Linn and Dalbey (1989) state:

> "*Students need to learn language features. However, such knowledge is of little general use or benefit. Students with an understanding of language features cannot compose programs that feature groups of commands working in concert.*" (Linn and Dalbey 1989 p 59)

In other words, students can alter, say, the number of loop iterations but would not know when or how to use appropriately that particular construct. This leads Linn and Dalbey (1989) to the next link of the chain, design skills. These skills enable students to put together a working program to solve a problem. "*Such skills are essential in order for students to write computer programs of any complexity.*" (Linn and Dalbey 1989 p 59)

Part of the set of design skills is the use of templates, sample programs that use more than one language feature: "*When students have a set of templates, they have a set of flexible and powerful techniques that allow them to solve many problems without inventing new code.*" (Linn and Dalbey 1989 p 60)

Linn and Dalbey (1989) also identify procedural skills, which include planning, testing and reformulating. Linn and Dalbey (1989) note, "*Planning is an important component of the behaviour of expert programmers.*" (Linn and Dalbey 1989 p 60) Students or novice programmers need an awareness of the importance of planning even if the tasks set do not require detailed planning. The same is true of testing. Expert programmers are generally skilled at testing and have well-developed debugging strategies (as well as strategies for trying to ensure bugs do not appear in the first place!).

In contrast, as the author can testify, novice programmers often do not test their code or test it only in the most obvious ways. Reformulating or rewriting code is another area in which the behaviour of novice and expert programmers can be contrasted. Linn and Dalbey (1989) note that novice programmers look for localized fixes for problems whereas expert programmers consider large changes to their code if appropriate.

This three level approach seemed promising but very difficult to model in simple CAL: it seemed to demand an intelligent tutoring approach. Since the author had wanted to keep the interventions as simple as possible, the idea of developing intelligent CAL was rejected.

Perhaps another, entirely different area that could be addressed by CAL is the idea of a concrete model of the computer. The Introduction to Programming Ideas does touch upon the operations of the computer but not in the way that would be most effective for students attempting to understand the relationship between their code and the machine upon which it runs or is supposed to run! Mayer (1989) explores how programming concepts can be integrated into existing knowledge, after being presented to the learner and held in short term memory. It is the *integration* of existing and new knowledge that is critical. Mayer (1989) suggests two approaches: "*...providing a familiar concrete model of the computer...*" and secondly, "*encouraging learners to put technical information into their own words...*" (Mayer 1989 p 131)

The use of concrete models may include the use of concrete objects such as bundles of sticks, used as manipulatives in mathematical operations. Mayer (1989) notes that related techniques such as the use of titles, short introductions (also called advance organizers) and concrete analogies (Ohm's Law described in terms of water flowing through a pipe). Mayer (1989) notes "*To date, advance organizers have been most effectively used in mathematics and science topics.*" (Mayer 1989 p135)

For programming a concrete model of the computer is effectively a 'glass box' approach: the programmer-user needs to understand what is going on inside the computer at a transaction level (not the circuit level!) Mayer (1989) describes work done on presenting concrete model to learners of BASIC. In later development of this study, Mayer (1989) concluded, "*...subjects given the model before learning showed* *evidence of more integrated and conceptual learning of technical information.*" (Mayer 1989 p 143) Mayer (1989) is also supportive of the positive effects of idea elaboration. Students who put technical ideas into their own words learn better.

Perhaps the tutorial CAL could be rewritten using Mayer's (1989) ideas about concrete models and advance organizers. This still seems a potentially useful way of revitalising some very basic CAL. However, the author decided not to revisit the CAL product any further. She still felt that the on-help approach should have been more successful and that it represented an approach that was genuinely responsive to the students' needs. What was lacking was a conceptual framework that would support and develop what the author felt to be a move towards a more constructivist way of supporting student learning.

Chapter 6: Conclusion

What can we say about what it means to teach novice programmers? That perhaps teaching is not a wholly appropriate view of complicated, even messy, process. Programming is more than learning syntax, more than knowing where the semi-colons go. It is easy to say this but less easy to identify what is most effective in giving a novice programmer a genuine understanding of the dynamic process of programming. Research has concentrated on various aspects of teaching programming from languages designed for new programmers to subtle, rich programming environments. What does creep into such technically-oriented research is the affective component but *how* students feel about programming is rarely explored.

Perhaps a rich return awaits the researcher who can begin to untangle the affective component of good programming practice. This issue is not likely to be addressed by standard CAL approaches: perhaps intelligent programming software might act as both source of programming understanding for a new programmer and as a source of positive encouragement!

What relevance does CAL have for the teaching of programming? The author believes it has shown itself to be, in a standard tutorial form, of limited use. Paradoxically, more traditional, paper-based materials appear to be extremely popular among beginning programmers, and that popularity seems to be unaffected by whether these are distributed by the tutor or made permanently available over the Department's server. It seems almost perverse to conclude, after investigating ways in which the teaching of programming can be supported by computer, that the best approach seems to be a blend of well-established resources (such as sample programs) and tutor help, week by week. Nevertheless, that is the author's deeply held conviction: there is no programming environment, textbook or CAL that can wholly replace, for the novice programmer, the knowledgeable tutor's focused and timely help during the creation and debugging of code.

However, it is not sufficient for those who engage with these issues to simply throw up their hands and declare that traditional methods are the favourite of students and therefore tutors need do no more than produce large numbers of lab sheets and sample programs. The onus is on professional practitioners to seek to embody in their teaching practice fresh ideas. With this in mind, the next chapter looks at the development of a pattern community among novice programmers, in the academic year 2003 - 2004, under the guidance of the author.

Introduction to Chapter 7

This chapter describes the work done by the author around using the idea of patterns in the teaching of programming to novice programmers. The concept of patterns and pattern communities is reviewed and the author's design and use of a pattern format is described. The research design is described for this latest phase. This chapter also examines some of the issues around the evaluation or grading of student code. The chapter also looks at the results obtained by the author, examining the ways in which the artefact was used, or not, by students. The chapter presents some initial evaluations of the qualitative methods used by the author, and describes the problems encountered by the author. A review of the whole set of research cycles is presented in the concluding chapter.

The concept of patterns

The idea of patterns has been described earlier in this work in more detail in Chapter 2) but it is useful to review the key points, to explain why the author chose the idea of patterns rather than a different approach for another cycle of her work on the teaching of programming.

The concept of patterns spring from the work of Christopher Alexander, an architect. Alexander (1979) identified and described what he felt were critical aspects of the art of creating living and working environments. One of the key aspects is the idea of the nameless, yet instantly known and responded to, quality. This concept of the nameless quality is central to Alexander's work: it is what causes a deep and life-affirming response in those who see it, whether it is in a building, a garden or a village. Patterns are a way of making an approach to embodying that nameless quality. Alexander offers patterns to be found in practical aspects of life such as the building of barns. Alexander's vision of patterns is the capturing of an essential rightness in the doing of some real world task, such as laying out a town or planning a garden. Patterns are the opposite of mechanical, dead frameworks.

The idea of patterns was taken up by software practitioners, those who had an interest in theories of software development as well as experience in coding. Ward and Kent are generally credited with being responsible for introducing patterns to software practice and development:

> "*In the software community, Ward Cunningham has a reputation for being a font of ideas.* [] *He invented the world's first wiki, a web-based collaborative writing tool, to facilitate the discovery and documentation of software patterns. Most recently, Cunningham is credited with being the primary inspiration behind many of the techniques of Extreme Programming.*" (http://www.artima.com/intv/simplest.html 30/03/2004)

Kent describes his discovery of patterns:

> "*I first discovered patterns as an undergraduate at the University of Oregon. Many of the students in my freshman dorm* [] *were in the School of Architecture. Since I had been drawing goofy house plans since I was six or seven, they pointed me in the direction of Christopher Alexander. I read all of The Timeless Way of Building standing up in the university bookstore over the course of several months.*" (http://c2.com/ppr/about/author/kent.html 30/03/2004)

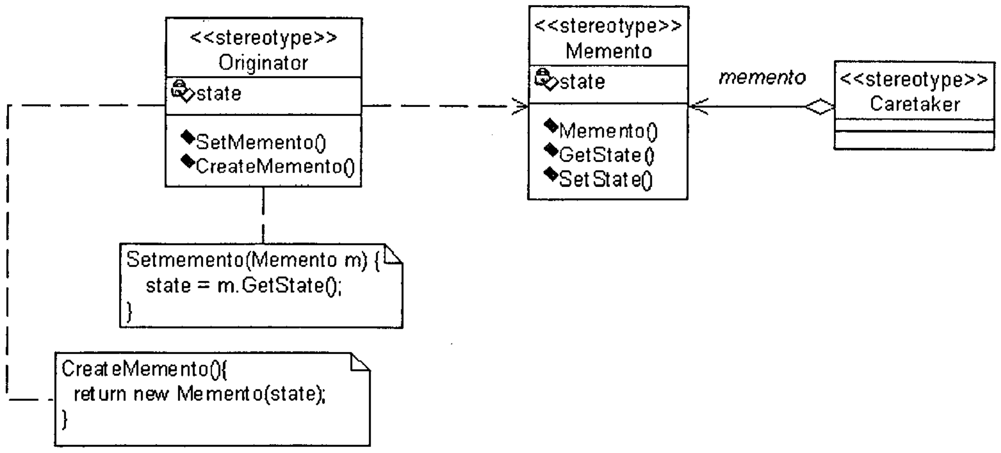
Ward and Kent applied the idea of a pattern language to a problem they were having designing a user interface with a group:

> "*In 1987,* [Ward and Kent] *were consulting with Tektronix's* [Semiconductor Test Systems Group] *that was having troubles finishing a design. They decided to try out the pattern stuff they'd been studying.* [] *Ward came up with a five pattern "language" that helped the novice designers take advantage of Smalltalk's strengths and avoid its weaknesses:* [] *Ward and Kent were amazed at the (admittedly spartan) elegance of the interface their users designed. They reported the results of this experiment at OOPSLA 87 in Orlando.*" (http://c2.com/cgi-bin/wiki?HistoryOfPatterns 13/08/2003)

From this fairly small-scale beginning, the idea of patterns was adopted by more experienced software developers. The first PLoP (Pattern Languages in Programming) gathering was held in 1994.
(http://pages.cpsc.ucalgary.ca/~kremer/patterns/history.html 30/03/2004)

> "*The first major compendium of patterns between two covers, ``Design Patterns: Elements of Reusable Object-Oriented Software" made it out in time for OOPSLA '94. It sold 750 copies at the conference--more than seven times the highest number of any technical book Addison-Wesley had ever sold at a conference.*" (http://c2.com/cgi/wiki?HistoryOfPatterns 30/03/2004)

Patterns for experienced software developers tend to be more complex and specialised, such as Memento. For the Memento pattern, the diagram below is generally offered as an introduction. (http://unicoi.kennesaw.edu/~jbamford/csis4650/uml/GoF_Patterns/memento.htm 30/03/2004, http://www.tml.hut.fi/~pnr/Tik-76.278/gof/html/Memento.html 30/03/2004 and http://www.dofactory.com/patterns/PatternMemento.aspx 30/03/2004):



(http://unicoi.kennesaw.edu/~jbamford/csis4650/uml/GoF_Patterns/memento.htm 30/04/2004)

This is a detailed and meaningful pattern format to a knowledgeable object-oriented programmer but for a novice programmer, it is obviously neither helpful nor appropriate. It is not just the complexity of the ideas embodied in the pattern: it is the format of the pattern also. The idea of the simplicity that is the essence of a pattern seems to be buried under the complexity of the concept to be conveyed.

In the author's view, patterns at this level and in this style are obviously appropriate for experienced programmers familiar with the concept of patterns but they do not embody what she felt to be the essence of Alexander's vision for patterns, their use and purpose. The patterns were impressive but did they strive to embody the nameless quality?

Patterns are increasingly to be found in general texts. In April 2004, Wiley published another volume in the Pattern-Oriented Software Architecture series, a book that focuses on the use of design patterns for techniques in implementing system resource management. Other texts examine patterns for object-oriented modelling and patterns for concurrent and networked objects.

Patterns can be light hearted: Pizza Inversion - A Pattern for Efficient Resource Consumption (http:///www.cmcrossroads.com/bradapp/docs/pizza-inv.html 18/03/2004) describes a pattern for the eating of a too-hot slice of pizza!

Whether the more complex patterns and patterns languages could be considered truly Alexanderian or not, these kinds of patterns were too complex for novice programmers. Pedagogical patterns such as those proposed by Bergin (http://csis.pace.edu/~bergin/papers/SimpleDesignPatterns.html. 07/08/2003) tended to be simpler in their structure and more immediately accessible. It is to these simpler patterns that the author looked initially.

While searching for material on the teaching of programming, the author found several websites dedicated to patterns and pattern languages. Some websites were for pattern languages in software development: others were for patterns in pedagogy. It seemed obvious to the author that sitting neatly between these two were pattern languages concerned with the supporting the learning of programming by novice programmers.

There must be either a pattern language concerned solely with that area of pedagogy, or pattern languages that supported the learning of a particular language, not for experienced programmers but for those new to programming. Sophisticated pattern languages for experienced programmers were easily found but were obviously aimed at very knowledgeable programmers, such as the Memento pattern, as referred to above.
(http://unicoi.kennesaw.edu/~jbamford/csis4650/uml/GoF_Patterns/memento.htm 30/04/2004)

The author could not find an example of a pattern language that was simple and aimed at novice programmers. It is possible that such languages exist but the author has not managed to find them or they are not yet in a public domain.

Not all pattern language creators skim over the idea of the nameless quality:

> "*The Hillside Group is founded on the observation that software creation is one of the most difficult human endeavors, requiring the creation of novelty under pressure, without the benefits of a long tradition to fall back on.* [] *The world of software development is a mixture of concerns ranging from correctness and execution efficiency to the beauty and elegance of the architecture, design, and internal structure of systems to the overall aesthetics, usability, and humanity of systems and all the way to the organization of development and the manner of software production.*" (http://hillside.net/vision.html 18/03/2004)

The mission statement of the Hillside group is perhaps somewhat unusual in that it acknowledges an artistic, even humanistic, element in software development.

Other pattern creators not only specifically acknowledge the nameless quality but make it central to the effort. One website defines how the quality would be made extant in software:

> "*...its modules and abstractions are not too big* [] *every module, function, class, and abstraction is small and named so I know what it is without looking at its implementation* [] *...if it was small, it was written by an extraordinary person, someone I would like as a friend; if it was large, it was not designed by one person, but over time in a slow,careful, incremental way* [] *...it is like a fractal, in which every level of detail is as locally coherent and as well thought-out as any other level.*"

(http://www.dreamsongs.com/NewFiles/AlexanderPresentation.pdf 18/03/2004)

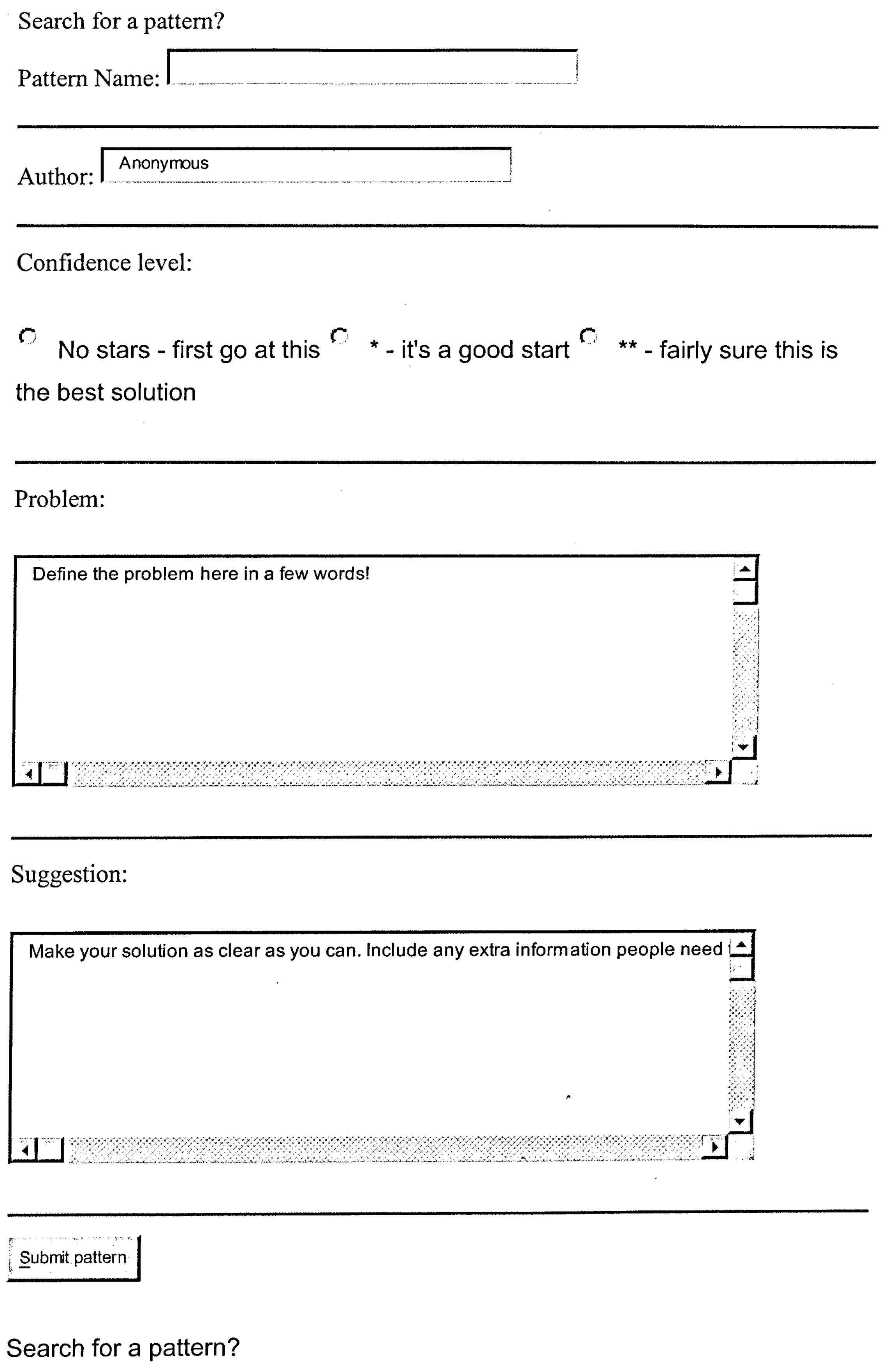
The ideas here are interesting because the writer has concentrated on the *purpose* of patterns, which is to foster the growth or creation of a genuinely meaningful product that embody the nameless quality. The writer focuses on what a piece of software that possesses that quality would be like, what its properties would be. There are some potentially fruitful ideas that could come out of this that the author has not had time to explore, such as introducing the concepts of the nameless quality to novice programmers and focusing on a top down implementation of the idea: producing software that embodies that nameless quality, by whatever means seem appropriate.

It seemed to the author that there were two choices open to her. She could create a pattern language based on the students' work; in some way, or she could pass control of the patterns to the students. Given the choice of writing a pattern language to support the teaching of programming or setting up a facility to allow students to develop a pattern language for the learning of Pascal, the author chose the latter.

The author felt it would be a comparatively trivial task (if an enjoyable one) to write, say, half a dozen patterns, around issues or concepts that regularly appear (in the author's experience) to be difficult for new programmers like the naming of files, saving the .obj file, the use of semi colons, or the problem with spaces in program and variable names. However, the author has already described how she wished to engage with her teaching in a more constructivist way and presenting students with a set of patterns seemed to subvert that goal. The author felt that if she wished in a genuinely constructivist way to engage her students in this work, then she had to present them with the opportunity to create the pattern language, rather than giving them a finished product. She therefore chose an 'empty vessel' approach.

The author's research design

Having decided on the development of a small-scale pattern language, the author chose to develop a minimal structure, as below. The first box was for pattern name, without a hint or prompt. (It probably would have been better to include a suggestion for pattern names.) The student was prompted to enter his or her name but could submit anonymously. The pattern had an associated confidence level. The problem and solution had separate boxes with an explanatory message in each one. Students then clicked on the Submit button to add their pattern to the pattern language. Students could also search previously submitted patterns using a key word to pull up relevant patterns.



Search for a pattern?

Pattern Name:

Author: Anonymous

Confidence level:

○ No stars - first go at this ○ * - it's a good start ○ ** - fairly sure this is the best solution

Problem:

Define the problem here in a few words!

Suggestion:

Make your solution as clear as you can. Include any extra information people need

Submit pattern

Search for a pattern?

(http://staff.dc.lincoln.ac.uk/~ambird/patterns 30/03/2004)

In the author's view, novice programmers who would be given access to the structure would not want to spend time and effort in creating a complex pattern. The idea of this minimally constituted pattern language was to develop a resource that answered the needs of those creating it. Having decided on a minimal structure, the author asked for help from the Department of Applied Computing's System Administrator who coded the form in Perl. This task took less than a couple of hours from giving the initial brief to the System Administrator to uploading the form on to the author's departmental webpage and making it available to students The first version of the form had no search facilities but this was added shortly after, at the author's instigation

The author's data gathering approaches

Having created the pattern template, the next stage was to look at ways to gather feedback about the usefulness of the pattern template. In Chapter 2, the author has described her focus in this research as qualitative, rather than quantitative, focusing on student perceptions of their learning experience. The question was, what instruments were most suited to gathering the kind of data the author felt was appropriate and meaningful in this context?

In previous efforts, the author had relied on use of questionnaires, which had proved less than satisfactory without other forms of data gathering to supplement them. The author decided to use a short version of the entry questionnaire that asked students to describe their programming background at the start of the unit, Introduction to Programming. The questionnaire asked 5 brief questions on a single side of A4. (See Appendix N) The author also decided to use a very brief exit questionnaire with just two questions. (See Appendix N) To give a more detailed picture over the life of the unit (12 weeks), the author decided to use observation (of a programming class), a focus group and unstructured interviews. The author hoped that by using various qualitative methods she would harvest a richer set of data than previously.

Each of these tools (questionnaires, focus group, observation and unstructured interviews) has its advantages and disadvantages. The author feels that it is incumbent upon her, as a researcher, to defend her choices of methods and to acknowledge alternative ways of approaching the task or tasks.

Questionnaires

Breakwell et al. (1995) note, "*the humble questionnaire is probably the single most common research tool in the social sciences.*" (Breakwell et al. 1995 p 174) Opie (Ed) (2004) notes the advantages of a questionnaire:

> "*It is relatively economical, respondents in distant locations can be reached, the questions are standardised, anonymity can be assured, and questions can be written for specific purposes.*" (Opie (Ed) 2004 p 95)

Questionnaires can be used for all kinds of purposes, from hypothesis generating, test development, population parameter estimation to hypothesis testing. However, as Opie (Ed) (2004) notes, designing a good questionnaire is not easy. These two issues, of purpose and design, have engaged the author a number of times. What is the rationale for using a questionnaire? The author would argue that the purpose of the questionnaire has changed over several incarnations of the instrument. Earlier versions used a Likert scale of 1 to 5 for statements about learning. For the exit questionnaire, in previous years, students have been asked to rate the importance of various resources and to rank them in order of use or significance.

Later versions dropped this in favour of completely open-ended questions about programming, together with a request for brief, factual information about the student's programming background and experience. As the questionnaire changed, it became perhaps less a questionnaire than a simple way of collecting on paper very basic information about the students and a way of trying to capture some of the students' attitudes to or thoughts on, learning to program.

In a sense then the questionnaire was not so much designed as grown organically. This is not, the author hastens to add, perceived by her as a strength! The use of a questionnaire format needs to be much more carefully thought through. What elements would be most usefully explored through the gathering of information in a questionnaire format? Some of the attitude exploration questions might have been usefully developed, looking at student attitudes to learning generally and to learning programming. What would be perhaps revealing over several iterations of the questionnaire's use would be to use a quantitative tool such as the Kruskal Wallis test, which is used to compare three or more independently sampled sets of data. Such a test might reveal whether three different student cohorts all exhibit similar attitudes to programming and learning. This would be a detailed, long-term undertaking and was rejected by the author as not manageable in the time available to her. However, for another cycle of the research, the author feels that some quantitative data should be more formally collected and processed.

The design of the questionnaire questions is also an area that needs some discussion. The questions in the versions used this cycle cannot be said to have needed designing as such. However, a more detailed or longer questionnaire, such as one that uses a Likert scale (or other measurements), does need careful question design. Here the questionnaire writers need to address issue such as the choice between open ended and closed ended questions.

The author adopted open-ended question format, due to the nature of the data she sought to gather. The exit questionnaire for January 2004 asked only two questions, designed to be as open ended as possible: What has been the hardest/most difficult aspect of learning to program? and What resource (book, friends, web page…) or strategy has been the most useful in learning to program? The author feels that these questions, in her own view, are written to invite a detailed and personal response. In practice, most students replied very briefly. (Questionnaire responses are discussed later in this chapter.)

Closed ended question are generally used wherever possible responses can be listed in advance. An example of this might be to ask students how often they attended lectures, with five possible replies ranging from 'Attend all lectures' to 'Attend only 1 or fewer a week'.

Breakwell et al. (1995) note that closed ended questions "*reduce the number of coding errors in the data set*." (Breakwell et al. 1995 p 177) The possibility of misinterpreting responses to open ended questions is an issue that needs to be acknowledged when working with qualitative data. However, the author felt that open-ended questions were most appropriate to the type of data she wished to collect.

As Breakwell et al. (1995) noted, "*The disadvantages of closed-ended formats are of many sorts but perhaps the most important is that they can create artificial forced choices and rule out unexpected responses*." (Breakwell et al. 1995 p 178) Since it was the unexpected or insightful response that the author hoped for, closed-ended questions were not used!

The author does not rule out using closed-ended questions in future work. Previously, she has used a Likert scale gather response to statements about learning and a ranking of resources question. The author feels it may be useful to revisit some of these at a later date, at the end of the academic year, with the cohorts who filled in the questionnaires in semester A.

With questionnaires about behaviours, the researcher needs to be aware that people do not always report accurately. They may under-report what they perceive as undesirable behaviours. Breakwell et al. (1995) also note a caveat about reporting of other behaviours by respondents: "*…studies suggest that over-reporting biases apply to socially desirable behaviours too…*" (Breakwell et al. 1995 p 185) So, if the author is asking about learning or study behaviours, students may report themselves as more diligent or knowledgeable than they are. This is a difficult issue to resolve. Ethically, the author feels uncomfortable about making the questionnaires traceable but to do that would enable the author to look at the self-reported behaviours of specific students against their final unit grade and their overall academic profile.

Focus groups

The author decided to use the idea of a focus group. Breakwell et al. (1995) describe the focus group as "a *discussion-based interview that produces a particular type of* *qualitative data.*" (Breakwell et al.1995 p 275) Breakwell et al. note that the focus group is used chiefly in marketing and that much of the work on using focus groups is centred on marketing issues. Social science and psychology have employed this approach to gathering qualitative data but Breakwell et al. (1995) sound a note of warning about the use of focus groups in psychological research: it is a fairly new tool and needs further work: "*It is clear, nonetheless, that the future of focus group research in psychology will depend on how rigorously it is conducted.*" (Breakwell et al. 1995 p 276)

As the name suggests, focus groups are centred on a specific topic. What makes the focus group different is its social nature: the interplay of the participants. "*The assumptions of focus groups is that people will become more aware of their own perspective when confronted with active disagreement and be prompted to analyse their views more intensely...*" (Breakwell et al. 1995 p 277) In using focus groups the author hoped that the discussion among the students would reveal attitudes to the learning of programming not previously articulated by the participants. The author envisaged a chronological sequence of one or more groups. She was hopeful of gaining a number of volunteers, given the cohort numbers on the first year. The reality proved to be very different. The focus group that did take place is discussed later in this chapter.

Observation

The other tool that the author hoped to use was observation of several programming laboratory sessions. The author felt this was important. However, 'observation' is rather a broad term and requires some clarification. Breakwell et al. (1995) note that "*the two possibly most influential trends come from very different theoretical perspectives: experimental psychology and ethology.*" (Breakwell et al. 1995 p 215) Systematic observation, as the name suggests, requires the observer to measure instances of previously categorised behaviours. This usually involves discrete measurements of time, frequency and occurrence.

Another approach is that of ethology. "*The ethological approach is characterised by particular method of direct observation which aims to record...behaviour completely impartially...*" (Breakwell et al. 1995 p 215) This approach does not attempt to infer or examine the motives or emotions of those observed.

A methodology that employs observation, in a number of ways is ethnography. Using observation as an ethnographical tool focuses on the describing of a culture. Breakwell et al. (1995) state, "*Ethnographic research often starts with observation and description, for it is in the process of observing that situation-specific questions emerge.*" (Breakwell et al. 1995 p 303)

The observation that the author undertook is closest in spirit to ethology. In observing the class, the author did not attempt to describe or explore the personal feelings and motivations of the students in the class. That element seems to properly belong in focus groups or unstructured interviews, in the author's view.

Opie (ed.) (2004) notes the advantages and disadvantages of observation. For the author, the two key ideas are "*The 'observer', unlike participants can 'see the familiar as strange'.* [] *Data collected can be a useful check on, or supplement to that obtained by other means.*" (Opie (ed.) 2004 p 122)

In observing laboratory sessions the author hoped to gain some insight into how the students engaged with programming tasks and activities. The author felt that observation might help her to understand better the questionnaire responses from that particular group of students. As a teacher, the author was aware that a great deal goes on in any classroom situation that even the best teacher is not cognisant of, for a variety of reasons. She felt it would be revealing to be the one 'on the sidelines' rather than the member of staff conducting the class. The author felt it would be useful to see how a colleague teaching the same unit approached the laboratory session.

For practical reasons the author decide not to video or record in any way the laboratory session observed by her. She felt it would be intrusive and undermine the purpose of the exercise, which was to observe the students as discreetly as possible. As Opie (2004) (ed.) notes, "*People, consciously or unconsciously, may change the way they behave when being observed.*" (Opie 2004 (ed) p 122) The author felt that videoing the session was most likely to affect the way the students behaved and rejected the idea because of this. Professional courtesy also dictated this choice: it is one thing to observe a colleague in an informal way, quite another to capture his or her work on tape.

In setting up an observation, a researcher must decide what stance he or she will take with regard to participating in the proceedings. (Opie 2004) The author decided to take a non-participatory approach to the observation.

Opie (2004) (ed.) lists a set of categories for observing classroom activities, including personal traits, verbal interaction, non-verbal signals, affective elements and cognitive aspects. (Wragg 1999 in Opie 2004 (ed.) p 125) One approach to structuring the observation is to note elements under the various categories. The most common category system is FIAC (Flanders Interaction Analysis Categories) which records the type of activity every 3 seconds. For example, Teacher praises or Asks questions, Student responds to teacher, student is silent. Each category has a code number and the observer records the code number, to give a pattern of classroom activity and interaction.

The author decided not to use FIAC on her firs observation but to use it subsequently, in later observations. This proved to be an over optimistic estimate of how much observing she would be able to do. The author strongly feels this instrument would be very useful particularly in observing programming students over a period of time. Use of FIAC might go some way to answering questions about whether novice programmers become more confident in their questions and responses to in a laboratory situation. It would also be useful to examine the topics asked about by students and the type of questions asked, as their programming knowledge (hopefully) grows. This type of observation work needs to be done over a whole semester, with a group that begins programming fairly early on.

Observation of classes encountered the same sort of problems as the focus group and questionnaires did. These problems are discussed later in this chapter.

The combination of these activities was designed to focus on qualitative data and enable the author to have some understanding of the various viewpoints that can have an impact on the teaching and learning of programming. "*The combination of evidence from the various methodologies should show where* [the evidence's] *limitations lie and point to an appropriate revision.*" (Breakwell 1995 p 15)

With the various tools selected, the author could then turn her attention to carrying out the relevant activities. The environment within which the research was to take place needs some description.

The institutional context for the author's research

In the summer of 2003, the Department of Computing had, along with other all the other departments on the site, moved out of the Cottingham Road campus, which was a few miles out of the city centre. The University moved some administrative functions entirely to Lincoln, leaving only a skeleton service in the Hull city centre. The University was now entirely located in Hull centre, in several buildings. The main building was in George Street a 1960s seven-storey tower block. The University hasd spent a considerable amount of money to refurbish the building and when staff moved in, in late July, substantial building work was still going on, and was finished only in late September.

The move was not a particularly popular one among staff and students. First year students who had been shown round the leafy and spacious Cottingham Road campus on open days now found themselves in the centre of.Hull, without a separate Students' Union building, no Union shop and no bar. Second and third year students were vocal in their dislike of the move.

For the Department of Computing, the strong recruitment to the Games degree in Lincoln meant that there were now two large cohorts in Lincoln, when all the computing staff were based in Hull. The imbalance in first year numbers was startling. As of March 2004, there were 139 students registered on the Introduction to Programming unit at the Lincoln Campus, all taking the Games Computing (Software Development) route. In Hull, there were 22 students registered on other BSc computing routes, taking the Introduction to Programming unit.

This division of student numbers between the Hull and Lincoln campuses generated organisational problems, as it has done over the past two years. Most of the Department's staff at the time of delivery of the unit in (September to January 2003 – 2004) were based in Hull. Staff were then required to split their teaching time between Hull and Lincoln. In previous years, the author has taught the Introduction to Programming unit in Hull, with a colleague taking the teaching of the unit in Lincoln. This has worked reasonably well, with some initial problems about the lack of resources in Lincoln being resolved (2002 – 2003).

However, in 2003 – 2004, the author was required to teach final year units in software development, to cover a colleague's absence on maternity leave, and the Introduction to Programming unit was handed over to another colleague. This raised two key problems for the author. Firstly, she would have no say over what was taught or in what order and secondly, she would not be able to build a relationship with those first year students. The author hoped that these problems would not be too deleterious in their effect on the latest phase of the research but as can be seen from her results, these two issues represented considerable handicaps to the success of the project. Because of the numbers of students taking the unit in Lincoln, other members of staff took some of the seminar groups. This generates another set of problems in itself.

Tutors can vary widely in how they interpret and deliver the unit co-ordinator's brief. Different staff may be less knowledgeable than others in the areas being taught. Students may perceive (with or without a basis in fact) some members of staff as being less academically credible than others. Something as simple as the time scheduled for a laboratory session can make a significant difference to the numbers of students attending.

All of these factors and others perhaps the author was not aware of make the collecting of data from these students problematic. An example of this is the initial information the author sought to collect. The member of staff delivering the lecture sessions in Lincoln, for introduction to Programming agreed to distribute the questionnaires but returned only a handful of replies. The students appeared to be very reluctant to respond. If the author had been delivering the lectures she would have not allowed the students to take the paper away but would have required them to return the material at the end other lecture, something the other member of staff did not do.

Each campus had its own problems: in Lincoln, the high student numbers were a significant issue for the Department, as were the questions of staffing and resources. In Hull, student numbers were much smaller so resources such as computers were more than adequate but the building's general facilities were limited. The author took the view that the George Street building, despite the money spent upon it (several million pounds) felt like an annexe to the 'real' University in Lincoln, with a bustling campus, and range of student facilities. After two weeks of the new term, the Faculty's management called a meeting, in which they announced that all the Department of Computing staff would be based in Lincoln from the summer of 2004. This created added difficulties for the author who could not transfer to Lincoln nor commute to teach there, leaving her with no option but to look for another post. The stress and uncertainty of this future move did not help the author in her efforts to engage with the latest cycle of her research.

In summary then there were a number of problems that the author faced in trying to gather data on the development of a pattern community. She had no control over, or influence on, the way the Introduction to Programming unit was taught. She had to rely on colleagues to administer the initial questionnaire to the Lincoln cohort. Students in both Hull and Lincoln were notably reluctant to engage with the author, with the exception of three students in Lincoln (out of student group of over a hundred!). Nonetheless, the author decided to persevere.

Evaluating student programs – an overview

The final element that needs to be discussed is the question of evaluating students' programming efforts. This is not a trivial question: it goes right to the heart of the research. When the author says she is looking for ways to support novice programmers, the implication is that she is looking for a way or ways to help them be better programmers. By extension, better programmers produce better programs. The issue here is, what do we mean by 'better'? Better than last year's cohort? Better than the average novice programmer? Better than they might have done without the intervention offered?

It is obvious very early on that some of the questions are unanswerable or, if an answer can be formulated, it is not meaningful. To compare cohorts is a very crude measure. Student intakes can vary across years. The games degree, when it recruited for the start of the academic year 2003, was so popular that the points count for applicants was raised, making the first years for 2003 – 2004 potentially very different in terms of academic profile from the year above them in Lincoln and the two years above them in Hull.

Programmer profiles

What do we mean by the average novice programmer? How do we define or measure 'average'? We could specify an A level points count or even an IQ score. We could administer some variety of learning style test or inventory. We could specify perhaps a typical personality profile for a programmer. Anecdotally, brilliant programmers tend to be obsessive, focused and poor at social skills. Yourdon (1976) notes

> "*many of the established superprogrammers are freaks in some sense of the word: they look funny; they wear funny clothes; they refuse to work regular hours; they don't get along with normal people; and so forth.*"
> (http://www.yourdon.com/articles/7602Infosystems.html 06/04/2004)

Conversely, this view of a typical programmer is not supported by some research into personality types: "*There is no real evidence to suggest that programming ability is related to any particular personality trait.* [] *it is probably impossible to identify* *programming personalities...*" (http://www.soften.ktu.lt/jep-06032/city/courses/IFPR402/human.html 06/04/2004)

With regard to the concept of emotional intelligence, one website claims

> "*those programmers who are in the top 10% of EQ on the following scales; willingness to collaborate, disinclination to compete, sharing information out-produce the average programmer by 320%. Those few programmers who are 'star performers' outperform them by 1272%.*" (http:// www.learninglinks.org/emotional_intelligence.pdf 06/04/2004)

Such claims seem a little hard to credit but perhaps the evidence really is as contradictory as it seems. There is no provable link between personality and competency on programming but there is a link between personality and attitudes and excellence in programming. This is a very complicated question and one that the author cannot even begin to wholly define, let alone explore in the space available here.

Software metrics

The question is also, what kind of product does the average programmer produce? How good is the code produced by the average programmer? Here we would have to unpack still further the questions. What do we mean by good? How many lines of code does the average programmer produce per day, week, month and how many bugs does that code contain? How is it commented? How is it tested? How well does it meet the design or specification? How well does it adhere to the standards or guidelines for that piece of code?

Some of these questions are specific to a particular project, piece of code or language. Can we agree that there are ways to classify whether a piece of software is 'good', whatever language it is written in and for whatever purpose?

The earliest measures focused on size of the program but this approach does not measure program complexity. A twenty-line program could be densely constructed and much harder to maintain than a hundred line programs of simple statements with no loops or branches.

There have been efforts over the past twenty years to develop and refine the broad measures used since the 1960s, such as LOC (Lines of Code):

> "*The obvious drawbacks of using such a crude measure as LOC as a surrogate measure for such different notions of program size such as effort, functionality, and complexity, were recognised in the mid-1970's. The need for more discriminating measures became especially urgent with the increasing diversity of programming languages. After all, a LOC in an assembly language is not comparable in effort, functionality, or complexity to a LOC in a high-level language.*"
> (http://www.dcs.qmw.ac.uk/~norman/papers/new_directions_metrics/HelpFile History_of_software_metrics_as_a.htm 06/04/2004)

McConnell (1993) lists some metrics for consideration. Metrics noted come under three headings: Size, Productivity and Defect Tracking. Under Size, McConnell (1993) lists Total lines of code, Total comment lines, Total data declarations and Total blank lines. Productivity metrics include Work-hours spent on project, Work-hours spent on each routine, dollars spent per line of code and dollars spent per defect. Under Defect Tracking, McConnell (1993) includes Severity of each defect, Number of lines affected by each defect correction, Number of attempts made to correct each defect and Number of new errors resulting from defect correction. (McConnell 1993 p 545)

Galin (2004) dedicates a whole chapter to software quality metrics and divides metrics into three classes: software process metrics, software process timetable metrics and software process productivity metrics. For software process quality metrics, Galin (2004) list three types: Error density metrics, Error severity metrics and Error removal effectiveness metrics. For software process timetable metrics, Galin (2004) lists TTO (Time Table Observance) and ADMC (Average Delay of Milestone Completion) as potential metrics. (Galin 2004 pp 416 – 421)

Criterion referencing

We can see that measuring software quality is not a trivial or always easy task. So, are there ways to assess what makes a piece of student code 'good'? We can assume that we cannot expect to apply complicated metrics to a novice programmer's first assignment, which might be a fairly basic task. In the past, the author has asked students to design first in pseudocode and then code a small program to calculate the cost of building a kit house or a swimming pool. The author has marked all the student efforts so there must be some template or approach for marking the early efforts of novice programmers, held in the author's head! Marking consistently is an issue that has been addressed over the last two years in the Department by the introduction of criterion referencing grids. The Department of Computing adopted criterion referencing as the only assessment in the curricula it delivered. All year 1 assignments had an associated criterion referencing grid created for them by the member of staff responsible for the assignment/unit in the academic year 2001 – 2002.

The following year all year 2 units also had criterion referenced assignments and in the subsequent year, all third year assignments would be criterion referenced. By 203-2004, all assignments set by staff in the Department would have a criterion reference grid that would be available to staff, students and external examiners. In fact, the set of learning outcomes for each unit was substantially reduced by the start of the academic year 2003 – 2004, as the University management felt that the five or six learning outcomes previously used for each unit were making the criterion referencing system unwieldy. Considerable effort had to go into redesigning the grids for well-established assignments.

The grid written and used by the author for the Introduction to Programming unit in 2002 – 2003 (using six learning outcomes) is included in Appendix L. Use of the grid went some way to making transparent the areas of interest to the author-as-marker, for students, colleagues and external examiners. Nonetheless, the author found the grid to be sometimes restricting. Work that met the criteria was felt to be in reality not worthy of the mark band in which it was placed and vice versa. What this suggests to the author is that assessing student programs is very much a qualitative activity, with a sense of 'This is good, solid code' that is reflected in a mark, rather than a ticking of points on a grid or checklist.

In reviewing the feedback given by the author to students in previous years, some points do emerge. Students submit code that does not compile. Sometimes this is because the program is hopelessly flawed and contains multiple errors: on other occasions the student has added a last minute comment without a closing parenthesis which means the program will not compile. Students frequently accept inaccurate output from their code without question: the classic example is calculating VAT. Students who could work out the VAT on, say, a hundred pounds, at least on paper will not check their code with simple values. Students struggle with variable names and the use of data types. The anecdotal list could go on and on. The author adopted the strategy of giving general feedback to her first year students after she had marked their initial assignment, a simple one-page summary of key points such as:

- It seems not many people actually tested their program against pen and paper calculation of their own. Why? It seems an obvious way to test what your program does.
- Comments also needed work
- Too many variables - re-use variables and overwrite values rather than setting up variables such as sal1, sal2, sal3, sal4 and so on.
- People are still handing in programs that do not compile. This worries me! If it is your program why can't you get it to compile? And if it isn't why are you handing it in as yours? (Feedback sheet written by the author October 2001)

It should be noted that this seemed to have no effect on the quality of their next piece of work!

Automated grading of student programs

So, if the author finds marking or grading student work a difficult process to fully articulate, have others found more accessible strategies? One approach that has been used fairly thoroughly twice, as far as the author can ascertain is automated marking of student code. The earliest example that the author found was the Ceilidh (Computer Environment for Interactive learning in Diverse Habitats) system of marking C code. The Ceilidh system was developed at the University of Nottingham in the mid 1990s.

The author can remember the creators of Ceilidh demonstrating their work at a conference in the mid 1990s. The authors of Ceilidh had plans to disseminate it widely but as with many innovative pieces of software in education it did not seem to translate well into use by other institutions. The author assumed that the Ceilidh system had effectively become extinct but interestingly, the approach appears to have been revived in the last two years.

Ceilidh was developed to mainly assist in the marking of coding assignments. Ceilidh also offered, according to http://www.doc.ic.ac.uk/~nd/surprise_95/journal/vol2/hst/article2.html support for the administration of courses with facilities for monitoring a student's marks and late or missing submission of work. Ceilidh also provided tutorial material, program skeletons and an editor/compiler. The core of the system was the automated marking of code, based on five main categories: dynamic correctness, dynamic efficiency, typographical analysis, complexity analysis and structural weakness. It should be noted that analysing program style was not, even in the early 1990s, a new idea: Redish and Smyth (1986) describe the creation of two style analysis tools for FORTRAN 77: AUTOMARK and ASSESS. (Redish and Smyth 1986)

In Ceilidh, dynamic correctness was tested by running to previously created test data with the student code. Dynamic efficiency checked for unused or inefficient code. Typographical analysis relied on measuring elements such as number of blank lines and comments. Complexity analysis counted the number of occurrences of assignment statements, loops, branches and function calls. Structural weakness is measured by counting instances such as variables declared but not used or value returned by a module but not used. The article at http://www.doc.ic.ac.uk/~nd/surprise_95/journal/vol2/hst/article2.html notes that "*The overall mark is calculated as weighted average of the marks under these five headings, the weights being specified by the teacher.*" (http://www.doc.ic.ac.uk/~nd/surprise_95/journal/vol2/hst/article2.html 31/03/2004)

The authors felt that automated assessment proved to be a fruitful approach. More specifically, students received immediate feedback and were more accepting of the software's judgment of their software. Also, the marking process was rule based and consistently applied. Ceilidh was revived after some years' lapse and it was noted that some of the elements of Ceilidh such as tutorial material could be more appropriately disseminated via web pages. (http://www.cs.nott.ac.uk/~smx/PGCHE/ceilidh.html 31/03/2004)

However, elements such as plagiarism tester and the automated test runs were extracted and put to use. The author of the article found at http://www.cs.nott.ac.uk/~smx/PGCHE/ceilidh.html notes that "*Anecdotal evidence from other courses suggests that students object strongly to having their programs marked in an entirely automated fashion.*" (http://www.cs.nott.ac.uk/~smx/PGCHE/ceilidh.html 31/03/2004)

Venables and Haywood 9 2001) describe the *submit* (the italic style is the creators' own) program as being "*developed so that students can receive instant and automated feedback to individual programming exercises.*" (Venables and Haywood 2001 p 1) *submit* differs from Ceilidh in that students can upload code as many times as they wish until a due date. "*After the due date passes, a tutor is able to review each student's final submission and provide further feedback, including a grade and comments on-line.*" (Venables and Haywood 2001 p 1)

submit analyses the student's program for elements of style such as use of comments and size of modules. The student's code is then compiled and the compiler output sent back to the student. Finally, the program is run using the test data provided by the lecturer (in much the same way as Ceilidh).

Venables and Haywood (2001) draw some general conclusions from their four separate trials of *submit*. The marking load for staff was reduced and the turnaround time for marking reduced from ten to five days. It should be noted that staff marked a randomly chosen exercise from the several put forward by a student and *submit* marked the others. (Venables and Haywood 2001 p 5) Venables and Haywood conclude that

> "*There have been a few unexpected benefits.* [] *Students ... seem more willing to take responsibility for their learning, in that they will keep improving their program until*

> *submit accepts it without complaint, whereas previously, they had a tendency to be sloppy when it came to program style and testing.*" (Venables and Haywood 2001 p 5)

Other systems for assessing student code are significantly later than the first version of Ceilidh: ASSYST is described by Jackson and Usher (1997) as a "*software tool…that is designed to relieve a tutor of much of the burden of assessing* [student]… *programs.*" (Jackson and Usher 1997 p 335) Similarly, Satratzemi et al. (2001) describe Telemachus as software developed to

> "*test and grade students' programs.* [] *The aim of Telemachus is not only to grade students' programs but more importantly to provide reliable performance data that could give a reasonable gauge of student knowledge and in this way contribute to teaching programming skills.*" (Satratzemi et al. 2001 p 30)

Satratzemi et al. (2001) refer to ASSYST and also to another system, called BOSS. (Satratzemi et al. 2001 p 31) Satratzemi et al. (2001) conclude that the

> "*…help that Telemachus offers to the tutor is likewise invaluable. Besides the fact the fact that the system saves the tutor from the laborious task of checking and marking students' programs manually, it also gives information that a human might have completely missed…*" (Satratzemi et al. 2001 p 36)

Ruehr and Orr (2002) offer a very different perspective on grading student code:

> "*…the pressure of class loads and the availability of sophisticated automatic tools have steered some instructors away from close interaction with students in computer science.*" (Ruehr and Orr 2002 p 65)

Ruehr and Orr (2002) see this loss of contact between student and tutor as a weakness.

Ruehr and Orr (2002) argue that interactive demonstration of their own code offers the student a much richer and more interesting experience than the automated grading approach. Ruehr and Orr (2002) list the technical criteria for grading student code. It should be noted that part of the aim of this grading is to identify students who are most suited to continue in computer science: this is not a consideration in the author's own grading.

Ruehr and Orr (2002) list seven areas of interest for grading: program design, class inheritance (for object-oriented code) coding style, program performance, program robustness, documentation and user interface design. (Ruehr and Orr 2002 p 66)

A key element of this approach is the opportunity for the instructor to gain a sense of the individual student's ability to communicate effectively about his or her code. Conversely, Ruehr and Orr (2002) feel that students benefit by watching the tutor work with their code. Overall, Ruehr and Orr (2002) feel that the technical element is comparatively easily assessed and the author would concur. A program that does not compile, no matter how elegantly commented, is of no value as software. The more intangible elements are more difficult: if a program compiles, runs and produces reasonable output, how do we distinguish the excellent from the simply OK?

Ruehr and Orr (2002) identify what they perceive to be critical elements of a "*good assessment process*." (Ruehr and Orr 2002 p 68) Such a process should be informative, focused, fair, feasible and rewarding for both student and instructor. (Ruehr and Orr 2002 p 68)

Ruehr and Orr (2002) argue that their approach to student code assessment, which centres on one to one program demonstration, "*offers many advantages over other common approaches used in the discipline, in that it provides for rich, informative and satisfying exchanges between student and instructor.*" (Ruehr and Orr 2002 p 68) The demonstration of code can be iterative, as Ruehr and Orr (2002) note, depending on time constraints.

Ruehr and Orr (2002) argue that interactive program demonstration has a key strength in guarding against the submission of plagiarised material: "*if the student wrote the program, he should of course understand how it works, and should be able to explain this to the instructor…*" (Ruehr and Orr 2002 p 71) Ruehr and Orr (2002) do conclude that a mix of assessment strategies will probably work best, given the constraints and demands of a particular academic environment.

A different approach again was offered by Edwards (2003) who described the linking of automated grading to a testing-driven style of code development. *"The core idea underlying this approach is that students should always practice test-first coding, also known as test-driven development (TDD) on their programming assignments from the beginning.*" (Edwards 2003 p 1)

Students are required to submit a test suite with their code and are encouraged to develop good, 'small iterations' (rather than 'big bang') testing strategies. Edwards (2003) notes that automated grading of code generally ignores any student testing so to fully exploit the possibilities of TDD, the assessment must focus on the students' testing performance. The Web-CAT (Web-based Center for Automated Testing) Grader therefore assesses the validity and completeness of student testing and also does static analysis on the code style. (Edwards 2003)

To sum up, the idea of automated grading of student programs seems to be a recurring theme among computer science and programming tutors. There are variations on the theme: the arrival of the Web seems to have given new impetus to the idea, as students can now submit work over the Internet. There are some who regard it as one, potentially weaker element in the set of assessment strategies.

Exploring how automated grading of code could support the development of programming skills, rather than simply providing a pass/fail or percentage mark feedback would be a rich area to explore but one that is outside the immediate scope of this author's work. In the case of the student cohort given the opportunity to develop patterns, the marking was done traditionally, by hand, by the tutors who taught the unit. However, the author does intend to use the data from the SLE (Specialised Learning Environment) which collates student feedback and results for each unit taught with in the Department of Computing, to look at the number of students who passed the unit. This raw measure is discussed later in this chapter.

The author's results

The author began by distributing the initial questionnaire to the students on the Introduction to Programming unit, in both Lincoln and Hull. She was able to introducer herself to the students in Hull and explain the purpose of the questionnaire. She was also able to distribute and collect the questionnaires in a short 10-minute session at the start of a colleague's scheduled lecture, thanks to the co-operation of that colleague. For the Lincoln students, the author had to rely on a different colleague who delivered the Introduction to Programming lecture, to deliver and collect the questionnaire. Despite the large cohort in Lincoln (139 students), the author was able to collect a total of only 26 replies from both cohorts – a 16% response rate. This was extremely disappointing. It was a salutary experience for the author, who had previously always taught the unit and had been able to manage data collection fairly closely.

The initial questionnaire

In collating the questionnaires, no distinction has been made between Hull and Lincoln students: had the response rate for Lincoln students been greater, it might have been worth processing the two sets of responses separately, as well as overall.

Out of the 26 replies, 14 students were aged 18. The remainder were aged 19 – 40.

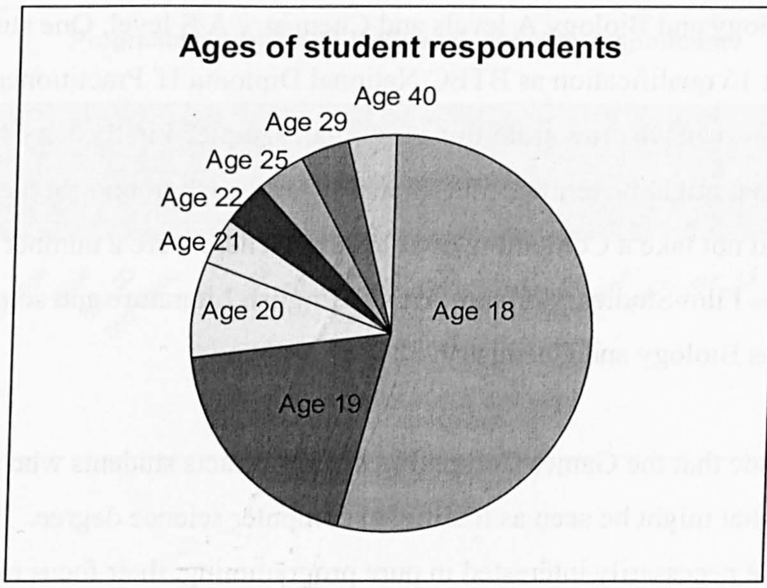


Figure 12: Ages of student respondents

Students were asked what qualifications they had at post 16.

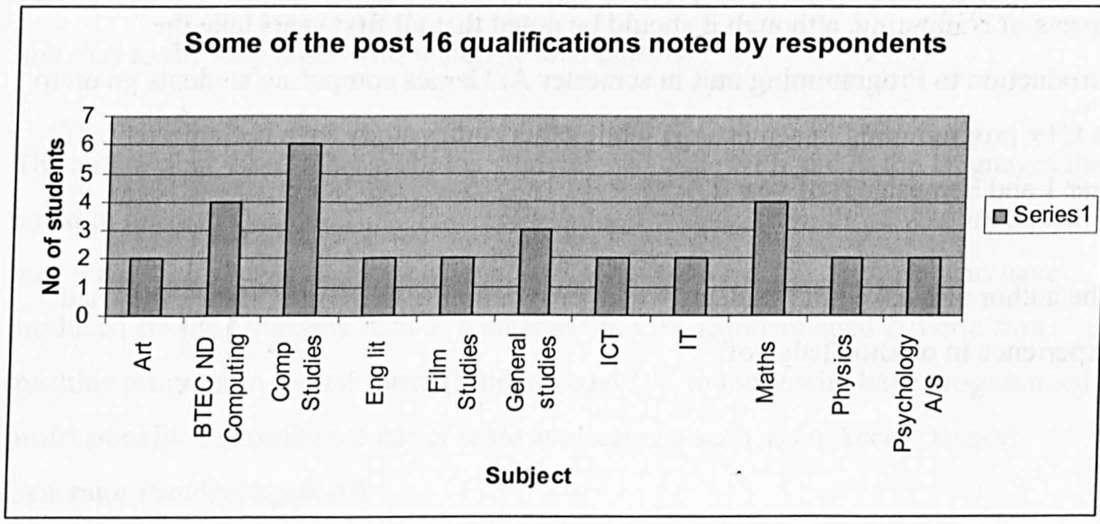


Figure 13: Some of the post 16 qualifications noted by respondents

To simplify the graph, the author has omitted subjects that were noted by only one respondent. These subjects were: Business and Economics, Psychology, Politics,

Design Technology and Biology A levels and Chemistry A/S level. One student also noted their post 16 qualification as BTEC National Diploma IT Practitioners.
What conclusions can we draw from this very small sample? Firstly, very few of the students had what might be termed a traditional A level background for a computing degree: they did not take a Computing or IT A level. There were a number of arts subjects such as Film Studies, German, Art and English Literature and science subjects such as Biology and Chemistry.

We can conclude that the Games Computing degree attracts students who are less interested in what might be seen as traditional computer science degree. These students are not necessarily interested in pure programming: their focus may well be game design or character creation for the games industry. These students may have some programming experience but it is likely to be focused on the production of game code rather than coding for coding's sake.

In Hull, the degrees recruited to include Computing (Software Development) and Computing (Internet Systems). These students may be more interested in the general aspects of computing, although it should be noted that all first years take the Introduction to Programming unit in semester A. Games computing students go on to do C++ programming in semester B while other routes study Java in Semester b of year 1 and Semester A of year 2.

The author also asked the students which programming languages, if any, they had experience in or knowledge of.

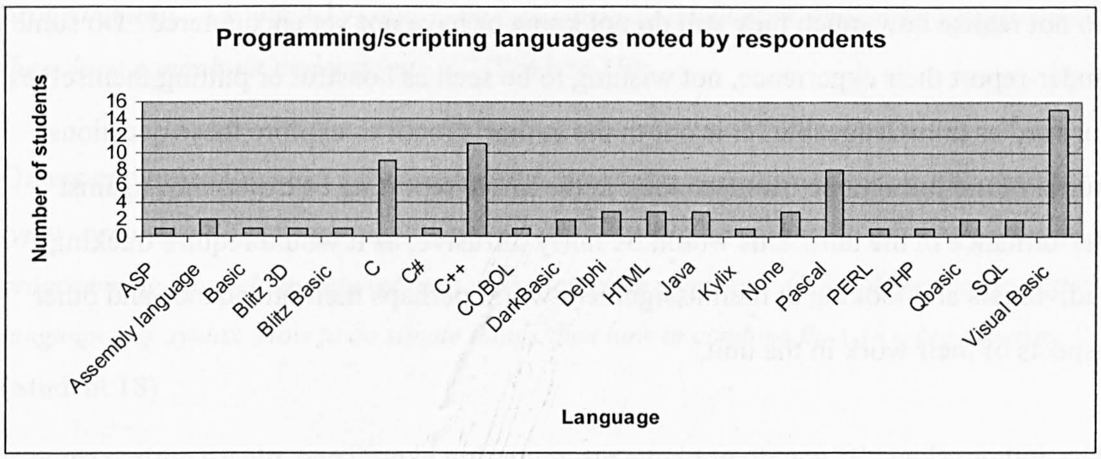


Figure 14: Programming/scripting languages noted by respondents

Visual Basic was the most frequently noted language (15) closely followed by C++ (11), then C (9). Pascal was also noted (8) and Delphi (3). Other flavours of Basic included QBasic (2), DarkBasic (2), Blitz Basic (1) and Blitz 3D (1). A small number of students accounted for many of languages with just one respondent. For example, one 21 year old student listed Pascal, Delphi, Kylix, C++, C ASP, PHP, Visual Basic and Perl as the languages with which he was familiar.

The author also asked what work the students had done with and in the languages they noted in the previous question. The replies varied widely, from those who indicated they were "*self-taught* [and] *messing about*" (Student aged 29), to those who have produced simple programs such as a database in C++ (Student aged 20) or a fruit machine program in Visual Basic (Student aged 19), to those who have programmed professionally and produced larger scale applications such as an Access Report Generator (Student aged 40).

Even 26 replies give the author and reader some idea of the diversity of the audience for the Introduction to Programming unit. Some students frankly admitted they had no programming knowledge or experience, while others claimed some acquaintanceship with at least one language. What that means exactly for each student is a matter of conjecture. Do some students over-report their programming knowledge, because they do not realise how much they still do not know or have not yet encountered? Do some under-report their experience, not wishing to be seen as boastful or putting themselves forward as knowledgeable? It is not in the author's remit to explore these questions but in future it might be useful to look at the initial reporting of experience against performance in the unit. This would be fairly intrusive, as it would require tracking individuals and looking at their assignment work, perhaps their attendance and other aspects of their work in the unit.

The author asked, "If you do not have programming experience, please outline some of the issues around learning programming that you think might concern you." Replies included "*Boring textbooks*" (Student 18), "*Mainly the mathematical skills involved…*" (Student aged 29), "*Very big subject – lot to learn!!*" (Student aged 25) and "*Learning coding, keeping up with the course outline and others with previous knowledge of programming…*" (Student aged 18). Out of the 26 replies only 5 made a response to this question, arguing that those who skipped this question felt that their programming knowledge was sufficient to give them an advantage in this unit.

The author also asked, 'What should the Introduction to Programming unit teach?' The author has asked this question of other groups and it is this question she hopes might produce clear insights. There does seem to be a paradox here, one that she has encountered herself, in learning to program: on first attempting to code, it seems so baffling as to be ungraspable, unfathomable but once one has understood the key elements it is hard to remember a time when one did not at least have some grasp of programming. Those who can't program yet don't know what it is they need to know or how it might be learnt and those who can program seem to have forgotten how they learned to program!

The idea that cropped up in this batch of answers was 'basic': "*A basic step by step introduction.*" (Student 29), "*Basic principles of programming…*" (Student 20), "*Basics and how to teach ourselves.*" (Student 18), "*The basics. Subroutines, algorithms, saving, loading and searching.*" (Student 18), "*Basic concepts of* *programming.*" (Student 19) and "*The basic principles behind syntaxing and to teach them how a machine understands it.*" (Student 19).

Others expanded the idea: "*Basic program structure, implementation of various data types, error handling/trapping.*" (Student 22) and *"How to program! Methods for good programming that help avoid problems when doing big programs. General knowledge of the language e.g. syntax. How to do simple things then how to combine them in a big program. "* (Student 18)

Other replies focused on broader questions of technique: "*Good practices and techniques not to be sloppy and lazy coders.*" (Student 21), "*Ways of programming. Standard industry techniques. The role of programmers etc.*" (Student 18)

It needs to be noted that the lecture series was based around theoretical problem and algorithmic thinking. The tutor's first lecture noted that this was not a programming course and students did not do any Pascal programming until mid-November, 2003. Even in the first questionnaire, students were raising this approach as one that could cause problems: "*More immediate practical work to familiar*[ise] *students with hand on programming.*" (Student 18).

In the collated comments from the study unit evaluation, collected anonymously via the SLE (Specialised Learning Environment), one student said "*I feel it would have been better to base this unit around the practical side of programming rather than a more theory-oriented approach.*" Others noted that starting programming in the last four weeks of the Christmas term disadvantaged them, as access to resources was difficult over the holidays. Another student notes

> *"I felt that we should have actually begun programming a little sooner. " Another said "We didn't start programming in Pascal until the last couple of weeks of the unit- I think maybe this semester should be all theory and straight into C++ next semester or otherwise learn Pascal side by side with the theory in the first semester. "*

The feedback via the SLE was both anonymous and voluntary and just 9% of students on the unit responded (15 students). Looking back over the student involvement with feedback opportunities, a picture emerges of a cohort where a comparative few are willing to engage in those activities, for whatever reason or reasons. The focus groups initiative suffered from the same lack of student involvement. Focus groups are discussed in the next section.

Focus groups

The author had planned to run a focus group over the course of the semester with students in both Hull and Lincoln. She envisaged a group of say 4 or 5 students from each campus who would meet with her for a short time (no more than 25 minutes) every three to four weeks. To this end, she emailed students across the unit and noted that she would be happy to meet with both Hull and Lincoln students. The meeting with Lincoln students would take a little more organising due to the problems of fitting in the travelling to and from Lincoln but the author felt it was worth making the time to work with a group of students who were prepared to discuss some of the key issues with her.

However, there was no response from any of the Hull students, and just a handful of replies from the Lincoln cohort. Out of those who replied, just three agreed to meet with the author in Lincoln. The author suggested a meeting in November but it was early December before the students were able to meet with her. The idea of the focus group as an evolving, continuing processes had to be abandoned by the author. Once again, the reluctance of first year students to engage with the author and her work was a major problem.

The three students who had volunteered to meet with the author were two mature students and a post A level entry student. The mature students were not typical in their entry profile of students on the Games Computing course and do distinguish themselves from most of their colleagues. The group expressed some disbelief at the attitudes amongst the cohort towards attendance and resit opportunities. The author taped her conversation with the students and a transcript of the meeting forms Appendix M. The students were articulate and able to reflect on their experience of the unit Introduction to Programming. They presented themselves as committed to the academic process, despite having other claims upon their time such as part time work.

The author asked them if they had any specific suggestions to make about the teaching of programming to new programmers. They focused on the choice of programming language, arguing that Pascal was not an optimal choice. The structuring of the unit also seemed to concern them with regards to moving on to learn C++ in semester B. The end of unit comments gathered via the SLE also confirmed this as an issue of concern among the students.

The creation and running of the focus group or groups proved to be intensely disappointing to the author. Her vision of an unfolding narrative gleaned from the students' developing knowledge of programming concepts was unrealised. In previous year, she has been able to observe students working on the programming exercises that she has set but the author hoped that a focus group or groups would make more explicit the process of learning to program and would help her to focus on the key issues as perceived by those students. This was not to be.

Observation

The author was able to observe one class session, with the Hull-based cohort. The class took place on the 14th November 2003. It was the first session in which the students were introduced to the Pascal editor and compiler. The author's colleague introduced the material using the interactive whiteboard. In fact it was hard for all the students to see the whiteboard comfortably as the desks and PCs were arranged in rows at right angles to the board. The room was also quite small for the number of machines but not all the machines were in use. There were 12 students in the class.

The author took notes as her colleague introduced the Pascal editor and compiler and stepped the students through the compilation process with a small two-line program, projected on the whiteboard. Some of the students were able to follow the keystrokes quite comfortably but others lagged behind. One student arrived half an hour into the session. One student was typing very busily and in fact turned out to be emailing, not programming!

By the end of the hour, the tutor had introduced the idea of output to screen, error messages, reading in data, commenting and the assignment statement. It was an intensive hour but the author noted that not all the students followed the step-by-step activities of the tutor on the whiteboard. Some found it difficult to keep up and others were obviously ahead.

It was an enlightening experience to watch the kind of teaching that the author has done many times herself, leading students through their first practical work with a programming language editor and compiler. What struck the author was the many ways in which students simply did not follow what was happening, for a number of reasons. The computer proved to be a distraction in itself, hence the emailing not programming, and other students tended to ask their immediate neighbour for help when they were having difficulties, rather than the tutor.

Exit questionnaire

The author gathered feedback from the Hull students on the 25th of February 2004. She travelled to Lincoln to meet with some of the first year students who were in seminars on the following day, 26th February 2004. The author met with three timetabled groups in Lincoln and gathered a total of 23 responses over the two days. The exit questionnaire asked only two questions: *What has been the hardest/most difficult aspect of learning to program?* and *What resource (book, friends, web page...) or strategy has been the most useful in learning to program?*

Answers to the second question fell into the following categories: Friends, Books, Websites (including message boards), the SLE and 'Practice', as in spending a great deal of time coding. The total number of replies does not add up to 22 because some students noted two resources as being equally important. As can be seen from the following graph, websites of various kinds, including message boards were most popular. No student made any reference to the pattern template created and offered by the author and no attempts were made by any student to add a pattern! It seems that students do not favour 'home-grown' resources, preferring the authoritative appearance of Internet material.

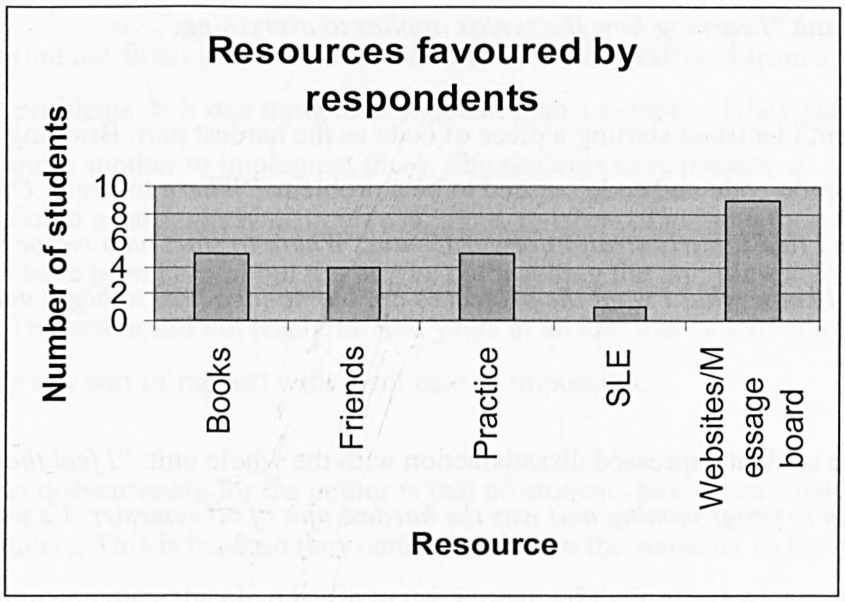


Figure 15: Resources favoured by respondents

Comments in response to the first question were varied. Some were crisp in expressing their views: "*…finding motivation to do it, since most of the examples are rather pointless.*", "*The hardest aspect of programming was being told that an assignment was due in using Pascal and not actually being taught how to program whatsoever!*", "*…a lack of hands-on practical exercises to demonstrate the concepts and realities of programming.*" and "*Learning what we need to know from the internet in 3 weeks. The practical lessons started too late.*"

Other students focused on the difficulties presented by having previous programming knowledge: "*Learning Pascal, after having used C++ for several years.*" and "*The beginning. Learning to use different syntax.*"

Syntax was frequently mentioned as presenting the biggest challenge: "*Probably learning the syntax of the languages or in other words, being able to convert pseudocode and ideas in my head into actual 'code'*", "*Learning the syntax for a program and then remembering it after a break in learning/practicing (sic) Also* *remembering the different rules from multiple languages and not getting them confused.*" and "*Learning how the syntax applies to everything.*"

Other student identified starting a piece of code as the hardest part. Bridging the gap between pseudo code and code seemed to be a problem: "*Where to begin. Once that part is over I find it fairly straightforward.*" and "*Where to start on a major project. Sometimes I know what I want the project to do, but no idea how to begin making it do it.*"

Finally, one student expressed dissatisfaction with the whole unit: "*I feel that the introduction to programming unit was the hardest unit of all semester A's units.*"

A number of issues emerged from this end of unit feedback that were more to do with the way the material was structured, paced and delivered than the learning of Pascal by and of itself. Nonetheless, it is possible to identify some key themes, such as the learning of the syntax of a language, the issues around good code production, the use of authoritative sources and the availability of meaningful examples of the language use.

Unit results

The results as presented by the University of Lincoln's own information system: 158 students on the unit overall. 114 pass, 39 fail, 3 no result and 2 decision deferred. No details are available for the no result and decision deferred students. The fails can be due to non-submission of work or failing one learning outcome for the unit. The cause of the fail is not part of the top level of the collated results. This gives a pass rate of 73% and an outright fail rate of 24%. But what do these results really mean for the students who took the unit? What make those statistics meaningful would be explore the causes for those 39 students failing. Did they all fail for very similar or very different reasons? There is insufficient time to look at this but it should be borne in mind that students fail for many reasons, not all them to do with the material being taught or the way it is taught.

Chapter 7: Conclusion

This latest (but not final) cycle of the author's research has suffered from a number of significant problems. It is one thing to design and plan a number of data gathering strategies: quite another to implement them. The students have presented, in the author's opinion a particularly difficult challenge, in terms of gaining their feedback. Colleagues have been helpful but the fact is, not teaching the unit presented real problems. The author did not teach the first years at all and this lack of contact made establishing any sort of rapport with them next to impossible.

What is also disheartening for the author is that no students have even tried to use the pattern template. This is because they came very late in the semester to the hands-on programming and once they had begin to use Pascal, all their effort was focused on gaining enough knowledge to pas the assignment, due in around early January. However, this does not invalidate the potential of the idea of a pattern community among novice programmers. The author feels that, had she been teaching the unit, she would have made the pattern resource a central feature of the unit. In this last iteration of the unit, students were required to keep a learning log and submit it for reading by the tutor. This was noted as a waste of time by one student, in the end of unit feedback via the SLE. Being part of developing the pattern community might have been more appealing (and less like being at school) for the first year students.

This is not intended as criticism of my colleagues: they conveyed central concepts for programming skills, which are an essential foundation for years two and three. However, the students felt encountering practice in Pascal late in the semester, rather than at the beginning, handicapped them. It is a persistent problem: there is a limit to what can be taught in any ten or twelve week period and in what order should the chosen elements be presented?

The author is disappointed that the pattern community she hoped would start in the first semester has not materialised but it dies not sake her belief that there something potentially very valuable in this approach, more so than in the two previous interventions. In the next chapter, the author reviews the three cycles of research she has undertaken and draws some general conclusions, rooted in her own experience, about approaches to the teaching of novice programmers.

Introduction to Chapter 8

This chapter reviews the three main cycles of research undertaken by the author and draws some personal conclusions about the work. It identifies what the author sees as the strengths and weaknesses of the work and presents an overview of the collected effort of the author to engage with the teaching of programming to novice programmers.

A note on viewpoint and voice

This chapter, more than the others, is an intensely personal one. Here, benign reflection perhaps no longer serves the author as well as it may have done in previous chapters. This chapter makes a narrative of the work done by the author and the narrative might be better framed in a truly reflexive way. This chapter does maintain the stance of benign reflection but the author feels it is important to acknowledge that there are tensions in and around this recounting of her effort and experience. A narrative also, by definition, makes a coherent story out of what may, in the experiencing, less than coherent. The unfolding of the story is to some extent done by the act of telling the story. False starts, dead ends and unproductive strategies may not be allowed to impede the momentum of the narrative, without any suggestion of dishonesty or deceit. All of these obtain in the author's story and it is worth acknowledging that here.

The research: an overview

The initial impetus for the work undertaken by the author over a number of years was one that sprang from her own experiences as a novice programmer.

To recap, the author took her first degree in English literature at Sheffield University in 1981. She took the PGCE (Post Graduate Certificate in Education) also at Sheffield University in 1983 and became a probationer (those were the days of a probation year for new teachers!) at Hornsea School in the East Riding in September 1983. She taught English language to pupils aged 11- 16, for five years, before leaving to take a conversion MSc in computing at Sheffield City Polytechnic in 1988 – 1989. Up to that point, the author's only experience with computers had been the use of a BBC master and a dot matrix printer to create Banda masters for copying at school. The author feels it is worth noting the reason for her change of direction.

After a fairly difficult first year, she did pass her probationary period and became fully qualified teacher in 1984. She gradually took on extra responsibility for the school libraries and in her last year at the school, re-organised and re-catalogued the sixth form library as well as staffing the lower school library every lunch hour. When she approached the school's management team about being moved up onto what was then a Scale 2, in recognition of her work, she was told that no money was available but that she should go on doing the work because it would get her good references! At this point, the author decided enough was enough and looked around for a new job or a new direction.

The author found the conversion course at Sheffield City Polytechnic, applied and was accepted. The move to computing was a calculated one: the author realised that senior positions in teaching English would be hard to get and computing might offer more opportunities. In a sense, leaving a job to take up the MSc was a gamble but the author was then only in her late twenties and had no dependents. It was a feeling of now or never for the author. There was a SERC (Science and Engineering Research Council) grant associated with the course, which helped to soften the financial transition from a salary to no salary.

The initial four weeks of the course were an intensive introduction to Pascal programming. The course may have assumed no prior knowledge (and the cohort was very mixed in terms of ages and backgrounds) but the pace was demanding and the tutors made no secret of the fact that at the end of four weeks, passing the Pascal assignment was a pre-requisite of continuing the course. Everyone did pass and the author wonders if the 'make or break' stance was a bluff? It hardly matters: it served its purpose as most of those on the course believed in it entirely (the author did) and were suitably frightened or motivated, depending on one's point of view.

The author can remember with absolute clarity (which may be illusory!) the first time she was required to write pseudocode. None of the lecture on pseudocode made much sense to the author and the idea that she could create this stuff out of her own head seemed both laughable and terrifying. The actual programming of Pascal was no better for the author. None of it had any relation to everything she had studied previously. In some kind of left-brain – right-brain burst of activity, writing poetry became easier than ever for the author but writing programs seemed out of the question. The author relied extensively on the help and support of other students on the course and a friend who was an amateur programmer, for the first few weeks of Pascal. She did pass but she is not sure how, in hindsight. None of it made much sense at all. Other aspects of the course were more enjoyable: logic programming and databases, for example. Both Prolog and SQL (Structured Query Language) seemed manageable and even intuitive to the author but was that because she had done those first weeks of Pascal? A sensitisation issue, perhaps.

For the dissertation, the author chose to write materials to support learning Prolog, aimed at novice programmers. Interestingly, in looking at the format of the work, the author can see elements that she re-used, unconsciously, in the first CAL workbooks that she developed for the Department of Engineering, as it was then. The idea of white space for students to record their own responses and checkpoint throughout the text were two key ideas. After gaining the MSc the author taught IT in a sixth form college for a while. Then she decided to move out of education into industry and gained a job as a junior technical author in a company in Birmingham. Life outside a school or college was very different in many ways but the author enjoyed her work. She documented large-scale systems for the motor trade.

During this time, she continued to teach, taking a range of adult education courses in word processing and introductory IT. After a year, she moved jobs, working as a technical author in Hull, this time writing manuals for legal software. After eight months in this job, which proved to be a fairly miserable experience, for a variety of reasons, the author was happy to make the move to what was then the Humberside college of Higher Education. Teaching in higher education had been the author's long term goal: her move to industry had been calculated as one that would make the possibility of getting a teaching job in higher.education more likely.

The author joined the Department of Engineering and was given a range of topics to teach from assembly language programming for HND year 1 students to basic spreadsheet concepts for business students. In her second academic year at the College (1993 - 1994) the author taught introductory programming using Pascal to BSc students studying business and software development. The issues around teaching novice programmers to program came to the forefront again, with particular significance. These students were unlikely to be full time, specialist programmers themselves but they needed enough expertise to work with and perhaps manage software development teams in a business context.

At the same time, students on BSc engineering degrees studied introductory programming in Pascal and then moved on to programming in C, as part of the control engineering element of their degree. Two very different audiences but the difficulties were similar. The kind of questions that concerned the author can be summed up as follows: What programming language should novice programmers use as a first language? What material does a student learning to program for the first time need? How should that material be structured and conveyed? What order should concepts be presented in? How should the students' programming effort be assessed? How much support should a novice programmer expect or get? What are the problems that seem to crop up repeatedly? What are the common misconceptions?

In a sense it is misleading to formulate these questions so neatly. Throughout this work, some of these questions have seemed clear, even obvious (although the answers less so) but at other times, it has been hard for the author to remain focused on attempting to form, let alone answer, those questions in any cogent fashion.

Realistically, all that the author has done over and above creating basic laboratory sheets and delivering material to students has largely had to be done in her own time. Workloads have been variable over the years but rarely have they been negligible!

To continue the chronological narrative: as noted in an earlier chapter, the member of staff, Roland Stokes, who later became acting Dean and then Dean of the Faculty of Engineering (which went through several name changes, becoming at one point the Faculty of Engineering and IT) initiated a project to develop CAL materials for the Department. The first project focused on selecting an appropriate platform for the development of CAL packages and using the platform to produce a piece of CAL on LANs (Local Area Networks) and CANs (Controlled Area Networks) for use within the Department. As the author had worked as a technical author she was asked to be involved in the project, as second supervisor and junior editor. The project, by three part time students who worked at the local British Aerospace site, was deemed a success. The students chose a flow chart-based programming platform, IconAuthor.

The perceived advantage of IconAuthor was that it had powerful template facilities. Once a flow of pages and a navigational structure was developed, the content of the pages could be changed by changing the pointer within the page to the text file stored elsewhere. In subsequent years, the initial work done by the British aerospace students was handed on to other final year project students, who used the templates and associated documentation to create new packages.

One team experimented with using video clips inside the CAL pages, in the academic year 1994 – 1995. The use of video was very much liked by students who saw the CAL on engineering materials but the University's network capacity at that time was insufficient to deliver appropriately the amount of data such clips required. Further CAL packages did not use video but only graphics.

One of the students who delivered the CAL on engineering materials noted in her report of May 1995 that there were in house CAL packages already written on propagation, computer memory, satellites, electronic principles of communication, waveguides, history of telecommunications, modulation and receivers, transmission lines and moments and force. (Cullum 1995 p 5) We can see from this that in about 18 months to two years a collection of packages (each of which would be around two hundred pages or screens) had been developed with fairly minimal resources: two full time placement students in 1994 – 1995 to create the CAL based on materials submitted by the staff who taught the subject.

This rapid development of CAL generated problems of its own. The CAL could be seen as little more than the text of lectures placed on screen with a handful of navigational buttons and some graphics. The end of section tests might be useful to students and were could be used by tutors to monitor student progress through the package. Statistics were collected for each section and sorted by student. A report was generated detailing the total time spent in the package by a student and the total score for the user across all the end of section tests. There were all kinds of pedagogical issues raised by the processes and products in CAL development. What concerned the author most was the lack of any theoretical model or framework for the design and use of CAL.

For example, the end of section questions were multiple choice. Questions were randomly allocated to each student from a bank of questions in the CAL, so two students working alongside each other, finishing the same section at the same time, would not se the same questions. The order of the answers was also shuffled so the correct answer to a question, say about line of sight in the package on satellites would not always be A. These features were much vaunted as cutting down on the possibility of sharing answers but there was no examination of whether such questions in themselves were meaningful or supported learning by the students.

In terms of learning theories, there was no model that underpinned the style and design of the CAL. It was accepted as a truism by those in charge of the project that CAL was better (and better was *not* defined!), cheaper in the long run and preferred by the students. There was no detailed or meaningful argument available to those using CAL about what models of teaching or learning were being embodied in the CAL and its delivery.

It needs to be noted that the author enjoyed being involved with the CAL. She took on responsibility for developing standards for the team including style standards. She also took on the role of style editor at the design stage, checking paper storyboards for the CAL. This close involvement meant that she could see very clearly the lack of theoretical underpinning for the work on CAL and this concerned her. Her initial idea was to examine a range of learning theories and look at the use of CAL within the Department of Engineering. She hoped her work would generate an appropriate framework for the design and use of CAL and produce meaningful arguments for the pedagogy of CAL written for use in the Department.

The author generated material on types of computer memory for a small piece of CAL. This was extended to become Introduction to Computers, a much bigger package. The author also felt that material on programming concepts was particularly difficult for many students new to programming so a first version of Introduction to Programming was written by the author and turned into CAL by the team. The author used this package but found it limited in its impact. Due to pressure of work, it was shelved for over a year and completely rewritten in the next academic year (1996 – 1997). It is this second version, with major changes in the order of the material that the author used for two years.

Given that the author's goal was to read widely around learning theories, that was certainly achieved. To map those theories on to the design of CAL packages and their use was quite a different matter. From the author's own reading many writers and researchers simply did not engage with this problem. Researchers around the use of CAL in the classroom were happy to quote student responses to the package (usually positive!) and/or detail how the CAL was written and/or how it was used but very few addressed the thorny issue of what pedagogical approach was being supported or embodied in the CAL. It was easy to find material that described different categories of CAL and reams of material on CAL projects, particularly with the advent of the TLTP and its funding of CAL development within consortia in higher education. It seemed to the author, in the mid 1990s, as if almost everyone in higher education was writing CAL and using it successfully with their students. It is worth saying that some packages stood out as particularly considered and well designed and some projects had a much more pedagogical approach but not all packages were excellent.

The number of packages produced within the Faculty of Engineering and IT as it has then become rose to 20 by mid to late 1996,. most of which were sold commercially through an arrangement with Feedback Instruments Limited. The author's work on learning theories suggested to her that what was being modeled in this CAL was a blend of a drill and practice approach (multiple choice questions) and page turning CAL (simple navigational structure). In itself, that was acceptable, if not innovative. The author decided to focus on the Introduction to Programming package and look at student responses to the material. This proved to be more difficult than she had ever imagined and now after three phases of her research, it remains a problem.

There are two key issues, in the author's view. First, what kinds of instruments are suitable for gaining student feedback and secondly, how can anyone (even the respondent) be certain that the answers truly reflect the learning or teaching that is happening? The second question arose strongly when the author conducted an interview with three first year students about the Pascal unit. (See Appendix M for the full transcript). From her conversation with the students, it was clear that, although they were very willing to discuss their experiences (and were articulate and apparently very frank) they had fairly limited insight into how they learnt and what strategies they could adopt, or be offered, to help improve their learning of programming. The author hastens to add, this is not a criticism of those students.

In her own experience, trying to offer insight into how one learns is very much like trying to look at the back of one's head with a very small mirror – one can only get partial views which may not give a true picture! It is difficult to describe exactly how one learns a specific skill. For example, the author can remember not understanding parameter passing at all: it was an utterly baffling concept that was unlikely to ever make sense. Now the author understand parameter passing and it seems elegant, intuitive and obvious but just when did the author's understanding go from the first to the second state? Was it a single moment of insight or a gradual 'coming into focus' of the ideas? She cannot say herself and this, in her opinion, illustrates part of the struggle in her work. Once a student grasps a programming concept it is very hard for him or her to recall that he or she once did not know that concept or misunderstood it.

To ask a student how he or she got to that level of understanding generally elicits a response along the lines of a shrug and "I just did." or "It sort of clicked."

So, to explore what most effectively moves a student (and of course, it is potentially misleading to talk of 'the student' when discussing teaching – all students are different and bring different expectations and abilities to the process of learning to program) from not understanding to understanding say, the idea of variables versus constants or the required program structure, is a complex undertaking.
At this stage, the author feels it is worth reiterating the questions that have sprung from of the experiences briefly noted here, and elsewhere, and presented first in Chapter 1:

- What are the key issues in learning to program?
- How significant is the choice of first programming language?
- Is there a meaningful distinction between learning a programming language and learning programming skills?
- What theories of teaching and learning can be most usefully employed to support the teaching of novice programmers?
- What theories drawn from various areas of psychology are of relevance to the teaching of programming?
- What aspects of CAL can be used to support the learning and teaching of a programming language?
- How can the author's work on the teaching of programming be embedded in an appropriate academic research framework?
- Having used a particular approach to the teaching of programming, what ways are there of gathering and collating meaningful evidence about that approach?
- Having gathered the evidence, what conclusions can be drawn from the author's work?

In a sense, the first question decomposes into all the subsequent questions and many other questions. There are probably as many issues in learning program as there are programmers who are learning!

For the question on choice of programming language, the author considers that the choice may not be as significant as some advocates of some languages would have us believe. There is a balance to be struck between presenting students with a very complex first language and one that is too simple. Students, in the author's experience have been scathing about learning Pascal as first language but when asked for an alternative, no consensus is forthcoming! What does seem clear is that student needs to meet programming concepts and genuinely grasp them, so that they can map those concepts (variables, parameters, assignment) onto other languages. The author encountered a student recently who had written shell script that used a variable in an arithmetic expression before he assigned a value to it. He did not see why the script did not do the required calculations and yet he was taught introductory programming concepts in the previous year (not by the author!)

The next question about teaching and learning theories is an equally difficult one. The author has read around the subject of learning theories and learning styles but the gulf between the idea of learning styles and the practicalities of teaching programming seems as wide as ever. Very little work seems to have been undertaken in this area, and it is potentially very interesting and productive. It would require careful testing of students to assess their learning styles and the design of multiple approaches to the delivery of material, a demanding and time-intensive undertaking. However, it could yield some useful suggestions for the shaping of the teaching and learning of programming.

In exploring some (a very few!) of the ideas to be found in psychology, the author found one concept very illuminating: that of confirmatory bias. It explained a great deal of her own early programming problems, in trying to debug her code, and helped her to understand why novice programmers genuinely do not see why their code is so poorly structured or bug-ridden! The question that springs out of this insight, is if the students know about confirmatory bias, does it change their coding and debugging approaches? On an anecdotal level, it seems not, but again, formal exploration of this idea would be worth undertaking.

The question about CAL supporting the teaching of programming is really where the author's research effort began. Naively, when she began, the author expected to undertake several iterations of the development of the Introduction to Programming Ideas CAL until it reached a level of polished effectiveness that made undeniable its positive impact on the learning of programming concepts. This was a vain, if long cherished hope. The Programming Ideas CAL underwent two major revisions and considerable development in term of its quantity of content but the last version seemed no more helpful or meaningful to the students than the initial version.

As with most CAL there were difficulties in gathering data about how effective the CAL was in delivering programming concepts. In all her reading, the general tenor of students' feedback is that they like the CAL, whether it is on diagnosing horse ailments or translation skills, but we have to ask some hard questions about *why* students like the CAL, whatever the package is, and what respondents really mean when they say they like the CAL.. We cannot discount all kinds of variables that affect individual and groups responses to new teaching approaches, computer-based or otherwise. It seemed to the author that page-turning, tutorial CAL had very little genuine benefit for the students apart from freeing them from the onerous duty of attending a set lecture time. The next question for the author was then, what other computer-based approaches might be relevant? The author then moved on to look at developing a help system that would grow in response to student queries, and finally, the use of a pattern language to support novice programmers. These three approaches are discussed further in this chapter.

What the author lacked was a framework for the research itself: an overview of the approaches, methods and tools that could support the author's investigation of the issues around the teaching of programming. In the author's opinion, it is a weakness apparently shared by some of the research efforts focused on programming. In discussing the latest new programming language or the use of an in-house programming environment, there is little or no acknowledgement of the dimensions of research paradigms, methodologies and methods. Obviously, the author cannot tell if this is because there is insufficient room to report on the intellectual and academic arguments advanced for the use of particular approach within a specific methodology, or whether it is because researchers in programming see no need to engage with ideas in human-centred research. The author would argue that an awareness of at least some of the ideas to be found in, say, educational research, at a bare minimum, is essential to any researcher working with human programmers!

The author's readings have led her over a number of areas in research methodologies, with particular emphasis, for obvious reasons, on educational research. Of necessity, human-centred research is hedged about with caveats and disclaimers and the author's own work is no different. What perhaps does make it not quite like much of the research in programming is that the author has attempted to develop at least some of her work with reference to theories of research that seem relevant, rather than simply technical issues. In the author's opinion it is not enough to examine questions of programming language constructs or graphical environments: there needs to be a foundation of research design. The author admits that her early worked lacked such a fully articulated design, but feels that her later iterations did have a sense of a mapping between ideas to be explored and approaches to those explorations.

The next question focused on the tools that might be meaningfully employed to gather the kind of data needed for real insight into the process of teaching someone to program. This has proved to be an area of ongoing challenge for the author. Possible and potential data gathering methods were reviewed in an earlier chapter, Chapter 2. The author has tried a number of methods, including the ubiquitous questionnaire in a variety of styles (Likert scale through to open ended questions). The success of these methods varies but overall, the author is left with a feeling of dissatisfaction. It is not that students are particularly difficult to survey in themselves (although this last round of questionnaires in the autumn of 2003 was a particularly frustrating experience) but that the author wonders if the data she is looking for is really available at all, in any truly meaningful way.

The idea in educational research seems to be that an educational strategy of some kind, from open plan classrooms and unstreamed classes of the Plowden Report era to tests at 7 and 14 and AS levels of the 1990s do have a positive impact on the education of those on the receiving end, or why else bother? The question therefore is how to measure and quantify the effect of the intervention or approach. On a very small scale, that has been the author's rationale: those using the CAL, or the help system or the pattern language are given an advantage. On a small scale, though, is the effect so subtle that is impossible to quantify? Are there things that a teacher of programming can do that seem like a good idea (and are well-received by the learners) but whose use cannot be fully supported by quantitative data? If so, what value does that strategy or intervention have?

The problem centres on two issues: the inherent variability of groups of learners and the control group problem. Many factors affect the makeup of a cohort of students coming into the University, from the demand for places on a course (a popular course can afford to ask for higher grades at A level) to the pressure on other courses within the University and at other institutions. The makeup of a seminar or study group can have noticeable effects on the ways the group members react and behave in the classroom. Ultimately, every student is unique and everyone can be surprising, either for good or bad. In the mid 1990s the Faculty ran the foundation year for students who had failed al their A levels and the clear philosophy was that everyone should have a chance. Some students flourished, others dropped out. The BSc in Business and Technology which recruited first in 1991 had a similar ethos: we recruited many students through Clearing for the first five years and most took the opportunity offered and did well.

A related problem is that, in examining the use of the three interventions, the environment in which the students studied changed, either dramatically or subtly, over the years. The differences might be as small scale as the arrival of one or two new members of staff or the installation of new computers or as momentous as the move to a completely different campus, as we did in September 2003. It is not possible to compare absolutely groups of students, year on year, for the reasons outlined above. We might be able to identify trends or commonalities but comparing marks, for example, can be only indicative, not definitive.

The second problem is that of the control group. Here the question of ethics in this type of research needs to be considered. If the author believes that the facility she is offering to students has the potential to be genuinely helpful, than withholding that opportunity from another group of learners is unethical and indefensible. There can be no argument for attempting to formulate control group based experiments over a semester, where the final assessment is graded or marked. There might be a place for a single lesson approach, which the author has tried, without any noticeable difference in test grades. For a single topic in number bases it does not seem to make any difference whether the material is presented in CAL or in a lecture. The test grades across the two halves of the groups (randomly assigned) were very similar.

Another question that cannot be set aside really closely concerns this chapter itself as well as the work as a whole. As Shermer (2004) notes, researchers in all fields of study are not free from bias. The "*cognitive barriers that color clear judgment*" (Shermer 2004 p 27) are as present in the researcher as they are in the subjects studied.

Scott-Kakures et al. (1993) describe Bacon's doctrine of the Idols: "*The Idols are certain typical ways in which the human mind is apt to go awry.*" (Scott-Kakures et al. 1993 p 103) There are four Idols but what most concern the author are the Idols of the cave and the tribe. Idols of the cave are personal and particular biases, habits or weaknesses. The question would be what are the author's own Idols of the cave? The question is not a comfortable one but one that needs asking. The Idols of the tribe are the "*inherited foibles of human thought*" (Shermer 2004 p 27) that all human beings are prone to. So, in a sense, in researching programming, the researcher must be very clear about his or her *own* Idols of the cave, have an awareness that others have different Idols of the cave (which they will find perhaps equally tricky to identify!) and also be aware that everyone is subject to Idols of the tribe.

Another way to talk about this is to compare our belief in love with our belief in software. People profess some scepticism about both love and software but deep down they believe that bad things won't happen to them – they won't fall out of love or be dumped in some humiliating scene in their favourite restaurant and the software they rely every day on won't destroy their vital data or cause some terrible accident. When it comes to the question of love, attempting to destroy all illusions is probably not a good strategy. Writing software and being a good programmer is a different matter.

Why is being human not an automatic qualification for being a programmer? One example, discussed earlier in this work: confirmatory bias. People naturally look for evidence to support their view or belief. They discount evidence that contradicts what they *know* (in their hearts) to be true and seize on anything that confirms that perceived truth. Since programmers are people too, and people tend to have an attachment to what they create (it's only human) they will look for proof that their code is good. What they *won't* do is assume it is riddled with flaws of logic and errors of syntax and act accordingly. The author is interested in ways novice programmers can be persuaded to critically examine their cherished code. That involves teaching debugging strategies and also persuading people to look close to home for the mistakes (only human to look for someone else to blame!).

Also, teaching someone to program involves a lot more than handing him or her the syntax of a language. It requires people to develop ways to handle complex programs inside their minds. Memory and understanding are required, in a subtle interplay. More recently, work has been done on using graphics and layout to aid program comprehension. Such work is about how people perceive the world visually and how making something clearer on the page (using indenting, fonts and white space) makes understanding easier.

Memory, vision, self: all aspects of being human that impinge on the creating of code. All of these are Idols of the tribe and need to be acknowledged and accommodated. To know how to help people become good programmers, a teacher of programming needs to have an understanding of human nature and ultimately, the researcher is human too.

The next stage: the help system

Having developed a fairly large piece of tutorial CAL on programming and found it did not seem to answer student needs, the author reflected on the elements that might best support student programming effort. The author frequently referred students to the error messages generated by the Borland Pascal compiler but the messages did not seem to be particularly meaningful to new programmers. For example, a wrongly spelt variable name will generate the message 'Unknown Identifier'. This does not tell the student exactly what the problem is although technically it is accurate. The author wondered if there was a way to make the error messages from the compiler more useful and meaningful for the users.

With this in mind, the author developed a help framework that was designed to grow as it was queried. Students could enter a Borland Pascal compiler error message number or a key word and the help would display an explanation written by the author. Initially the author wrote a handful of explanations for errors that seemed to occur frequently such as the missing semi colon at the end of the line. The Borland Pascal compiler places the cursor on the line *below* that line missing the semi colon so the help written by the author directed the student to look at the line above! The help was meant to be simple and practical. The author also envisaged that as students queried the help, she would add the text for missing errors. To this end, the author asked the developer to add a file to store the queries entered by the students. Students accessed this help by opening a window for the software alongside the Pascal editor window.

The author envisaged harvesting queries on a twice-weekly basis and expanding the help significantly. However, the students did not use the help. Students entered a few queries but some of those were nonsense or expletives. The author feels that she should have approached the use of the help in a much more structured manner. The students should have been directed to the help in the laboratory session and encouraged to use it while programming in the University without tutor support. The author feels she could also have employed a more constructivist approach and used some of the laboratory time to work on the help text with the class. Surveying the preferred resources of the students who had been offered this initial version of the help system showed very little difference to the responses of the students in a previous year, who had used the CAL.

Surprisingly, programming students are very traditional in their choice and use of academic resources. Only in the past year has using web sites for programming information and guidance become more prominent. Programming students do seem to like books, up to date ones preferably but even old texts are acknowledged as being occasionally useful. Novice programmers also like to adopt a constructivist approach, with various shades of collegiality. Some, as noted by the students the author interviewed, make meeting to discuss or work on assignments, including programming, a regular and social activity. Others, of course, prefer to work individually, seeking help only for very specific problems.

Having worked with CAL and a self-written on line help system, the author felt that the next stage was a difficult one. In reading round the topic of programming generally, she came across the work of PloP and the concept of pattern languages. She read some of the work of Christopher Alexander (1979) and was enthused by the ideas contained in his book. The author was sure that extensive work must have been done in using pattern languages to support a constructivist thread in the teaching of novice programmers. However, her reading lead her to conclude that a great deal of very technical pattern could be found for experienced developers to draw upon, and patterns languages were being developed to model pedagogical strategies but there was no pattern language aimed at novice programmers, at least not that the author could locate.

The author was keen to employ the concept of pattern languages in the teaching of novice programmers and asked the Department of Applied Computing's System Administrator to create a simple pattern template in September 2003.

A pattern language for novice programmers

A problem immediately emerged with the work on the pattern language: the author would be teaching final year students only and the control of the Introduction to Programming unit would pass to two of her colleagues. This presented the author with a dilemma. Should she abandon all attempts at fostering the use of pattern language or try to convey to the students the purpose and rationale for the work, despite not teaching them? As described earlier in this work, she chose to continue with her research, with some misgivings (which proved to be justified).

With the pattern template available to students at http://staff.dc.lincoln.ac.uk/~ambird/patterns/ the author gave her colleagues the introductory sheet on the template and the initial entry questionnaire to distribute. The unit was delivered on two campuses, in Hull and Lincoln. Visiting the Lincoln campus required putting aside a whole day so the author relied on colleagues teaching in Lincoln to give out and collect the questionnaires. The rate of return was extremely disappointing.

To sum up, no students used the pattern language template, mainly because they studied concepts in programming first and started programming in Pascal later in the semester, in late November. This does not constitute a criticism of my colleagues, only an observation! It does reinforce the notion that there are as many ways to teach introductory programming, as there are people to teach it. In interviewing the students on the course in Lincoln, the question of what to teach and in what order and the question of a first programming language were all touched upon. Again, different approaches suit different people.

The overall results were very disappointing for this phase of the research. The author feels that to successfully use the pattern template, it needs to be introduced early on alongside actual programming exercises. Again, she envisaged adding sample patterns based on initial student ideas, as a laboratory exercise. There are a number of ways the template could be employed, perhaps across the whole of the first year not just semester A.

Students go on to program either in C or Java and the pattern templates could be sub divide into those for each language and perhaps general problem solving ones. The author intends to continue her interest in the teaching of programming as she moves into teaching other languages such as PHP, for web programming. The author can envisage several more cycles of research based on using pattern languages and developing a pattern language community among new web programmers.

What of ethnography and the TLTP?

The author's reading has ranged fairly widely over topics related to the teaching of programming, from research methodologies (ethnography as part of the hermeneutic framework or view) to a broadly historical view of initiatives in CAL development in the US and the UK such as the Stanford arithmetic project of the 1960s in America through to the UK Government's funding for CAL in higher education, the TLTP. Between those two have been forays into psychology, the history of philosophy, learning theories, child development, language, the workings of the brain and mind and some material now entirely excised from this version.

In re-reading the material, the question occurs to the author, how does all this fit together? (If it does.) Is it meant to? How does knowing what Bacon's Idols of the cave are help the author be a better (contentious word!) teacher of programming? Does the author really need to know about the work of Skinner or Piaget to use a piece of CAL in her teaching?

Were the hours the author spent reading reports written in the 1960s and 1970s about CAL projects, now long gone, wasted? Is it pointless to know how various committees and projects spent the funding allocated to them over twenty-five years ago? The questions are disingenuous: the author feels that her reading around broad topics was never wasted effort. Knowing the history of a range of CAL projects (and some smaller scale projects must have passed beyond even dedicated research effort, with lost reports and long obsolete software) makes any sensible developer or user of CAL alert to the issues around CAL, such as evaluation of the material and its structure, its pedagogical foundations (or lack of them), its genuine effectiveness and measurement of that effectiveness. The author has long noted that the fashion in educational computing was poised to change and research is now emerging that is much generally more sceptical about the benefits of using computers in the classroom.

Knowing some of the philosophical roots of learning theories might be only an exercise in overview but the author feels that her interest in such antecedents justified her wider reading. Making a coherent, unseamed whole out of *everything* she has read is not possible but she would like to feel that her work is informed by an awareness of a range of ideas, issues and debates.

Chapter 8: Conclusion

This chapter has presented no more than an overview of some of the elements of the author's various approaches to one aspect of her professional practice, the task of teaching novice programmers. The question about how to teach novice programmers might be formulated more usefully as, what does not seem to work very well with those learning to program for the first time? The answer to that appears to be, just about everything! Human-centred research is unlikely to produce definitive, unarguably correct answers but it is frustrating to have expended considerable effort for very little return! However, the author believes that the idea of developing a pattern community is one that has a potential and one that is worth another research phase.

Introduction to Chapter 9

This chapter offers, very briefly, some initial conclusions offered by the author, based on the three approaches undertaken by her and described in this work.

Suggestions for the teaching of programming

The author felt it was appropriate to summarise the elements of practice that she felt could be identified from her various efforts around the teaching of programming. She offers these as a short list, with brief comments.

- Offer a collected set of disparate resources and identify early on strategies for employing those resources.
  - For example, books are still liked and used by novice programmers. It would be useful to make distinctions between material and code presented in texts and similar material that can be found on the Internet, focusing on validity and audience.

- Make explicit the distinction between learning to program in a specific language and learning to problem solve and write algorithms.
  - Students could be asked to identify their area of interest and guided to the most appropriate resources. In the author's experience, sample code is easily found but examples of good, non-trivial pseudocode are less widely available.

- Offer some element or resource that supports a constructivist approach and make this a non-optional part of the course. That usually means making it part of the assessment or monitoring process, in some way.
  - That element may be on-line help that is added to via some regular update mechanism (an electronic suggestion box?) or a pattern repository. It might also be the use of an approach such as pair programming.

- Students do not often get to see or use bad code. The author found that debugging error-ridden code in laboratory time was very popular with novice programmers.
    - A library of examples of poor coding might be useful. This could form part of a CAL package that is more than just page turning. It might be useful to automate the evaluation of the tidied up version offered by the student. This removes the need to debate the appropriateness of assessing formative or summative assessments automatically, while still giving students practice in improving a piece of code.

- Acknowledge the affective domain and make explicit the issues around identifying very closely with one's own code.

A view of the author

In 1968, Sackman et al. (1968) noted

> "*There is, in fact, an applied scientific lag in the study of computer programmers and computer programming – a widening and critical lag that threatens the industry and the profession with the great waste that inevitably accompanies the absence of systematic and established methods and findings and their substitution by anecdotal opinion, vested interests and provincialism.*" (Sackman et al. 1969 p 3)

It seems to the author that efforts to remedy that deficit have been problematic, to say the least. To the author, part of the problem seems to be that researchers in programming look at the issues that reside in their areas of expertise or interest. To generalise a little wildly, programmers who research how to support novice programmers write software for graphical editing of code or tracing of code execution. Psychologists look at issues of memory or bias. Those who are interested in the minutiae of coding look at two different types of loop and how their use affects understanding of code.

Is the author any different? She must have undoubtedly focused on what interests her (the human aspects of learning to program) to the detriment of some other aspect – her own particular Idol of the cave! She has tried to make this work address some of the lag described above by taking a wider view, but at the same time has sought to engage personally, in her own practice, with approaches to teaching programming.

Her interest in supporting novice programmers remains a central focus for the author but perhaps that does not emerge strongly enough on a personal level, from her work overall. Therefore the author feels it is worth articulating clearly: her goal in this work was to find ways to help novice programmers develop confidence in their programming and to have the ability to choose from and use all the tools and resources available to them in a considered way.

This last point is a critical one. The author feels that is not enough to present students with a varied and good (however we define 'good') set of resources: novice programmers need to be able to distinguish between resources that serve them well at one point and not at another. The author's work suggest to her that students do have a sense of what they like best as support, which is largely an individual choice, but that developing more sophisticated strategies choosing resources for finding solutions and problem solving in programming is more demanding.

The author chose to focus on the supporting strategies and resources but one area of interest is still that higher level of resource choices and employment. From the author's own understanding of her reading around expertise, what makes someone good at his or her subject is a nose for what seems productive and promising. Expert programmers focus very quickly on the strategies and resources that most closely address the issue at hand. In the author's experience, novice programmers take a very long time to develop that second level of review and choice. They can spend a long time hammering out a solution based on an erroneous assumption or on faulty information, never questioning their choice of resource or strategy.

So, the next stage for the author is to look at offering a broad set of resources (that might once again include some variety or flavour of CAL?) and to look at ways of fostering that more reflective choosing of strategies and employing of resources in novice programmers: making them more expert at being programmers. Perhaps there is room her for some early examination of learning styles, with a cohort of students.

Allied with this will be the use of a pattern language to foster what the author hopes to be a genuinely vibrant, if small-scale, pattern community.

Because this work reflects the author's continuing interest in what it means to teach a programming effectively, any conclusion is at least partly a convenient artifice. The work goes on, the author's ideas develop and change and new approaches are always possible.

Nothing is ever wholly complete: revisions, new ideas and fresh approaches are always possible and the author welcomes the sense of excitement that these possibilities generate for her in her teaching. This thesis is about learning: learning by the author as well as (hopefully!) by her students, past and future.

The work undertaken by the author seems, in review, to have been a process of winnowing out rather than inclusion. The author rules out tutorial CAL as meaningful support for the teaching of programming and offers rather pessimistic views of computer-based support that requires students to engage in dialogue with the tutor via the technology, as in the author's Pascal help system. The conclusion for the author is that students can be notably passive in their learning and alarmingly rigid in their expectations. One approach to this might be to make explicit these issues early on the life of a novice programmer and work on combating the repetitive cycle of programming code from a lab sheet, without ever truly reflecting on what it is he/she is doing and what it is the code is supposed to be doing.

The author believes that her work on a pattern community for novice programmers is one that holds out promise of a more productive, student-centred approach, if not a complete answer to some of the problems and issues outlined in this work. It is the thirds and final approach to the teaching of programming that seems, to the author, to be the most potentially fruitful and one that merits further development, of both the central idea and the research framework around it, from methodology to data gathering tools.

Conclusion to Chapter 9

This chapter has offered a brief summary of the author's suggestions for the teaching of novice programmers and offered a short general conclusion to this work.

Conclusion to this work

The author began with an apparently relatively straightforward question: what sort of CAL can be developed and used so that it supports effectively the teaching and learning of a programming language, for a student new to programming? It seemed to the author at the outset that an answer to this question must be capable of being formulated, even if it was hedged with uncertainties. On closer examination, the question unpacks itself quite dramatically, to encompass theories of learning (how do people learn?), human physiology and psychology (vision, confirmatory bias and affect), theories of personality, ideas and concepts around the field of programming generally (from programming paradigms, environments and languages to individual constructs) and a range of ideas about using computers to support learning, from drill and practice CAL to collaborative learning using a network, in some way. All these elements are a vital part of the question the author began with and part of the answer, even if the answer is, in some aspects, a negative one.

The author feels that she has identified what does not work so effectively – tutorial CAL is too static, and on-line help equally unloved by students! The idea of developing a pattern community is one that seems to the author to hold genuine promise but one that requires further work and explication by the author.
In summary, the author feels that the focus of her effort has shifted during the life of the research but that shift represents her continuing engagement with the question: how best to help novice programmers learn to program? The answer is perhaps not monolithic but many faceted: using computers as supporting devices is essential (no programming without them!) but is not the whole story: the human element (the novice programmers themselves <u>and</u> the tutor) is critical.

Volume 2

Appendix A

Getting started.. simple!

To access Turbo Pascal:

* Log on to Network
* Choose Menu system
* Choose More Applications
* Choose Software Development
* Choose Programming Languages and....
* Choose Procedural Languages
* Choose Turbo Pascal

To set up a new file, press ALT and F together to bring up the file menu.

Move the highlight to New, using the arrow key and press Enter.

You type in your program into the Edit window.

When it is typed in you compile and run it.

Typing in a program is like typing in any text. The difference comes later.

Type in the following program EXACTLY as below.

```
program newprog (input, output);

begin

        writeln ('Hello world')

end.
```

To compile the program press ALT and F9 together.

If your program is OK you get a compile successful message. Press any key to continue.

If your program does not compile you need to de-bug it. When your program has compiled you can then run it.

To run the program press CTRL and F9 together.

The screen doesn't seem to change.... has anything happened?

Press ALT and F5 together.

What you see is the Run window where your program shows the ouput.

You should see

```
Hello world
```

in the Run window..

Press ALT and F5 to go back to the Edit window. You can now re-edit your program so it says something more complicated or you can type in a new program. To type in a new program, choose New from the File menu.

Each time you edit a program you need to re-compile it.

Appendix B

Introduction to Appendix B

This appendix offers an outline definition of positivism and hermeneutics. It also describes very briefly phenomenology, phenomenography, ethnography and ethnomethodology.

Positivism

Carr and Kemmis (1994) note:

> "*'Positivism' is not a systematically elaborated doctrine. Rather, [], it is the name usually associated with the general philosophical outlook which emerged as the most powerful intellectual force in western thought in the second half of the nineteenth century.*" (Carr and Kemmis 1994 p 61)

The nature of positivism is summed up by Melrose (1996): "*Positivists value concrete and factual bodies of technical knowledge and generalisations, arrived at by repeated and confirmatory experimentation and observation.*" (Melrose 1996 p 50) This assumes that there is an external world that exists independently of those who seek to know it. If the world exists independently of those who study it, there must be a clear distinction between the studied world and those doing the studying: there is no blurring of this world/person boundary.

Such clear distinction implies that the subjective views of the person doing the studying do not have any impact on, or relevance to, the world being studied. Different people, presented with events in this external world, should be able to agree upon the nature of those (measurable) events and agree conclusions based upon those measurements.

This is as true for the social sciences as it is for the natural sciences. "*The social world* [in this view] *is very much like the natural world.*" (Usher 1998 (a) p12) We can immediately question this viewpoint for work in social science and educational research: in such human-centred situations, the interactions of the observer and observed are both gross and subtle, observable and hidden, in ways that are not true in a chemistry experiment.

Appendix B **Positivism and hermeneutics**

The positivist view of cause and effect is not an adequate model for understanding human activities:

> "*...meaningful actions are reduced to patterns of behaviour which, [], are assumed to be determined by some external forces so they can be made amenable to conventional scientific explanations.*" (Carr and Kemmis 1994 p 89)

The positivist approach emphasises rationality, determinacy, impersonality (removing/denying the subjective) and prediction: in the real world, events are repeatable. This model of scientific enquiry has, says Usher (1998 a) come to dominate research in both the natural and social sciences. Emphasis is placed on formulating generalisations for both areas of research, perhaps to the detriment of work in social sciences and education.

Hermeneutics

Positivism can be contrasted with the ideas of a hermeneutic/interpretive approach. Such an approach "*assumes that all human interaction is meaningful and hence has to be interpreted and understood within the context of social practices.*" (Usher 1998 (a) p18) The question of context or background is crucial. Those doing the research and those who are being researched are embedded in a rich background of "*assumptions and presuppositions, beliefs and practices...*" (Usher 1998 (a) p19) From this, then, must come an understanding of the researcher as part of that background. Separateness is not possible. (Usher 1998 (a) p 22)

If we accept that we, as researchers, are part of the world we are researching, then we have gained both a valuable insight and a valuable tool. There are pitfalls: the extreme position of reflexivity is to argue that all research in a hermeneutic/interpretive framework is ultimately research about the researcher. All the research activity does is reflect back the researcher's own goals, ideas and needs.

Part of the answer to this is to make explicit the uses of reflexivity and critically examine the reflexive elements present in research, while it is being carried out.

Appendix B — Positivism and hermeneutics

Positivism denies reflexivity: the identity of the observer is both irrelevant and unexplored. In hermeneutics, that identity is both central and should be articulated.

Within the hermeneutic framework, we can identify different approaches, such as phenomenology, phenomenography, ethnography and ethnomethodology. Each of these is briefly described below.

Phenomenology and phenomenography

Phenomenology advocates the study of experience taken at face value. In this paradigm, *s*ubjective consciousness is the prime factor. This consciousness is active, and bestows meaning. A variation of phenomenology is phenomenography:

> "*... phenomenography investigates the qualitatively different ways in which people experience or think about different phenomena.* [] *Phenomenology is concerned with the relations that exist between human beings and the world around them.*" (Marton 1988 p144)

The word phenomenography was coined in 1979 and appeared in print for the first time in 1981. The key idea is that an idea, concept or principle can be understood in a number of different ways.

> "*Phenomenography involves analyzing and categorizing peoples' different ways of understanding or conceiving of a certain phenomenon, as this manifests itself in different discourses; although most generally in a conversation between the phenomenographer and an interview person.*"
> (http://www.ped.gu.se/biorn/phgraph/wild/briefing.html 15/02/2001)

The methodology of phenomenography is unformalised. The idea of conversation as the basis for research is an attractive one but perhaps lacks rigour.

There are issues with regard to the substance of phenomenography:

> "*The interest shown by phenomenographers for methodological problems and the meaning of the procedures they carry out when doing their research has been small, even though examples of the opposite exist, which means that they have, as yet, barely formulated the theoretical bases of their way of doing research; something which in its turn makes it difficult to actually pin down what it is that phenomenographers do.*"
> (http://www.ped.gu.se/biorn/phgraph/wild/briefing.html 15/02/2001)

Ethnography and ethnomethodology

Ethnography has its origins in anthropology, in such works as Malinowski's study of Trobriand Islanders. Ethnography can be defined as concentrating on describing the culture of those being studied rather than seeking to validate or create generalisations. This is not to imply that ethnography is a monolithic approach with no variations of methods or disputes and debates about its tools. Hammersley (1992) argues that ethnography suffers from a "*disabling defect* []*: it is guided by an incoherent conception of its own goals*." (Hammersley 1992 p11)

Nevertheless, we can find clear definitions of the role of the ethnographer:

> "*the ethnographer wants to understand what one has to know, as a member of a particular group, to behave competently as a member of that group. A "good" ethnography describes a cultural reality in such a way that a non-member of the culture could "pass as an insider" if he or she had internalized the cultural features of the particular setting. To a certain extent ethnographers are interested in taxonomizing or categorizing the cultural perceptions in the ethnographic account*." (http://www.phenomenologyonline.com/glossary.cfm?range=e-h 08/05/2003)

Traditional ethnography can be seen to have four key areas: the researcher's role, natural settings, reflexivity and the relationship of ethnography "*to the practice of education*." (Scott 1998 (b) p144)

In ethnography, the researcher can take the role of participant. This participation can be classified as complete, participant-as-observer, observer-as-participant and complete observer. (Scott 1998 (b) pp144 - 145) The question needs to be asked, can a researcher ever be a full participant? In the second case, the researcher participates in order to experience at least some elements of the culture/system they are studying. Researchers using this model are open about their role and have to negotiate their participation. Complete observer is a model where the researcher adopts "*a passive role and concentrate*[s] *on minimising their 'contamination' of the setting*." (Scott 1998 (b) p145)

Hammersley (1992) argues that ethnography's greatest commitment is to description, however the description is defined and bounded. (Most ethnographers acknowledge the idea of multiple views/individual realities present in any social system or event).

Hammersley (1992) argues that such a commitment is misleading: it is not possible to produce neutral descriptions and explanations of those descriptions. The assumptions and judgements that underpin any description and explanation need to be "*made explicit and justified where necessary.*" (Hammersley 1992 p28)

What emphasis should be placed on the validity of ethnographic research? Hammersley (1992) presents a definition of validity: "*An account is valid or true if it represents accurately those features of the phenomena that it is intended to describe, explain or theorise.*" (Hammersley 1992 p 69) This definition does require some unpacking and Hammersley (1992) offers three areas for consideration: plausibility and credibility, the relationship of claim and evidence and, finally, the type of claim made.

More significantly still, the idea of ethnography can be said to embody a telling flaw. The more deeply ethnography engages with the subjects of study, the less sure it is of the conclusions or ideas generated by that study, "... *the more deeply it goes the less complete it is. It is a strange science whose most telling assertions are its most tremulously based...*" (Geertz 1993 p 9)

Scott (1998 b) characterises the idea of reflexivity as "*central to the ethnographic enterprise.*" (Scott 1998 9b0 p153) The key point is that the ethnographic approach does not pretend or suggest that the data collected is utterly uncontaminated by the presence and activities of the researcher:

> "*...ethnography demands a level of introspection both about how the data were collected and the positioning of the researched account in those social and political arrangements that constitute society.*" (Scott 1998 (b) p155)

Ethnomethodology

> "*studies the "methods" that people employ to accomplish or constitute a sense of objective or social reality. The purpose is to elucidate how taken-for-granted or seen-but-unnoticed "rules" lie at the basis of everyday communications and interactions among social actors.*"
> (http://www.phenomenologyonline.com/glossary.cfm?range=e-h 08/05/2003)

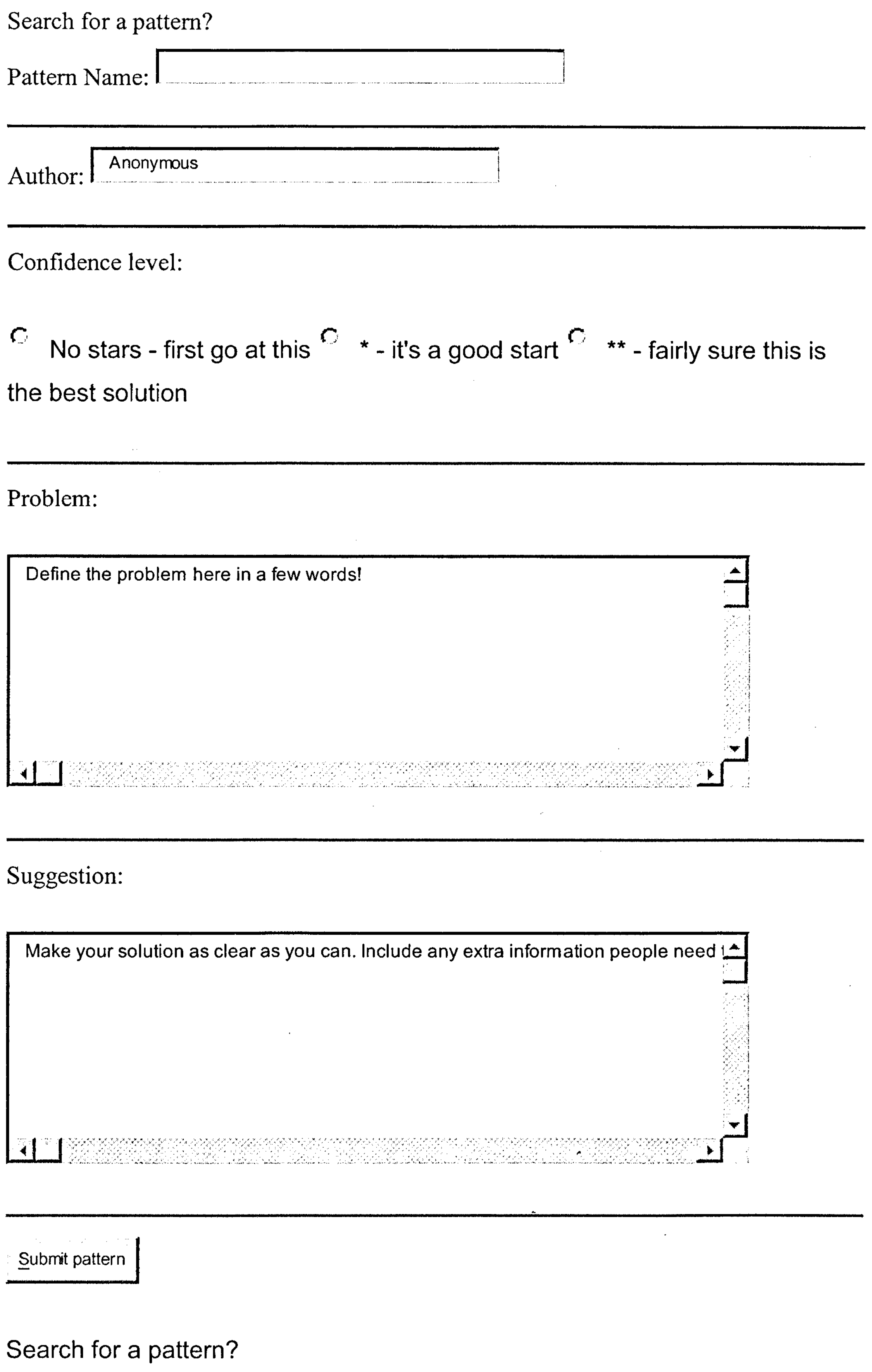
Cohen and Manion (1994) distinguish two types of ethnomethodology: linguistic and situational. Linguistic ethnomethodology focuses on the use of language in everyday situations, whereas situational ethnomethodology looks at the ways people deal with the different and varied social contexts in which they exist. (Cohen and Manion 1994 p 32)

Appendix B: Conclusion

The brief outline given here cannot hope to do justice to the rich complexity of these philosophical issues, whether they are viewed as a set of theories or as a framework for investigating and interpreting human activities, but this chapter has presented some initial, elementary definitions.

Appendix C

Appendix C **Pattern Form**



Search for a pattern?

Pattern Name:

Author: Anonymous

Confidence level:

○ No stars - first go at this ○ * - it's a good start ○ ** - fairly sure this is the best solution

Problem:

Define the problem here in a few words!

Suggestion:

Make your solution as clear as you can. Include any extra information people need t

Submit pattern

Search for a pattern?

Appendix D

Introduction to Appendix D

This appendix will outline the first Pascal programs offered in a number of introductory Pascal texts, chosen at random by the author from those available to her students in the University library in the academic year 1996 – 1997.

Review of selected Pascal texts

One of the key sources for a programming beginner is a book or books. Books on programming languages are numerous and varied. A new language or a new paradigm will inevitably generate a flurry of new titles from 'How to get started with ...' to 'Expert use of...'

The Internet also provides a wide range of resources for those working with programming languages: such resources are reviewed later in this chapter. This section concentrates on as handful of introductory Pascal texts. Such a review cannot be exhaustive: the author has chosen a number of introductory texts at random, without regard for their date of publication.

How should we classify the different ways different authors tackle the question of writing about Pascal? One measure might be to examine the opening paragraphs of the first chapter and/or the introduction. Another might be to examine the first piece of Pascal code presented to the reader. The author proposes to use these (fairly arbitrary) measures in discussing the selected texts.

Before looking at Pascal books it is worth noting that some educators have reservations about the pedagogical value of such texts. McKeown and Farrell state:

> "*Programming texts appear to pay little or no attention to instructional design principles. Texts merely present programming concepts in a conceptual order. Simple principles are first presented and later built upon as more abstract principles are introduced. There appears to be little attention paid to problem-solving principles or debugging issues, areas that have consistently proven to be major stumbling blocks for novice programmers.*"
> (http://homepages.dsu.edu/mckeownj/CPCCCSCpaper.html 04/12/2000)

Appendix D **Review of a selection of introductory Pascal texts**

In *Pascal programming: a beginner's guide to computers and programming*, Hawksley (1988) opens the introduction to the book by describing learning to program as being analogous to learning to play a musical instrument. He extends this analogy and describes how a basic knowledge of the physical construction of, say, a piano, can improve the learner's playing of that instrument. So, he says, with programmers: they should have a basic understanding of the general elements of a computer.

Hawksley (1988) then goes on to describe the pedagogical structure of the book. Key skills in music can be acquired and improved by practice and programmers need to learn the ways a language's constructions can be used: "*With this in mind, the later chapters of part 1 introduce fundamental constructions of the Pascal programming language together with short examples to illustrate their use.*" (Hawksley 1988 p 2)

In Chapter 5, Hawksley (1988) tackles Pascal program construction. The first Pascal program appears on page 34:

```
program sayhello (output);
        begin
        write ('hello')
        end.
```

(Hawksley 1988 p 34)

In Brown's *Pascal from BASIC* (1982) the preface states, "*This book assumes that you are a reasonably competent BASIC programmer and that you want to learn Pascal.*" (Brown 1982 Preface) The first example program Brown (1982) describes is a BASIC program to find the average of *n* numbers.

Brown (1982) then shows the Pascal equivalent:

```
program average (input, output);
{ --- find the average of N numbers ---}

var
n:      integer;
        k:      integer;
s:      real;
x:      real;

begin
        read (n);
        s: = 0;
for u: = 1 to n do
        begin
                read (x);
                s: = s+x;
        end;
                writeln ('Average = ', s/n);
        end.
```

(Brown 1982 p 2)

Brown then goes to discuss the key differences between the BASIC version of this activity, with its REM and use of line numbering and the Pascal version.

In Farmer's *The Intensive Pascal Course*, (1984) the author opens with the statement:

> "*This book assumes that the reader already has experience of programming in a high-level language (e.g. Algol 60, BASIC, FORTRAN, etc) and therefore has a 'feeling' for the structure of a program.*" (Farmer 1984 p 4)

Farmer then goes on to discuss the notation and characteristics of Pascal and simple data types. Interestingly, at the end of the first chapter, Farmer has some pragmatic advice about learning one text editor under one operating system very thoroughly and also the use of one Pascal compiler, so that the user becomes familiar with its diagnostics. (Farmer 1984 pp 10-11)

In *Simple Pascal*, McGregor and Watt (1981) open their introduction with a short program and then discuss each line of the program in turn, describing the role of reserved words such as 'program' and the creation and formatting of output using quotation marks and expressions. McGregor and Watt (1981) note:

> "*in order to make it easy for a program to be processed by a machine, the grammatical rules in PASCAL are very strict. You will find, for example, that you must put commas and semicolons in just the right places.*" (McGregor and Watt 1981 p 3)

In *Pascal for Electronic Engineers* Attikiouzel (1990) introduces concepts such as algorithms, different programming languages and software tools before giving a broad view of Pascal. He begins with an outline of the relationships between data types and lists Pascal's special symbols, much as <> := and *. Also listed are Pascal identifiers such as false, get, readln and so on (Attikiouzel 1990 p 6)

Attikiouzel (1990) then moves on to describe the structure of a Pascal program and the role of comments. A complete program is given on pages 10 to 11. This program converts a decimal number input by the user to binary. The program uses the const statement as well as declaring three variables, with a REPEAT ... UNTIL loop inside the ELSE of an IF...THEN...ELSE. This is a rather more meaningful, even challenging, piece of code than the earlier examples reviewed.

In the next chapter, Attikiouzel (1990) introduces the concepts of constants and variables, output and input from a program and data types such as real and integer.

Appendix D **Review of a selection of introductory Pascal texts**

In Eisenbach and Sadler (1981) the first Pascal program is:

```
PROGRAM EVENINGALL;
BEGIN
WRITE ('HELLO');
WRITE ('.' , 'HELLO HELLO');
WRITE (* MOVES TO A NEW LINE*);
WRITE ('AND WHAT DO WE HAVE HERE?');
WRITE ('HELLO HELLO HELLO')
END.
```

(Eisenbach and Sadler 1981 p 12)
(Note: the line numbering in the original has been omitted.)

Beaumont (1989) introduces the assignment statement in the first Pascal program:

```
PROGRAM ADDUP;

VAR first number; second number; result: REAL;

BEGIN
First Number:=       2,3;
Second Number:= 4,1;
Result:=       First Number + Second Number
END.
```

(Beaumont 1989 p 83)

This program appears *after* the introduction of data types such as integer, real, char, features such as operators for arithmetic expressions, mod and div and input and output.

What can we conclude from what is a very brief review of a very small number of Pascal texts? We can see that the preferred approach is generally to present a very short program, usually one that outputs a short message to the screen, and to broaden the discussion from this point onwards, adding new features and increasing the complexity of the programs offered. We could term this the 'Hello world' approach, as the convention is that the output program delivers a message something like Hello or Hello world:

> "*Before you start any programs you need to have an idea of what you want to do.* []... *the "Hello world" program is famous in many way as it's generally a 1 line program (or as few as possible)...*" (http://www.xcalibur.co.uk/training /programming/start.html 03/05/2000)

The converse approach is to present a fairly complex program that does some non-trivial processing and decompose it in detail, discussing each of the elements in turn.

Wirth notes: "*Programming is usually taught by examples. Experience shows that the success of a programming course critically depends on the choice of these examples.*" (http://www/acm.org/classics/dec95 20/10/2000) Wirth argues that these examples too often are chosen to show what the computer can do rather than embodying a more generally applicable concept.

Another weakness, he notes, is the way in which example code is presented as finished:

> "*As a consequence of these teaching methods the student obtains the impression that programming consists mainly of mastering a language* [] *and relying on one's intuition to somehow transform ideas into finished programs.*" (http://www.acm.org/classics/dec95.html 20/10/2000)

It should be noted that the original article was published in April 1971 but Wirth's aside on programming languages is as true as it was thirty years ago.

Wirth goes on to argue that new programmers should be taught design and program construction and that example programs should demonstrate a development of idea and code.

Some concepts prove difficult for students across a range of structured languages, including the idea of parameter passing and the scope of variables. One Internet message describes such concepts being simplified by presenting the concept of global variables only. (http://www/uni-giessen.de/hr2/tex/more.info/info/mailarchiv/litprog.1/msg00507.html 03/05/2000) It is probably not an adequate solution, when encountering areas of difficulty, to ignore those areas entirely!

Appendix D: Conclusion

What can be gleaned from looking at the texts on Pascal is that most authors take a features-driven approach. They describe constructs and ways of doing a particular task in the language of choice and note any quirks or peculiarities in the language. It is hard to discern any pedagogical structure in these texts, although perhaps that is unfair. These books do not make exaggerated educational claims and, on the whole, the reader is left to supply his or her own pedagogical framework.

Appendix E

Introduction to Appendix E

This appendix reviews in outline some of the work done in the comparison of expert and novice programmers, in terms of their approach to implementing algorithms, solving problems and mental models.

Experts versus novices

One aspect of research in psychology that might initially seem very promising is that of exploring and comparing the different models and strategies developed and employed by expert and novice programmers. If we could quantify the ways in which an expert approaches the coding of a new program or comprehends a program, then we might identify strategies (or as noted earlier, models) that can be explicitly taught to a novice.

Learning programming has been described (by Arzarello et al. 1993) as a cognitive apprenticeship. Arzarello et al. (1993) define such an apprenticeship as a process of:

> "*building new (cognitive and metacognitive) knowledge and abilities, and* [] *a process of revision of already acquired knowledge, and of* [] *organization of a new framework in which old and new knowledge must be compared and integrated.*" (Arzarello et al. 1993 p 284)

Arzarello et al. (1993) make what they say is an important distinction between the cognitive and practical apprenticeships. Practical elements of learning to program are fostered by appropriate teaching strategies, such as scaffolding. The cognitive aspects are brought into being through the observation of the expert by the novice, where the expert makes external and knowable his/her cognitive processes.

Part of the role of the expert is to identify areas of cognitive conflict for the novice and help the novice develop ways of meeting these difficulties. The learning of programming then, is more than learning the syntax and semantics of a language: it is about adopting ways of thinking about programming problems and recognising the existence of better, more productive strategies for certain situations or problems and applying them in a considered way.

Winslow (1996) has a more positive view than some authors of what psychology might offer:

> "*Can we turn novices into experts in a four year undergraduate program? If so, how? If not, what is the best we can do? While every teacher has his/her own opinion on these questions, psychological studies over the last twenty years have started to furnish scientific answers.*" (Winslow 1996 p 17)

Winslow (1996) identifies some of the results that research about novice programmers turns out repeatedly. Novices understand syntax and semantics but cannot move from the individual elements to whole programs. Even where students do understand a problem, the act of coding that understanding by translating an algorithm into a program seems beyond them.

Winslow (1996) sums up the salient points that research seems to make again and again. Novices lack adequate mental models of programming. Novices do not have a deep understanding of the subject and often fail to apply the knowledge that they do have, even when it is relevant. They rely on general (inappropriate?) strategies rather than generating more specific solutions to a problem. Novices look first to use constructs and code line-by-line rather than thinking in overall terms.

Davies (1993) makes a central point about novice versus expert programmers. It is assumed that enabling novices to become good programmers (and by extension, eventually experts) involves the transmission of programming knowledge. Davies says that studies suggest, "*novice programmers display strategic as opposed to knowledge-based difficulties.*" (Davies 1993 p 262)

Winslow (1996) sums up the research on expert programmers:

> "*On the other hand, similar studies have found that experts:*
> *Have many mental models and* [] *mix them in an opportunistic way.*
> *Have a deep knowledge of their subject which is* [] *many layered* []
> *Apply everything they know.*" (Winslow 1996 p 18)

Experts work forward from the problem's givens, are good at spotting problems that have partial solutions or solutions already in existence, concentrate on data structures in code and use algorithms rather than focusing on a particular language. Finally, experts have better tactical skills. It is unrealistic to expect student programmers to achieve that level of expertise but should teachers should aim to equip students to adopt some of the strategies demonstrated in expert programming practice. For Winslow (1996) that involves being able to apply general problem solving approaches to a specific problem, distinguish between task specific and general rules (and combine them) and "*increase ability by practice.*"(Winslow 1996 p 19)

Blum (1996) notes, "We *know that experts have considerable knowledge.*" (Blum 1996 p 122) This rather trite and obvious statement can be unpacked quite usefully, however. Glaser et al. (1988) note the following characteristics of expertise:

> "[] *Experts perceive large meaningful patterns in their domain...*
> *Experts are fast; they are faster than novices at performing the skills of their domain, and they quickly solve problems with little error...* []
> *Experts see and represent a problem in their domain at a deeper (more principled) level than novices; novices tend to represent a problem at a superficial level...* []
> *Experts have strong self-monitoring skills ... "* (Glaser et al. 1988 pp xvii - xx)

What is required, argues Winslow (1996), is the learning of problem solving. By problem solving, what Winslow (1996) appears to mean is the whole sequence of understanding the problem domain, formulating a solution and translating the solution into code, which needs to be tested and debugged. Winslow identifies a critical weakness in this cycle:

> "*Every teacher has had troubles with students who do not read the problem carefully, or read it in a superficial manner. Mathematicians consider "word problems" one of the most difficult things to teach, yet programming is almost all word problems.*" (Winslow 1996 p 20)

Once students have understood the problem, the second major difficulty is the combining of syntactically valid statements into a working program.

Students know the individual constructs, such as IF...THEN or loops but do not see how to meld these constructs into meaningful code:

> "*The difficulty is knowing where and how to combine statements to generate the desired result. As noted earlier, experts think in terms of algorithms and not programs. The actual translation of an algorithm into a working program is a task, not a problem.*" (Winslow 1996 p 20)

When the program is written the student must test and debug the code: another point of difficulty:

> "*Studies have shown that there is very little correspondence between the ability to write a program and the ability to read one. Both need to be taught along with some basic test and debugging strategies.*" (Winslow 1996 p 21)

So, what pedagogical strategies can be identified to address the issues identified here? Winslow (1996) makes some suggestions based on the work of Linn and Dalbey (1989, cited in Winslow 1996). Novice programmers should learn the syntax and semantics of a language one element at a time. The learning of a language's features needs to be combined with problem solving and program design skills, which need to be taught explicitly.

Winslow concludes:

> "*Psychological studies of expertise in general and computer programming expertise in particular show that turning a novice into an expert is impossible in a four year program. Competence, however, is possible.*" (Winslow 1996 p 21)

Davies (1993) argued that expert programmers have undergone a process of "*knowledge restructuring*". (Davies 1993 p 238) and from that restructuring comes the different strategies employed by expert programmers. One of the strategies identified is that of 'chunking' a program during comprehending the code. Davies cites the work of Widowski and Eyforth (1986) who extended the work of Chase and Simon (1973).

Widowski and Eyforth's (1986) work confirmed that expert programmers' comprehension worsened when programs were presented randomly to them whereas novice programmers understanding was "*affected little.*" (Davies 1993 p 246) Davies reviews extensively the work done by various researchers on expert programmers strategies but concludes, "*...no clear picture emerges from this work*" (Davies 1993 p 263).

Davies describes a study that recorded the eye fixations of programmers asked to study a binary search program, work done by Crosby and Stelorsky (1989). The researchers found no correlation between patterns of fixation and expertise: in other worlds there is no single, expert way to read and scan a piece of code.

Davies also reviews Rist's model of focal expansion (1986 and onwards) He notes:

> "*Rist ... has proposed a model of scheme creation in programming that describes the way in which plan-based knowledge structures are transformed into programs and accounts for the differences in generation strategy that have been observed to be associated with differences in expertise.*" (Davies 1993 p 250)

As the theory name suggests, Rist (1986) proposes that single lines of code, each containing a central action to be performed, are combined to form a plan. Rist's (1986) model makes two claims about the effect of knowledge on behaviour. Firstly, if knowledge does guide program design, then that design will be top-down. If no knowledge applies to the program then the "*solution will be created by focal expansion* [or] *a bottom-up process.*" (Davies 1993 p 250)

Davies goes on to discuss extensions of Rist's work by Green, Bellamy and Parker (1987) Green et al. (1987) proposed:

> "*a comprehensive model of coding behaviour which highlights those features of the device, task, interaction medium and user knowledge that contribute to the use and development of particular forms of programming strategy.*" (Davies 1993 p 251)

Davies (1993) notes, "*Green, Bellamy and Parker introduce the "parsing-gnisrap" cycle to describe the cyclical alteration between code generation and recomprehension.*" (Davies 1993 p 253)

In this model, code is externalised (presumably typed into an editor) when the programmer can no longer hold in working memory the program structure: this is the gnisrap element. When the code is externalised, it needs to be parsed or re-understood at intervals by the programmer. This model, notes Davies (1993), makes different predictions about the order in which a program is generated, compared to Rest's model. In the parsing-gnisrap model, programs are accretions of code segments that are externalised (typed in) when the programmer can no longer hold the full code in his/her mind. Looking at the accretions of code requires the programmer to reconstruct the original plan of the program.

The medium in which the externalisation takes place affects the programmer's strategy. An obvious difference is that between the use of pen and paper and text editor. The language chosen (Pascal, Basic etc) also affects the strategies employed by the programmer. Some languages support a non-linear approach better than others.

In the parsing-gnisrap model, there may be no differences predicted between expert and novice programmers. However, it should be noted, "*the data used to support the parsing-gnisrap model was collected from expert programmers.*" (Davies 1993 p 256)

Kahney (1983) summarises the problem solving behaviour of novice programmers as possibly consisting of method finding (recall of similar problems), evaluation of possible methods, ways in which the chosen method can be framed in a program, coding, evaluation of code and testing. Kahney (1983) concludes that is possible to distinguish between talented novices and the rest. Talented novices are more expert-like in their strategies where the "*average novice is quite unlike the expert.*" (Kahney 1983 p 139)

Petre and Blackwell (1999) investigated the idea that programmers (particularly expert programmers) visualise or image/imagine the code or program structure and that visual programming languages must be closer to the human internal representation of code than textual languages such as Pascal.

Petre and Blackwell (1999) describe two studies in their paper. One study concentrated on eliciting comments about mental imagery from expert programmers. The other was a questionnaire-based study of 200 users of a visual language, LabVIEW.

The programmers surveyed in the first study used a variety of ways to describe what they were thinking. The imagery might be of symbols, of an internal conversation or auditory. Petre and Blackwell (1999) note that visual images appeared in all the descriptions given. The images were of mind machines or of the flow of data or a combination of both. Programmers also described idea of landscapes, with the solution forming the landscape over which the programmer could 'fly'.

Some common elements emerged, such as the dynamic/static interplay present in the images. The programmers also were able to focus on elements of their images to switch their attention to other areas. They could also change the quality of their focus: look at details or the general image. The programmers' images were flexible and variable and could take account of areas of the unknown or incomplete. Also, the images were perceived as being multi-dimensional, and to be multiple themselves.

The second study looked at users of LabVIEW, a visual programming language that emphasises data flow. Petre and Blackwell (1999) seem reluctant to draw strong conclusions from their study, leaving the question of whether there are broadly applicable programming strategies that can be inferred from an investigation of expert programmers' mental imagery.

Another, learner-centred approach is described by Davy and Jenkins (1999). They describe the background to the project, which focused on applying elements of research in teaching programming to undergraduate classes. The students started learning programming through Pascal, then moved on to C++ in year two. Aspects of the object-oriented paradigm were introduced toward the end of the C++ material: "*This structure proved largely unsuccessful, despite much effort from staff in terms of delivery and support.*" (Davy and Jenkins 1999 p 5)

Interestingly, Davy and Jenkins (1999) note, "*A consistent model of student learning appears a fundamental of effective teaching.*" (Davy and Jenkins 1999 p 6)

Davy and Jenkins (1999) describe the use of a model described by Laurillard, itself a synthesis of various research approaches. Laurillard's model "*emphasises students'* *conceptions and can be described by a 'conversational framework' with four components.*" (Davy and Jenkins 1999 p 6)

These four components are: discursive, interactive, adaptive and reflective. Davy and Jenkins (1990 describe the discursive component as a:

> "*dialogue in which learning goals are agreed, the conceptions of both students and tutor become accessible to each other, and students receive feedback on their descriptions of their concepts.*" (Davy and Jenkins 1999 p 6)

In the interactive component students receive both intrinsic and extrinsic feedback, from the task performance and from tutor evaluation of their work: "*In the* adaptive *component the tutor alters the focus of continuing dialogue and tasks on the basis of reflection on the discursive component.*" (Davy and Jenkins 1999 p 6)

The reflective component is where students use their learning experiences to develop both their understanding and their task performance.

To use this model, changes were made to the organisation of teaching programming: "*The largest innovation was the weekly tutorials, in groups of about 15.*" (Davy and Jenkins 1999 p 6) These group tutorials were designed to support the discursive elements of Laurillard's model, with groups streamed by programming experience.

Adaptive feedback was provided in the work on exercises throughout the teaching period:

> "*Explicit reflection on the students' own activities was encouraged in the first assignment* [] *which required students to evaluate a range of motivations for learning to program. The exercise was completed after an interactive lecture where students had substantial opportunity to express their views.*" (Davy and Jenkins 1999 p 7)

Davy and Jenkins (1999) conclude:

> "*Rigorous evaluation of these innovations is impossible, as the many simultaneous changes preclude proper controls. Even where clear evidence of improvement can be seen, it is hard to establish cause and effect conclusively.*" (Davy and Jenkins 1999 p 7)

Nevertheless, they present raw marks for the new style programming unit and its predecessor and state that marks show a "*significant upward shift for most students.*" (Davy and Jenkins 1999 p 8)

Appendix E: Conclusion

The comparison of expert programmers' strategies with those of novice programmers does seem initially very promising but it is quickly becomes obvious that modelling explicitly what it is expert programmers do, that could be conveyed to novices as 'winning strategies' is not a trivial task. It is a potentially useful but not definitive answer to improving the programming of novices.

Appendix F

Introduction to Appendix F

This appendix examines the Stanford project of the early 1960s. It could be argued that CAL, in more theoretical forms, began even earlier but the Stanford project was a well-funded and well-managed project that foreshadowed many the recurring themes in CAL research over the next thirty years. It also discusses briefly the two projects set up in 1971 by the NSF (National Science Foundation of America), PLATO and TICCIT.

The Stanford Arithmetic project

Computer Assisted Instruction at Stanford began with a program of research in early 1963. In 1964, Stanford won a contract from the US Office of Education to set up a computer-assisted learning laboratory at a state elementary school. After initial trials with small groups in 1963 and 1964, the Stanford project's mathematical material was further developed and used with three schools during the school year 1965-1966. In the 1966-1967 school year, the numbers of children using/testing the software/hardware had grown from a few groups of four and five to over 100 pupils.

The arithmetic drill-and-practice material being developed alongside the mathematical logic programs was also more widely trialled during 1966-1967. The program initially centred on a particular school, with an initial cohort of 288 students at the start of the project, rising as other classes at the school were offered use of the teletype facilities. For the material of the project:

> "*emphasis was placed on arithmetic skills for two reasons: first, because these skills continue to represent a large portion of the elementary school mathematics curriculum; and second, the performance level on these skills of students in traditional or modern mathematics has not been at a fully satisfactory level.*" (Suppes et al. 1968 p 20)

Concepts covered included: addition, subtraction, multiplication, addition with carrying, money (equivalence) multiplication tables, division and long division. Time spent on each block of material ranged from 3 days to 12 days.

Within each block (or concept) were five levels of difficulty:

> "*On the first day of a new block, every member of class was given the same lesson. This lesson was of average difficulty (level 3). Those students who scored between 60% and 79% were given a third level lesson the following day; those who scored above 79% were given a lesson on the next higher level (level 4); those who failed to score at least 60% were given a simpler lesson on a lower level (level 2). This procedure was followed throughout a concept block...*" (Suppes et al. 1968 p 26)

Monitoring of students was a important part of the project and teachers were given outline reports, sharing student activity within the software (problems attempted at each level, lessons missed). There were, given the newness of the technology, a number of support issues: technically competent staff were required to create, maintain, update and support such leading edge technology.

Some of the log of the runs is worth quoting from:

> "*One boy should have had a second day drill but got drill 501013. I sent him back and completed the drill myself. Students before and after him got the correct drills for the second day. The machine stopped when a boy was working. I sent him back and as a result he got two time outs when the machine started again. The timing was also wrong. After lunch, the boy who had got the wrong drill was given the correct one, but the machine stopped and printed garbage...*" (Suppes et al. 1968 p 49)

The list and catalogue of problems continues, with time outs, problems with printers, system shut downs and pupil difficulties with the material. Other problems centred on the teachers' use of the teletypes, either forgetting the routines for use or re-start or not time-sharing the teletype efficiently, leaving it standing idle (Suppes et al. 1968 pp 58 - 72).

The Stanford team distributed questionnaires to students, parents and teachers. The questionnaires are described as open-ended, as the respondents had to fill in blanks or give a free-text answer. For students, the questionnaire was anonymous. It was filled in by 149 students across a range of ages. Suppes reports that 71 per cent of students liked the drill and test approach; 72 per cent of the students liked the teletype machines. The questionnaire also explored children's attitude to school and found that 42 per cent of answers indicated a more positive attitude towards other subjects (not involving the teletype).

With regard to the operational problems of the system, the teachers were critical and negative whereas the children "*did not resent errors made in their drills and did not regard operational problems as a major factor in their own evaluation of the program.*" (Suppes et al. 1968 p 97)

The questionnaire indicated a need for clearer explanation and communication between project staff and the schools in which the project ran (Suppes et al. 1968 p100):

> "*The majority of parents reacted favourably to the teletype project, but their answers reflected a need for better communication.... especially concerning the purpose of the drills and date summaries. When asked how they felt about having their children work on the teletype . . . 57 of the 101 parents who replied said that they felt it was "good" or "worthwhile", and only two parents were strongly displeased with the project.*" (Suppes et al. 1968 p 120)

Even at this early stage, parents were expressing concern in two areas: integration of the teletype drills into the classroom experience and the reduction of class-teacher contact time. The authors see both these issues raised in comments on the questionnaires, as products of poor communication about the project and its purpose and nature.

The drills were designed to rehearse traditionally taught skills but both pupils and parents seemed hazy about this. The drills also did not replace teacher time but again, this does not seem to have been clear to parents, examining the general comments made (Suppes et al. 1968 pp 134-137). The teacher responses to the project tended to focus on the potential benefits and actual problems of the projects. Suppes et al. (1968) note:

> "*The teachers were convinced of the potentiality of computer-assisted drills as a supplement to regular classroom instruction. They pointed out at least five areas in which they felt such a program would be especially beneficial. [They] felt that the drill program must be improved before it can become a practical and valuable addition to the curriculum.* [They also] *questioned the appropriateness of the computer-based environment for some children...*" (Suppes et al. 1968 p 138)

Suppes et al. (1968) discuss in detail research in the teaching of arithmetic and the performance models for addition, subtraction, multiplication, fractions and division

(Suppes et al. 1968 pp 155-276). "*One of the major tasks of the project has been the coding, inputting, and debugging of the drills themselves.*" (Suppes et al. 1968 p 325)

What the authors found was that an initially well-tailored language supporting particular hardware or application development would grow increasingly sophisticated until its usability was overwhelmed by the added-on features. The authors note that this happened a number of times, especially where the added features were very narrow in scope and tended to make the new version much less general purpose. The answer was to build a general-purpose language, which raised a number of design issues:

> "*Several fundamental properties emerged as goals in the design of the proposed programming system. The most basic of these* [was] *the separation of instructional procedure from lesson material.* [] *The language should facilitate* [] *experimental modifications of the format or timing of problem presentation, or even changes in the type of terminal used, without the need for time-consuming and costly re-coding...*" (Suppes et al. 1968 p 301)

The project developed Teacher-Student ALGOL, a language designed with the above concept in mind that also was accessible to non-programmers: "*The aim was to furnish generalized power for handling complex information structures in an on-line, real-time environment, while maintaining the procedural framework of ALGOL.*" (Suppes et al. 1968 p 303)

For a detailed discussion of the language features see Suppes et al. pp 303-324. It is interesting to note that features of structured programming such as IF...THEN...ELSE and FOR... loops were being used, as were procedures, assignment statements and variables.

In 1968, the Stanford authors took an optimistic view the use of CAL: "*There are at least four major aspects of computer-assisted instruction that seem to offer great potentiality for education at all levels.*" (Suppes et al. 1968 p 3) The four aspects were identified as: the ability of computerised instruction to deal with students of varying ability and different levels of knowledge, the individual attention (or a satisfactory impression of it) that can be given to a student, the management of some of the learning process, giving human teachers more time away from administrative tasks and finally, the flow of information about a student's performance, to teachers, researchers and administrators.

The Stanford authors identified all of these as potentially major advantages of computerised learning. Yet they also identified problems, springing from their own experience with the Stanford arithmetic project: the reliability of the computers, the constraints and demands of curriculum design for a new medium, the question of capturing and maintaining student interest and the problem of cost. The hardware used was relatively expensive: certainly schools would not have had access to such equipment without the research project.

Display devices are one example of the limitations of hardware at that time. Early display devices for the CAI used microfilm source material. The display device used more widely was a cathode ray tube device and it is worth quoting the description:

> "*It can display points of light in an area 10 inches high by 10 inches wide. There are 1024 possible positions on the horizontal and vertical axes. In addition to individual points, there are 120 prearranged characters which may be displayed in five different sizes. It is also possible to display vectors by simply identifying the end points. A typewriter keyboard is attached to the scope and may be used to send information from the student to the computer. An audio system designed by Westinghouse Corporation can play pre-recorded messages to the user, through individual speakers in each student booth. The messages are recorded on magnetic tape 6 inches wide; two tape transports may be assigned to each of the six student stations. Each transport has a capacity of about 17 minutes, which can be used in any combination, from one message of 17 minutes in length to 1020 messages of one second each. The random-access time to any stored message is approximately one second. The main memories of the central computer, a PDP-1, designed by Digital Equipment Corporation, are a 32,000-word core, and a 4,000-word core which can be interchanged with any of 32 bands of a magnetic drum on files stored on an IBM 1301 disk.*" (Suppes et al. 1968 p 9)

From the Stanford project it is possible to identify a number of themes that can be seen in future projects. The first is that of the optimistic tone of the researchers: obviously, no researcher develops and trials a system with a wholly negative or disappointing result as a potential goal but in this, as in other projects, the review of the project is perhaps over-optimistic, tending to gloss over some of the problems and emphasising the potential benefits. As with other projects in the future, evaluation of the outcomes is done through questionnaires.

The subjective opinions of participants are quoted in support of the project's success (with some concession made to problems of communicating the purpose and goals of the project). These are not criticisms of the Stanford project in particular, but observations that can be repeated for many other projects, both large and small, over the subsequent years.

Pedagogical aspects of CAL use in the classroom is also a theme that features strongly in future CAL research: again and again, authors seek to develop material that will most effectively support the learner and improve learning outcomes: again and again, students, and even teachers/lecturers, seem hazy about the actual measurable benefits of using that CAL at that time. At its crudest, the question is: does a students who uses this CAL know more and/or do better on tests and exams than a student who has not had access to the CAL? Teachers' responses to the arithmetic test and drill routines devised by the Stanford researchers were to call for improvement in the type of material, while the researchers felt that improvements in the instructional design were also required.

The technical problems encountered by the Stanford project are also echoed in future projects: the unreliability or limitations of hardware and software is one example. With advances in display technology and the rise of the PC (Personal Computer) some of the obstacles to interacting with the computer have been removed but no system can ever be said to work perfectly all the time. Where CAL is installed on a network, students may have problems logging on or the software might run very slowly, due to high demand on the network.

The student response to the CAL is another example of a potentially difficult area. The rise of the PC means that, in Britain, America and Europe, students of almost any age are likely to have some acquaintance with computers and familiarity with the use of computers is not an issue, as it might have been for the pupils using the Stanford teletypes. Since 1995 IT has been part of the National Curriculum in England and Wales: in the 1995 document on IT within the curriculum, pupils as young as five to seven years were to be given IT experience, with pupils at eleven being "*critical and largely autonomous users of IT*" (DFE 1995 p 4)

The survey of information technology provision in schools also reveals the extensive financial investment made in computing facilities in both secondary and junior schools. In junior schools "*The average number of computers per pupil was 19 in 1995 - 96, compared with 23 in 1993 - 94.*" (DFEE 1997 p 2) and in secondary schools, for 1995 - 96, the average number of pupils per computer was 9.

For junior schools, one fifth of schools reported that IT made a substantial contribution to teaching and learning; for secondary schools it was one third. These figures will, of course, hide wide variations in the quality of the hardware and software used and the teaching practices into which these elements are embedded. Nevertheless, it is true to say that pupils are experiencing IT at all levels throughout their school career. With such familiarity goes a certain sophistication and students at all levels are likely to demand a well-designed interface for the CAL they use: anything less is likely to be heavily criticised or rejected totally.

TICCIT and PLATO

The NSF decided, in view of the waning energies of CAL initiatives at the end of the 1960s in America, to fund two major projects in this area. The Foundation invested $10 million over five years, from 1971 onwards.

TICCIT (Time-shared Interactive Computer Controlled Television) was run by the MITRE company. MITRE was to develop the hardware and software to deliver course materials, developed by Brigham Young University.

Appendix F **The Stanford Project, PLATO and TICCIT**

The key concept was to develop a complete system of instruction rather than one to act as a supplement to traditional teaching and one that would reach and be appropriate for the largest possible audience:

> "*One of the main characteristics of this approach* [was] *its factory-like production of course material.* [] *...the TICCIT premise* [was] *that the effectiveness of a particular learning strategy* [was] *independent of subject-matter.*" (O'Shea and Self 1983 p 87)

TICCIT was to be used at two community colleges and to offer courses in mathematics and English composition. TICCIT was controlled by the learner, who could use various buttons to look at the objectives for the course segment he or she was working on. There were a total of nine buttons, allowing the user to look at the MAP of the segment, and to ask for HARD or EASY material. (For a graphic representation see O'Shea and Self (1983) p 88) TICCIT hardware (two minicomputers) supported up to 128 terminals, each with a keyboard, colour television set, a loudspeaker and a light pen. The system used colour videotapes.

A report on TICCIT was produced in draft form in 1977. The overall response at the colleges that used TICCIT appears to have been generally negative, with some evidence that students failed to complete TICCIT courses in greater numbers than those taking conventional courses. Staff were also "*lukewarm*" (O'Shea and Self 1983 p 92) towards the new technology and the report concluded that the potential of CAI has been demonstrated, rather than definitively confirmed.

The PLATO (Programmed Logic for Automatic Teaching Operation) project was based at the University of Illinois. O'Shea and Self note:

> "*The PLATO system has a long history: from 1960 one-terminal PLATO I to the NSF-funded PLATO IV with about 950 terminals located at about 140 sites and about 8000 hours of instructional material contributed by over 3000 authors.*" (O'Shea and Self 1983 p 93)

Aims of PLATO IV included the development of new materials for the new computer-based education network and the conclusive demonstration of the feasibility, adaptability and educational appropriateness of the system and its associated materials:

> "*PLATO is a timesharing system. (It was, in fact, one of the first timesharing systems to be operated in public.) Both courseware authors and their students use the same high-resolution graphics display terminals, which are connected to a central mainframe. A special-purpose programming language called TUTOR is used to write educational software. Throughout the 1960's, PLATO remained a small system, supporting only a single classroom of terminals. About 1972, PLATO began a transition to a new generation of mainframes that would eventually support up to one thousand users simultaneously.*" (http://thinkofit.com/plato/dwplato.htm 14/09/2000)

The key difference with PLATO was the provision of a courseware authoring language so that teachers and developers could write materials for PLATO with no particular restrictions on style or approach. This obviously generates academic freedom but leads to material of widely varying quality (a feature seen in other large scale CAL projects)

Programs developed for PLATO in just one subject area, chemical engineering, show the range of material created. Topics included modelling of a draining tank, use of steam tables, steam turbines, compression of a gas, gas-phase chemical equilibrium, Rankine refrigeration cycle and a utility program for the calculation of chemical equilibrium constants.
(http://www.che.udel.edu/faculty/full/sandler/comp.html 14/09/2000)

In common with other projects, the development of material was more time-consuming than first estimated and the demonstration of PLATO IV began a year late. Demand for processing power was also more than expected and 1000 terminals were used rather than the projected 4000 in order to get the system to run at acceptable levels. Again, questionnaires were key instrument of feedback gathering and student response top PLATO were generally favourable. Teachers found that TUTOR was too difficult to use as an authoring language.

PLATO and TICCIT demonstrated two different approaches. Both projects, as the funding dried up, withered on the vine somewhat. However PLATO did generate spin-off software, important in its own right:

> "*As an educational/multimedia system, PLATO has many offspring. Its most successful direct descendant is Ten CORE, an authoring system for DOS and Windows. Macromedia's Authorware, an authoring system for the Macintosh and Windows, is also firmly rooted in PLATO.* [] *Many people who experienced the online PLATO community were inspired to replicate it on other platforms. Lotus Notes is the best-known example. It was developed by Ray Ozzie, Tim Halvorsen, and Len Kawell, all of whom had worked at CERL in the late 1970's. It would be an exaggeration to call Lotus Notes a clone of PLATO Notes, because Ozzie expanded the concept to include powerful capabilities that were never contemplated for PLATO. But many of its basic features were modeled after PLATO Notes.*" (http://thinkofit.com/plato/dwplato.htm 14/09/2000)

PLATO developers produced software to record interactions, initially about software bugs. The idea grew increasingly sophisticated and increasingly recognisable as a conferencing system.

The author David Woolley describes the development process:

> "*I came up with a design that allowed up to 63 responses per note, and displayed each response by itself on a separate screen. Responses were chained together in sequence after a note, so that each note could become the starting point of an ongoing conversation. This is what John Quarterman calls a star-structured conferencing system, and PLATO Notes was apparently the first of its kind.*" (http://thinkofit.com/plato/dwplato.htm 14/09/2000)

Other software was developed that allowed users to talk on-line and the concept of notes and on-line chat quickly took off:

> "*The sense of an online community began to emerge on PLATO in 1973-74, as Notes, Talkomatic, "term-talk", and Personal Notes were introduced in quick succession. People met and got acquainted in Talkomatic, and carried on romances via "term-talk" and Personal Notes. The release of Group Notes in 1976 gave the community fertile new ground for growth, but by that time it was already well established. The community had been building its own additions to the software infrastructure in the form of multiplayer games and alternative online communications. One such program was Pad, an online bulletin board where people could post graffiti or random musings.*" (http://thinkofit.com/plato/dwplato.htm 14/09/2000)

Appendix F: Conclusion

The Stanford project was only one of hundreds of projects on CAL, undertaken in America during the 1960s. It was this concentration of effort and funding that left educationalists in Britain feeling that, if the Government did not take steps, then Britain would never develop its own expertise in CAL and would be wholly reliant on American CAL, with serious implications for both education and the economy:

> "*American researchers have gone ahead and there are now as many as 50 or 60 CAI installations in the USA at an estimated cost (in 1967) of $8m whilst the rest of the world, Britain included, has made very little progress.*" (Annett 1970 p 63)

This issue so concerned the Government of the UK that successive governments have spent money on various initiatives around the development, use and evaluation of CAL, in various educational sectors. Some of these initiatives are reviewed in Appendix G.

Appendix G

Appendix G **The NCET, NDPCAL, MEP, CTI, TLTP and the ILT**

Introduction to Appendix G

This appendix describes the work of the NCET (National Council for Educational Technology) from its inception in the late 1960s to its disbandment. This appendix also outlines the work of the NDPCAL (National Programme in CAL Development) and examines the work of the CTI (Computers in Teaching Initiative) from 1989 onwards. It looks at the work done within the TLTP (Teaching and Learning Technology Programme) up to the start of Phase Three. Various other initiatives in learning and teaching are also discussed, together with selected reports.

The NCET 1967 – 1973

The first report of the NCET, *Towards more Effective Learning*, covered the work of the Council from its inception in early summer 1967 to the end of 1968.
The NCET (National Council for Educational Technology) traced its earliest roots back to the Brynmor Jones Report on Audio-Visual Aids in Higher Scientific Education, which appeared in October 1965. This report constituted a "*wide ranging discussion of the current state of developments in educational technology, together with detailed and practical recommendations for future action.*" (NCET 1969 (a) p 1) The NCET sprang out of the Report's recommendation for both a National Centre and a Council to oversee such a centre. The Government adopted the idea of a Council, part of whose remit was to consider the setting up of a National Centre.

The work of the NCET up to 1969

Interestingly, the NCET saw educational technology as a potential answer to problems of mixed ability teaching and the lack of specialist subject teachers. (NCET 1969 (a) pp 11-12) Issues around resources staff and timetabling were also seen as part of the NCET's concern, as was educational research based on the use of various technologies and their effective use.

The NCET saw some key concerns arising from the idea of applying educational technology such as the fear of teachers that centralised material production would infringe pedagogical freedom of choice. The NCET envisaged that, as the cost of labour remained comparatively high and the cost of technological equipment fell, the emphasis would be on the use of various technologies within education.

Allied with this area of effort was that of awareness of what was currently available, a task too large to be undertaken solely by the NCET. The NCET showed itself to be aware that educational technology was developing rapidly and that the greatest area of change was likely to be in the field of computing, as evidenced in their early publication of a working paper on computers in education.

Part of the NCET's work was the dissemination of their findings. The NCET agreed that a balance must be struck between publication for a specialist audience and publication for a general audience, both of which needed to reflect the NCET's work. The NCET also decided to publish 3 times a year the Journal of Educational Technology.

The NCET realised that the technological advances in computing were inevitably going to impinge on British educational practices, especially through the pressure to match the sophistication of American efforts in applying computing to education, in many guises:

> "*...with the rapid advances being made in computer technology, and the growing efforts being devoted to the educational use of computers in the USA since 1960, it became clear to the National Council for Educational Technology that some evaluation of the role of the computer in British education and training was called for. Accordingly in December 1967 the Working Party* [] *was set up: one of its tasks was to look at the prospects for computer-based learning systems.*" (NCET 1969 (b) p vii)

The Report produced by the Working Party identified investment by American government agencies and private foundations at approximately $50 million dollars a year in the mid 1960s. In contrast no coherent, centrally funded initiative was underway in Britain. The Working Party's role was to investigate the potential role of computers in education and to outline such a coherent programme in computer based learning systems, including financial recommendations. Much of their material appears to have been drawn either from conferences in the mid 1960s, where American delegates took centre stage, such as the conference in 1966 in London, sponsored by the US Office of Naval Research. (See Duke 1970 p 66)

In 1969, the authors were optimistic about the future of computer-based learning systems:

> "*There is virtually no limitation to the type of student that can be accommodated by a computer based learning system.* [] *likewise there is no limit to the type of subject that can be presented...*" (NCET 1969 (b) p 21)

For the authors, early work should concentrate on subjects where the student responses can be evaluated easily and the material logically structured but the driving force should be educational objectives, not the environment. They saw such systems in use in schools, colleges, universities and in industry, as well as defence. The Report also considers the costs of adopting computer-based learning systems, using as their basis the "*present average costs per student hour and those on the horizon for computerised systems.*" (NCET 1969 (b) p 25)

The authors, in the final paragraph, hand over the work to the full-time study team looking at educational uses of computers, applauding the Council's decision to establish such a team. The were many areas in education in which computers could make significant contributions, such as administration and time-tabling but the key area identified was computer based learning.

The next publication to look at the question of computers in education with reference to computer-based learning was the slim volume (16 pages) Computer Based Learning: A Programme for Action. Annett describes this paper as "*succinct, general and optimistic in tone.*" (Annett 1970 p 63)

The themes that will emerge in later reports and projects summaries over the next thirty years or so can be identified here again: here is the optimism about what can be achieved through the application of computers. Although the authors are hampered by technological limitations, they can foresee the increasing sophistication of hardware and software leading to almost boundless opportunities to improve teaching and learning.

Here too is the problem of quantifying expected benefits and measuring actual benefits when examining the use of computers to teach. Underlying such a problem (which is many problems, once closely analysed) is a broader philosophical question about what is meant by learning. These and other themes are taken up by later work in computer-based learning, either implicitly or explicitly.

The authors go on to identify the areas of computer-based learning they perceive as being significant. They discuss simulation, computer assisted instruction, (which they see as an area of both greatest potential and greatest uncertainty), computer-managed instruction, (where the computer is a tool for diagnosis and testing) and data storage and retrieval. Also described is the beginning of the infiltration of computers into diverse areas of work and education:

> "*The computer...* [is] *likely to play such a major part in our commercial and industrial life that it is particularly important that those who are likely to go forward into commerce and industry where the computer can be of direct use, should thoroughly understand its capabilities.*" (NCET 1969 (d) p 17)

The role of the computer as a valuable administrative tool is also identified but the authors envisage schools and colleges using a time-share system, rather than having their own computing facilities.

For both further and higher education, the authors see a pedagogical role and a commercial one:

> "*...the computer has an important and far reaching role to play for simulation purposes and as a computational tool.* [For re -training]... *the computer will certainly have a potential contribution to make both as a computational tool and in CAI/CMI.*" (NCET 1969 (d) p 29)

Again, less accurately, the authors predicted the possible disappearance of traditional examining methods, to be replaced by computer-based assessment. The authors mention in one brief paragraph, the use of computers as a store of information. Their vision was of a centralised cataloguing and storage facility with documents on line: "*User interest profiles could be built up and documents located at speed, thus providing the means of effectively meeting individual requirements.*" (NCET 1969 (d) p 42)

We can see that to these authors, as others of the time, the potential of computers appears almost limitless, with predictions based on current developments and some wishful thinking about how computers, in some future manifestation, will radically alter the nature of education. The predictions seem both too optimistic and at the same time not bold enough.

The work of the NCET 1969 - 1973

The report *Educational Technology: Process and Promise* was the final one from the NCET, summarising its work from its inception in April 1967 to its closedown in September 1973 when it was succeeded by the Council for Educational Technology for the United Kingdom. Some of the initial part of the Report was scene setting, recapping the creation up of the Committee. (NCET 1973 pp 9 - 18) This section examines the work of one of the six committees active under the NCET, the Joint Committee.

The Joint Committee concerned itself with the development of teaching and learning materials. Two of the large scale projects set in motion by the activities of the committee (in one or more of its previous incarnations) were the teaching of maths to non-maths science students at post-A level and the development of computer aided learning materials (the NDPCAL project).

In Computer Assisted Learning: 1969 - 1975, Annett says:

> "*These papers [the working papers of 1969], ultimately digested by DES, resulted in the funding of a National Development Programme in Computer Aided Learning (NDPCAL) in January 1973 at a cost over five years of £2 million.*" (Annett 1976 p 1)

Appendix G The NCET, NDPCAL, MEP, CTI, TLTP and the ILT

In early 1973 the NDPCAL was created under the directorship of Richard Hooper, a project largely based on the work of the NCET in computer-based learning systems. The next section looks at the work of the NDPCAL.

The work of NDPCAL

The NDPCAL sprang directly out of the work of the NCET in the late 1960s. In 1972, Margaret Thatcher, then Secretary of State for Education, approved the setting up of the programme. Richard Hooper took up the post of Director in January 1973. The original budget was £2 million, increased to £2.5 million over the five-year life of the Programme, due to inflation.

The breakdown of costs is interesting: the bulk of the spending (£1.6 million) was on staffing. The tertiary sector had the largest project share at £1, 249 000. The Director estimated that the Programme had attracted approximately £2 million in matched funding but warned that the figures were not precise. Matched funding included expenditure on hardware and staff time.

The main aim of the Programme, formulated in its first month was: "*to develop and secure the assimilation of computer assisted and computer managed learning on a regular institutional basis at a reasonable cost.*" (Hooper 1977 p 15)

This definition offers some interesting insights into the Programme. What was lacking is the grander scale of some of the NCET's earlier visions. The emphasis was on the institution-based uptake of CAL at a reasonable cost. The focus was on developing CAL that could, and would, be used by institutions after central funding had been exhausted. Allied to this aim of institutionalisation was the concept of transferability: this is discussed later in this appendix.

The Programme funded 35 main projects and feasibility studies. By 1977, 27 of the projects or studies involved 44 institutions. For these 44, in the summer of 1977, it looked as if 32 would maintain the impetus of the Programme, giving a success rate of

70%. Two areas were particularly successful: the further/higher education sector (with 22 out of 27 projects looking healthy) and industrial training (with six out of seven projects apparently viable in the long term).

Clear strategies evolved over the life of the Programme for promoting both institutionalisation and transferability. Some of the strategies for the former included: matched funding, organisational structure and curriculum integration. Strategies for transferability included technical policy, existing communication channels and inter-institutional projects.

The Programme produced a number of technical and future study reports, around the areas of CAL and CML. The Programme's first technical report was published in October 1973 and looked at problems of implementing CML (Computer Managed Learning). The report examined a number of issues, including the diagnostic assessment of learners and the sequencing of learning materials. Once again, Britain is seen as somewhat lagging behind the US but benefiting from the American experience:

> "*The criticisms of the Open University, PLAN and IPI reflect upon areas where the problems have not yet been solved and ignore deliberately the great contributions that each of the three have made towards improving learning.*" (Hawkridge 1973 p 16)

In March 1973 Richard Hooper gave a paper at the first British conference on Computers in Higher Education. The paper critically examined some of the claims made for the role of computers in education. Interestingly, Hooper noted that the role of computer as a medium for teaching about computers was secure but the place of computing in the curriculum was not!

Hooper (1973) was sceptical of the claims made for the cost-effectiveness of CAL and points out that, despite extensive American work, even large-scale projects such as PLATO have not definitively delivered cost-effective CAL. On the contrary, Hooper (1973) notes that the introduction of CAL to a subject or institution may well increase staff costs to that institution as well as the concomitant investment in computer rooms and equipment.

Hooper (1973) concludes that the claims for cost-effectiveness:

> "*cannot be taken too seriously at present.* [] *It is difficult to establish cost* []. *It is difficult to establish educational effectiveness. It is difficult to relate one to the other.*" (Hooper 1974 (a) p 361)

Hooper concludes that the debate about CAL is not likely to die away in the near future. He draws general conclusions about how a CAL model might best use the experiences of the academic community in the States to create what he calls the adaptive laboratory, a model that draws on the idea of computer as tutor and as arena for simulations and problem solving.

Despite the Director's wariness of formal evaluation, accompanying NDPCAL's development investment and effort was the evaluation of the educational benefits of CAL (Other evaluations were financial and technical.) UNCAL stood for UNderstanding Computer Assisted Learning. UNCAL represented, in 1975, an estimated 4% of the total NDPCAL spend. The establishment of UNCAL was initially a one-person job, which grew to a staff of seven, including a part time director. It is important to note that UNCAL evaluated the projects initially at least on a general basis rather than carrying out fine-grained analysis of particular projects.

UNCAL's aims can be summarised as follows: to develop a self-critical element within NDPCAL: to aid the development of projects by offering appropriate evaluation tools; to help the NDPCAL directorate in the overall management of the programme; to aid the Programme Committee and finally to display the work of the NDPCAL to an appropriate audience.

From this summary, it is obvious that UNCAL was not set up to carry out fine-grained testing of achievable learning outcomes. This fits with the clearly stated belief of the Programme Director that formal evaluation at this level may hamper rather than help development. UNCAL's role was broader than that and focused at programme as well as at project level. From reading the UNCAL companion, it is possible to gather a sense of the turbulence that accompanied the genesis of UNCAL. Significantly, the authors note that UNCAL was not subject to veto of publications: its work was both open and independent.

To sum up, the authors' survey of the work of UNCAL gives a clear picture of the issues that were emerging as early as 1975. The work of the UNCAL team identifies factors influencing the potential success of projects but does not seek to give definitive solutions to the problems it describes. The report also describes some of the positive aspects of the projects examined.

As well as evaluating the educational aspects of projects funded by NDPCAL, there was a detailed evaluation of the financial aspects of such work. Fielden and Pearson's (1978) report on the introduction of computers into teaching focused on a detailed cost benefit of such introductions. The authors were financial experts employed by a large firm of accountants.

Given the diversity of projects funded by the NDPCAL, the wide range of individual project objectives and the different institutions in which research took place, it is not surprising that the auditors found it difficult to develop or apply any firm baselines to the costs and benefits incurred by or derived from those projects:

> "*One of the hardest questions for the evaluation to answer has been: what is the nature of the* [educational] *change. In some projects there is no simple answer since CAL material has been used differently according to the teachers' preferences and teaching style and according to the CAL package or simulation in question.*"
> (Fielden and Pearson 1978 p 62)

Fielden and Pearson (1978) conclude that:

> "*CAL is now a tried supplement and alternative to conventional teaching. 35,000 student terminal hours were recorded in 1976/77 in the projects []. While we have shown that it is nearly always more expensive in total cost terms than existing methods, there are many cases where the particular marginal costs of adopting CAL in the institution are low. In terms of extra cost, CAL could cost less than extra teaching staff. The educational evaluation is reporting on the qualitative aspects of the change. There are many claims for added educational value, enhanced understanding, immediacy flexibility, etc. which require examination. When answers are received to these questions, the decision-maker can then face the ultimate value judgement. In the probable extra cost of CAL worth the extra claimed educational*

> *value?* " (Fielden and Pearson 1978 p 79)

Their conclusion very much echoes the view of Richard Hooper who warned at the beginning of the project, in his paper Making Claims for Computers, that hard facts and figures about the more intangible educational benefits of CAL may be difficult to garner successfully.

The CET's annual report for 1977 - 1978 notes that the NDPCAL came to an end in December 1977, as planned:

> "*It had spent £2,600,000, an increase of only £600,000 over the original allocation of £2,000,000 despite the unprecedented inflation rates of recent years. It left computer-assisted and computer-managed learning established on a sound and continuing basis in 32 institutions.*" (CET 1978 p 45)

This report views the NDPCAL as an "*unqualified success*" (CET 1978 p 46) and is strongly optimistic about the value of the initial programme and the after-care provided for projects. This after-care took the form of "*six programme exchange services,* [] *an information service on computer-assisted and computer-managed learning* [] *conferences and seminars on the subject.*" (CET 1978 pp 46 - 47) It should be noted that this optimistic view is presented only months after the end of NDPCAL and that, as with other projects, a longer-term view was needed to gauge the true impact of NDPCAL.

NDPCAL focused on learning *through* computers, in various forms, on varieties of CML and on the concept of tutorial dialogue. There are two interesting points raised in Tawney's (1979) introduction: at the time of writing (1979) computers were too expensive to be regarded as a cheaper alternative to lecturers and secondly, NDPCAL researchers/developers regarded the CAL as an addition to conventional teaching not a replacement for it: a view taken by Richard Hooper in earlier papers, with his emphasis on the teacher a valuable, unique medium of communication and the computer as a separate medium with its own strengths and weaknesses.

Appendix G **The NCET, NDPCAL, MEP, CTI, TLTP and the ILT**

Even at this comparatively early stage of CAL use, Tawney (1979) identified two approaches to CAL that can be seen wherever CAL is used today: the fitting of CAL into a pre-existing, largely unchanged course, replacing some lectures/tutorials with CAL-based actively or re-designing a course around CAL.

What can be concluded from the work undertaken for NDPCAL? A number of points emerge. The first is that what constitutes CAL varies widely and different institutions tackled the development and integration of CAL in different ways. Secondly, those who developed and used the CAL were enthusiasts and the reporting of projects and their outcomes reflect a generally positive attitude. Thirdly, most projects were hampered by the display technology available, which limited the types and nature of output to students.

Some views of NDPCAL were, in the long term, much harsher:

> "*The idea that in 1973 there existed a body of knowledge about computer assisted learning which it was worth developing without further associated research seemed at the time fanciful, and in retrospect, absurd. The absence of a proper experimental design resulted in a proliferation of 'case-studies', the significance of successes of which is virtually impossible to determine. The actual developments embodied in the projects were marginal in relation to existing knowledge... Technologically the ... projects were unadventurous.*" (O'Shea and Self 1983 p 101 - 102)

This negative view is capped by their verdict: "*The report's conclusions on technical matters have thus been rendered obsolete* [by the introduction of microcomputers] *and much of the original development work is now seen to be irrelevant.*" (O'Shea and Self 1983 p 103)

The CET continued its work in educational technology but Government support for large-scale, centrally funded projects was largely missing.

As Marshall (1988) points out:

> "*It was the Labour Government which promoted the introduction of microtechnology through the MEP [Microelectronics Education Programme] and £12.5 m was the sum proposed in 1979... Sadly, this initial enthusiasm was not maintained and, in 1980, the programme's budget was cut to £8m over a three year period, i.e. 0.02% of the overall education budget...*" (Marshall 1988 p 272)

MEP (Microelectronics Project)

The emphasis of government intervention in educational technology passes onto the secondary sector by the end of the 1970s. In March 1980 the Under Secretary of State at the Department of Education and Science announced a four-year programme in microelectronics. The MEP was set up in November 1980. The £8 million funding of the MEP (over 1981 - 1984) meant that it could not help schools buy the micro computers that would enable them to use the material or training offered by the MEP itself but would concentrate on preparing children in schools "*for life in a society in which devices and systems based on microtechnologies are commonplace and pervasive.*" (Fothergill et al. 1983 p 3)

For a detailed discussion of the aims and strategies of the MEP see Fothergill et al. 1983 p 3 – 16. Two key features are worth noting: firstly, part of the work of the MEP was to set up 14 pilot information centres around the country, serving local education authorities and secondly, the brief was much wider than anything previously attempted, covering teaching about computers as well as through computers.

A special project, Micros in Schools was launched by the Department of Industry. This gave schools the chance to buy one or two brands of microcomputers at 50% discount. One of the microcomputers offered was the machine developed by the BBC. One writer made the accurate prediction that the BBC micro would be widely used in homes and schools. The BBC micro proved to be immensely popular with a wide variety of users. For schools purchasing a microcomputer, training for at least two teachers was offered as part of the scheme. A course in microcomputers was distributed by the MEP.

Under the heading of curriculum development, the MEP looked at the impact of new technology upon traditional subjects e.g. changes in syllabus content and teaching methods.

It also looked at the area of information studies:

> "*This is one part of education that will be profoundly affected by the computer, for access to information and in retrieval will be revolutionised by the technology.* [] *Far more important will be the abilities to use the technology to locate information and to make use of it effectively and efficiently. Thus there is a need to help all children to undertake information studies and this will be an important development for the curriculum.*" (Fothergill 1983 pp 136 - 137)

In the early 1980s the new technology was seen as creating many new needs within schools, particularly for teachers who understood the need to teach the new technology and for courses that would give pupils experience of this technology.

Computer Studies in its previous forms as a narrowly focused, mathematically-based subject could no longer fully answer the needs of schools in the early 1980s. The Alvey Committee reported in 1982 that:

> "*Action must start in the schools. We support the moves which are now putting computing on the curriculum. But, it is no good just providing schools with microcomputers. This will merely produce a generation of poor BASIC programmers...The teaching of computer science in schools must be increased substantially, in quality and in quantity.*" (Report of the Alvey Committee 1982 p 62)

Here the emphasis was on producing pupils with a thorough grounding in computing, to improve the quality of A level computing candidates and undergraduates in computing. The report noted that universities had to give remedial education to undergraduates with A level computing. It is interesting to note that the recommendations of the Alvey Committee on developing IT skills in Britain are a direct response to what the Report describes as "*a major competitive threat*" (Report of the Alvey Committee 1982 p 5), posed by Japan's Fifth Generation computing project.

In the early 1980s two main arguments can be identified. Firstly, that all pupils should have contact with the new technology, from the point of view of information studies. Secondly, the teaching of computing as a specialist subject needed to be vastly improved. For secondary schools the arguments about CAL seem to have become largely irrelevant, given the pressures generated by the rise of the microcomputer, the introduction of the National Curriculum and the need for British education to once more rise to a technological and educational challenge in equipping students with appropriate computing skills. CAL is mentioned as part of the MEP but the emphasis has undoubtedly shifted to learning *about* computers.

The next significant Government initiative was the CTI (Computers in Teaching initiative). This is discussed in the next section.

CTI

In 1982 the Nelson working party (set up by the Computer Board for Universities and Research Councils) noted serious shortages in the provision of computers for teaching in higher education. "*As a result, the Board set three tranches of £500,000 at half yearly intervals to fund 23 pilot projects in the use of computers in teaching.*" (Gershuny and Slater 1990 p 55) This initial project was followed by the setting up of the CTI, which was established in 1985 by the Computer Board for the Universities and Research Councils (later known as the Information Systems Committee).

It initially supported 139 courseware development projects, with at least one project at each UK university, across most disciplines. "*19 CTI Centres were set up in April 1989 with a further 2 following in October 1989.*" (Darby 1990 p 3) Even in its early stages, the CTI was evaluating thousands of pieces of hardware and software. Other bodies were also contributing to CTI Centres. The Institute of Chartered Accountants gave £35,000 to the Centre for accountancy, for example.

In 1989, Gardner and Slater reported on the work of the first phase of the CTI. By the middle of 1989, most of the 139 projects had filed reports on their work and the process of reviewing end products and lessons learned could begin.

The authors were optimistic about the success of the CTI's work from 1985 to 1989 and saw a positive role for the CTI in the 1990s:

"*Just as the main phase of the CTI has done much to stimulate nationwide activity in*

> *the eighties, the programme's second phase will, if successful, lead to more widespread and imaginative uses of computers in teaching in the nineties*." (Gardner and Slater 1989 p 3)

The need for high quality courseware was identified as being a key aspect of higher education teaching by the early 1990s:

> "*Following a period of considerable growth in the amount of courseware being produced in the UK... all disciplines represented stated that currently the lack of high quality courseware was seen as a major blockage to the uptake of CAL in universities*." (Darby 1991 p 13)

Other projects were set up in the early 1990s, notably the ITTI (Information Technology Training Initiative.) The ITTI received £3 million over three years to improve training material availability, develop software and training materials and develop CAL. Grants were made to 20 universities. Nottingham received £124, 790 to develop CAL standards and "*authoring standards, delivery specifications and training materials as a pilot* [] *for computer-based training and interactive learning....*" (Shields 1991 p 59)

Other bodies were also funding research into technology for teaching and learning. The ESRC (Economic and Social Research Council) set up a co-ordinating centre, a programme evaluation unit and three research centres, with funding of £1 million for the years 1989 – 1991:

> "*ESRC has the responsibility of playing a critical role in the evolution of new ways in which technology may contribute to learning: no other UK agency carries the brief to undertake and stimulate the necessary basic research. The uncertain pragmatics of early work on classroom technologies must now give way to more clearly defined studies*." (Lewis 1989 p 2)

This project followed a three-year pilot programme undertaken from 1985 to 1988.

Much effort in the field of teaching and learning technologies was being duplicated, with diverse initiatives and projects soaking up considerable funding. A plethora of initiatives does not necessarily guarantee progress in any area of research or development.

This point was made most strongly in the 1998 Atkins Report:

> "*There are now too many different projects and programmes in the use of ICT,* [information and communication technology]*funded jointly and severally by the Funding Councils and DENI (and indeed by other central agencies such as the DfEE). At Funding Council level the situation looks particularly messy, with variable memberships of different programmes (and stages of programmes), overlaps in remit and inconsistencies in budgetary mechanisms...* " (Atkins Report 1998 paragraph 17)

The CTI annual report for 1992 - 1993 notes that the year was one in which: "UK higher education took in many more students than ever before. Universities and colleges are now bursting at the seams and teaching loads have increased sharply." (CTI1993 p 5)

It is worth noting that at the time of the report the CTI serviced exclusively the 43 'old' universities. The report notes the work done by the CTI including the production of 18 resource guides, published on paper and on disk at a net cost of £30, 400, the production of newsletters and the creation of over 6,000 screens of electronic information on bulletin boards. Total CTI funding was £1,033,907 received from Information Systems Committee of the University Funding Council with a contribution from the Department of Education in Northern Ireland.

The CTI also reported on the running of a wide range of workshops, with an average attendance of 19 people. Subjects included CAL in tropical medicine and CAL in language learning. From this we can see that encouraging the use of CAL in higher education by individual staff and departments was a key aspect of the work of the CTI. The report concluded, "*The discipline-based approach of the CTI has proved both powerful and resilient enabling a range of resources to be provided in a highly cost-effective manner.*" (CTI 1993 p 7)

The annual report for 1993-1994 notes that the funding of the CTI was taken over by the Higher Education Funding Councils for England, Scotland and Wales. A review of the CTI by the Funding Councils' Review Group led to the renewing of CTI funding for a further five years. The CTI remit was extended to the all higher education establishments. The CTI continued its work in publishing (producing 12 new editions of resource directories, 54 newsletters and maintaining and establishing electronic information sources, including, for the first time, World Wide Web pages).

The CTI also received attention in the Dearing Report. The Report's authors were convinced of the value of the educational technology, noting:

> "*we see a growing contribution to learning form the extensive use of computer-assisted learning materials, communication technology and new, enhanced delivery systems.* [] *One of the benefits of new technology lies in providing a learning environment that may succeed improving understanding where other methods have failed.* [] *By using computer-based learning material, students can receive immediate feedback to assist learning complex concepts.*" (Dearing Report 1997 paragraphs 8.23 - 8.26)

The report goes on to note that the CTI has done "*valuable work in providing institutions with subject-specific advice on technology-based educational practice...*" (Dearing Report 1997 paragraph 8.71) and recommends that the CTI centres have a central role in the Institute for Teaching and Learning. The Institute for Teaching and Learning is discussed later in this appendix.

In 1997 the four higher education funding bodies established a Review Group to examine the work of the CTI and the TLTSN (Teaching and Learning Support Network):

> "*It ran from November 1997 to June 1998. The Review had three aims. The first was to determine the extent to which the Computers in Teaching Initiative (CTI) and the Teaching and Learning Technology Support Network (TLTSN) had fulfilled their terms of reference. The second aim was to capture some of the lessons learned during the lifetime of these two initiatives. The third was to gather views and make recommendations about what should happen to these programmes at the end of their current period of funding, in context of other programmes and other developments in higher education...* " (Atkins Report 1998 paragraph 1) .

The Review also looked at the work of the TLTP (Teaching and Learning Technology Programme), the projects of the Joint Information Systems Committee and took note of the early work done towards the setting up of the ILT (Institute for Learning and Teaching)

Appendix G The NCET, NDPCAL, MEP, CTI, TLTP and the ILT

The Report notes that:

> "[the] *under-utilisation of CAL and ICT* [Information and Communication Technologies] *remained a continuing, major problem for virtually all higher education establishments... Many existing academic staff development programmes were relatively ineffective in supporting the use of CAL and ICT.*" (Atkins Report 1998 paragraph 6)

However, the Review Group found that the users of CTI centres were consistently happy with the service received: "*The CTI has fulfilled its terms of reference and in the eyes of its direct users has provided a good and valued service.*" (Atkins Report 1998 paragraph 8)

The Report concluded that the CTI centres should remain in place, with some development of their work:

> "*There is substantial support across the sector for taking forward a reconceptualised programme on a subject basis. CAL and ICT would be an important but not exclusive focus of the new programme which would embrace innovatory and good practice in learning and teaching in the subject...* " (Atkins Report 1998 paragraph 25)

This involved winding up the CTI and TLTSN in their present forms and establishing subject centres, covering all subjects on a UK-wide basis, setting up a generic technology unit and creating a single central unit to manage all the programmes. The subject centres should act as knowledge brokers, advocating and disseminating good practice (ideally based on up-to-date research), reviewing new materials and hardware and enabling the sharing of institutional and individual experience. Interestingly, the Report recommends that the ESRC carry out research on the effectiveness of CAL and ICT.

The Report concluded that, while CAL was certainly an important element in teaching and learning strategies in higher education, there was still an under use of CAL in any coherent fashion. The Report identified a number of barriers to the development and adoption of CAL, summed up in the Report as "money, *materials and mindset.*" (Atkins Report 1998 paragraph 90) Within these categories, these factors are very much as previously identified by other projects and initiatives.

The Report goes on to identify specific strategies or levers to address the problems of CAL development and adoption including staff development, institutional strategies for the development of teaching and learning and consortia for the development of CAL.

It seems that even where an initiative such as the CTI is felt to be successful, changes are seen as necessary after a number of years to focus once more upon relevant issues such as the transferability of material and the effective adoption of technology-based teaching and learning approaches. The Report appears to stress the dissemination and sharing of materials, without emphasising the creation of those materials. Creation of materials is an area of endeavour that presumably will be left to individual staff or departments, either under a broader project umbrella such as the TLTP or funded by the institution, although the Report notes "*There is no doubt courseware development will remain a substantive issue for the funding bodies...*" (Atkins Report 1998 paragraph 132)

The TLTP

The TLTP (Teaching and Learning Technology Programme) was launched in February 1992 by the UFC (Universities Funding Council). The funding for the initiative was £7.5 million. 160 bids were submitted and a total of 43 projects were funded. Over three years, the funding totalled £22.5 million. The TLTP distributed grants totalling £35 million to 76 projects over its first two phases. As the TLTP co-ordinator, Turpin notes:

> "*two types of project were considered: single institutional projects addressing the problems of implementation and staff development and consortia projects based within subject disciplines.*" (TLTP Science Case Studies Introduction)

The UFC was succeeded by separate bodies for England (HEFCE), Wales (HEFCW), Scotland (SHEFC) and Northern Ireland (DENI). These four bodies agreed to commit further funding to the second phase of TLTP. The second call for proposals generated 367 submissions. "*A total of £3.75 million was made available to fund a further 33 projects under the second phase of TLTP.* (Turpin (Ed.) 1995 p 1) The total funding from the higher education bodies is £33 million. The institutions who have had bids funded contribute directly and indirectly to the TLTP work, giving "*overall funding for TLTP... somewhere in the region of £75 million...*" (Turpin (Ed.) 1995 p 1) This makes the TLTP probably "*the largest learning technology initiative ever undertaken within UK higher education...*" (Turpin (Ed.) 1995 p 1)

The concept of the TLTP can be traced back to CTI activity during the 1980s. As noted earlier, initial projects were hardware-based (the acquisition of workstations) but later projects focused on software and emphasis was placed on the use of computers in non-scientific subject areas. This was an initially unfamiliar concept. The second phase of CTI was the setting up of subject-based centres, to support the uptake of new technology-based approaches to teaching and learning.

The TLTP set out to address the problem of lack of subject materials. The project was to find the development of materials that would underpin the delivery of courses to increased student numbers. Quality was not a focus of these materials but quality improvements would, of course, be welcome.

The TLTP also attempted to address a crucial issue in higher education in the 1990s: that of cost. Universities (both 'old' and 'new') had seen funding per student drop while student numbers have increased. The question of managing resources effectively to meet the needs of a diverse student population has lead universities to see the use of CAL as a sure route to cost savings and teaching efficiencies. "*There are several studies coming out of TLTP pointing in better learning, as yet there are relatively few that show any direct cost savings*."(Slater 1996 p 4) Equipment costs remain relatively stable (the cost of PCs stays the same but functionality increases) and support for undergraduates cannot be done away with entirely.

It is interesting to note that:

> "*Some of the emerging programme's early advocates attributed to information and communication technologies a near miraculous capacity to palliate the effects of rising student numbers, secure efficiency gains and generally provide a magical panacea to many problems facing our universities as they struggled to come to terms*

> *with the new demands of the 1990s.*" (Gardner 1996 p 6)

Other authors took a more optimistic view:

> "*There are clear signs from institutional plans and strategies that senior managers in HEIs now largely understand these issues and see their role as encouraging uptake of TLTP material. Some interviews have indicated that many overestimate material available and underestimate the need for update and support, but TLTP forms an increasing part of the proposed future.*" (Slater 1996 p 4)

The CTI publication, Active Learning, devoted its July 1996 edition to articles detailing response to the TLTP and reports from consortium members. The theme was TLTP: what has been achieved?

Clark (1996) points out a number of worrying elements in what he calls the "*fierce rush for grants*" (Clark 1996 p 11) to develop CAL materials. He takes as his basis for argument the report on a CD-ROM based package that covered 20 hours of lectures on weed biology (Teaching with Multi-media: a case study in weed biology by Lisewski and Settle):

> "*This course was originally composed of... 20 lectures, 12 hours of practicals, 10 hours of workshops and 48 hours of private study, a total of 90 hours of assigned time. The students in the raised regime had unlimited access to Macintosh computers running the programme between 9am and 5pm throughout the duration of the course...* " (Clark 1996 p 9)

When Clark sought to quantify the cost of the project, he found active figures very difficult to obtain: not because there was a conspiracy of silence but because the project had been funded (as many internally funded projects are) by a blend of shifted resources and re-allocated funds, which represented incredibly complex institutional accounting.

Clark does, however, put a figure on that package. He estimates its cost at £200,000. He then goes on to point out that if the 16 hours released by giving students access to the CD-ROM (rather than requiring them to attend a lecture) is costed at £500, then to recover the £200,000, the CD ROM would have to be used 80 times in a five year period. He also points out that the lecture component was actually the second <u>cheapest</u>

element of the course, private study being the first. The argument is, of course, that the development cost can be recouped by scaling up the delivery of courses: 300 students not 30 and or selling the package outside the university. Clark examines both these arguments. He points out that if it costs tens of thousands to create CAL material to cover say 3 hours of a 400-hour degree, how will the other 397 hours be funded?

For the external sales aspect, he identifies several problems. With reference to TLTP material, the first phase deliverables were to be made available, without charge to all institutions. Secondly, materials bought in acquired by an institution are likely to need tailoring to a particular course and this returns us neatly to the cost of staff hours. It takes time and resource investment to overcome the 'not invented here' problem.

What many give such heavy investment its final value is increased quality of student learning. Here again, figures are difficult to obtain, as gains are hard to quantify:

> "*This is yet another case in which the aspirations and expectations invested in computer technology have failed to materialise. ...To say that students really enjoyed the learning experience is not enough...*" (Clark 1996 p 10)

The problems and costs of producing sufficient hours of CAL (and similar material) are simply too large to allow a straightforward scaling of production. It costs thousands to produce an hour of CAL and that cost does not drop if there are 40 hours to be produced rather than four. Clark identifies the problem of production of academic materials from a copyright view point: if such materials are produced by a academic, then the university for which s/he works undoubtedly has provided subsidy in terms of time and resources, whether or not this is explicitly understood. In that case, to whom does the copyright and therefore the revenue belong? It is a complex legal question that has yet to be fully tested.

Mogey (1996) identifies a weakness in the TLTP, that of dissemination. Some individual projects clearly identified a strategy for circulating materials and updates: others had no such strategy. Mogey goes on to say that her university, Heriot-Watt decided to collect as much TLTP software as possible, for distribution to staff throughout Scotland. This activity is part of the Learning Technology Dissemination

Initiative (LTDI), which is part of the detailed implementation support service, funded by Scottish Higher Education Funding Council (SHEFC).

The goal of TLTP was the dissemination of materials developed under its aegis: the experience of lecturers may be less positive than that envisaged by the project's funding bodies. Part of the issue may well be a practical one (different hardware and operating environments) and some of it may be the perennial problem of academic reluctance to use external materials in any great detail or to any great extent.

TLTP has invested resources in the development of a wide subject base for the CAL packages, from general skills (word-processing) to language translation.

Coopers and Lybrand produced an evaluation report for the first two phases of TLTP:

> "*The Report commended the programme's let a thousand flowers bloom approach, which involved many staff across a wide range of departments working together. [] It recommended continuing the programme to encourage widespread use of... materials, and proposed further development in specific areas to build on the achievements so far. The HEFCE Board agreed that in future the programme should be underpinned by stronger central arrangements to assist the selection, guidance, monitoring and co-ordination of projects.*" (Turpin 1997 p 1)

We can hear echoes of the NDPCAL approach, with its centralised control of projects. Funding of up to £3.5 million annually was made available.

In February 1998, Phase 3 funding of 32 projects was announced. Phase 3 focused on the use of technology-based teaching and learning strategies within higher education, rather than on the development of technology-based materials:

> "*HEFCE and DENI have commissioned the Tavistock Institute to work with TLTP over the three years of this phase on the formative evaluation: they will be preparing an evaluation framework and guidelines that will support individual projects and the programme as a whole. The main purpose of this work will be to ensure that lessons and emerging understanding from the programme are made available to the higher education community.*" (TLTP 1998 p 4)

However, new materials are part of the deliverables from the Phase 3, as well as good practice guidelines, case studies and workshops. Four new development projects have been funded in the subject areas of nursing, art and design, music and medicine. Phase 3 subject areas include archaeology, economics, health sciences, history and music. Generic projects have also been funded, looking at technology in key skills development, distance learning and computer based assessment.

Phase 3 of TLTP coincided with a flurry of Government activity in the field of teaching and learning. In February 1998, the Government published the Green Paper *The Learning Age: a renaissance for a new Britain.* The Planning Group for the Institute of Learning and Teaching was established, with the aim of establishing the Institute by the end of 1998. Also announced during 1998 were the University for Industry and the National Grid for Learning. The TLTP co-ordinator notes "*The planned outcomes of Phase 3 fit well with the education sector's overall focus on lifelong learning...*" (TLTP 1998 p 2)

This shift in the teaching and learning debate was could probably be traced back some years but could be clearly seen in the ALT-C conference in September 1997. The conference was titled Virtual Campus, Real Learning. The focus of debate on technology for education could be seen as shifting away from the hardware (which platform, how fast will it run the software) and software (is it PC or Mac, does it require Windows 95) to the pedagogical issues: how does the technology support learning?

The call was for the technical aspects of the technology to take second place to educational imperatives. At ALT-C 98 the themes included learning environments, virtual reality and, significantly, evaluation, an examination of the use of learning and teaching technologies. As we have seen, this re-focusing of debate away from the impact of the technology towards the educational outcomes facilitated by the use of the technology is a theme that shapes much of the thinking in TLTP Phase 3 and a variety of publications during 1998.

The ILT (Institute for Learning and Teaching)

The concept of an Institute for Teaching and Learning was put forward in the Dearing Report (published in July 1997) in recommendation 14. The report *Implementing the vision*, arose from a round of consultations and various projects, culminating in a residential meeting by the ILT Planning Group in June 1998. The report provides "*recommendations on the proposed key functions of the ILT, its structure and staffing, governance and funding*." (Institute for Learning and Teaching 1998 paragraph 1)

As described in the report on the projected Institute, the purpose of the Institute is "*to provide 'professional standing' for teachers in higher education, comparable to that in other professions...*" (Institute for Learning and Teaching 1998 paragraph 4) Part of its role will be to redress the perceived imbalance between teaching and learning in higher education, with teaching taking second place. The ILT will also seek to improve the quality of teaching and learning, with particular emphasis on teachers in higher education meeting the challenges that face them, including staff: student ratios, the diverse range of students, including mature students and "*the enhanced application of communication and information technologies...*" (Institute for Learning and Teaching 1998 paragraph 6)

It is important to note that the ILT will not be a funding body but will certainly have a major role to play in further projects centred on technology for teaching learning.

One of the ILT's key activities is defined as:

> "*the development and encouragement of generic and subject-based good practice in teaching and learning, including appropriate and effective application of communication and information technologies...*" (Institute for Learning and Teaching 1998 paragraph 17)

In summary, the work of the ILT is likely to impinge upon the funding and nature of future projects in CAL and associated technologies. The ILT, once established, will define potentially fruitful and relevant areas of research and prospective researchers would be foolish to ignore the ILT's identified areas of interest. If the ILT proves able to provide a respected and powerful professional focus for teachers in higher education, the ILT may become the major force in shaping the nature of teaching and learning in higher education over the next decade and beyond.

Appendix G: Conclusion

This appendix has reviewed very briefly a range of Government projects, initiatives, reports and funding from the late 1960s to the late 1990s. A number of interesting points emerge, from a wealth of material (much of which does not appear here, for reasons of space) spanning several decades. There are common features that emerge over a diverse range of projects. Firstly, the participants tend to be (unsurprisingly) enthusiastic promoters of the development and use of CAL. A cynical view would be that funding undoubtedly promotes enthusiasm but it also apparent that funding is given to those who are enthusiastic about CAL in the first instance. It could be argued that only the enthusiastic will attempt the often complicated and time-consuming task of securing funding.

Secondly, projects, however well funded, tend to have a very limited life. Original proponents move on, (often to sit on committees that allocate funding or evaluate later projects), the hardware and software age badly and the syllabus changes, leaving the CAL to gather dust. It seems almost impossible to produce CAL that does not have a built in obsolescence.

Thirdly, the project's outputs (videodisks, floppy disks, CDs or web pages containing the lovingly crafted CAL) are generally received with limited enthusiasm outside the department or institution that created them, and other staff or institutions tend to be critical or even dismissive.

Finally, the evaluation of the use of CAL is notable weakness. Some small-scale projects attempt evaluation of the CAL, usually by asking students' opinions of the material. The larger scale efforts of UNCAL discussed earlier in this appendix are a notable exception. A structured, coherent approach to evaluating both the project and its CAL output are rare. Most researchers appear to content to accept as a basis of evaluation, a handful of questionnaires filled in by the users of the CAL. That is not to say that questionnaires are not a valid instrument, but that more rigorous evaluation would have been useful for many of the CAL projects attempted over the past decades.

Whether one examines a government funded project that cost hundreds of thousands of pounds (or even millions of pounds) or a small-scale institutionally funded piece of work, the results are much the same: in the long term, value for money does not seem to be a notable feature.

Appendix H

Contents

1 INTRODUCTION TO WORKBOOK
- 1.1 Pre-requisites
- 1.2 Learning outcomes
- 1.3 Suggested reading

2 THE COMPUTER ENVIRONMENT
- 2.1 Hardware
- 2.2 The microcomputer
- 2.3 The microprocessor
- 2.4 Peripherals
- 2.5 Computer memory
- 2.6 Software

3 THE HISTORY OF PROGRAMMING LANGUAGES
- 3.1 Stage 1 of programming language development
- 3.2 Stage 2 of programming language development
- 3.3 Stage 3 of programming language development

4 INTRODUCTION TO STRUCTURED PROGRAM DESIGN
- 4.1 The development of software engineering
  - 4.1.1 Real time embedded systems
  - 4.1.2 The software crisis
- 4.2 The GOTO statement
  - 4.2.1 Arguments against GOTO
  - 4.2.2 Arguments in favour of GOTO
- 4.3 The debate about structured programming
  - 4.3.1 Modularization
  - 4.3.2 The algorithmic tool kit

5 THE SOFTWARE DEVELOPMENT LIFE CYCLE
- 5.1 Introduction
- 5.2 Requirements analysis
  - 5.2.1 Problem solving
  - 5.2.2 Specification (input / output)
- 5.3 Program design
  - 5.3.1 Definition of the problem
  - 5.3.2 Expressing algorithms
  - 5.3.3 Algorithms
  - 5.3.4 Top-down design
  - 5.3.5 Modularization
  - 5.3.6 Incorporating top-down design with modularization
- 5.4 Implementation
- 5.5 Executing

5.6 Debugging
5.6.1 Detailed problem analysis technique
5.6.2 Desk checking / walkthrough
5.6.3 Syntax or compile-time errors
5.6.4 Run-time errors
5.7 Program verification and testing
5.7.1 Black box and white box testing
5.7.2 Unit testing
5.7.3 Integration testing
5.7.4 Alpha and Beta testing
5.8 Documentation
5.9 Program maintenance

6 PROGRAMMING LANGUAGES
6.1 Logic languages
6.2 Prolog
6.3 Procedural languages
6.4 Examples of procedural languages
6.4.1 FORTRAN
6.4.2 COBOL
6.4.3 BASIC
6.4.4 Pascal
6.4.5 C and C++
6.5 New programming languages
6.5.1 Visual Basic
6.5.2 Java
6.5.3 Java Script

7 SAMPLE PROGRAMS
7.1 Pascal program: 1
7.2 Pascal program: 2
7.3 C++ program: 1
7.4 C++ program: 2
7.5 Visual Basic program: 1
7.6 Visual Basic program: 2

1 Introduction to student workbook

You use this workbook with the Computer Aided Learning (CAL) material. This book has a suggested reading list, to help you with wider reading on the subject. It also contains activities for you to carry out while you work with the CAL. The workbook has gaps for you to fill in from the text on the screen. It also has activity boxes that ask you to undertake some wider reading on the subject or to summarise the text on screen.

1.1 Pre-requisites

None.

1.2 Learning outcomes

At the end of this unit the student should be able to:

- understand and explain the basic principles of structured programming
- explain key elements of the software development life cycle
- be able to describe features of different programming languages
- outline how various programming languages were developed and why
- understand the importance of structured programming

1.3 Suggested reading

The number for each book is the ISBN for that book. This number uniquely identifies the book. You can use this number to order the book from a library or bookshop.

- Principles of Programming Languages
  Bruce J Maclennan
  CBS College Publishing
  0-03-005163-6

- Foundations of Computer Technology Systems
  A J Anderson
  Chapman & Hall
  0-412-59810-8

- Programming Languages - Structures and Models
  H L Dershem & M J Jipping
  PWS Publishing Company
  0-334-94740-9

- Algorithmics - The Spirit of Computers
  David Harel
  Addison-Wesley
  0-201-50401-4

- Software Tools & Techniques for Electronic Engineers
  Keith Jobes
  McGraw Hill
  0-07-707720-2

2 The computer environment

Before anybody really understood the concept of computers, they were feared due to their supposed capacity of being able to take over positions within the workforce. Now it is realised that computers do actually need human intervention to initiate their functions. The truth is a computer is just a piece of machinery that has extensive capabilities, if humans create the appropriate software structure for them to work.

In a broad context there are two basic aspects of a computer, [ ]. Generally it is said that if you can touch part of the system, then it is hardware. To define hardware you could say it consists of all the physical components that make the computer system. The hardware and software elements have a wide range of functions, but are collectively known as the [ ].

2.1 Hardware

2.1.1 The Microcomputer

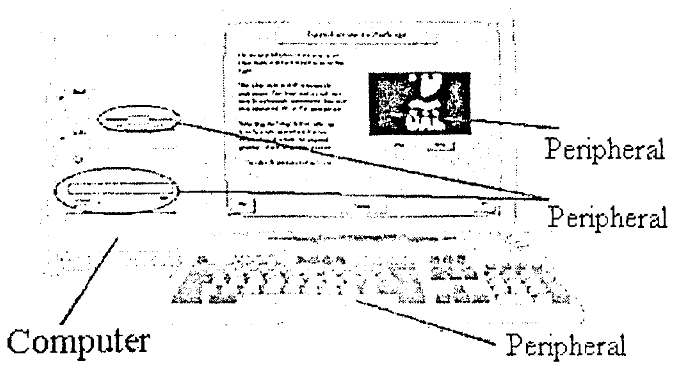


Figure 2.1 A microcomputer

This shows a collective picture of the some of the hardware used in a system. A complete microcomputer is hardware and software.

2.1.2 The microprocessor

Also known as the CPU [ ] is often referred to as the *brain* of the computer. The CPU is a circuit chip which is actually inside the microprocessor. The CPU is responsible for the retrieval of any instructions from the computers memory, deciding their type and splitting each instruction into a series of smaller actions. The speed at which this can be done depends on the capacity of the chip, which can vary depending on the actual system.

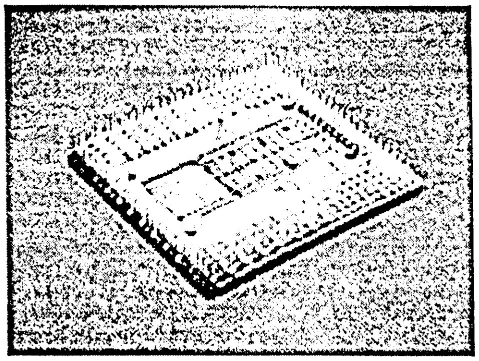


Figure 2.2 A CPU chip

Activity 1

How much does a PC cost?

The answer to that is, it depends! You should research the cost of a Pentium PC with an SVGA monitor and a CD-ROM drive. How much (roughly) would you expect to pay? Make clear where you got the information from.

Activity 1 cont'd

2.1.3 The peripherals

Output peripherals

- **Monitor**
- **Printers**
- **Speakers**
- **Modem (transmitter)**

Input peripherals

- **Keyboard**
- **Scanner**
- **Mouse**
- **Joystick**
- **Modem (receiver)**
- **Microphone**
- **Lightpen**

2.1.4 Computer memory

Computer memory has two main categories because most programs are too large to be stored within the CPU itself. A separate memory is required:

1) Primary memory

This is where the data is temporarily held during processing. Again, this can be split down into two different working elements.

i) RAM [ ]. Here the data is held for processing. RAM is volatile. This means that any information stored when the [ ] is cut will [ ], hence the need for backups.

ii) ROM (Read Only Memory). This is only read by the CPU. ROM consists of programs and instructions which are required for the [ ] to be loaded. Usually held here is Basic Input Output System (BIOS) data.

2) Secondary memory

When large amounts of data need to be stored, [secondary memory is used]. This comes in numerous forms. Although useful, the speed of data transfer onto secondary memory devices is quite slow. As technology advances the number of devices available for secondary memory are increasing, along with the speed of transfer and the capacity of the device. A few examples follow:

- 3.5 inch [ ]

- [ ]

Selection

Selection involves selecting a particular sequence of instructions in preference to another. Below are the two main types of selection instructions that can be found in programming languages:

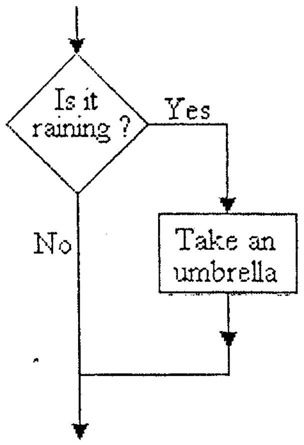
- IF condition
  THEN *action*

  Note! the condition must evaluate to TRUE, which can be seen in the IF....THEN example.

- IF condition
  THEN *action*
  ELSE *different action*

The first selection control structure is the IF....THEN statement. As already mentioned, the condition must evaluate to TRUE therefore, in the following example, it must be true that it is raining.

FLOWCHART	*PSEUDOCODE*
IF condition	IF it is raining
THEN *action*	THEN take an umbrella



There are certain disadvantages associated with white box testing. These disadvantages include:

- the number of distinct paths in most programs tends to be extremely large resulting in impractical testing
- a program that is completely path tested could still contain many errors
- a program may contain several missing paths
- there are problems with while loops which are not properly tested, as a result of loop execution from the **Boolean** test data never being TRUE

In summary, white box and black box testing strategies complement each other.

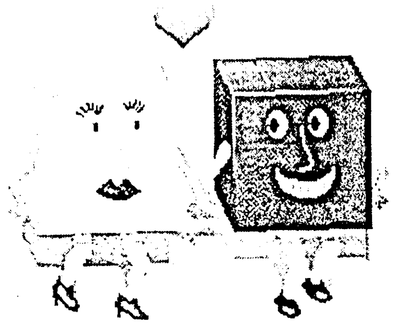


Figure 5.35 White box and black box testing complement each other

5.7.2 Unit testing

Unit testing involves testing individual modules in a program and is a similar method to white box testing. Unit testing is the second option of the *program verification and testing* phase of the software development lifecycle.

Unit tests verify the design and implementation of all the components. Special test software must be used to act out the roles of the modules which the one under test will interact. When the lowest level components have been tested, modules are combined into subsystems that are tested in a similar manner. The procedure continues until the complete system is finally assembled and tested as a whole.

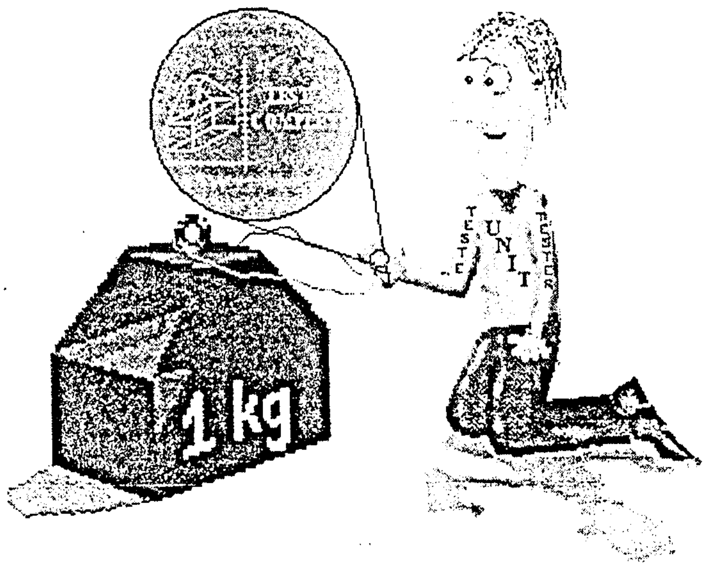
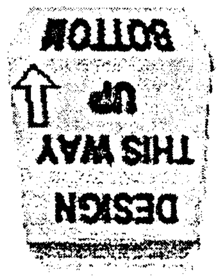


Figure 5.36 Unit testing

There are however problems that occur when one module is tested in isolation from other modules:

- the test software may itself be a complex program, requiring considerable development and testing time

- preparation of the test data may involve considerable effort



Due to its manner of testing, unit testing can be associated with bottom-up rather than top-down design and is therefore not a

5.9 Program maintenance

Once the system has been shown to work and has fulfilled the original requirements the software can be released to the customer. The development phase may be completed but it is still normal for the system developers to continue to maintain and possibly upgrade the hardware and software for the particular design. *Program maintenance* is the eighth phase of the software development lifecycle.

The maintenance phase is concerned with:

- [ ] resulting from a customer's observations of any shortcomings following the initial installation and use
- [ ] over the lifetime of a program
- adaptation to run on hardware different to that for which the software was originally intended

6 Programming languages

Programming languages can be seen as either close to human languages (high level languages) or close to binary (low level languages).

With programming languages as with other goods and services, consumers and producers seek constant improvement. The more control and power each progression in programming has the more the programmers see the possibilities in increasing the capabilities further. The numerous advancements in programming languages means that even though there is still software being used that was written in languages such as BASIC and COBOL, these languages are becoming gradually [ ].

When looking at high level programming languages specifically within the high level sector there are three main categories in which they can be placed:

- Logic languages
- [ ]
- New programming languages

6.1 Logic languages

These types of programming languages are also known as non-procedural programming languages, they tend to be associated with **artificial intelligence** as it concentrates on [ ]. For example a question is asked and the computer derives a logical answer instead of the programmer programming in how to obtain the answer. For the computer to be able to find the answer a description of the relationship between input data and the output data is needed. These programs are successful in that they are concise.

There are many characteristics of logic programming. One of the main features is the use of control statements.

Logic programming concentrates on what, not how, it uses a collection of facts and rules about those facts to draw up solutions or answers to input questions. It does not use the algorithmic toolkit.

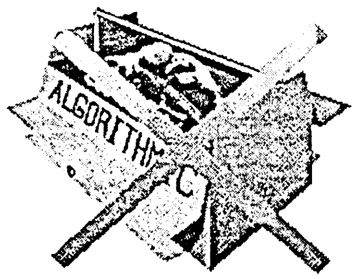


Figure 6.1 Logic programming does not use the algorithmic tool kit

6.2 Prolog

Prolog stands for PROgramming LOGic. It is a high level language which is based on formal logic system where by the programmer states the relationship between the assumptions and conclusions.

It was designed in the 1970s by Alain Colmerauer and colleagues at the University of Marseilles, France. Prolog has been used in numerous ways but is most predominantly known for its use in fifth generation computing.

The language itself is used in expert systems. These are programs which have knowledge about a particular subject built into them. Expert systems have helped in various areas, for example to diagnose some illnesses. Expert systems consist of facts (about the illness) and rules (which state how the illness is diagnosed). So, measles is a disease with symptoms spots on the skin and if the patient has itchy red spots, then the chances are they have measles.

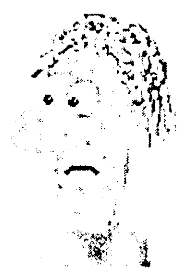


Figure 6.2 A patient with measles

A prolog program will consist of a known set of facts, and rules about those facts. Prolog actually performs pattern matching. It puts facts and rules together to give an answer to the appropriate question.

Examples of possible facts:

In English	**In Prolog**
`sheep eat grass`	`eat(sheep,grass).`
`cows eat grass`	`eat(cows,grass).`

Examples of possible questions:

```
do sheep eat grass ?-eat(sheep,grass).

do cows eat grass  ?-eat(cows,grass).
```

You ask, `eat(sheep,grass)` and the program looks for a match among the facts and returns true (if it matches) or false (no match found).

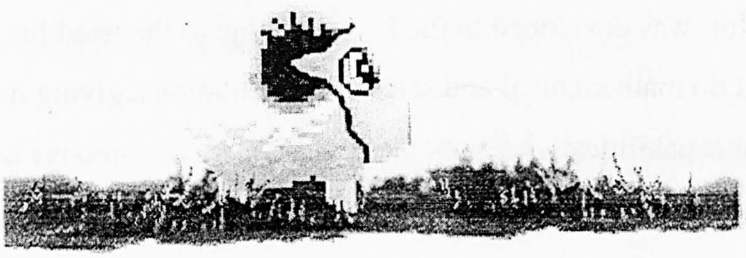


Figure 6.3 A sheep eating grass

6.3 Procedural languages

These languages require the programmer to define clearly all the steps of the solution. Most well known languages are procedural. Pascal is a [ ] language.

The program is split into a number of steps called procedures which are then carried out in sequence. These steps indicate the way the program runs or the flow of control. Procedural languages tend to be close to human language, which enables programmers to read them with more ease than languages such as assembly language.

Procedural languages are much more exhaustive than many other forms of programming language. For any non-trivial program there are [ ] to write, especially when it is used to produce a complete software package. This is due to every step needs to be clearly specified by the programmer and all possibilities covered by the action of the program. Procedural languages demand [intensive effort] and [ ] to create.

6.4 Examples of procedural languages

6.4.1 FORTRAN

Formula TRANslator was developed in the [ ] due to the need for a programming language that could do mathematical and scientific calculations, giving it extensive and powerful numerical capabilities.

The language itself is based on simple English type statements, such as:

```
integer I, MX, MN, A(100)
real RS
read (A(I), I=1, 100)
MX = A(1)
MN = A(1)
do 10 I = 2, 100
if (A(I).gt.MX) = A(I)
if (A(I).lt.MN) MN = A(I)
```

This is just a small fragment of FORTRAN code

Starting at the top of this code the first line introduces the 3 integers. This is followed by the real number variable of RS. Line 3 reads the contents of A from the input devices, this indicates that both MX and MN are the contents. The do command indicates that everything from here to line 10 should be run in a loop.

FORTRAN was one of the first high level languages to be implemented and also one of the most popular in its time. Although widely used FORTRAN had limitations due to its mathematical and scientific nature. Therefore if a company wanted a software package primarily for wordprocessing FORTRAN would not be the most applicable language.

When it was designed there was no consideration to the complexity of the language, its primary aim was to be concise. This made FORTRAN lack various aspects for program enhancing which are usually associated with high level languages.

There is an updated version of FORTRAN called FORTRAN 77, which has incorporated more aspects, such as running programs within a program (subroutines) and structural nesting for example a loop within a loop. Due to its early popularity there are many amount of programs written in FORTRAN.7. A new version has been developed called FORTRAN 90.

FORM ULA
TRAN SLATION

Figure 6.4 FORTRAN

6.4.2 COBOL

COBOL is described as been completely opposite to FORTRAN in almost every aspect, other than they are both high level languages and very popular.

COBOL is a multipurpose language that is [ ], and is useful for [ ]. It is very popular for business applications, especially ones being used on larger computer systems.

COBOL was designed to be clear and have good readability, which reduced the look of being mathematical that some other programming languages have. This gives the language a human perspective. The COBOL programming language itself has similarities to natural language.

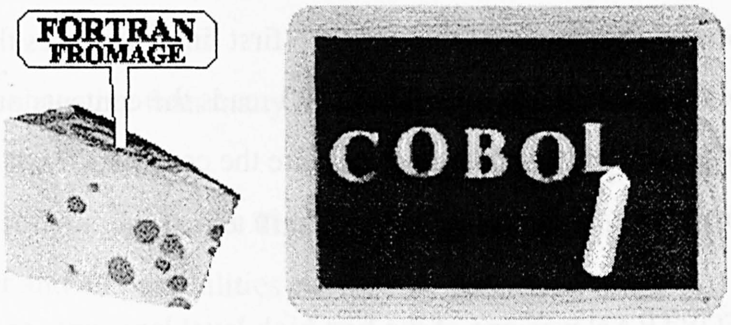
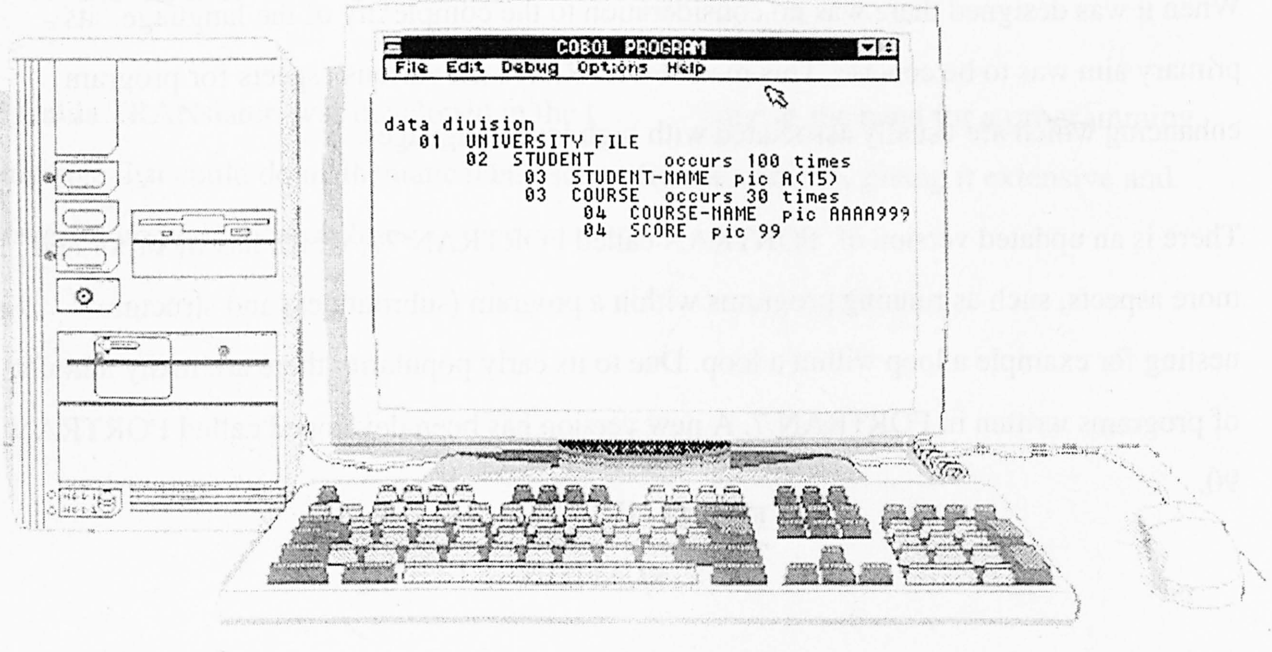


Figure 6.5 COBOL is different to other languages

COBOL fragment:



```
data division
   01  UNIVERSITY FILE
       02  STUDENT        occurs 100 times
           03  STUDENT-NAME  pic A(15)
           03  COURSE  occurs 30 times
               04  COURSE-NAME  pic AAAA999
               04  SCORE  pic 99
```

Figure 6.6 COBOL fragment

This is a small section of COBOL programming. The sample shows a university students name, relevant course and the score attained.

6.4.3 BASIC

BASIC consists of hundreds of instructions all of which resemble a natural language. In order for these instructions to work there is also a set of rules to follow, called syntax. All instructions must have a line number:

```
10   CLS
20   PRINT "hello"
30   END
```

If a line is not numbered then the instruction will be carried out as soon as the program is run, and it will not be in sequence.

The language was originally designed in [   ], to help teach the concepts of programming. BASIC was classed as the [   ] language.

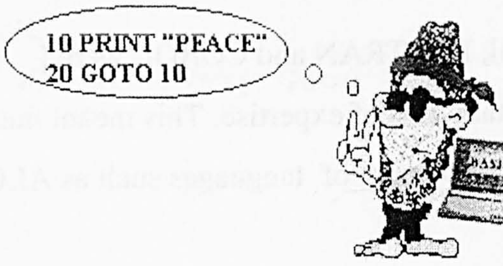


Figure 6.7 BASIC was designed in 1963

Early BASIC could sometimes look confusing as GOTO commands would be followed by a line number, which meant the program would jump to the line number indicated.

Fragment of Basic:

```
10   REM - PRIME NUMBERS LESS THAN 100
20   N=N+10
30   IF N=100 THEN GOTO 120
40
↓
120  END
```

This fragment of code takes a value N and adds 10 to it. When N is equal to 100 the program ends. The REM statement at line 10 indicates that the line is only a remark and not part of the actual program.

The complexity changed as BASIC followed the improvements in the personal computer revolution. Microsoft enhanced the languages capabilities and upgraded to QuickBasic, which moved the language forward. The line numbers were eliminated and modern features such as **data types** and **subprograms** were introduced.

6.4.4 Pascal

As the predecessors of pascal, FORTRAN and COBOL were [ ] specifically designed for certain uses of expertise. This meant there was a need for an all round language hence the development of languages such as ALGOL60 and Pascal.

There are various rules that are associated with Pascal. These rules dictate the variables that can be used whilst programming. Pascal's facilities allow the definition of data types, such as only real numbers.

Pascal has data types that are both program defined and user defined, usually placed at the beginning of the program. Pascal is used in many different ways due to its versatility, although it is one of the preferred high level languages that is taught at most levels of programming.

Along with most other high level languages Pascal is unable to run directly on the machine as the program is not written in binary. It is therefore necessary to have [ ]. The compiler turns the program into [ ] so that it can run. The Pascal language itself is quite easy to follow as the instructions are based on natural language.

Fragment of Pascal:

```
IF score >=70 THEN
     BEGIN
       Write ('Successful');
       IF SCORE >=80 THEN
          BEGIN
            Write ('and')
            IF Score >=90 THEN
               Write ('distinguished')
            ELSE
               Write ('Very Good')
          END
END
```

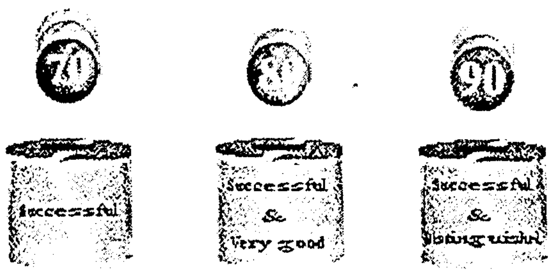


Figure 6.8 Graphical representation of the code

There is a variable score which has a value (taken in elsewhere in the program). This section of the code carries out one action which writes a phrase on the screen, depending on the value of the score. If the score is greater that or equal to 70 (>=70) then the output is the word '`Successful`'. The code continues to run until it has correctly categorised the score.

6.4.5 C and C++

C was designed by Bell Laboratories in the early [ ]. C soon became one of the most important development languages for MS-DOS and UNIX enthusiasts. This was mainly due to the lack of other languages that could perform as well within these environments.

The language itself is high level and uses the natural language statements IF, THEN, ELSE. It is a well structured language that is extremely well suited to engineering but has excellent all round abilities. Due to its immense use C has a large range of standard functions available. One major advantage of C is that it can be compiled into a stand alone application, such as an .exe file.

As a language C can adopt a [ ] which means the final program can be split into several more manageable separate parts or files. When learning C it can be made easier if the student already knows Pascal or a different procedural language.

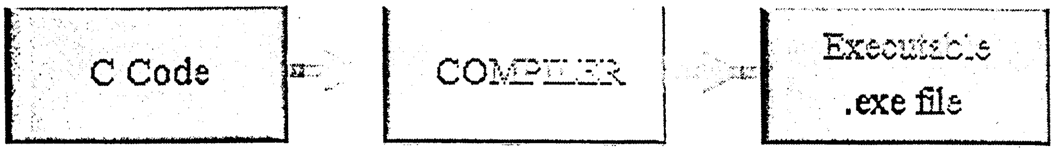


Figure 6.9 C can adopt a modular approach

Small C program:

```
line
 1    /* simple program to write on the screen */

 2    #include <stdio.h>

 3    main()
 4    {
 5        printf("Hello World!");
 6    }
```

This program starts off with line 1 a comment, this is the text between the symbols /* and */. Line 2 is the standard input/output library of the program, this allows a selection of pre-written functions to be used. Line 3 is the point at which the program executes (runs) from. The braces of the program are shown in lines 4 and 6, these enclose the main routine. Line 5 is the main routine.

Once this program has been run with aid of a C compiler it will put up on screen Hello World!

In an attempt to add the features of a general purpose object-oriented language to C, Bjorne Stroustrup created C++. Not only is C++ an improvement on C but it also fixes some of the problems that exist in the C programming language.

If you look at the first section of the C example again only in C++, it will look similar to the following.

C++ program:

```
void main()

{

    cout<<"Hello World!"

}
```

This C++ program produces the same output as the previous C program, Hello World!

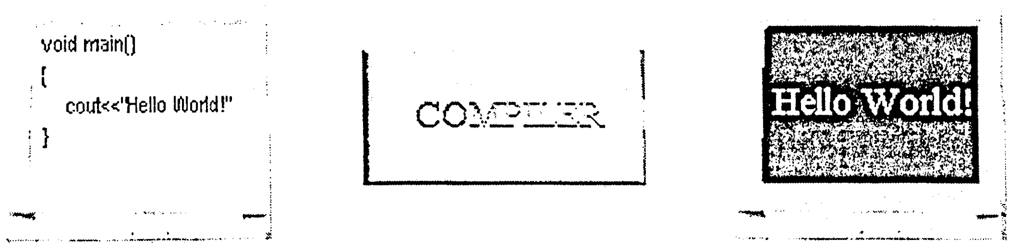


Figure 6.10 A program producing an output

6.5 New programming languages

Programming languages are no different from any other technological advancements. There is always a need for faster more powerful languages. This has never been more apparent then with the vast use and demand for the [ ]. Although the existing languages are compatible with Internet programming, the actual programmers felt there was a gap in the programming which could be filled by new specific languages.

Primarily most new programming languages are object oriented, for example [ ]. Object oriented programming enables the programmers to not only define the data types and structure but also the functions. An object is basically a unit of data and code which enable the programmers to use pre-built objects. Having objects both reduces the amount of actual code needed.

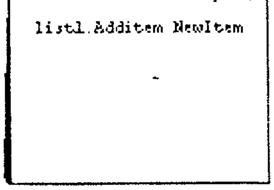
The uses and fragments of a selection of new programming languages will be explained in this section.

6.5.1 Visual Basic

Visual Basic was based around an older language BASIC. It is seen as a good language to use for a graphical programming environment. Visual Basic is not completely object oriented but is based around some object oriented concepts. The language was designed by Microsoft several years ago to enable programmers to create Windows applications with greater ease and speed.

As with the majority of programming languages once they have been released the actual programmers demand more from them. Visual Basic is now in [ ] since its introduction in [ ].

The actual programs readability is good as it is based on natural language. For example the instruction to add a new item is:



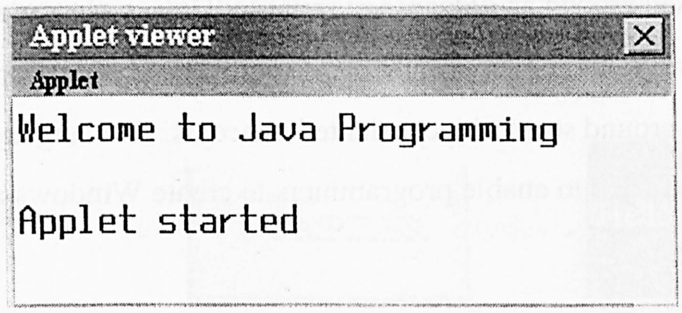
```
list1.AddItem NewItem
```

Figure 6.11 The instruction to add a new item

```
//A first program in Java
    import java.applet.Applet;  //import Applet class
    import java.awt.graphics;   //import Grphics class

    public class welcome extends Applet  {
        public void paint ( Graphics g )
        {
          g.drawstring ("Welcome to Java Programming!",25,25);
        }
    }
```

This fragment of java would appear on the screen as follows:



```
Welcome to Java Programming

Applet started
```

Figure 6.13 A fragment of Java

6.5.3 JavaScript

JavaScript is not the same as Java or produced by the same company. JavaScript was designed by [ ] specifically for use with the [ ]. JavaScript is a simpler language that can be used by non-programmers. It does not create applets or stand alone applications.

Although they are not the same language JavaScript does support Java's expressions and basic flow control. JavaScript is not fully object oriented as it does not use classes or inheritance. Unlike Java the programming is not compiled by the server just interpreted by the client. The objects that are present in the program are checked at the run time as the program is not compiled.

In order to use JavaScript programming language it is essential to be using a web browser that is JavaScript embedded, for example Netscape Navigator (post version 2.0) or Microsoft Internet Explorer (post version 3.0).

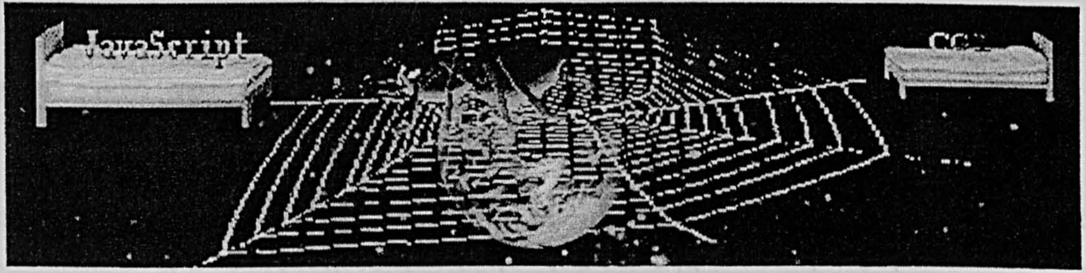


Figure 6.14 A JavaScript embedded web browser

Basically JavaScript looks very similar to many other HLL, with a natural language look to it.

```
<HTML>
<HEAD>
<SCRIPT LANGUAGE = JavaScript">
document.write ("Hello World.")
</Script>
<BODY>
</BODY>
</HTML>
```

Starting from the top of this fragment of code, the first line is opening the HTML page. The HEAD statement indicates that what follows will be at the top of the page, the type of language is the depicted. Next the contents is entered: Hello World, this is done using the line of code: `document.write ("Hello World.")`is actually the only piece of JavaScript the rest is part of the HTML page. Following this is the closing of the script then the body and the HTML page.

Learning outcomes

At the end of this material you should be able to:

- understand and explain the basic principles of structured programming
- explain key elements of the software development life cycle
- be able to describe features of different programming languages
- outline how various programming languages were developed and why
- understand the importance of structured programming

Suggested reading

The number for each book is the ISBN for that book. This number uniquely identifies the book. You can use this number to order the book from a library or bookshop.

- Principles of Programming Languages
  Bruce J MacLennan
  CBS College Publishing
  0-03-005163-6

- Foundations of Computer Technology Systems
  A J Anderson
  Chapman & Hall
  0-412-59810-8

- Programming Languages - Structures and Models
  H L Dershem & M J Jipping
  PWS Publishing Company
  0-334-94740-9

- Algorithmics - The Spirit of Computers

  David Harel

  Addison-Wesley

  0-201-50401-4

- Software Tools & Techniques for Electronic Engineers

  Keith Jobes

  McGraw Hill

  0-07-707720-2

The microcomputer

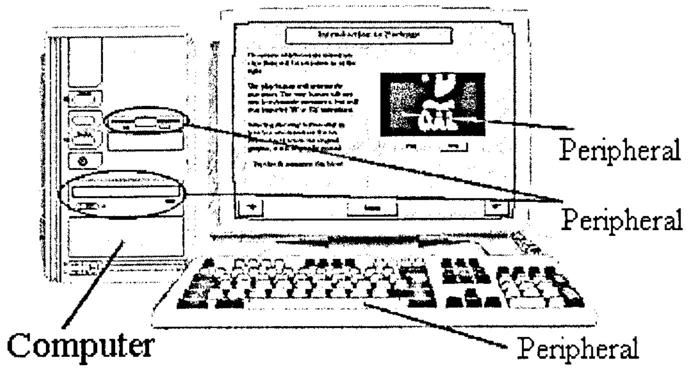


Figure 2.1 A microcomputer

A microcomputer or PC (Personal Computer) consists of hardware and software. Hardware are items like screens (monitors), the peripherals (such as printer or keyboard) and the microprocessor itself.

The microprocessor is also known as the CPU (Central Processing Unit) and is often described as the brain of a microcomputer.

Output peripherals

- **Monitor**
- **Printers**
- **Speakers**
- **Modem (transmitter)**

Input peripherals

- **Keyboard**
- **Scanner**
- **Mouse**
- **Joystick**
- **Modem (receiver)**
- **Microphone**
- **Lightpen**

Computer memory

Computer memory has two main categories: primary & secondary memory

Primary memory has two main types

i) **RAM** (Random Access Memory). Here the data is held for processing. RAM is volatile. This means that any information stored when the power supply is cut will be lost, hence the need for backups.

ii) **ROM** (Read Only Memory). ROM consists of programs and instructions which are required for the operating system to be loaded. Usually held here is Basic Input Output System (BIOS) data.

Secondary memory

When large amounts of data need to be stored, secondary memory is used. This comes in numerous forms. As technology advances the number of devices available for secondary memory are increasing, along with the speed of transfer and the capacity of the device. A few examples follow:

- 3.5 inch floppy disk
- CD - WORM
- magnetic tapes

5.2 Requirements analysis

In order to analyse a problem the following stages have to be determined:

- problem solving
- [specifications (input / output)]

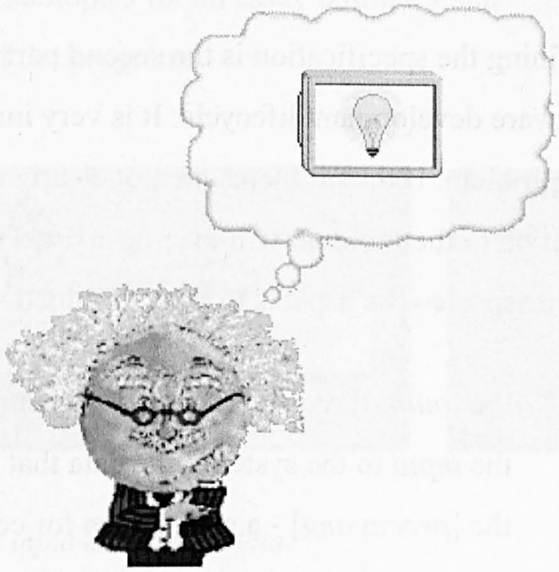


Figure 5.4 Problem solving

5.2.1 Problem solving

Problem solving is the first part of the [*requirements analysis*] phase of the software development lifecycle and:

- involves careful analysis and investigation to discover the nature of the problem
- includes development of a [logical process or design of a solution] before use of a computer

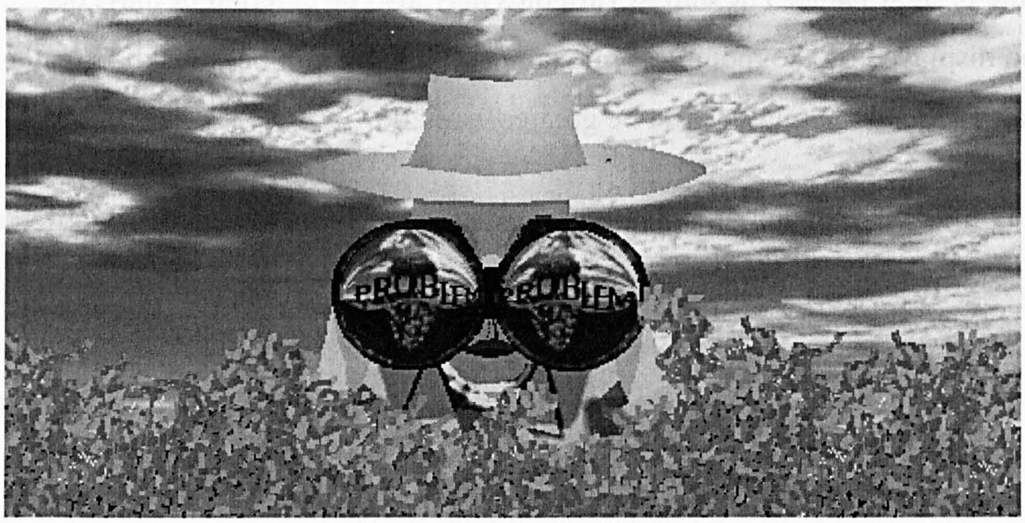


Figure 5.5 Discovering the nature of the problem

5.2.2 Specifications (input / output)

Defining the specification is the second part of the *requirements analysis* phase of the software development lifecycle. It is very important to clarify the processes needed to solve the problem. If the problems are not clearly defined then it is unlikely that a satisfactory solution to the problem will ever be arrived at. It is important to split up the problem into three aspects. The aspects to be determined are as follows:

- the [*output*] required from the system - the materials generated by a program
- the *input* to the system - the data that a program requires to produce its results
- the [*processing*] - a mechanism for converting input to output

A well designed program will always have this input, process, and output, as shown in the diagram below.

$$\begin{tikzcd}\text{INPUT} \arrow[r] & \boxed{\text{PROCESS}} \arrow[r] & \text{OUTPUT}\end{tikzcd}$$

Figure 5.6 The aspects of the problem

The specifications should clearly state what inputs the program should accept and what outputs will be produced by these inputs. The input variables may be either external variables, such as speed and temperature, or user controls, such as asking the user to enter a control value or their name. The output variables may also be external or may be information displayed on the monitor. For example, an input number must be greater than 0 but less than 100, and the output must be greater than 200.

The working limits that the input and output must adhere to should also be specified along with other information regarding them such as [tolerance, resolution and precision].

The process box can be broken down into more detailed process boxes with data flowing into and out of each. This is similar to developing subtopics for an essay outline. Each process box represents a [*module* or major task] of the program.

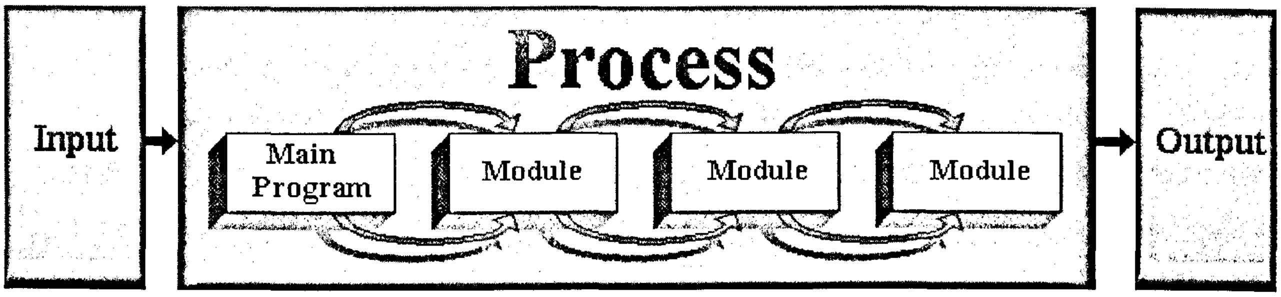


Figure 5.7 A process box with input and output data

5.3 Program design

5.3.1 Definition of the problem

Before any preparation or programming can begin, the problem has to be defined. Definition of the problem is the first part of the [*program design*] phase of the software development lifecycle. The problem must be understood, and a solution must be engineered. It is logical to begin with a requirements list. This is a set of written statements that specify what the software must do or how it must be structured. These requirements are specified by a systems analyst or a software engineer.

A number of problems should also be studied including:

- [how would people use the system]
- what data will the system be processing
- [what hardware will be used to implement the system]
- what factors that might influence the performance of the system

The factors and the requirements should then be compared. This process is valuable because it helps to produce a completed program or system that will be more naturally organised than if the requirements alone drove the initial design.

Activity 8

Find another diagram or graphic describing the software lifecycle. Sketch the diagram below and indicate the source (book title, author or other source).

Answer

Students should submit any reasonable diagram or graphic.

5.3.2 Expressing algorithms

Flowcharts

Flowcharts are a very important aspect of programming and are the first option of *expressing algorithms* in the [*program design*] phase of the software development lifecycle. They can be seen as the blueprint of the programs duties. These visual, diagrammatic techniques are one way of presenting the [control flow] of an algorithm in a clear and readable fashion. A flowchart shows the logic of an algorithm, emphasising the individual steps and their interconnections. This illustrates the way in which control flows from one action to the next.

Over the years a relatively standard symbolism emerged when presenting a flowchart and below are examples of some of the representations.

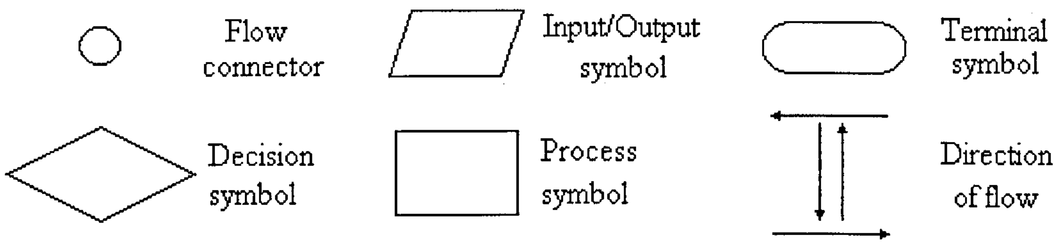


Figure 5.8 Representative symbols used in flowcharts

To the right is a flowchart that shows how to work through a bag of pick and mix sweets:

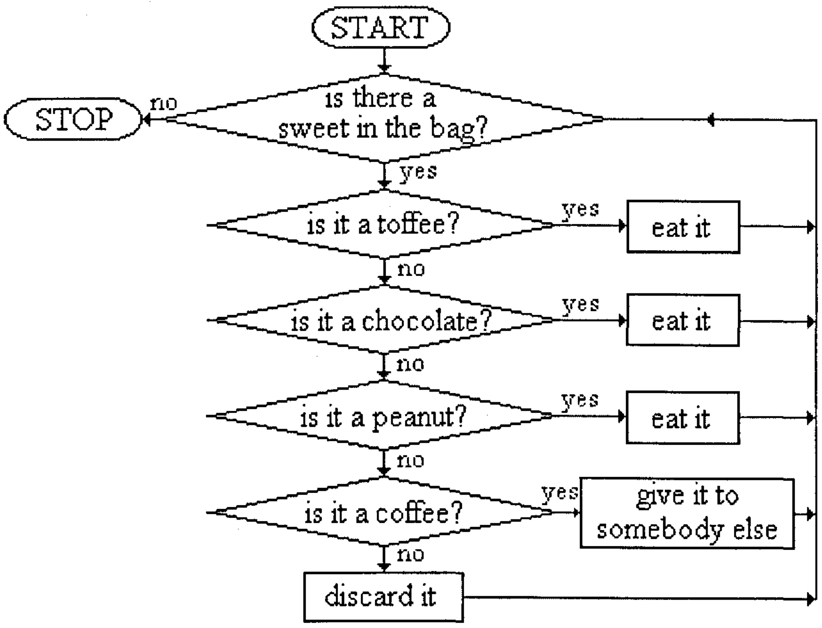


Figure 5.9 A flowchart showing how to eat a bag of sweets

Activity 9

Draw a flowchart to outline the process of going out for an evening eg get ready, meet friends, decide where to go depending on how much money you have and so on.

Answer

Students should have some decision boxes eg cinema or night-club. Otherwise, any reasonable flowchart is acceptable.

The advantages of using flowcharts are as follows:

- there is close correspondence with underlying code
- flowcharts are good for procedural specifications

Figure 5.10 Advantage of using flowcharts

Pseudocode

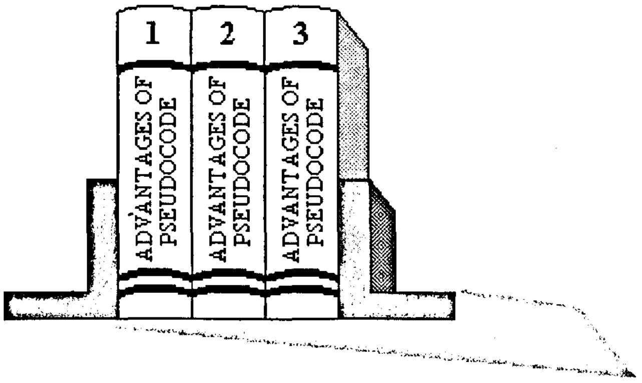
Pseudocode is an alternative method to using flowcharts and is the second option of *expressing algorithms* in the *program design* phase of the software development lifecycle. Pseudocode, also called Structured English, is the least formal means for stating the program ideas. The idea behind pseudocode is to write English-language statements that are almost as specific as the code itself but flexible enough to be translated into more than one programming language.

An example of pseudocode can be seen below:

```
for every sweet in a packet
    do pull out a sweet
    if it is a toffee
        then eat it
    else
        if it is a chocolate
            then eat it
        else
            if it is a peanut
                then eat it
            else
                if it is a coffee
                    then give it to somebody else
                else discard it
```

The advantages of using pseudocode are as follows:

- it is an appropriate method when detail is not completely known
- it is quick and easy
- it is a good method for communicating algorithms to end-users



Once the requirements, the specifications and the problem have been defined and it has been decided that the problem can be solved with a suitable structured solution then it is time to start coding the problem.

6 Programming languages

Programming languages can be seen as either close to human languages (high level languages) or close to binary (low level languages).

With programming languages as with other goods and services, consumers and producers seek constant improvement. The more control and power each progression in programming has the more the programmers see the possibilities in increasing the capabilities further. The numerous advancements in programming languages means that even though there is still software being used that was written in languages such as BASIC and COBOL, these languages are becoming gradually [obsolete].

When looking at high level programming languages specifically within the high level sector there are three main categories in which they can be placed:

- Logic languages
- [Procedural languages]
- New programming languages

6.1 Logic languages

These types of programming languages are also known as non-procedural programming languages, they tend to be associated with **artificial intelligence** as it concentrates on [what is to be done rather that how to do it]. For example a question is asked and the computer derives a logical answer instead of the programmer programming in how to obtain the answer. For the computer to be able to find the answer a description of the relationship between input data and the output data is needed. These programs are successful in that they are concise.

There are many characteristics of logic programming. One of the main features is the use of control statements.

Logic programming concentrates on what, not how, it uses a collection of facts and rules about those facts to draw up solutions or answers to input questions. It does not use the algorithmic toolkit.

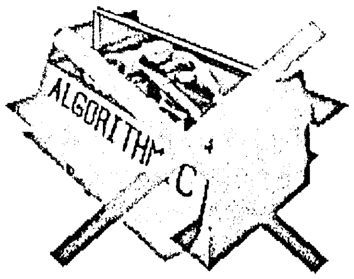


Figure 6.1 Logic programming does not use the algorithmic tool kit

6.2 Prolog

Prolog stands for PROgramming LOGic. It is a high level language which is based on formal logic system where by the programmer states the relationship between the assumptions and conclusions.

It was designed in the 1970s by Alain Colmerauer and colleagues at the University of Marseilles, France. Prolog has been used in numerous ways but is most predominantly known for its use in fifth generation computing.

The language itself is used in expert systems. These are programs which have knowledge about a particular subject built into them. Expert systems have helped in various areas, for example to diagnose some illnesses. Expert systems consist of facts (about the illness) and rules (which state how the illness is diagnosed). So, measles is a disease with symptoms spots on the skin and if the patient has itchy red spots, then the chances are they have measles.

Figure 6.2 A patient with measles

A prolog program will consist of a known set of facts, and rules about those facts. Prolog actually performs pattern matching. It puts facts and rules together to give an answer to the appropriate question.

Examples of possible facts:

In English	In Prolog
`sheep eat grass`	`eat(sheep,grass).`
`cows eat grass`	`eat(cows,grass).`

Examples of possible questions:

```
do sheep eat grass ?-eat(sheep,grass).

do cows eat grass ?-eat(cows,grass).
```

You ask, `eat(sheep,grass)` and the program looks for a match among the facts and returns true (if it matches) or false (no match found).

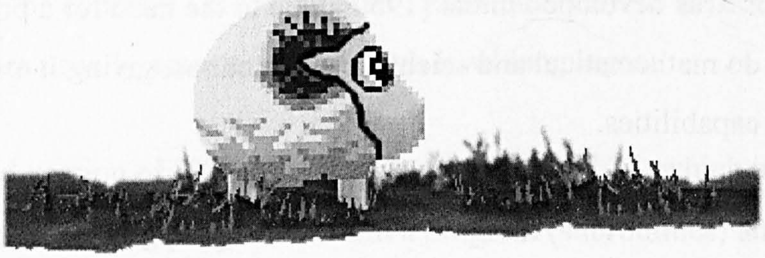


Figure 6.3 A sheep eating grass

6.3 Procedural languages

These languages require the programmer to define clearly all the steps of the solution. Most well known languages are procedural. Pascal is a [procedural] language.

The program is split into a number of steps called procedures which are then carried out in sequence. These steps indicate the way the program runs or the flow of control. Procedural languages tend to be close to human language, which enables programmers to read them with more ease than languages such as assembly language.

Procedural languages are much more exhaustive than many other forms of programming language. For any non-trivial program there are [many lines of code] to write, especially when it is used to produce a complete software package. This is due to every step needs to be clearly specified by the programmer and all possibilities covered by the action of the program. Procedural languages demand [intensive effort] and [considerable time] to create.

6.4 Examples of procedural languages

6.4.1 FORTRAN

Formula TRANslator was developed in the [1950s] due to the need for a programming language that could do mathematical and scientific calculations, giving it extensive and powerful numerical capabilities.

The language itself is based on simple English type statements, such as:

```
integer I, MX, MN, A(100)
real RS
read (A(I), I=1, 100)
MX = A(1)
MN = A(1)
do 10 I = 2, 100
if (A(I).gt.MX) = A(I)
if (A(I).lt.MN) MN = A(I)
```

This is just a small fragment of FORTRAN code

Starting at the top of this code the first line introduces the 3 integers. This is followed by the real number variable of RS. Line 3 reads the contents of A from the input devices, this indicates that both MX and MN are the contents. The do command indicates that everything from here to line 10 should be run in a loop.

FORTRAN was one of the first high level languages to be implemented and also one of the most popular in its time. Although widely used FORTRAN had limitations due to its mathematical and scientific nature. Therefore if a company wanted a software package primarily for wordprocessing FORTRAN would not be the most applicable language.

When it was designed there was no consideration to the complexity of the language, its primary aim was to be concise. This made FORTRAN lack various aspects for program enhancing which are usually associated with high level languages.

There is an updated version of FORTRAN called FORTRAN 77, which has incorporated more aspects, such as running programs within a program (subroutines) and structural nesting for example a loop within a loop. Due to its early popularity there are many amount of programs written in FORTRAN.7. A new version has been developed called FORTRAN 90.

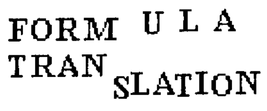


Figure 6.4 FORTRAN

6.4.2 COBOL

COBOL is described as been completely opposite to FORTRAN in almost every aspect, other than they are both high level languages and very popular.

COBOL is a multipurpose language that is [problem oriented], and is useful for [commercial data processing]. It is very popular for business applications, especially ones being used on larger computer systems.

COBOL was designed to be clear and have good readability, which reduced the look of being mathematical that some other programming languages have. This gives the language a human perspective. The COBOL programming language itself has similarities to natural language.

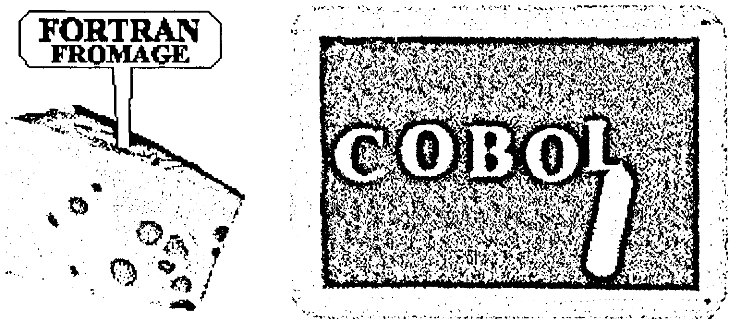
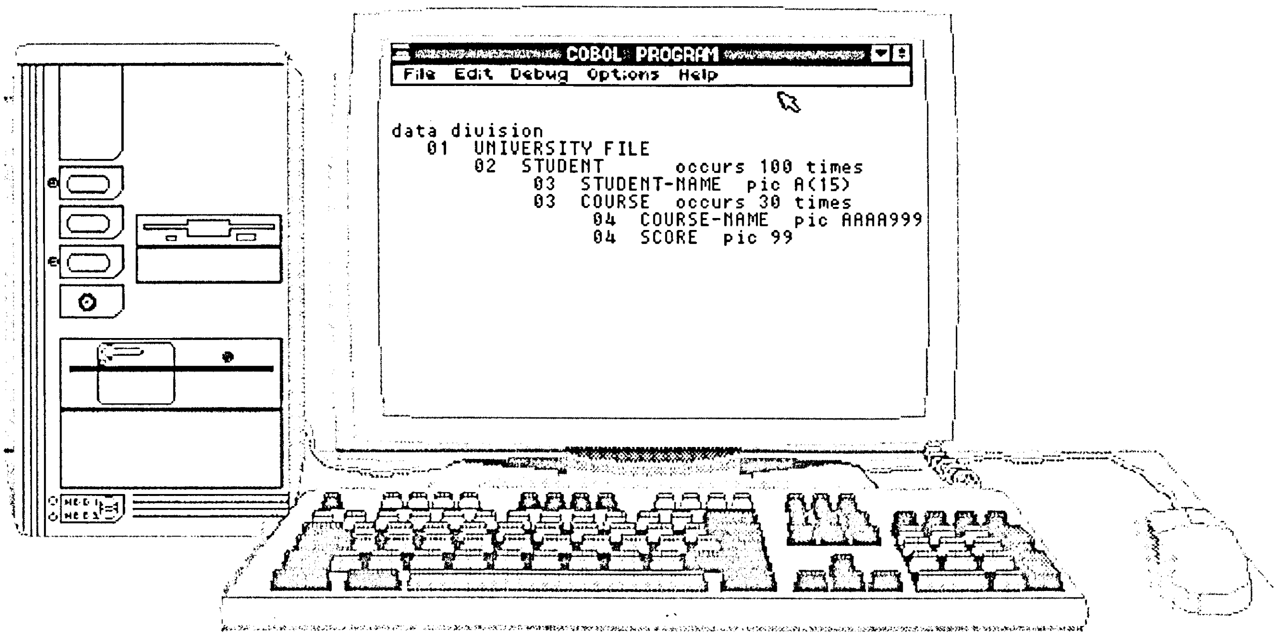


Figure 6.5 COBOL is different to other languages

COBOL fragment:



```
data division
   01  UNIVERSITY FILE
       02  STUDENT       occurs 100 times
            03  STUDENT-NAME  pic A(15)
            03  COURSE  occurs 30 times
                 04  COURSE-NAME  pic AAAA999
                 04  SCORE  pic 99
```

Figure 6.6 COBOL fragment

This is a small section of COBOL programming. The sample shows a university students name, relevant course and the score attained.

6.4.3 BASIC

BASIC consists of hundreds of instructions all of which resemble a natural language. In order for these instructions to work there is also a set of rules to follow, called syntax. All instructions must have a line number:

```
10   CLS
20   PRINT "hello"
30   END
```

If a line is not numbered then the instruction will be carried out as soon as the program is run, and it will not be in sequence.

The language was originally designed in [1963], to help teach the concepts of programming. BASIC was classed as the [first easy to use] language.

Figure 6.7 BASIC was designed in 1963

Early BASIC could sometimes look confusing as GOTO commands would be followed by a line number, which meant the program would jump to the line number indicated.

Fragment of Basic:

```
10   REM - PRIME NUMBERS LESS THAN 100
20   N=N+10
30   IF N=100 THEN GOTO 120
40
↓
120  END
```

This fragment of code takes a value N and adds 10 to it. When N is equal to 100 the program ends. The REM statement at line 10 indicates that the line is only a remark and not part of the actual program.

The complexity changed as BASIC followed the improvements in the personal computer revolution. Microsoft enhanced the languages capabilities and upgraded to QuickBasic, which moved the language forward. The line numbers were eliminated and modern features such as **data types** and **subprograms** were introduced.

6.4.4 Pascal

As the predecessors of pascal, FORTRAN and COBOL were [specialist languages] specifically designed for certain uses of expertise. This meant there was a need for an all round language hence the development of languages such as ALGOL60 and Pascal.

There are various rules that are associated with Pascal. These rules dictate the variables that can be used whilst programming. Pascal's facilities allow the definition of data types, such as only real numbers.

Pascal has data types that are both program defined and user defined, usually placed at the beginning of the program. Pascal is used in many different ways due to its versatility, although it is one of the preferred high level languages that is taught at most levels of programming.

Along with most other high level languages Pascal is unable to run directly on the machine as the program is not written in binary. It is therefore necessary to have [compilers]. The compiler turns the program into [binary] so that it can run. The Pascal language itself is quite easy to follow as the instructions are based on natural language.

Fragment of Pascal:

```
IF score >=70 THEN
     BEGIN
       Write ('Successful');
       IF SCORE >=80 THEN
          BEGIN
            Write ('and')
            IF Score >=90 THEN
               Write ('distinguished')
            ELSE
               Write ('Very Good')
          END
END
```

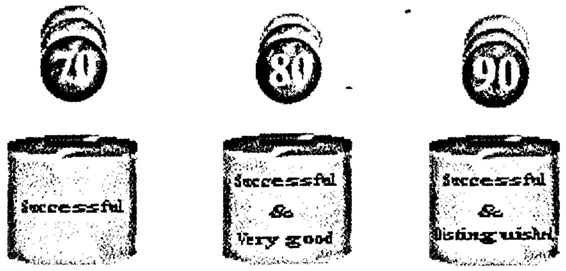


Figure 6.8 Graphical representation of the code

8 Workplan

No	Suggested Activity	Assessment
1	Work through sections 2 and 3: The computer environment The history of programming languages	Complete student activities 1, 2, 3, 4, 5 and end of section questions
2 and 3	Work through section 4: Introduction to structured program design	Complete student activities 6, 7 and end of section questions
4 and 5	Work through section 5: The software development lifecycle	Complete student activities 8, 9, 10 and end of section questions
6	Review section 4 Work through section 6: Programming languages	Complete end of section questions
7	Review section 5 Work through section 7: Sample programs	

9 Suggestions for assignment

Here are three programs littered with mistakes. Type them in as they are here: *one* program per file and de-bug them. Use the error messages generated when you compile to correct the mistakes. The mistakes can be trivial (wrong punctuation) up to more serious (missing variable declarations) and beyond, to missing chunks of the program, giving wrong answers. You should not assume that if the program runs after you have debugged it that it does anything sensible or useful! Always look at the run window using Alt & F5. The aim is to get the programs working!

The students should take the three sample programs and de-bug them. This can be done in any language and is good initial practice in the use of a programming language without the demands of writing a program.

9.1 Suggestion for Pascal assignment

Program 1

```
program repeating(input,output)

var first_no: interger;

beginn

writeln('Hello, please give me a number..');

readln(first_no)

writeln(@and another number please');

readln(sec_no);

while first_no > 5 do
begin
secno:=secno + 1
first_no:=first_no - 1
end;

writeln('Thank you')

end.
```

Program 2

```
program if_prog(input, output);

const weighting1 = 0.6;
     weighting2 = 0.4;

var result: real;
     mark1:integer;

behin

writeln('Please enter assiognemnt mark');
readln(mark1);
writeln('Please enter exam mark');
readln(mark2);

result := (mark1 * ??) + (mark2 * weighting2);


if result > 60 write('well done')

else

write('OK I suppose)

end.
```

Program 3

Students need to look carefully at this program and ask what its purpose is: this is an advance on de-bugging only.

```
program more_if(input,output);

var something: interger;

begin

writeln('Some number please? Your number is given a rating');

readln(something);

if something < 5 then

    write('Low!')

else

    if (something > 5) and (something < 10) then

        write('Better')

    else

        if something > 10 then

            write('OK')

{endif}

    {endif}
```

```
              {endif}

writeln('Why?')

end.
```

9.2 Assignment 1: sample answer

Program 1

```
program repeating(input, output)

var first_no, secno: integer;

begin

writeln('Hello, please give me a number..');

readln(first_no);

writeln('and another number please');

readln(secno);

while first_no > 5 do
begin
secno := secno + 1;
first_no := first_no - 1
end;

writeln('First no is', first_no);
writeln('Second no is', secno);
writeln('Thank you')
```

```
end.
```

Program 2

```
program if_prog(input, output);

const weighting1 = 0.6;
     weighting2 = 0.4;

var result: real;
     mark1, mark2: integer;

begin

writeln('Please enter assignment mark');
readln(mark1);
writeln('Please enter exam mark');
readln(mark2);

result := (mark1 * weighting1) + (mark2 * weighting2);


if result > 60 then
writeln('Well done')
else

writeln('You need to improve your mark');

end.
```

Appendix I

```
#!/usr/bin/perl

$default = "CHECKED";
$filename= "knowd.dat";
#-------------------------------------------------------------
sub getareas
{
        $num_acts = 0;

        if (-r $filename)
        {
                open(FILE,$filename);
                while (<FILE>)
                {
                        $line = $_;
                        chomp($line);
                        ($areas[$num_acts][0],$areas[$num_acts][1]) = split /,
                        $num_acts++;
                }
                close (FILE);
        }
}

#-------------------------------------------------------------
print "Content-type: text/html\n\n";
print "<HTML><TITLE>Knowledge Database</TITLE></HEAD>\n";
print "<BODY  background=\"../images/back.gif\">";
print "<A NAME=\"TOP\">\n";
print "<SCRIPT> window.defaultStatus = \"Helpdesk Database - NEW ENTRY\"</SCRI
print "<IMG SRC=\"../images/know.jpg\" ALT=\"Logo\" ALIGN=CENTER>";
print " <B> HelpDesk Database - NEW ENTRY</B>\n";
print "<HR>";
print "<FORM ACTION=\"../cgi-bin/enterc.cgi\" METHOD=\"POST\">\n";
print "<IMG SRC=\"../images/tick.gif\"> ";
print "<B>Step 1</B> Click on the entry area:<P>\n";

getareas();
for($loop = 0; $loop < $num_acts; $loop++)
{

        print " <INPUT TYPE=\"radio\" NAME=\"ACTION\" VALUE=\"$areas[$loop][1]
        if ( ($loop%5) == 0)
        {
                print "<BR>";
        }
        $default = "";
}
print "<HR>";
print "<IMG SRC=\"../images/tick.gif\"> ";
print "<B>Step 2</B> Type in the new query below :<BR>";
print "<INPUT NAME=\"QUERY\" SIZE=\"100\" MAXLENGTH=\"500\" VALUE=\"\">";
print "</TEXTAREA>";

print "<HR>";
print "<IMG SRC=\"../images/tick.gif\"> ";
print "<B>Step 3</B> Enter keywords to find this query (6 Maximum)\n";
print "<BR><INPUT NAME=\"KEYS\" SIZE=\"50\" MAXLENGTH=\"50\" VALUE=\"\">\n";
print "</TEXTAREA>";

print "<HR>";
print "<IMG SRC=\"../images/tick.gif\"> ";
print "<B>Step 4</B> Authorisation Password and Submit :";

print "<P>Enter Password : ";
print "<INPUT TYPE=\"password\" NAME=\"PASS\" SIZE=\"8\" MAXLENGTH=\"7\">";
print "<P> <INPUT TYPE=\"submit\" VALUE=\"SUBMIT\">\n";
print " or to start again click here ";
print "<INPUT TYPE=\"reset\" VALUE=\"Clear Form\">";
print "</FORM>\n";
print "<HR><FORM><B>OR</B> click here to : ";
print "<INPUT type=\"button\" value=\"Home Page\" name=\"button4\"onClick=\"wi
print " <INPUT type=\"button\" value=\"Query Database\" name=\"button5\"onClic
=\"window.location='../cgi-bin/queryk.pl'\">\n";
print "</FORM>\n";
print "<P><I>(c) University of Humberside - Darren Bird</I>\n";
print "</BODY></HTML>\n";
exit;
```

Appendix J

For - Introduction to Programming – devised by Amanda Bird, programmed by Darren Bird.

At

http://staff.dc.lincoln.ac.uk/~ambird/patterns

you will find a repository for *your* programming knowledge.

The idea is that you can pass on to all your colleagues (here and in Lincoln!) doing the Intro to Programming unit any really good ideas you have developed while programming. The ideas or patterns you add become a resource for everyone doing the unit. The more everyone uses this to note their ideas, the better it will be!

There is a simple structure to the repository. Every idea or pattern has a name that you give it. Please make the name explanatory e.g. The semi colon problem

You can enter your name if you wish but the default is Anonymous.

You can then check one of the boxes for a confidence level. If you think your idea is a solid and tested solution to the problem, give it a high confidence rating. If you think it could be improved (by you or someone else) give it the appropriate number of stars!

You then type in a very brief description of the problem and in the next text area, your solution.

When you are happy with the idea, click on the Submit Pattern button.

I will get the facilities for searching the collection of patterns added in the next week or two...

Thank you.

Appendix K

```
#!/usr/bin/perl

ny $title = "New Patterns";
ny $dbname = "/home/ambird/patterns/dbfile.txt";
ny $searchpage = "search.html";


#------------------- subroutines -------------------

#-------------- MAIN PROGRAM ------------------------

@cgipairs = split (/&/, $ENV{'QUERY_STRING'});
foreach $pair (@cgipairs)
{
        ($name,$value) = split(/=/,$pair);
        $value =~ tr/+/ /; $value =~ s/\\//;
        $value =~ s/%([a-fA-f0-9][a-fA-f0-9])/pack("C", hex($1))/eg;
        $value =~ s/~!/ ~!/g;
        $value =~ s/\r//g;
        $value =~ s/\n/\<BR\>/g;
        $PARAM{$name} = $value;
}

print "Content-type: text/html\n\n";    # Start HTML
print "<H2>$title</H2>";

print "<P>You have entered a new pattern\n";
print "<P>Press the back button to return or\n";
print "<A HREF=\"$searchpage\">search</A>.<P>\n";

$name = $PARAM{'NAME'};                     # getCGI keys
$conf = $PARAM{'CONF'};
$author = $PARAM{'AUTHOR'};
$desc = $PARAM{'DESC'};
$pattern = $PARAM{'PATTERN'};

if ($name eq "")
{
        print "<B>Warning : No field entered for name</B>\n";
        exit;
}

if ($author eq "")
{
        print "<B>Warning : No field entered for author</B>\n";
        exit;
}

if ($desc eq "")
{
        print "<B>Warning : No field entered for description</B>\n";
        exit;
}

if ($pattern eq "")
{
        print "<B>Warning : No field entered for pattern</B>\n";
        exit;
}

if (-e $dbname)
{
        open(FILE,">>$dbname") or die "<P>Couldn't open $dbname\n";
        print FILE "\n";
        print FILE "NAME:$name\n";
        print FILE "CONFIDENCE:$conf\n";
        print FILE "AUTHOR:$author\n";
        print FILE "DESCRIPTION:$desc\n";
        print FILE "PATTERN:$pattern\n";

        close(FILE) or die "<P>Can't close $dbname\n";
}
else
{
        print "<P>File $dbname does not exist!\n";
}

print "<P>\n";

exit;
```

```
#!/usr/bin/perl

my $title = "New Search";
my $dbname = "/home/ambird/patterns/dbfile.txt";
my $count = 1;


#------------------- subroutines --------------------

#-------------- MAIN PROGRAM ------------------------

@cgipairs = split (/&/, $ENV{'QUERY_STRING'});
foreach $pair (@cgipairs)
{
        ($name,$value) = split(/=/,$pair);
        $value =~ tr/+/ /; $value =~ s/\\//;
        $value =~ s/%([a-fA-f0-9][a-fA-f0-9])/pack("C", hex($1))/eg;
        $value =~ s/~!/ ~!/g;
        $PARAM{$name} = $value;
}

print "Content-type: text/html\n\n";    # Start HTML
print "<H2>$title</H2>\n";
print "<P>You have search for : $search\n";

$search = $PARAM{'SEARCH'};

if ($search eq "")
{
        print "<P>\n";
        print "<B>WARNING : No search pattern entered!</B>\n";
        print "<P>\n";
        exit;
}

if (-e $dbname)
{
        open(FILE,"<$dbname") or die "<P>Couldn't open $dbname\n";

        while(<FILE>)
        {
                if (/^NAME:/)
                {
                        $name = $_;
                        $conf = <FILE>;
                        $author = <FILE>;
                        $desc = <FILE>;
                        $pattern = <FILE>;
                        $allofit ="$name $author $desc $pattern";
                        $allofit =~ s/<BR>/ /g;

                        if ($allofit =~ /$search/oi)
                        {
                                print "<P><U><B>Matching entry $count.\n";
                                print "</U></B><P>\n";
                                print "$name<BR>\n";
                                print "$conf<BR>\n";
                                print "$author<BR>\n";
                                print "$pattern<BR>\n";
                                $count++;
                        }
                }
        }
        close(FILE);
}
print "<P>\n";
print "End of Search\n";
print "<P>\n";


exit;
```

Appendix L

UNIVERSITY OF
LINCOLN

AP10088 Structured software development

Learning outcome	Criterion	Fail	3	2ii	2i	1
present structured techniques in software development [40]	Identification of the Waterfall model stages used and presentation of a pseudocode description of your solution [358]	No or incorrect identification of relevant Waterfall model stages. No indentation or use of Keywords. No solution or a sparse solution that is fundamentally incorrect. No attempt made to identify appropriate data values or their proper use.	Correct identification of relevant Waterfall model stages. The solution captures the basics but omits some points of significance. Most steps presented and in the correct order. Some data values correctly identified and manipulated	Correct identification of relevant Waterfall model stages. The solution captures the basics but omits a few points of significance. Some data values identified and manipulated correctly. Minimal use of decomposition as a structuring mechanism .	Correct identification of relevant Waterfall model stages. A complete solution but occasional errors in use and consistency of pseudo code. Some use of decomposition. Majority of data values identified and manipulated correctly.	Correct identification of relevant Waterfall model stages. A complete solution that has keywords and indentation correctly and consistently applied. All steps are listed one step per line and in the correct order. Good use of decomposition. All pertinent data values identified and manipulated correctly.
	Identification of the Waterfall model stages used and presentation of a flowchart description of your solution. [359]	No or incorrect identification of relevant Waterfall model stages.The solution is incoherently represented and /or uses incorrect notation.	Correct identification of relevant Waterfall model stages.Only a single level of description is presented. Correct notation used. Only a rudimentary visual organisation and choice of text attempted.	Correct identification of relevant Waterfall model stages.Some mismatch between the level of decomposition used and the complexity of the problem. Correct notation used. Control constructs, in general, correctly applied. Visual organisation is adequate. In some cases text used to express actions and decisions could be improved.	Correct identification of relevant Waterfall model stages.The level of decomposition correctly matches the detail of the problem. The correct notation has been applied. Control constructs correctly applied. Visual organisation is generally sound but open to improvement. In some cases text used to express actions and decisions could be improved.	Correct identification of relevant Waterfall model stages.The level of decomposition correctly matches the detail of the problem. The correct notation has been consistently applied. A correct and consistent application of appropriate control constructs. The visual organisation of the solution is excellent. The text used to express actions and decisions is appropriate and succinct.
apply structured techniques in the development of simple software [41]	apply structured techniques in the development of simple software [360]	If any of the following holds a fail will apply. The program fails to compile under the corporate environment. No trace table is presented. Source code is undocumented.	Program compiles but executed code is incomplete. A basic trace table presented. A basic interface is presented. Source code layout and commenting is rudimentary. Some relationship between pseudo-code and implementation is evident.	All of the following must be present. The program compiles without errors and is error tolerant. Rudimentary trace table is presented. The solution matches the specification. An interface is used that meets most user and results presentation requirements. Source code layout and commenting is acceptable. The program generally matches the pseudo code design decomposition.	All of the following must be present. The program compiles without errors and is error tolerant. Trace table is appropriate. The solution matches the specification. An interface is used that meets user and results presentation requirements. Source code deploys best practice documentation standards for easy understanding and maintenance. The program matches the pseudo code design decomposition.	All of the following must be present. The program compiles without errors and is error tolerant with graceful recovery. Trace table is appropriate. The solution matches the specification fully. An interface is used that meets user and results presentation requirements well. Source code deploys best practice documentation standards for easy understanding and maintenance. The program matches the pseudo code design decomposition well.

Department of Computing
3:36 pm Nov 19, 2002

UNIVERSITY OF
LINCOLN

AP10119 Introduction to Programming

Learning outcome	Criterion	A1	A2	A3	Fail	3	2ii	2i	1
Describe the software life-cycle and apply its principles in staffal practice	Student develops appropriately structured and presented pseudo-code.	•			Pseudo-code lacks structure and coherence. Top-level statements are not decomposed. No evidence of stepwise refinement.	Some attempt at structure. Some top-level statements are decomposed. Some evidence of refining to several levels but levels partially conflated.	Top two levels have good structure. Decomposition at third level is weaker. Some separation of levels but more detail needed at lower levels.	Decomposition of levels is good and shows clear understanding of top down method. Lowest levels are primitives.	Decomposition of levels is clear, appropriate and well laid out. No errors of logic. Lowest levels are primitives.
	Student describes role and nature of pseudo-code placing it in context of software lifecycle	•			Student does not describe a coherent or meaningful lifecycle model. No sources or quotations.	Student partially describes a meaningful and recognisable software lifecycle model. At least two sources listed and at least one relevant quotation.	Student describes a meaningful and recognisable software lifecycle model and makes reference to other models. At least four sources listed and two relevant quotations.	Student describes a meaningful and recognisable software lifecycle model and makes reference to other models. At least four sources listed and more than three relevant quotations.	Student describes a meaningful and recognisable software lifecycle model and shows understanding of alternative models. At least five sources listed and more than four relevant quotations.
Design and write short programs that compile and execute	Student's program compiles runs and processes input appropriately.	1	•	•	Student's program is not on disk or does not compile.	Student's program compiles with warnings. Runs but the processing and/or results are partial or inaccurate. Poor user interface.	Student's program compiles. Runs and the processing and/or results are largely accurate. Reasonable user interface.	Student's program compiles. Runs and the processing and/or results are fairly accurate. Good user interface. Program terminates with user help.	Student's program compiles. Runs and the processing and/or results are accurate. Excellent user interface. Program terminates cleanly.
Employ good programming practice in code development	Student's code reflects accurately the pseudo-code submitted.	2	•	•	Student's pseudo-code bears no relation to the code or is a minor rewriting of the code with language specific features included	Student's pseudo-code bears some relation to the code. Some language specific features included.	Code has obviously been based on the pseudo-code. No language specific features in the pseudo code.	Code reflects pseudo code very closely. No language specific features in pseudo code.	Code entirely reflects pseudo code. No language specific features in pseudo code.
Present annotated program code in a clear and structured fashion	Student's code contains meaningful comments that explain key elements of the algorithm.	3	•	•	There are no comments or comments are trivial and/or do not contribute to understanding of the algorithm.	Comments are minor but do attempt to explain some algorithm elements.	Comments attempt to explain critical elements of the algorithm such as branches or data validation.	Comments highlight some key algorithm elements and explain clearly critical design decisions.	Comments highlight all key algorithm elements and explain clearly critical design decisions.
Select appropriate constructs and concrete datatypes for implementing algorithms	Student uses appropriate datatypes for required input and processing.	4	•	•	Datatypes are inappropriate or inadequate for purpose.	One or two datatypes used but others also required.	Small range of datatypes used, largely appropriately.	Range of datatypes used appropriately. Data structures also employed where required.	Data types and data structures obviously reflect considered design decisions. All are appropriate.
	Student uses appropriate constructs for the algorithm's requirements.	5	•	•	Program is sequence only with no tests branching or loops. No parameters.	Program contains some branching but is mainly sequence only. No parameters.	Program has branching and a loop used appropriately. Parameters and global variables used.	Program has branching and types of loop used appropriately. Parameters and local variables used.	Program has branching and types of loop used appropriately. Highly modular use of parameters and local variables.
Devise, implement and document a structured software test plan	Student devises appropriate test data.			•	Test data absent or completely inappropriate.	Test data included but lacks range. Test only obvious cases.	Reasonable range of test data included. Test a wider number of cases.	Reasonable range of test data included. Test cases cover good range of possible data.	Excellent range of test cases. Difficult or obscure cases also covered.
	Student reports on the use and results of test data.			•	Test results missing or meaningless. Student does not report on the test.	Report is minimal and does not demonstrate grasp of key issues.	Report demonstrates clear understanding of the purpose of testing.	Report demonstrates clear understanding of the purpose of testing. Student demonstrates understanding of what testing of code has achieved.	Report demonstrates clear understanding of the benefits and limitations of testing. Student demonstrates understanding of what testing of code has achieved.

[illegible] 2 - 2003

Appendix M

Introduction to Appendix M

This is a transcript typed by the author of a meeting with first year students taking the Introduction to Programming unit as part of the Games Computing degree.

Present Amanda Bird (A), Suzanne Bowen (S), Suneithi Chand (Su) and Andy Walters (An)

A And we're recording...so thank you very much...fire away.
S Well we mean we've got ours split into two now which makes it a bit easier, 'cos we just... have the unit Introduction to Programming and we have Data Types Structures and Algorithms as well. So, the DTSA one is actually harder so the actual Introduction to Programming has been ... is fairly easy really from what we've been doing ... I mean I haven't really done any programming before...erm – well, a long time ago I did, I'm a mature student
A Are you? You don't look it!
Su That's what I said when I first saw her!
S Oh no I'm 30.
A No
Su: That's what I thought
A If you had told me you were 19 I wouldn't have blinked.
S No I'm not.
A How do you stay so youthful? What is your secret? Share!
S I've no idea. All my family are the same!
A So basically you did a bit of programming in the past. What did you do?
S I did Pascal
A Yeah
S Using the same books that I'm using now [laughs] to try and teach myself again but I've done a bit of Pascal, a bit of C... I was ...obviously when I was younger, like Basic and Visual Basic but we don't really use them now. The only thing with this programming course is that we have to do Pascal and a lot of the people on the course don't really... you know, are wondering why we are doing a language that nobody

uses any more to teach us to program. 'Cos I think it would be better to just to ...cos we're doing C++ next year...to actually just...

A So are you on the Games route?

S Yes

Su Yes

S Yes we're both on the games industry one...

A So you won't be doing Java.

S We'll be doing C++ after Christmas

S But to teach us how to program, we're learning Pascal. I don't see the point in it

A That's interesting. So you've got some experience of programming.

S A bit...

A Did you actually teach yourself or were you...

S Just teach myself from a book...

A Can you see any arguments for doing Pascal rather than say plunging straight into C++?

Su It is easier to learn...cos there are a lot of people who have never programmed before. S Yeah, I think if you have never programmed before you might as well do C++ still

S You might as well learn the language that you're gonna be using rather than a language that nobody ever uses and hasn't used for what, 10 years

A So the actual programming of Pascal, you've just started doing that, you've done quite a lot of theoretical stuff first...

S Yeah we've only done two weeks of programming out of the twelve weeks

A How have you found that? Would you have preferred to leap straight in to programming? Would you have preferred to sit down in front of a machine first?

S No I think it's been really interesting actually

Su Yeah

S I mean that side of it you know is quite difficult if you've never done anything like that before which I mean even though I've like had a go at programming I've never done any of this sort of background... background work

A So what cos I don't teach on this unit now, I did last year, what have covered in the last 9 weeks or so?

S You can speak [to Suneithi] You don't know do you? [laughs]

Appendix M Transcript of meeting with students in Lincoln 02/12/2003

Su Last thing we did was debugging

S Yeah, we've done all the waterfall method…like, we've done it all in order so we've done like the first two parts of it and then we'd have lectures [muffled]

Su Top-down step-wise refinement

S Yes

A So what you're learning is a methodology rather than just pure programming

[Another student arrives, An]

A Yes, I…the more the merrier! Grab a chair, please do join us.

S We're just saying what we've been doing in the lectures so far… We've [muffled]...lectures…[muffled] so far…lectures..

Su Static typing

S That's datatypes…oh sorry , yeah

A [Muffled] Variables, datatypes so that kind of thing… so all of that is kind of completely divorced from any language – it's all pure.

S Yeah… and then we've done…

Su So the language that we learn is just Pascal…

A Do a bit of sort of editing...actually hands on Pascal…

S Only in the last couple of weeks, yeah

A So you're only going to get, like another, well this week and next week of Pascal…then like January 's blank and then you're onto your C++

S Yeah that's why…we didn't really see the point in doing Pascal but...

A Could I ask how other people have found it? Can you give me some feel for the rest of your group…how they feel about this…doing Pascal…

Su There's quite a lot…there's quite a lot of [muffled]…programmers...so they don't do much

S I think most of 'em find it easy because they have done programming before

A Have you come quite new to programming?

Su I did some Pascal at college but not it wasn't that much, so it's not that new but….

S I did A Level French.

Su Ancient history.

A So now you're back in the games...you're doing a games degree. How do you see these and games going together?

Appendix M Transcript of meeting with students in Lincoln 02/12/2003

An I've done a bit of programming before for about two years [muffled] Pascal, I can see why we do it... [muffled] I think if you don't [muffled]

A Yeah, that's usually been the argument we've advanced... but I think as time goes on maybe the students that come in become more sophisticated in what they do know, so maybe it is time to overhaul it. What I'm trying to... I was saying to Suzanne and Suneithi that what I'm trying to get at would be seeing what we could do to make teaching programming to new programmers more effective. We've tried various things in the past and none of them seem particularly useful like doing on-line help to supplement the Pascal help or I mean the thing that I've directed people to which is like a little box to fill in to solve problems and suggest solutions to problems for your colleagues, I suspect if you've only done 2 weeks of Pascal that nobody would have bothered with that, so that's another washout but there you go... I really suppose, on a basic level I'm just sort of throwing my hands up and saying we've tried lots of things, I can't think of anything else, what do you guys think?

S I mean, I think... I know a lot of people have thought it's been a bit slow, I think basically because they know what they're doing [laughs] and they think its... like the first few lectures of DTSA were originally all one lecture, and then they thought it was too difficult for people so they made it into one like...yeah but programming, programming's...you know... we've basically had sort of 1 lecture sort of per issue so it's not been too bad but..

An [muffled – large portion of dialogue lost] try to get into the frame of mind...most of it's to do with theory

S Yeah but I think that's the point of it, yeah

An [muffled]...rather than concentrating on particular language

A Yeah I suppose the argument we'd advance is the stuff you're gonna be coding later is so complicated that there's no question of sitting there and going I'll do it like this, you need a methodology, you need a set of stages to go through, a set of thinking tools, like you say, that will pick that up but I don't know... I'm still trying to get at, is there, is there something I'm missing, 'cos I always come away from whenever I have a go at this every year, there must be something really simple, that would just make it click for people who haven't encountered this before, but I don't know. I'm just.. Some years we've packed things in a lot quicker, some years we've done things a lot slower, some years we've done lots of Pascal first, some years we haven't, I don't know, so...

S I don't think we should...

Su I think that this unit for this first semester should be just theory

A Um, no programming at all?

Su Yeah, just theory and then after semester B it should start programming then.

S Even though it's not that difficult but our assignment that is due in after Christmas is to write a big long algorithm in Pascal that some people have only done for 2 weeks, [laughs] you know it's like, program this!

Su Where to start? Oh right. I'm finding I don't know where to start with that one.

S Whereas you know if they'd have started... you know done the theory before Christmas then started people off with C++ ... maybe, I don't know.

A I need to get hold of a copy of the assignment 'cos I've not seen it.

Su I've got a copy I think...no I don't.

A What about the things that support the programming, like the Pascal help or lecture notes or books... what sort of things do you find most useful? Other people, friends, what sources really solve problems?

S Well, we have a study group

A Is that informally constituted? Just something you put together?

S Yeah, something I organised round my flat, so – all the clueless people all come to my flat once a week,

Su We get studying done eventually. [laughter]

S Yeah, we're trying to get the ratio right

A Socialising to... Pizza versus Pascal

S Yeah. Pizza usually wins over Pascal, I must admit but ...

Su Pizza always wins. It does [laughter]

S Yeah, 'cos we have seminar questions and stuff that we have to do, in fact some of the seminar stuff we have to do is harder than the assignment! Yeah. We've had to write, is it linked list... linked list deletion pseudo code and I find pseudo code harder than actually doing it in Pascal.

A That's interesting.

S 'Cos I never know what to write

Appendix M **Transcript of meeting with students in Lincoln 02/12/2003**

Su Everyone says that there's no right or wrong answer for pseudo code but there always is a right and wrong answer. No matter how you write it you have to write it some other way. It really annoys me

S I hate it when tutors say there's no right or wrong answer because obviously there is a wrong answer. Obviously, you can't put anything. There's no right or wrong answer, but you should do it like this.

A That's interesting,

S They do like to say that...

A We try to lull you into a false sense of security and pounce upon your helpless answers...

S Every one of our tutors have said at some point, oh there's no right or wrong answer for this assignment, but...

Su But ...there's always a but...

S If you don't write it this way like we say you'll fail.

A So programming is not much of a problem but some of the things that go round it like pseudo code that's harder to get your head round?

S Yeah, I think it is...it's probably not for most people.

An It probably depends on what kind of background you have. What kind of person you are. 'Cos there's a lot of people who are doing this course who've got no programming background...

S I think the majority...

An [muffled] ...people are struggling and I think it's the programmers who are going to struggle with other parts...

S Yeah, they're going to struggle next semester when we're doing mainly animation and that.

A Do you feel there are people on your course, who are struggling with introduction to programming stuff, the theoretical pseudo code-y stuff ?

Su I know quite a few who struggle with pseudo... but the theory stuff we can ...we can figure out.

An [muffled]

A It's the classic dilemma, isn't it?

An [muffled]

Appendix M Transcript of meeting with students in Lincoln 02/12/2003

A That's another very interesting point I think you've made. People in the most need to address that problem, not getting to grips with it are the people most likely to think that's it, I'm out of here, they don't turn up.

S I've heard quite a few people saying, and I think it's the wrong approach but they'll go, I don't get it, I'll just sit the resit.

Su I know

S As if they're going to get it but they'll say oh I don't get it and I'll fail it ...

A Really?

S I'll fail it this time

A Such a high-risk thing strategy! Do they not realise that if they do that often enough they're going to fail the year? Do they not realise... you know that's like walking on a tightrope with a river full of crocodiles below... one false move and...

Su No assignment support over summer as well so...

A No, well, lecturers have to take holiday some time, quite frankly when you all bogged off in the July, we do too I mean we're back for boards and that sort of thing you know... If we take two weeks off and it just happens to coincide with the two weeks you need us, it's difficult....

S But I think if you don't get it now you're not gonna get it when you do the resit.

A No I'm interested in that attitude, I've come across that but I didn't realise it was as overt as that – people actually say say, bugger it, I'll do it in the summer 'cos how do you know what's going to happen to you between now and the summer – what if one of your family is really ill or the next units are equally as hard or something...there's always something, isn't there?

S A lot of people have sort of cottoned on to the fact that your first year doesn't really count towards your grades so a lot of people don't turn up, they think, oh well I'll struggle, get a 3rd in my first year and then I'll like do some work next year.

A If you're still here next year!

Su That's actually demotivated me though since you told me that it doesn't really matter what we get, it's just demotivated me and I ... I was working really hard before

A That's OK to say I'm not going to knock myself out to the nth degree 'cos all I've got to do is get a reasonable mark, just pass everything but to kind of just say well I'm not gonna do anything...

Su [muffled]...in the first place, really irritating

Appendix M Transcript of meeting with students in Lincoln 02/12/2003

S I think your pride gets in the way of your… well, in my case it does, I still want to, I haven't missed one single lecture or seminar… I haven't been late for any of 'em.

A Yes I think that's a very relevant view… it's like booking theatre tickets then you don't bother going – you're paying for them so it's cold hard cash, isn't it?

S You need to get your money's worth.

Su Yeah

A I like that, I might tell that to my third years as they don't turn up either. I've got 100 in the 3rd year at George Street in Hull, in this morning's lecture there were about 30, so that was 70 at home listening to Viking radio eating toast in bed, great! So, what's coming across to me is that, this group of people, and the rest of your group who aren't here are extremely motivated and you've kind of taken a step back from the process and said, well you know this is how it is working for us. Do you think you are kind of, you are a self-selected group, and that if I collared some of the bog standard 19 year olds, would I get a very different picture of how they see things?

S Probably would, but bog standard 19 year olds wouldn't have even bothered filling in your questionnaire.

A No they haven't, I've had a poor response..

S [muffled]

Su Most people when they come to uni don't realise that they've gotta do some work! They should realise it. They think they can do naff all

S Anything extra curricular like this they wouldn't even bother so …but

An ...stereotypical student [muffled]

Su Lots of people just waiting to be told what to do.

A Interesting

S I mean we've got a typical mature student outlook, I'm paying for it so I'm coming.

Su where do I fit in? [laughs]

Su I'm here because I want to be here not because my mum and dad sent me.

A You are going to uni whether you like it or not!

S Basically because we've had jobs before as well you know it's not that ghard to get out of bed in the morning. It's like ooh 9 o clock lecture, that's so early'. I mean I used to get up at 6 and go towork all day and I work now I work nights and come here and don't miss any lectures.

A Is that true for you as well? You've worked previously, presumably?

Appendix M **Transcript of meeting with students in Lincoln 02/12/2003**

An Yes.

A So what made you decide to come back into higher education?

An Personal circumstances. I'd never been to university, didn't really know what to do… Wish this course had been running 20 years ago. This is the course I enquired about…

S Cos there's only 2 universities in the country that do games computing, the other one's down London.

Su There's Bradford

S And Scotland…

A We've been going quite a while now. 4 coming up to 5 years on it now, so we were the first, the absolute first. We've had 2 cohorts go through.

S Two lots?

Various [muffled]

S And have they done well 'cos there seems to be a large drop out rate...about a third?

A Yes, I think that's partly because of the people it attracts. We get a lot of people who think, oh yes, games computing – what I'm gonna do is, I'm going to play games for 3 years and they're going to give me a degree.

S That's what some people actually do on our course

AWell they did when they were based in Hull, we had a games room in Hull, one of the top floor labs and basically, sort of from first thing in the morning till last thing at night and all through the summer, they were just like Doom-ing

Su That just echoes, that just echoes what happens in the game lab here

A I mean people play games a lot, not because they are analysing the game, they're just playing it and I think, the things that you talked about like not coming to lectures and not engaging with the subject, that's why the people drop out.

S I must admit I was surprised we don't have more lectures. I know it sounds [?] but I can't work out why we don't have more lectures and seminars.

Su We only have nine…

S We only have 1 hour for each unit – I expected to come to uni and like work all day,

A No…

S I'd prefer more, I'd prefer more… well at least, more seminars to go along with the lectures

Appendix M Transcript of meeting with students in Lincoln 02/12/2003

A It's a fairly standard model, 1 hour lecture, 1 hour seminar per unit for your subjects and what we usually say is that kind of you double that by your own self study so I don't know, if you do 10 hours, you should do at least another 10 hours and another 5 on top of that, and it rises sharply when your assignments are due in, and in your final year, when you are doing dissertation then it is almost a full time job really.
Su Third year is going to be hard I think...
S Well I try and do it as if I'm doing a full time job, you know sort of get up, do lectures and work until five even when I haven't got them and then go to work!
A Do the evening shift! Well, when do you sleep, if that's not an impertinent question?
S Well, I only work... I only work every other day. So like last night I finished work at 1 o clock, then I couldn't get to sleep when I got home, so it was about 3 o clock and then we had our modelling lecture at 9 this morning
A So you're feeling really sharp this morning!
S I've had lots of coffee today!
A Speaking of which, shall I get a round of drinks in?
Su Sure.
S Sure
A [Arranges drinks orders]

Five minute break for fetching of coffees

A So, one of the things, one of the things that started us off was I was saying about resources that support your programming learning, things like books and you were talking about your study group that you've got sort of going – what about books and on line resources and stuff like that, how much do you use of that?
S Erm...
Su we do have a few Web links, Terry put some links up, tutorials on Pascal.
S One of the PhD guys that's tutoring us lent me a book so...
A Do you prefer to look on the internet for stuff, or are books OK? Is the library much...?
S I prefer to look up stuff on the internet, but I can't understand stuff if I'm reading it off a screen. I print it out. I'm one of those weird people. Are you the same

Appendix M **Transcript of meeting with students in Lincoln 02/12/2003**

An Agrees

S I can't take it in until…

A [muffled]

S – so much for a paperless society!

A Ah it'll never happen…

S I print out all my lecture notes and I carry them round with me and then people…

A Back strain!

S I know

Su Human SLE

S I am, yes. I have the entire SLE in my folder [laughs]

S But yeah, we… I'm learning from that Illustrating Pascal and I've got a Illustrating C for next year by the same author

Su It's quite old isn't it? [muffled]

S Not sure if I've got it in my bag or not –

A It's called Illustrating Pascal?

S Illustrating Pascal. It's quite old, about 1990

A Well it doesn't have to be up to date [muffled] last twenty years or so…

S And there's an Illustrating C and [muffled] the book and basically it's just double pages on one subject, so one subject is just a page and that's it

A So short and to the point?

S Drawn, things drawn in pictures just to show you. I got it off Amazon but they've got Illustrating Pascal on Amazon as well it's only about a tenner…

A That's quite cheap, isn't it?

S Only get books… I only get text books if they're like under 20 pounds…

A I think that's fair…

S Cos like I put all the suggested reading books in the basket to see how much they cost – suggested reading for this semester and next semester and it was like 200, 300 pound – A A Not going to happen is it?

Su [muffled]

S Yeah, got the UML one though cos it was 9.99!

Su SAMS, SAMS Teach Yourself UML

S In 24 hours...I'm just waiting for a free 24 hours to actually read the book [laughter] Just to prove it, cos it's proven, it's 24 hours

Appendix M Transcript of meeting with students in Lincoln 02/12/2003

An You can read it in 24 hours but you won't understand it in 24 hours.

A That takes 24 weeks. So, what I'm getting correct me if I'm wrong, is a picture that the student body is 2 separate cohorts – there's people like you who are really engaged with it and are quite reflective on the process. They can kind of take a step back and think 'I do it like this because...' and then there's like everyone else, bimbling around, going I'll do it in summer. Is there much communication between those 2 groups? – do they go oh wow you're really swotty or are they like - got the answers?

S Um... I don't think so, even though I'm like, even though I do all my stuff I still struggle but I just try, try hard, try harder but, um, there's not that much communication, I can't say, I can only say I know about 10 people...

An [muffled]

Su Plagiarism

S Yeah

An [muffled]

A Well the thing is, it's...

An [muffled]

A Well the thing is I agree, it's a very grey area in programming, isn't it because no programmer ever works in complete isolation but you're doing an assignment and someone looks at your code and thinks that's very good I'll have that.. we've had cases where people have literally nicked it off the printer and submitted it as their own, write out the name at the top and it's very difficult, sometimes when someone says well actually I wrote it you know and what do we do

S You have to penalise both sides, don't you?

A It's been really difficult, there's been a couple of cases where we had to decide, who's ripping off who? So I think it's a fair point.

S I know what you mean because I mean like we have our study group and stuff like that...you talk to loads of people about your assignments. When you sit down to actually write it, you can't remember which thoughts were your own.

Su Yeah

A Actually, I don't think...

S It's sort of like subconscious plagiarism. You write something and you think did I write that because I cannot remember if somebody told me that or what so you end up like trying to change it but really I mean you are gonna get some, you are going to have to use some things..

A There's always some overlap because there are only so many ways you can do something. There is always gonna be a kind of similarity.

S ..variables..thy're always gonna have similar anmes because people are gonna think oh count OK yeah call that count, call that, that but I think there's probably less of motivated people on course than motivated people

A Do you know how many.. I'm trying to get a feel for the size of the cohort..is it about 150 people on the first year?

S 151

Su You know the exact figure?

S Well, Kevin keeps saying 'I've got 151 of these to mark! Yeah 151 five girls...

A Really?

S Five girls, 140 spotty teenagers

A 145 that leaves 6 unaccounted for...how many mature students are there?

S I'm the second oldest

Su There's you two, there's Steve and [?] isn't there?

S There's, there's...

Su About four...

S I mean mature students, if you're over 21 I don't really call them mature students

A Call somebody mature if they have worked for a few years and then come back into education

S There's only a handful of people that are in their late twenties and then there's me that's 30 and Andy over that's ancient! You're old!

A So, what do you all want to do when you graduate?

S I thought when you grow up I thought you were gonna say then!

Su when you grow up she's immature... [laughs]

A [muffled] So when you graduate where do you see yourselves going after this course? What do you fancy doing?

S Even though... I actually want to program, even though, even though we always get told that's like the lowest of the low of games computing but I actually want to. Most people are here for the animation

Su I am [muffled]

Appendix M Transcript of meeting with students in Lincoln 02/12/2003

S…which surprised me, the people who are actually here for animation, why didn't they do, apply for the games design?

A Higher points count

Su and the portfolio. I haven't got the patience to do a portfolio, I really don't, which is why I'm not doing an art degree… I have no patience

A A fair analysis really which is if you want the more media side of things you usually are required to have some sort of track record in doing this whereas I suppose you're degree will take zero start point…

Su I think it's good though the way it …cos when you do get into the industry, it will help you

S It does make it more interesting. I mean the stuff that we've done so far's been really interesting except for communication skills

Su [muffled]

S… necessary evil, I suppose but…

An [muffled]

A So you're quite experienced programmer?

An [muffled]

A That's really what's drawn you in?

Su [muffled]

An [muffled]

S There is an interesting mix.

Su Most people think… I only know a handful of people who want to do design

S People think, people do think it's an easy degree and it's not.

Su Mickey Mouse degree

S Yeah, Mickey Mouse degree. A lot of 'em call it Mickey Mouse degree, yeah

A Well I think when you hit dome of the 2d & 3d animation, some of the heavy duty programming…I can see why for the outsider although I don't think I agree when it comes to some of the topics, stuff that you're gonna do…

S No … I've downloaded the maths for next semester

A [muffled] ..skinning and splining and..

Su Modelling… yeah

A To do it well I think is quite demanding, I mean anyone can faff about a bit…

Appendix M Transcript of meeting with students in Lincoln 02/12/2003

S But I mean they do literally let anybody onto this course though so that is a major problem

A Well it is, we had wastage, we've always traditionally taken in the computing department, sort of taken people with low A level points, everybody gets a chance, everyone gets an equal playing field and if they won't take the chance, fair enough they are out in year one or year two and that's entirely their choice. You can't make someone go for it you can't say this is the best opportunity you'll ever have in your life, they go Yeah OK, whatever...you know

S Yeah, there's a lot of people like that.

A OK, so some of the ideas I'm getting from are very useful. I don't suppose anybody has looked at the URL that I've set up where you've got …?

Su No, not yet.

A I think I might e-mail and ask people to look at in, might put a couple of samples up next week... see that's week 12 and then it's Christmas. When's your assignment due in?

Su Which one? For programming?

A For programming

Su 13th Jan

A I'll put some stuff in this week and email…

S We'll have to do it before Christmas

A So it's got to be done really, hasn't it? Um, OK.

END OF TAPE

Appendix M: Conclusion

This meeting was meant to one of a series but due to difficulties in finding student volunteers, this proved to be the only meeting that took place.

Appendix N

End of unit – Introduction to Programming.

Please write as much as you can in answer to the two questions below. Thank you!

What has been the hardest/most difficult aspect of learning to program?

What resource (book, friends, web page...) or strategy has been the most useful in learning to program?

September 2003 – Introduction to Programming questionnaire

Please note: *completing this questionnaire is voluntary but the information you supply will help support this unit's development! Thank you for your time.*

Please return your questionnaire to the tutor.

Name: (leave blank if you wish)
Course:
Age in years:

1. What qualifications do you have at post 16? (A levels, BTEC, NVQ, others)

2. What programming experience do you have? (If none, please go to question 4)

3. If you have programming experience, please describe briefly some of the work/programs you have done: (Then please go to question 5)

4. If you do not have programming experience, please outline some of the issues around learning programming that you think might concern you.

5. What do you think the Introduction to Programming unit should teach new programmers?

Please continue any answers over the page if you wish

Student Profile

Please fill in the following details:

Gender:

Age (at 1st Sept 2000) Years Months

GCSEs obtained. (Please include grades)

A levels/further qualifications. (Please include any HND/Btec qualifications)

What programming experience have you had? Please note the languages you have used, if any!

Questionnaire

Please circle one number beside each statement. (1 is <u>strongly disagree</u> and 5 is *strongly agree,*)

	Disagree → Agree
Learning is easier in small groups	1 2 3 4 5
I find it difficult to grasp a new topic when I cannot relate to it	1 2 3 4 5
Correct lecture notes should be given for reference	1 2 3 4 5
Learning is a process of being told what to do and then going away to practise it.	1 2 3 4 5
I find working in groups difficult because people learn differently	1 2 3 4 5
Learning from a booklet of printed lecture notes is difficult	1 2 3 4 5
Practical work helps the learning process	1 2 3 4 5
Lectures are not that easy to follow all the time	1 2 3 4 5
Learning is easy when you have enthusiastic lecturers	1 2 3 4 5
Lectures provide the necessary framework to understanding topics	1 2 3 4 5
I like to learn in my own time away from the university	1 2 3 4 5
There is some useful material to learn from on the Internet	1 2 3 4 5
In order to learn the student must work as well	1 2 3 4 5
I find it easier to learn and absorb information in silence	1 2 3 4 5
Computer-based learning can be tedious	1 2 3 4 5
Seminars are sometimes a waste of time as you come out feeling no different	1 2 3 4 5
Learning is easier in lectures as information needed is just handed to you	1 2 3 4 5
Assignments that are long don't help you learn a topic any more easily	1 2 3 4 5
One of the best ways to learn is to discuss the problems with friends	1 2 3 4 5
You should learn by using the library	1 2 3 4 5
Learning should be backed up with more face to face work with tutors	1 2 3 4 5
Learning takes concentration	1 2 3 4 5
Learning takes a lot of hard work and can't be done solely from lectures	1 2 3 4 5
Learning is more effective through practical examples	1 2 3 4 5
It is important to attend both lectures and seminars to fully understand things	1 2 3 4 5
Group work is a more active and interesting way of learning	1 2 3 4 5
It is easier to remember facts if you actually write about them as well as read them	1 2 3 4 5

You do not have to give your name but all named questionnaires will be placed in a draw to win a £5 HMV voucher.

1) What programming language have you studied this semester ?

2) What other programming languages have you previously studied?

3) Do you program on your home computer?

If so in what language(s)?

4) What aspects of the language you have studied *this semester* have you found ..hardest?

...easiest?

5) What suggestions would you make to improve the teaching of the language you have studied *this semester*?

6) What suggestions would you make for the teaching of **introductory programming?**

7) Please place the following resources in order of importance to you in learning programming.
(If you felt help from other students was the most important place a 1 beside it and so on. Any resources you did not use, leave the box blank)

☐ The language's help system	☐ Individual programming practice
☐ Books from the library	☐ Sample programs
☐ Books bought (new or 2nd hand)	☐ On-line help set up by the University
☐ Internet sources	☐ Magazines
☐ Myton House help desk	☐ Students on the same course
☐ Friends/colleagues not at the University	
☐ Lecture notes	☐ Lab worksheets
☐ Computer Aided Learning material	

Please add any other comments below:

THANK YOU

Questionnaire on CBT. You do not have to give your name.

1) Please outline your educational background (eg A levels taken - you do not have to specify grades)

2) How do you prefer to learn a new topic or subject? Lectures, tutorials, private reading, discussion with friends, through assignments, seminar or lab work, through CBT? Try to describe how you tackle a completely new subject.

3) How do you feel about the CBT you have done up to now? If you could identify clear advantages and/or disadvantages, what would they be?

4) How do you see CBT fitting into the learning environment for your course/subject? Should there be more or less CBT? Is CBT better than any other method?

5) What could be added to CBT to improve its effectiveness? Any ideas welcome!

6) Should CBT replace the bulk of mass lectures? Does CBT neeed even scheduled classes?

CBT QUESTIONNAIRE

STUDENT NAME:	
AGE:	
COURSE & YEAR:	
SEMINAR TUTOR:	
DATE & TIME:	
TITLE OF PACKAGE USED:	
SECTIONS COVERED:	
Did you enjoy using the package?	Yes/No
Is the package you used, of (tick as appropriate)	a) Too high a level b) About the right level c) Too low a level
Could the package you used, replace the lecture needed to cover this material?	Yes/No
For how long at one time could you use this CBT package? (tick as appropriate)	20 mins or less 30 mins 40 mins 1 hr 1 hr 20 mins 1 hr 30 mins or more.
How would you rate the overall presentation of the package? (tick as appropriate)	Very Good Good Average Poor
Does the use of graphics, animation and video,	Help keep your attention Yes/No Help reinforce the material Yes/No

How would you rate the quality of graphics in this package? (tick as appropriate)	**Very Good** **Good** **Average** **Poor**
How would you rate the quality of animation in this package? (tick as appropriate)	**Very Good** **Good** **Average** **Poor**
Do you think the addition of video is a valuable addition?	**Yes/No**
Are there any features in particular you would like to see added to CBT software? If so, please indicate alongside.	
Would you prefer to see problems to solve in the package?	**Yes/No**
Why are you using the CBT system? (tick as appropriate)	**a) To reinforce lectures** **b) Instead of lectures** **c) Own interest** **d) You have been told to.**
Any other comments?	

Thankyou.

Bibliography

ACM Transactions on Computer-Human Interaction Sept 1994 Volume 1 Number 3

Acton H B (ed.) (1972) John Stuart Mill Utilitarianism Liberty Representative Government, London: Dent

Adams W (1973) *The Use of APL in Teaching Programming* in Turski (ed.) (1973) Programming Teaching Techniques

Ainley J and Goldstein R (1988) Making Logo work: A guide for teachers, Oxford: Blackwell

Alexander C, Ishikawa S, Silverstein M, Jacobson M, Fiksdahl-King I and Angel S (1977) A Pattern Language Towns Buildings Construction, New York: Oxford University Press

Alexander C (1979) The Timeless Way of Building, New York: Oxford University Press

Alexander S and Bond D (2001) *Learners still learn from experience when online* in Stephenson (ed.) (2001) Teaching and Learning Online Pedagogies for New Technologies

Allen P, Booth S, Crompton P and Timms D (1996) *Added value: quality rather than quantity* in Active Learning Number 4 July 1996

Alspaugh C (1972) *Identification of Some Components of Computer Programming Aptitude*, Journal of Research in Mathematics Education Volume 3 No 2 Found at http://www.nctm.org/jrme/abstracts/volume_03/vol03-02-mar1972.html 24/08/2000

ALT-C (1997) Virtual campus, real learning Conference programme and abstracts, Oxford: ALT

ALT-N Association for Learning Technology Newsletter No 18 July 1997

ALT-N Association for Learning Technology Newsletter No 20 January 1998

ALT-N Association for Learning Technology Newsletter No 21 April 1998

ALT-N Association for Learning Technology Newsletter No 22 July 1998

ALT-N Association for Learning Technology Newsletter No 23 October 1998

Alvey Committee (1982) *A programme for advanced information technology: the report of the Alvey Committee*, London: HMSO

Ambler S (2000) *The extreme programming software process explained* Computing Canada, Willowdale, March 2000, Volume 26 Issue 5

Bibliography

Andrews J, Garrison D R and Magnusson K (1996) *The Teaching and Learning Transaction in Higher Education: a study of excellent professors and their students*, Teaching in Higher Education Vol. 1, No 1 1996

Anderson F H (ed.) (1960) The New Organon and Related Writings, New York: Liberal Arts Press

Anderson J (1988) *The Expert Module* in Polson and Richardson (1988) Foundations of Intelligent Tutoring Systems

Angrist J and Lavy V (2002) *New Evidence on classroom computers and pupil learning*, The Economic Journal 112 (October) pp 735 – 765, Oxford: Blackwell Publishers

Annett J (1970) *Computers in education I: The Black, White and Blue Papers* in Journal of Educational Technology, Vol. 1 No.1 Jan. 1970 London: NCET

Annett J (1976) Computer Assisted Learning 1969 - 1975, London: SSRC (Social Science Research Council)

Annett J and Duke J (eds) (1970) Proceedings of a Seminar on Computer Based Learning Systems, Bodington Hall Leeds 8 - 12 September 1969, London: NCET

Appel M H and Goldberg L S (eds) (1977) Topics in Cognitive Development Volume 1 Equilibration: Theory, Research and Application, New York: Plenum Press

Appleton B (2000) *Patterns and Software: Essential Concepts and Terminology* found at http://www.cmcrossroads.com/bradapp/docs/patterns-intro.html 09/03/2003

Arblaster A (1982) *Human Factors in the design and use of computing languages* in International Journal of Man-Machine Studies (1982) 17, pp 211-224

Arblaster A (1983) *The evaluation of a programming support environment* in Green et al (eds) (1983) The Psychology of Computer Use

Armour P (2000) *The five orders of ignorance* Association for Computing Machinery Communication of the ACM New York: October 2000 Volume 43 Issue 10 pp17-20

Arzarello F, Chiappini G, Lemut E, Malara N and Pellerey M (1993) *Learning to Program as a Cognitive Apprenticeship Through Conflicts* in Lemut et al (eds) (1993) Cognitive Models And Intelligent Environments for Learning Programming

Atkins M (1998) An evaluation of the Computers in Teaching Initiative and the Teaching and Learning Technology Support Network, HEFCE

Atkinson C (1983) Making Sense of Piaget The Philosophical Roots, London: Routledge Kegan Paul

Bibliography

Atkinson R C and Wilson H A (eds) (1969) Computer Assisted Instruction A book of readings, New York: Academic Press

Atkinson R L, Atkinson R C and Hilgard E R (1983 8th Edition) Introduction to psychology, New York: Harcourt Brace Jovanovich

Attikiouzel J (1990) Pascal for Electronic Engineers, London: Chapman and Hall

Aune B (1970) Rationalism, Empiricism and Pragmatism An Introduction, New York: Random House

Austen E and Enns J (2000) *Change Detection: Paying Attention to Detail* PSYCHE, 6(11), October 2000 found at http://psyche.cs.monash.edu.au/v6/psyche-6-11-austen.html 22/02/2001

Ausubel D, Novak J and Hanesian H (1978) Educational Psychology A Cognitive View Second Edition, New York: Holt Rinehart and Winston

Ayer A J (1990 edition) Language Truth and Logic, London: Penguin Books

Ayscough P B (1979) *Preparation for laboratory exercises* in Learning through computers An introduction to Computer Assisted Learning in Engineering, Mathematics and the Sciences at tertiary level

Bacon R (1996) *The effective use of computers in the teaching of* physics in Active Learning, Number 4 July 1996

Baecker R M and Marcus A (1990) Human Factors and Typography for More Readable Programs, Reading, Massachusetts: Addison-Wesley

Baecker R M and Marcus A (1998) *Printing and Publishing C Programs* in Stasko et al. (eds) (1998) Software Visualization

Baecker R M (1998) *Sorting out Sorting: A case study of Software Visualization for Teaching Computer Science* in Stasko et al. (eds) (1998) Software Visualization

Baecker R M and Price B (1998) *The Early History of Software Visualization* in Stasko et al. (eds) (1998) Software Visualization

Bajpai A C and Leedham J F (eds) (1970) Aspects of Educational Technology IV, London: Pitman

Balz G A (1967) Descartes and the modern mind, Hamden: Archon Books

Barker J and Tucker R N (eds) (1990) The Interactive Learning Revolution Multimedia in Education and Training, New York: Kogan Page

Barker P (1989) (ed.) Multi-media Computer Assisted Learning, London: Kogan Page

Bibliography

Barker P (1989) *Multi-media CAL* in Barker (1989) (ed.) Multi-media Computer Assisted Learning

Barker P and Yeates H (1985) Introducing Computer Assisted Learning, Englewood Cliffs: Prentice Hall

Barr A, Beard M, Atkinson R (1976) *The computer as a tutorial laboratory: the Stanford BIP Project* in International Journal of Man Machine Studies Volume 8 pp 567 – 596

Bartlett F C (1932) Remembering: A study in experimental and social psychology, Cambridge: Cambridge University Press

Bazik J, Tamassia R, Reiss S and Van Dan A (1998) *Software Visualization in Teaching at Brown University* in Stasko et al (eds) (1998) Software Visualization

Beauchamp T L (ed.) (1999) Hume An Enquiry into Human Understanding, Oxford: Oxford University Press

Beaumont G (1989) Preliminary Turbo Pascal, London: Kogan Page

Becker H J (1994) Analysis and Trends of School Use of New Information Technologies, U.S. Congress Office of Technology Assessment Found at http://www.gse.uci.edu/EdTechUse/c-tblcnt.htm 12/03/2002

Beer G (1996) Open Fields: Science in Cultural Encounter, Oxford: Clarendon Press

Beetham H (ed.) (1997) IT & Dearing: The Implications for HE Colloquium Proceedings, CTISS

Bell D, Morrey I and Pugh J (1997) The Essence of Program Design, Harlow: Prentice Hall

Bell J (1999) Doing your research project A guide for first-time researchers in education and social science, Buckingham: OU Press

Bem S and Looren de Jong H (1997) Theoretical issues in psychology, London: SAGE

Bem S and Looren de Jong H (1997) Theoretical issues in psychology, London: SAGE

Benford S, Burke E and Foxley E (1993) *Obtaining and using the Ceilidh system for teaching programming* in CAL into the mainstream CAL 93 Conference handbook

Bernard H R (1994) Research Methods in Anthropology Qualitative and Quantitative Approaches, Thousand Oaks, California: SAGE Publications

Bibliography

Berryman J, Hargreaves D, Hollin and Howells K (1987) Psychology and You, London: British Psychological Society/Routledge

Beswick N (1987) Re-thinking Active Learning 8 - 16, London: Flamer Press

Beynon J and Mackay H (1993) Computers into classrooms More Questions than Answers, London: Falmer Press

Biermann A W (1997) Great Ideas in Computer Science A Gentle Introduction, Cambridge, Massachusetts: MIT Press

Biggs M (2000) *Pair programming: Development times two* Infoworld, Framingham, July 2000 Volume 22 Issue 30

Bird G (1972) Philosophical Tasks, London: Hutchinson University Library

Bishop J (1993) Turbo Pascal Precisely, Wokingham, England: Addison-Wesley

Blackwell A (1996) *Metaphor or Analogy: How Should We See Programming Abstractions?* in Vanneste, Bertels, De Decker and Jaques (eds) Proceedings of the 8th Annual Workshop of the Psychology of Programming Interest Group, pp 105-113.

Blackwell A and Bilotta E (eds) (2000) 12th Workshop of the Psychology of Programming Interest Group, Cozenza, Italy, April 2000

Blackwell A and Green R G (2000) *A Cognitive Dimensions Questionnaire Optimised for Users* in Blackwell and Bilotta (eds) (2000) 12th Workshop of the Psychology of Programming Interest Group

Blum B (1996) Beyond Programming to a New Era of Design, New York: Oxford University Press

Bonar J and Soloway E (1989) *Pre-programming Knowledge: A Major Source of Misconceptions in Novice Programmers* in Soloway and Spohrer (eds) (1989) Studying the Novice Programmer

Booth S (1992) *Learning to program: A phenomenographic perspective*, Göteborg, Acta Univertsitatis Gothoburgensis found at http://www.ped.gu.se/biorn/phgraph/civil/graphica/diss.ab/booth.html 15/02/2001

Booth S (1997) *On phenomenography, learning and teaching*, Higher Education Research & Development, 1997, Volume 16, No. 2, pp 135-158

Bork A (1983) *Computers and the future: education* in CAL 83 Selected papers from the Computer Assisted Learning symposium

Bower G H and Hilgard E R (1981) Theories of Learning (5th edition), New Jersey: Prentice-Hall

Bibliography

Boyd-Barret O and Scanlon E (1991) (eds) Computers and Learning, Wokingham, England: Addison-Wesley

Boyle T (1997) Design for multimedia learning, Hemel Hempstead: Prentice Hall

Boyle T, Gray J, Wendl B and Davies M (1993*) Taking the plunge: A case study in the intensive use of computer-assisted learning* in CAL into the mainstream CAL 93 Conference handbook

Bradshaw P M (1985) *CAL in the teaching-learning process* in Reid and Rushton (eds) (1985) Teachers, computers and the classroom

Brahan J W and Godfrey D (1981) A *marriage of convenience: Videotex and Computer Assisted Learning* in Computer Assisted Learning Selected papers from the CAL 81 symposium

Breakwell G (1995) *Research: Theory and Method* in Breakwell et al. (eds) (1995) Research methods in psychology

Breakwell G, Hammond H and Fife-Schaw C (1995) (eds) Research Methods in Psychology, London: SAGE

Briggs L (ed.) (1977) Instructional Design Principles and Applications, Englewood Cliffs, New Jersey: Educational Technology Publications

Brittan G (1978) Kant's Theory of Science, Princeton, New Jersey: Princeton University Press

Brown M and Hershberger (1998) *Program Auralization* in Stasko et al. (eds) (1998) Software Visualization

Brown M and Sedgewick R (1998) *Interesting Events* in Stasko et al. (eds) (1998) Software Visualization

Brown P (1982) Pascal from BASIC, London: Addison-Wesley

Bruner J (1985) Child's Talk, Oxford: Oxford University Press

Bruner J (1986) Actual Minds Possible Worlds, Cambridge, Massachusetts: Harvard Press

Brusilovsky P (1993) *Towards an Intelligent Environment for Learning Introductory Programming* in Lemut et al (eds) (1993) Cognitive Models And Intelligent Environments for Learning Programming

Bryant I (1998) *Action research and reflective practice* in Scott and Usher (1998) (eds) Understanding Educational Research

Bibliography

Burkhardt H, Fraser R and Wells C (1981) Teaching *style and program design* in Computer Assisted Learning Selected papers from the CAL 81 symposium

Burroughs G (1971) Design and Analysis in Educational Research, Oxford: Alder Press

Butcher P G (1985) Computing *aspects of interactive video* in CAL 85 Advances in computer-assisted learning: selected proceedings from the CAL 85 symposium

CAL95 (1995) CAL 95 Learning to succeed Conference Handbook

CAL97 (1997) International Conference WWW Proceedings

CAL Group (1994) CAL Group News Issue 2 November 1994, CAL Group

CAL into the mainstream (1993) CAL 93 Final programme

Callear D (1999) *Intelligent Tutoring Environments as Teacher Substitutes: Use and Feasibility* in Educational Technology Volume XXXIX No 5

Calvin A D (1969) (ed.) Programmed Instruction Bold New Venture, Bloomington: Indiana University Press

Calvin W (1997) How Brains Think Evolving Intelligence, Then and Now, London: Wiedenfeld and Nicholson

Canfield Smith D (2000) *Building personal tools by programming* Association for Computing Machinery Communications of the ACM, New York: August 2000 Volume 43 Issue 8 pp 92-95

Canfield Smith D, Cypher A and Teler L (2000) *Novice programming comes of age* Association for Computing Machinery Communications of the ACM, New York March 2000, Volume 43 Issue 3 pp 75-81

Carr W and Kemmis S (1994) Becoming Critical Education, Knowledge and Action Research, London: Falmer Press

CET (1977 a) Technical Report No 14 Computer assisted learning in higher education - the next ten years A future study report, London: CET

CET (1977 b) Technical Report No 15 Educational Computing in the local authority sector - the next ten years A future study report, London: CET

CET (1978) Annual report for the year October 1977 - September 1978, London: CET

Chambers J (1992) Empiricist Research on Teaching A Philosophical and Practical Critique of its Scientific Pretensions, Dordrecht, Kluwer Academic Publishers

Chomsky N (1968) Language and Mind, New York: Harcourt Brace and Jovanovich

Bibliography

Chomsky N (1980) Rules and Representations, New York: Columbia University Press

Clark D (1996) *Interactive media programmes and the problem of scaling* in Active Learning Number 4 July 1996

Clements D H (1985) Computers in Early and Primary Education, Englewood Cliffs: New Jersey: Prentice Hall

Coburn P, Kelman P, Roberts N, Snyder T, Watt D, Weiner C (1982) Practical Guide to Computers in Education, Reading, Massachusetts: Addison-Wesley

Cohen L and Manion L (1994) Research Methods in Education, London: Routledge

Conrad B (2000) *Taking programming to the extreme edge* Infoworld, Framingham, July 2000, Volume 22 Issue 30

Cooper D (1985) Teaching Introductory Programming, New York: WW Norton & Co

Coplien J (2000) Software Patterns, New York: SIGS Books & Multimedia

Coulson J E (ed.) (1961) Programmed learning and computer-based instruction Proceedings of the conference on Application of Digital Computers to Automated Instruction October 10 - 12 1961, New York: Wiley and Sons

Cranston M (ed.) (1965) Locke on Politics, Religion and Education, New York: Collier Books

Creative Computing August 1984

Creative Computing August 1984 *Learning to Program* pp 58-59

Crook C (1994) Computers and the collaborative experience of learning a psychological perspective, London: Routledge

CTI (1992 a) Annual report of the Computers in Teaching Initiative, CTI

CTI (1992 b) Computers in university teaching: core tools for core activities, CTISS

CTI (1993) Annual report April 1992 to March 1993, CTI

CTI (1994) Annual report 1993-1994 of the Computers in Teaching Initiative, CTI

CTI (1996) Annual report 1994-1995, CTI

CTI (1998 a) A review of learning and teaching in UK higher education from the Computers in Teaching Initiative, Oxford: CTI

CTI Active Learning No 3 December 1995 Oxford: CTISS

Bibliography

CTI Active Learning No 4 July 1996 Oxford: CTISS

CTI Active Learning No 6 July 1997 Oxford: CTISS

CTI Active Learning No 8 July 1998 Oxford: CTISS

CTI Computing (1998 b) Journal of the CTI centre for computing Number 10 Summer 1998, CTI

CTI Computing (1998 c) NewsSheet Autumn/Winter 1998, CTI

CTI Engineering Software for engineering education, Issue 14 1998

CTISS (1988) The CTISS file No 6 April 1988 Oxford: CTISS

CTISS (1988) The CTISS file, No 5 February 1988 Oxford: CTISS

CTISS (1990) The CTISS File No 10 September 1990 Oxford: CTISS

CTISS (1990) The CTISS File No 9 February 1990 Oxford: CTISS

CTISS (1991) The CTISS File No 12 September 1991 Oxford: CTISS

CTISS (1992) The CTISS File No 13 April 1992 Oxford: CTISS

CTISS (1992) The CTISS File, No 14 October 1992 Oxford: CTISS

Cullen J, Hadjivassiliou K, Hamilton E, Kelleher J, Sommerlad E and Stern E (2002) *Review of the current pedagogic research and practice in the fields of post-compulsory education and lifelong learning Final report (revised) submitted to the Social and Economic Research Council by the Tavistock Institute*, London: Economic and Social Research Council

Cullum H (1995) CBT – Engineering materials, Final year project report, A J Bird's personal copy

Cummins R and Cummins D (2000) (eds) Minds, Brains and Computers the Foundations of Cognitive Science An Anthology, Malden, Massachusetts: Blackwell Publishers

Cunniff N, Taylor R and Black J (1989) *Does Programming Language Affect the Type of Conceptual Bugs in Beginners' Programs? A Comparison of FPL and Pascal* in Soloway and Spohrer (eds) (1989) Studying the Novice Programmer

Cvetkovic S, Chirico S, Larco-Burcheli L and Okretic V (1993) *CAL systems developed in advanced programming environments for teaching signals and systems* in CAL into the mainstream CAL 93 Conference handbook

Bibliography

Dancy J (ed.) (1998) A Treatise concerning The Principles of Human Knowledge, Oxford: Oxford University Press

Darby J (1990) *The Computers in Teaching Initiative: A Progress Report* in The CTISS File No 10 September 1990

Darby J (1991) *Computers in Teaching: The Needs of the 90s A report from the Computers* in Teaching Initiative The CTISS File No 12 September 1991

Darby J (1992) *Editorial* in The CTISS File No 13 April 1992

Darby J (1993 a) *Editorial* in The CTISS File No 15 April 1993

Darby J (1993 b) *Teaching and Learning Technology Programme* in CAL into the mainstream CAL 93 Conference handbook

Darby J and Martin J (eds) (1995) *Teaching with multimedia,* Active Learning No 3 December 1995, CTI

Darby J and Martin J (eds) (1996) Active Learning No 4 July 1996, CTI

Davert C E (1974) Procedural Guidelines for the Design of Mediated Instruction, Great Plains National Instructional Television Library, Washington (D.C.): Association for Educational Communications and Technology

Davies S (1993) *Models and theories of programming strategy* in International Journal of Man-Machine Studies 39 pp 237-267

Davy J and Jenkins T (1999) *Research led innovation in teaching and learning programming* in SIGSCE Bulletin Conference Proceedings 1999

de Cecco J P (1964) Educational Technology Readings in Programmed Instruction, New York: Holt Rinehart and Winston

Dearing R (1997) Higher education in a learning society: Report of the national committee of enquiry into higher education Found at http://www.leeds.ac.uk.educol/ncihe 12/08/1999

Dekeyser H, Goeminne K, Schuyten G, Dunantlaan H, Protier S, Martens R and Valcke M (1995) *Computer-assisted learning in an applied statistics course: what kind of benefit is there?* in CAL95 Learning to succeed Conference Handbook

Dennen V (2000) *Task Structuring for On-line Problem Based Learning: A Case Study* Educational Technology and Society 3 (3) 2000

Dennett D (1996) Kinds of Minds Towards an Understanding of Consciousness, London: Weidenfeld and Nicolson

Bibliography

Denning P J, Comer D, Gries D, Mulder M, Tucker A, Turner J and Young P (1989) *Computing as a discipline*, Communications of the ACM, 1989, Volume 32 no 1 pp 9-23

Department of Industry (1982) A programme for advanced information technology: the report of the Alvey Committee, London: HMSO

DePasquale P *Subsetting language elements in novice programming environments* Found at http://people.cs.vt.edu/~edwards/RESOLVE2002/proceedings/DePasquale.html 04/02/2003

Derry S J (1990) *Metacognitive models of Learning and Instructional Design* in Jones and Winne (eds) (1990) Adaptive Learning Environments Foundations and Frontiers

DES (1967) Children and their Primary Schools A Report of the Central Advisory Council for Education (England) Vol. 1: The Report, London: DES

Dewey J (1933) How we think: a restatement of the relation of the reflective thinking to educational practice, Boston: D C Heath and Company

DFE (Department of Education) and Welsh Office Education Department (1995) Information Technology in the National Curriculum, London: DFE and Welsh Office Education Department

DFE (Department of Education) and Welsh Office Education Department (1995) Information Technology in the National Curriculum, London: DFE and Welsh Office Education Department

DFEE (Department for Education and Employment) (1997) Survey of Information Technology in Schools 1996, London: DFEE

Dick B (1999) *What is action research*?
Found at http://www.scu.edu.au/schools/gcm/ar/whatisar.html 29/01/2004

Dickson W P, Neal V A and Gillingham M G (1984) *A Low Cost Multimedia Microcomputer Systems for Educational Research and Development* in Educational Technology Volume XXIV No 8 August 1984

Draper S (1996) *Programming skills, visual layout design, and unjustifiably useful testing: Three reports in the psychology of programming* Found at http://wildcat.psy.gla.ac.uk/~steve/PPIG96.html 06/12/2000

Draper S, Henderson F, Brown M, McAteer E, Smith E and Watt H (1995) *The TILT evaluation instruments and method: the state of our practice* in CAL95 Learning to succeed Conference Handbook

Dreyfus H (1992) What Computers Still Can't Do A Critique of Artificial Reason, Cambridge, Massachusetts: MIT Press

Bibliography

Du Boulay B, O'Shea T and Mark J (1989) *The Black Box Inside the Glass Box: Presenting Computing Concepts to Novices* in Soloway and Spohrer (eds) (1989) Studying the Novice Programmer

Duffy T, Lowyck J and Jonassen D (eds) (1993) Designing Environments for Constructive Learning, Berlin: Springer-Verlag

Duit R (1994) *The Constructivist View: A Fashionable and Fruitful Paradigm for Science Education Research and Practice* in Steffe and Gale (1994) (eds) Constructivism in Education

Duke J (1970) *Computers in Education II: The Leeds Seminar* in Journal of Educational Technology Vol. 1 No. 1 Jan. 1970 London: NCET

Dunn S and Morgan V (1987) The Impact of the Computer on Education A course for teachers, Englewood Cliffs, New Jersey: Prentice Hall

Durgin F (2000) *The Reverse Stroop Effect* Found at http://www.swarthmore.edu/SocSci/fdurgin1/publications/ReverseStroop/PBRStroop.html 22/02/2001

Eckstein J and Voelter M (2001) *Patterns and pedagogy – A winning team* Found at http://www-106.ibm.com/developerworks/webservices/library/j-patt.html 07/08/2003

Educational Technology Research and Development, Volume 45 No 3 1997 and Volume 45 No 2 1997

Educational Technology, Volume XXIV No 8 August 1984

Edwards S (2003) *Teaching software testing: Automatic grading meets test-first coding*, OOPSLA '03, October 26 – 30 2003, Anaheim, California

Eick S (1998) *Maintenance of Large Systems* in Stasko et al (eds) (1998) Software Visualization

Eisenbach S and Sadler C (1981) PASCAL for Programmers, Berlin: Springer-Verlag

Eisenstadt M (1993) *Human Cognition Research Laboratory The Open University (U.K.)* in Bridges between Worlds INTERCHI 1993 Human Factors in Computing Systems Conference Proceedings

Eisenstadt M and Brayshaw M (1998) *The truth about Prolog execution* in Stasko et al (eds) (1998) Software Visualization

Eisner E (1985) The Art of Educational Evaluation A Personal View, London: The Falmer Press

Bibliography

Eisner E and Peshkin A (1990) (eds) Qualitative Enquiry in Education The Continuing Debate, New York: Teachers College Press

Elkind D and Flavell J H (1969) Studies in Cognitive Development Essays in honour of Jean Piaget, New York: OUP

Engle G W and Taylor G (1968) Berkeley's Principles of Human Knowledge Critical Studies, Belmont, Wadsworth

Ericsson K A and Hastie R (1994) *Contemporary approaches to the study of thinking and problem solving* in Sternberg (1994) (ed.) Thinking and Problem solving

Espich J E and Williams B (1967) Developing Programmed Instructional Materials, London: Pitman

Evans G, Dodds A, Kemm R, Weaver D and McCarroll G (1999) *Individual differences in strategies, activities, and outcomes in computer aided learning: a case study* Found at http://www.aare.edu.au/99pap/dod99220.htm 15/02/2001

Evans KM (1991) Planning small scale research A Practical Guide for Teachers and Students, Windsor, Berkshire: NFER-NELSON Publishing Company

Eysenck M (1998) Psychology an integrated approach, Harlow: Addison Wesley Longman

Farmer M (1984) The Intensive Pascal Course, Bromley: Chartwell-Bratt

Feynman R (2000) The Pleasure of Finding Things Out The Best Short Works, London: Allen Lane The Penguin Press

Field J (ed.) (1997) Electronic Pathways Adult Learning and the new communication technologies, Leicester: NIACE

Fielden and Pearson P (1978) The cost of learning with computers The report of the Financial Evaluation of the National Development Programme in Computer Aided Learning, London: CET

Firdyiwek Y (1999) *Web-Based Courseware Tools: Where Is the Pedagogy?* in Educational Technology, Volume XXXIX No 1

Flavell J and Markman E (eds) (1983) Volume III Cognitive Development, New York: Wiley

Flavell J H (1963) The Developmental Psychology of Jean Piaget, Princeton, New Jersey: D Van Nostrand

Flugel J C (1964 3rd Edition) A hundred years of psychology, London: Duckworth and Co

Bibliography

Formanek R and Gurian A (1976) Charting intellectual development A practical guide to Piagetian tasks, Springfield: C C Thomas

Fothergill R (1983) *The Microelectronics Education Programme in the United Kingdom* in World Yearbook of Education 1982/983, New York: Kogan Page

Fothergill R, Anderson J, Aston M, Bevis G, Irving A, Russell P and Wheeler P (1983) Microelectronics Education Programme Information Guide 4 Microelectronics Programme policy and guidelines, London: CET

Fox R and Tiger L (1972) The Imperial Animal, London: Secker & Warburg

Fox R and Tiger L (1998 Edition) The Imperial Animal, New Brunswick: Transaction Publishers

Fricke A and Voelter M (2000) *Seminars A Pedagogical Pattern Language about teaching seminars effectively*, EuroPLoP 2000

Fund for the Advancement of Education (1964) Four case studies of Programmed Instruction, New York: Fund for the Advancement of Education

Fyfe S, Slack-Smith L, Edwards P, Fyfe G and Kennedy D (1995) *Cross Check on Quality: Use and Evaluation of Interactive Multimedia Courseware in the Biomedical Sciences* in CAL95 Learning to succeed Conference Handbook

Gage N and Berliner D (1982) Educational Psychology, Chicago: Rand McNally College Publishing Company

Gagne R (1982) *Developments in Learning Psychology Implications for Instructional Design; and Effects of Computer Technology on Instructional Design and Development* in Educational Technology June 1982

Gagne R and Briggs L (1979) Principles of Instructional Design, New York: Holt, Rinehart & Winston

Gagne R M (1965) *The Analysis of Instructional Objectives for the Design of Instruction* in Glaser (1965) Teaching Machines and Programed Learning II Data and Directions

Galin D (2004) Software Quality Assurance From theory to implementation, Harlow, England: Pearson Addison Wesley

Gardner N (1996) *Avarice versus ideology: a perspective on TLTP* in Active Learning Number 4 July 1996

Gardner N and Slater J (1989) Computers in Teaching Initiative Detailed Descriptions, CTI

Bibliography

Gaskin J C A (ed.) (1999) Thomas Hobbes Human Nature and De Corpore Politico, Oxford: Oxford University Press

Geertz C (1993) The Interpretation of Cultures, London: Fontana Press

Gelman R and Baillargeon R (1983) *A review of some Piagetian concepts* in Flavell and Markman (eds) (1983) Volume III Cognitive Development

Gentner D and Stevens A (1983) (eds) Mental Models, Hillsdale, New Jersey: Lawrence Erlbaum Associates

Gerard R W (1969) *Shaping the mind: Computers in Education* in Atkinson and Wilson (eds) (1969) Computer Assisted Instruction A book of readings

Gergen K (1994) *Social Construction and the Educational Process* in Steffe and Gale (1994) (eds) Constructivism in Education

Gershuny J and Slater J (1989 a) Computers in Teaching Initiative An executive summary, Bath: CTI

Gershuny J and Slater J (1989 b) Computers in Teaching Initiative Report, Bath: CTI

Gershuny J and Slater J (1990) *Computers in Teaching Initiative: A Brief Review* in The CTISS File No 9 February 1990

Giarelli J and Chambliss J (1988) *Philosophy of Education as Qualitative Enquiry* in Sherman and Webb (1988) (eds) Qualitative Research in Education: Focus and Methods

Gibbs W W (1994) *Software's Chronic Crisis*, Scientific American, September 1994

Ginsburg H and Opper S (1979) Piaget's Theory of Intellectual Development, (2nd Edition) New Jersey: Prentice-Hall

Ginsburg H P (1997) Entering the child's mind, Cambridge: Cambridge University Press

Glaser M, Chi R, Farr M (ed.) and Glaser R (1988) The Nature of Expertise, Hillsdale, New Jersey: Lawrence Erlbaum Associates Publishers

Glaser R (ed.) (1965) Teaching Machines and Programed Learning II Data and Directions, Washington: National Education Association of the United States

Glassborow F with Allen R (2004) You Can Do It! A Beginner's Introduction to Computer Programming, Chichester: Wiley and Sons

Glassman W E (1979) *The Cognitive Approach* in Medcof and Roth (1979) Approaches to Psychology

Bibliography

Glynn I (1999) An Anatomy of Thought The Origin and Machinery of the Mind, London: Orion Books

Goldsworthy R (1999) *Lessons on Learning and Technology: Roles and Opportunities for Design and Development* in Educational Technology Volume XXXIX No. 4

Goodman R (1967) Programmed Learning and Teaching Machines An Introduction, London: English Universities Press

Gopnik A and Meltzoff A (1997) Words, Thoughts and Theories, Massachusetts: MIT Press

Gosman A D, Launder B E and Reece G J (1979) CAL *in an integrated course on fluid mechanics and heat transfer* in Learning through computers An introduction to Computer Assisted Learning in Engineering, Mathematics and the Sciences at tertiary level

Green T (1980) *Programming as a Cognitive Activity* in Smith and Green (1980) Human Interaction with Computers

Green T (1983) *Learning Big and Little Programming Languages* in Wilkinson (ed.) Classroom Computers and Cognitive Science

Green T (1990 a) *Programming Languages as Information Structures* in Hoc et al. (eds) The Psychology of Programming

Green T (1990 b) *The Nature of Programming* in Hoc et al. (eds) The Psychology of Programming

Green T, Payne S and van de Veer G (1983) (eds) The Psychology of Computer Use, London: Academic Press

Greenbaum R and Tilker H (1972) The challenge of psychology, New Jersey: Prentice Hall

Greenfield S (1997) The Human Brain A Guided Tour, London: Weidenfeld and Nicholson

Greer J and Mandinach E (1990) *Environments to support learning* in Jones and Winne (eds) (1990) Adaptive Learning Environments Foundations and Frontiers

Gross R D (1994 2nd Edition) Key studies in psychology, London: Hodder and Stoughton

Gross T F (1985) Cognitive Development, Monterey, Brooks/Cole Publishing Company

Bibliography

Guiffre Cotton E (2000) The Online Classroom: Teaching with the Internet, Bloomington, Indiana: EDINFO Press

Guyer P (ed.) (1992) Cambridge Companion to Kant, Cambridge: Cambridge University Press

Guzdial M (1994) *Software-realized scaffolding to facilitate programming for science learning*, Interactive Learning Environments 4, 1 pp 1- 44

Habermas J (1972) Translated by Sapiro J Knowledge and Human Interests, London: Heineman

Halff H (1988) *Curriculum Instruction in Automated Tutors* in Polson and Richardson (1988) Foundations of Intelligent Tutoring Systems

Halford G S (1978) *Introduction: The structural approach to cognitive development* in Keats, Collis and Halford (eds) (1978) Cognitive Development Research based on a neo-Piagetian approach

Hall J, Colbourn C, and Light P (1995) *"The knee bone's connected to the ..." Evaluating educational technology in medical education* in CAL95 Learning to Succeed Conference Handbook

Hall M *Teaching with Electronic Technology* Found at http://www.ala.org/aasl/kqweb/28_5_webprofilefulltext.html 14/09/2000

Hall P, Hovenden F, Rachel J and Robinson H (1998) *Postmodern Software Development* Found at http://mcs.open.ac.uk/mcs-tech-reports/96-07.pdf 07/01/2004

Hall S (1996) *Reflexivity in Emancipatory Action Research: Illustrating the Researcher's Constitutiveness* in Zuber-Skerritt (ed.) (1996) New Directions in Action Research

Hammersley M (1992) What's wrong with ethnography? London: Routledge

Hammond N (1993) *Editorial* in CAL into the mainstream CAL 93 Conference handbook

Hammond N and Trapp A (eds) (1993) CAL into the mainstream CAL 93 Conference handbook , CTI Centre for Psychology

Hampton J A (1979) *Service mathematics teaching in further and higher education* in Learning through computers An introduction to Computer Assisted Learning in Engineering, Mathematics and the Sciences at tertiary level

Hanson L F and Komoski P K (1965) *School Use of Programed Instruction* in Glaser (1965) Teaching Machines and Programed Learning II Data and Directions

Bibliography

Harasim L (ed.) (1990) On Line Education Perspective on a New Environment, New York: Praeger

Harding R and Robinson B (1995) *Welcome message,* CAL95 Learning to succeed Conference Handbook

Harding R D (1979) *The computer as an aid in applied mathematics* in Learning through computers An introduction to Computer Assisted Learning in Engineering, Mathematics and the Sciences at tertiary level

Hartson H (ed.) (1985) Advances in Human-Computer Interaction, Norwood, New Jersey: Ablex Publishing Corporation

Hauswirth M, Jazayeri M and Winzer A (1998) *A Java based environment for teaching programming language concepts* Found at http://www.infosys.tuwien.ac.at/Staff/pooh/papers/Simplesem/1190-HTML.html 03/05/2000

Hawkes M (1999) *Exploring Network-Based Communication in Teacher Professional Development* in Educational Technology Volume XXXIX No. 4

Hawkridge D G (1973) Technical Report No 1 Problems in Implementing Computer Managed Learning, London: CET

Hawksley C (1988) Pascal Programming A Beginner's Guide to Computer and Programming, Cambridge: Cambridge University Press

Hayes N (1994) Foundations of Psychology An Introductory Text, London: Routledge

Hearnshaw L S (1989) The shaping of modern psychology An historical introduction, London: Routledge

Helm P (1997) *Teaching and Learning with the new technologies: for richer, for poorer; for better, for worse…* in Field (ed.) (1997) Electronic Pathways Adult Learning and the new communication technologies

Hirschorn P (1979) *The Behaviourist Approach* in Medcof and Roth (1979) Approaches to Psychology

Hoc J (1983) *Analysis of beginners' problem-solving strategies in programming* in Green et al (eds) (1983) The Psychology of Computer Use

Hoc J, Green T, Samurçay R and Gilmore D (eds) (1990) The Psychology of Programming, London: Academic Press

Hock R R (1995) Forty studies that changed psychology Explorations in the history of psychological research, New Jersey: Prentice Hall

Bibliography

Hofmeister A (1984) Microcomputer Applications in the Classroom, New York: Holt, Rhinehart and Winston

Hofstetter F (1979) *The meaning of PLATO at the University of Delaware* in Lewis and Tagg (eds) (1980) Computer Assisted Learning Scope, Progress and Limits

Holt R (1972*) Teaching the Fatal Disease (or) Introductory Computer Programming using PL/I* Found at http://plg.uwaterloo.ca/~holt/papers/fatal_disease.html 19/10/2000

Honey P and Mumford A (1992) The Manual of Learning Styles, Maidenhead, Berkshire: Peter Honey

Hooper R (1974 a) Technical Report No 3 Making Claims for Computers, London: CET

Hooper R (1974 b) Computers and Sacred Cows, London: CET

Hooper R (1977) Final Report of the Director, London: CET

Hooper R and Toye I (eds) (1975) Computer assisted learning in the United Kingdom Some case studies, London: CET

Howe J A M, Ross P M, Johnson K R, Plane R and Inglis R (1981*) Teaching mathematics through programming in the classroom* in Computer Assisted Learning Selected papers from the CAL 81 symposium

Howe J, Ross P, Johnson K, Plane R and Inglis R (1981*) Teaching mathematics through programming in the classroom* in Computer Assisted Learning Selected papers from the CAL 81 symposium

Howes D H and Boring E G (eds) (1966) Elements of psychophysics, New York: Holt Rinehart and Winston

http://antioch-college.edu/~andrewc/home/kant/kant_glossary.html 29/06/1999

http://aral.cps.msu.edu/CPS101SS99/CPS101Visitor/InstructionalTheory.htm 25/04/2001

http://ascilite.org.au/conferences/brisbane99/program/snapshot-abstracts.htm 15/02/2001

http://ascilite.org.au/conferences/brisbane99/program/snapshot-abstracts.htm 15/02/2001

http://atschool.eduweb.co.uk/mbaker/worksht/pup-recs.html 03/05/2000

http://byte.com/art/9509/sec7/art19.htm 04/12/2000

Bibliography

http://c2.com/cgi-bin/wiki?HistoryOfPatterns 13/08/2003

http://c2.com/ppr/about/author/kent.html 30/03/2004

http://classics.mit.edu/Aristotle/memory.html 08/06/1999

http://coglab.psych.purdue.edu/coglab/Labs/StroopEffect.html 22/02/2001

http://cogprints.soton.ac.uk/documents/disk0/2000/2000/10/50/index.html 22/02/2001

http://courses.cs.vt.edu/~cs3604/lib/Therac_25/Therac_1.html 07/01/2004

http://csis.pace.edu/~bergin/papers/SimpleDesignPatterns.html. 07/08/2003

http://csis.pace.edu/~bergin/patterns/codingpatterns.html 09/03/2003

http://csis.pace.edu/~bergin/patterns/ExperientialLearning.html 09/03/2003

http://csis.pace.edu/~bergin/patterns/fewpedpats.html 09/03/2003

http://csis.pace.edu/~bergin/PedPat1.3.html

http://cui.unige.ch/Visual/local/Blackwell96c.html 25/04/2001

http://cuiwww.unige.ch/OSG/info/Langlist 04/12/2000

http://edu-ss10.educ.queensu.ca/~brownan/courses/aqcsdp97fall/learningtoprog.htm 15/02/2001

http://el.www.media.mit.edu/people/acs/chapter1.html 30/07/1999

http://ericae.net/tc2/TC830363.htm 08/12/2000

http://eyelab.psy.msu.edu/people/henderson/research.html 22/02/2001

http://faculty.insead.fr/adner/projects/E-procurement%202.pdf 13/11/2003

http://foldoc.doc.ic.ac.uk/foldoc/foldoc.cgi?query=axiomatic+semantics 19/02/2003

http://gear.kku.ac.th/Tutorial/Pascal/default.htm 04/12/2000

http://hc.les.dmu.ac.uk/michael/qual_aims.htm 13/01/2004

http://hc.les.dmu.ac.uk/tim/pysc1010/L1_handout.html 01/06/1999

http://hc.les.dmu.ac.uk/tim/pysc1010/L2_handout.html 01/06/1999

http://hc.les.dmu.ac.uk/tim/pysc1010/L3_handout.html 01/06/1999

Bibliography

http://hillside.net/vision.html 18/03/2004

http://homepages.dsu.edu/mckeownj/CPCCCSCpaper.html 04/12/2000

http://http.cs.berkeley.edu/~abegel/ed222a/ed222a-paper.html 25/04/2001

http://ironbark.ucnv.edu.au/courses/subjects/c101.html 04/12/2000

http://keirsey.com/personality/nfij.html 28/09/2002

http://kmi.open.ac.uk/people/paulm/aied/ 04/12/2000

http://lieber.www.media.mit.edu/people/lieber/Lieberary/Softviz/CACM-Debugging/Hairiest.html 18/04/2001

http://lrs.ed.uiuc.edu/Guidelines/ 14/09/2000

http://mcs.open.ac.uk/mcs-tech-reports/96-07.pdf 07/01/2004

http://news.com.com/2100-1017-913223.html 04/03/2003

http://pages.cpsc.ucalgary.ca/~kremer/patterns/history.html

http://panizzi.shef.ac.uk/medtl/lrnjrnl.html 09/12/2000

http://people.cs.vt.edu/~edwards/RESOLVE2002/proceedings/DePasquale.html 04/02/2003

http://pizza.cs.ucl.ac.uk/teaching/html/Programming-I.xml.html 13/02/2003

http://plg.uwaterloo.ca/~holt/papers/fatal_disease.html 19/10/2000

http://psyche.cs.monash.edu.au/v6/psyche-6-11-austen.html 22/02/2001

http://psy-test.com/Bsys.html 08/12/2000

http://reports-archive.adm.cs.cmu.edu/anon/usr/anon/home/ftp/1998/CMU-CS-98-101.html 24/04/2001

http://seeri.etsu.edu/SECodeCases/ethicsC/DeathByWire.htm 07/01/2004

http://socrates.berkeley.edu/~maccoun/ar_bias.html 25/04/2001

http://spicy.atd.depaul.edu/jumpstart/assess/gloss/glossary.html 14/09/2000

http://taylorandfrancis.com/psypress/BKFILES/0863776124.htm 22/02/2001

http://thinkofit.com/drwool/dwconf.htm 14/09/2000

Bibliography

http://thinkofit.com/plato/dwplato.htm 14/09/2000

http://trochim.human.cornell.edu/kb/qualdata.htm 07/01/2004

http://uk.geocities.com/balihar_sanghera/qrmparticipantobservation.html 02/02/2004

http://unicoi.kennesaw.edu/~jbamford/csis4650/uml/GoF_Patterns/memento.htm 30/04/2004

http://userpages.umbc.edu/~schmitt/331S96/dhubba1/report.html 04/12/2000

http://waltoncollege.uark.edu/disted/effective_pedagogies_for_managin.htm 14/09/2000

http://web.cps.msu.edu/~urban/ITS.htm#_Toc355707493 19/11/1999

http://wildcat.psy.gla.ac.uk/~steve/PPIG96.html 06/12/2000

http://wind.cc.whecn.edu/~gnelson/paper/TCC-98.html 14/09/2000

http://www4.thny.bbc.co.uk 13/11/2003

http://www-106.ibm.com/developerworks/webservices/library/j-patt.html 07/08/2003

http://www.aare.edu.au/99pap/dod99220.htm 15/02/2001

http://www.acm.org/classics/dec95 20/10/2000

http://www.acm.org/classics/oct95 20/10/2000

http://www.acm.org/sigplan/ 04/12/2000

http://www.adlnet.org/home.cfm 12/03/2002

http://www.ai.mit.edu/ 12/03/2003

http://www.ala.org/aasl/kqweb/28_5_webprofilefulltext.html 14/09/2000

http://www.anova.org/jsmill.html 10/06/1999

http://www.art.ttu.edu/arted/syllabi/res%20biblio/qual.html 14/02/2001

http://www.artima.com/intv/simplest.html 30/03/2004

http://www.ascusc.org/jcmc/vol4/issue2/larose.html 13/01/2004

http://www.atl.ualberta.ca/articles/idesign/learnchar.cfm 20/09/2000

http://www.auckland.ac.nz/cpd/caleval.html 15/02/2001

Bibliography

http://www.bc.edu/bc_org/avp/eve/02-03mt35001fall.pdf 13/02/2003

http://www.borland.com/pascal/ 04/12/2000

http://www.br.cc.va.us/vcca/i11tayl.html 09/12/2000

http://www.cals.ncsu.edu/agexed/aee735/ppt2/tsld007.htm 09/12/2000

http://www.che.udel.edu/faculty/full/sandler/comp.html 14/09/2000

http://www.chelt.ac.uk/el/philg/gdn/discuss/kolb1.htm 09/12/2000

http://www.chesworth.com/pv/languages/index.html 04/12/2000

http://www.chisp.net/~dminer/c64/gazette/8801/8801090.html 04/02/2003

http://www.cica.indiana.edu/cscl95/jonassen.html 25/04/2001

http://www.cis.gsu.edu/~mmoore/CIS3300/handouts/SciAmSept1994.html 20/11/2003

http://www.classes.cec.wustl.edu/~cs504/home.html 04/12/2000

http://www.cmcrossroads.com/bradapp/docs/patterns-intro.html 09/03/2003

http:///www.cmcrossroads.com/bradapp/docs/pizza-inv.html 18/03/2004

http://www.cnn.com/TECH/space/9909/30/mars.metric/ 07/01/2004

http://www.columbia.edu/itc/english/f2007/jameson/concepts/dialectic.html 08/03/2001

http://www.computer.muni.cz/student/looking/spring97/janlee/ 04/12/2000

http://www.cs.cmu.edu/afs/cs.cmu.edu/user/mleone/web/language/publications.html 04/12/2000

http://www.cs.cmu.edu/afs/cs.cmu.edu/user/mleone/web/language-research.html 04/12/2000

http://www.cs.helsinki.fi/research/aaps/Jeliot/ 08/12/2000

http://www.cs.iastate.edu/~leavens/ComS227.html#Overview 13/02/2003

http://www.cs.iupui.edu/~aharris/n301/alg/tmcm-java-labs/labs/xTurtleLab1.html 04/12/2000

http://www.cs.montana.edu/~dynalab/description/index.html 08/12/2000

Bibliography

http://www.cs.nott.ac.uk/~smx/PGCHE/ceilidh.html 31/03/2004

http://www.cs.utexas.edu/users/ethics/SE_education/se_edu_uk.html 13/11/2003

http://www.csa1.co.uk/htbin/ids51/txtdisp.cgi 03/08/2000

http://www.csa2.co.uk/htbin/ids51/procskel.cgi 02/08/2000

http://www.csc.liv.ac.uk/~ken/cpp/prog1.html 03/05/2000

http://www.cse.unsw.edu.au/~cs3111/Lectures/ 2003S2SA/newIntro.pdf 20/11/2003

http://www.cti.ac.uk 14/09/2000

http://www.cti.ac.uk/info/handbook.html 19/08/1999

http://www.cti.ac.uk 18/02/1999

http://www.cti.ac.uk/news/ 18/02/1999

http://www.curtin.edu.au/conference/ASCILITE97/papers/Sanders/Sanders.html 15/02/2001

http://www.cvcp.ac.uk/ 18/02/1999

http://www.cwu.edu/~ceps/constdef.htm 30/07/1999

http://www.dacs.dtic.mil/techs/history/His.RL.1.0.html 04/03/2003

http://www.developercareers.com/ddj/articles/1999/199914/199914b/199914b.htm 08/12/2000

http://www.dcs.qmw.ac.uk/~norman/papers/new_directions_metrics/HelpFileHistory_of_software_metrics_as_a.htm 06/04/2004

http://www.dfee.gov.uk/pns/DisplayPN.cgi?pn_id=2001_0070 15/02/2001

http://www.doc.ic.ac.uk/~nd/surprise_95/journal/vol2/hst/article2.html 31/03/2004

http://www.dofactory.com/patterns/PatternMemento.aspx 30/03/2004

http://www.dreamsongs.com/NewFiles/AlexanderPresentation.pdf 18/03/2004

http://www.eduard-rhein-stiftung.de/html/T2002_e.html 20/1/2003

http://www.educationau.edu.au/archives/cp/09.htm 15/02/2001

http://www.educationau.edu.au/archives/cp/09.htm 15/02/2001

Bibliography

http://www.emporia.edu/math-cs/simpson/cs250/cs250.htm 04/12/2000

http://www.engin.umd.umich.edu/CIS/course.des/cis400/euphoria/euphoria.html 04/12/2000

http://www.epicent.com/journals/computer/smith002.html 25/04/2001

http://www.ericfacility.net/ericdigests/ed355205.html 29/01/2004

http://www.extremeprogramming.org 18/01/2001

http://www.gise.org/JISE/Vol1-5/DOESINST.htm 24/04/2001

http://www.gregorc.com/ 01/11/2000

http://www.gse.uci.edu/EdTechUse/c-tblcnt.htm 12/03/2002

http://www.hcc.hawaii.edu/intranet/committees/FacDevCom/guidebk/teachtip/keirsey .htm 08/12/2000

http://www.hcc.hawaii.edu/intranet/committees/FacDevCom/guidebk/teachtip/modalit y.htm 08/12/2000

http://www.holtsoft.com/turing/essay.html 31/10/2000

http://www.icbl.hw.ac.uk/ctl/msc/ceejw1/paper1.html 30/07/1999

http://www.ics.ltsn.ac.uk/pub/conf2000/Papers/jenkins.htm 13/02/2003

http://www.ils.nwu.edu/~e_for_e/nodes/I-M-NODE-4159-pg.html 15/02/2001

http://www.ils.nwu.edu/~e_for_e/nodes/NODE-43-pg.html 15/02/2001

http://www.ils.nwu.edu/edoutrage/edoutrage11.html 15/02/2001

http://www.ilt.columbia.edu/ilt/papers/ILTpedagogy.html 15/02/2001

http://www.ilt.columbia.edu/K12/livetext/docs/construct.html 30/07/1999

http://www.inform.umd.edu/UMS+State/UMD-Projects/MCTP/WWW/Essays/Constructivism.txt 30/07/1999)

http://www.inform.umd.edu:8080/UMS+State/UMD-Projects/MCTP/WWW/Essays.html 30/07/1999

http://www.infosys.tuwien.ac.at/Staff/pooh/papers/Simplesem/1190-HTML.html 03/05/2000

http://www.insightsvancouver.com/comparisons.htm 24/04/2003

Bibliography

http://www.internetvalley.com/intval.html 14/09/2000

http://www.irisa.fr/manifestations/1995/AADEBUG95/welcome.html 12/11/2000

http://www.it.bton.ac.uk/staff/rng/teaching/notes/Soar.html 13/03/2003

http://www.jdl.co.uk/training/4-siders/softwEng_4s.html 20/11/2003

http://www.jmu.edu/gened/Knowledge/Chap12.htm 04/12/2000

http://www.kablenet.com/kd.nsf/Frontpage/A8F5EAC1009EEC2780256C950060ADCF?OpenDocument 03/2/2003

http://www.kmi.open.ac.uk/~paulm/ppig/spring97.html 09/07/1998

http://www.knight.org/advent/cathen/02004a 04/06/1999

http://www.lawandco.com/wwwboard/messages/2.html 09/07/1998

http://www.leeds.ac.uk.educol/ncihe 18/02/1999

http:// www.learninglinks.org/emotional_intelligence.pdf 06/04/2004

http://www.lifelonglearning.co.uk/greenpaper/ 18/02/1999

http://www.louisepryor.com/showTopic.do?topic=6 13/02/2003

http://www.mc.maricopa.edu/users/vaughan/text/lex/defs/dialectic.html 0803/2001

http://www.mii.kurume-u.ac.jp/~leuers/Freud.htm 17/06/1999

http://www.nae.edu/NAE/naehome.nsf/weblinks/MKEZ-5KYNFL?OpenDocument 19/11/2003

http://www.nasponline.org/publications/cq312cbattery.html 24/04/2003

http://www.nctm.org/jrme/abstracts/volume_03/vol03-02-mar1972.html 24/08/2000

http://www.neoscience.org/reflexiv.htm 13/01/2004

http://www.nickerson.to/visprog/ch2/progvis24.htm 20/02/2003

http://www.niss.ac.uk/education/hefce/pub98/ 18/02/1999

http://www.ovum.com/go/content/c,36416 19/11/2003

http://www.parliament.the-stationery-office.co.uk/pa/cm199900/cmselect/cmpubacc/65/6503.htm 07/01/2004

Bibliography

http://www.peak.org/~sechrest/talks/internet.howto.html 10/02/2004

http://www.ped.gu.se/biorn/phgraph/wild/briefing.html 15/02/2001

http://www.ped.gu.se/biorn/phgraph/civil/graphica/diss.ab/booth.html 5/02/2001

http://www.pentile.com/Human_Vision.html 06/03/2001

http://www.phenomenologyonline.com/glossary.cfm?range=e-h 08/05/2003

http://www.ppig.org/resources.html#newsletters 04/12/2000

http://www.programmersheaven.com/ 04/12/2000

http://www.python.org/cp4e/ 06/05/2003

http://www.python.org/doc/essays/cp4e.html 09/03/2003

http://www.rolemodelsoftware.com/moreAboutUs/publications/thereforeBoom.php 18/03/2004

http://www.saginaw.k12.mi.us/~mobility/vsionsys.htm 06/03/2001

http://www.sci.brooklyn.cuny.edu/~arnow/ED/coreloopinvar.html 04/12/2000

http://www.scit.wlv.ac.uk/~cm1995/cbr/library.html 26/11/2003

http://www.scu.edu.au/schools/gcm/ar/whatisar.html 29/01/2004

http://www.sd.monash.edu.au/blue/papers.html 03/05/2000

http://www.si.umich.edu/ICOS/gentleintro.html 12/03/2003

http://www.sigmod.org/pubs/contents/proceedings/series/popl/ 04/12/2000

http://www.smccd.net/accounts/mecorney/f2000foto/color/color.htm 06/03/2001

http://www.smccd.net/accounts/mecorney/f2000foto/color/light.htm 6/03/2001

http://www.smccd.net/accounts/mecorney/f2000foto/color/matter.htm 06/03/2001

http://www.smccd.net/accounts/mecorney/f2000foto/color/vision.htm 06/03/2001

http://www.soften.ktu.lt/jep-06032/city/courses/IFPR402/human.html 06/04/2004

http://www.space.com/news/mco_report-b_991110.html 07/01/2004

http://www.swarthmore.edu/SocSci/fdurgin1/publications/ReverseStroop/PBRStroop.html 22/02/2001

Bibliography

http://www.tall.ox.ac.uk/alt/alt-c98 18/02/1999

http://www.tau-web.de/hci/space/i3.html 06/03/2001

http://www.tec.puv.fi/~kimmos/News/0574 15/02/2001

http://www.tec.puv.fi/~kimmos/News/0575 15/02/2001

http://www.tec.puv.fi/~kimmos/News/comp/programming/1626 15/02/2001

http://www.tec.puv.fi/~kimmos/News/vanhat/Comp.edu 15/02/2001

http://www.telalink.net/~hauk/pascal/ 04/12/2000

http://www.templetons.com/brad/alice.html 26/10/2000

http://www.tml.hut.fi/~pnr/Tik-76.278/gof/html/Memento.html 30/03/2004

http://www.ucc.ie/hfrg/projects/respect/urmethods/interviews.htm 03/02/2004

http://www.ulst.ac.uk/cticomp/bowman.html 28/11/2000

http://www.ulst.ac.uk/cticomp/papers/traxler.html 09/07/1998

http://www.unb.ca/web/units/psych/likely/headlines/C1650_99.htm 01/06/1999

http://www.unb.ca/web/units/psych/likely/headlines/C1700_49.htm 01/06/1999

http://www.unb.ca/web/units/psych/likely/headlines/C1780_99.htm 01/06/1999

http://www.unb.ca/web/units/psych/likely/headlines/C1800_29.htm 01/06/1999

http://www.unb.ca/web/units/psych/likely/headlines/C1830_49.htm 01/06/1999

http://www.unb.ca/web/units/psych/likely/headlines/C1850_69.htm 01/06/1999

http://www.unb.ca/web/units/psych/likely/headlines/C1870_79.htm 17/06/1999

http://www.unb.ca/web/units/psych/likely/headlines/C1880_89.htm 08/07/1999

http://www.unb.ca/web/units/psych/likely/headlines/C1890_99.htm 08/07/1999

http://www.unb.ca/web/units/psych/likely/headlines/C1900_14.htm 08/07/1999

http://www.unb.ca/web/units/psych/likely/headlines/C1915_24.htm 08/07/1999

http://www.unb.ca/web/units/psych/likely/headlines/C1925_29.htm 08/07/1999

Bibliography

http://www.unb.ca/web/units/psych/likely/headlines/C1940_49.htm 08/07/1999

http://www.unb.ca/web/units/psych/likely/headlines/C1950_59.htm 08/07/1999

http://www.uni-giessen.de/hrz/tex/more_info/info/mailarchiv/litprog.1995/msg00507.html 03/05/2000

http://www.userlab.com/Vision1.html 06/03/2001

http://www.wam.umd.edu/~mlhall/teaching.html 04/12/2000

http://www.well.com/user/smalin/miller.html 05/03/2002

http://www.wikipedia.org/wiki/Context-free_grammar 27/02/2003

http://www.wjh.harvard.edu/~viscog/lab/ 22/02/2001

http://www.xcalibur.co.uk/training /programming/start.html 03/05/2000

http://www.yorku.ca/dept/psych/classics /Aristotle/memory.htm 01/06/1999

http://www.yorku.ca/dept/psych/classics /suggestions.htm 01/06/1999

http://www.yourdon.com/articles/7602Infosystems.html 06/04/2004

http://www2.lucidcafe.com/lucidcafe/lucidcafe/library/96may/freud.html 17/06/1999

http://www2.ucsc.edu/people/anoe/GrandIllusion.html 22/02/2001

http://www-hcs.derby.ac.uk/tip/models.html 13/03/2003

http://www-isu.indstate.edu/ctl/styles/learning.html 08/12/2000

http://www-jime.open.ac.uk/97/1/isvl-03.html 04/12/2000

http://www-2.cs.cmu.edu/~mleone/language/projects.html 09/03/2003

http://www2.sis.pitt.edu/~peterb/0012-031/syllabus.html 13/02/2003

http://www.Xprogramming.com 18/01/2001

http://xrs.net/resources/Pascal/ 13/02/2003

Hubbard G (1986) *Organising for innovation: A Comparative Commentary on two Government Programmes* in Journal of Computer Assisted Learning Vol. 2 No 1 March/April 1986

Hudson K (1984) Introducing CAL A practical guide to writing Computer Assisted Learning programs, London: Chapman and Hall

Bibliography

Inglis A, Ling P and Joosten V (1999) Delivering Digitally: managing the Transition to the Knowledge Media, London: Kogan Page

Inhelder B and Piaget J (1964) The early growth of logic in the child Classification and Seriation, Bristol, Routledge

Inhelder B, Sinclair H and Bovet M (1975) Learning and the Development of Cognition, London: Routledge

Innovations in Education and Teaching International, February 2002, Volume 39 Number 1, SEDA, London: Routledge

Institute for Learning and Teaching (1998) Institute for Learning and Teaching: Implementing the vision http://www.cvcp.ac.uk

Irwin T (1988) Aristotle's First Principles, Oxford: Oxford University Press

Irwin T and Fine G (1995) Aristotle: Selections, Indianapolis: Hacket

ISC Courseware Development Working Party (1992) The CTISS File No 14 October 1992

Jackson D and Usher M (1997) *Grading student programs using ASSYST*, Technical symposium on Computer Science Education, Proceedings of the twenty-eighth SIGCSE technical symposium on computer science, San Jose, California: pp 335 – 339

Jamison D, Suppes P and Butler C (1970) *Estimated Costs of Computer Assisted Instruction for Compensatory Education in Urban Areas* in Educational Technology, September 1970

Jaynes J (1993) The origin of consciousness in the collapse of the bicameral mind, London: Penguin

Jegede O, Fraser B and Curtin D (1995) *The Development and Validation of a Distance and Open Learning Environment Scale*, Educational Technology Research and Development 1995 Vol. 43 Pt 1 pp 90 - 94

Jenkins T and Davy J (2000) *Dealing with Diversity in Introductory Programming* Found at http://www.ics.ltsn.ac.uk/pub/conf2000/Papers/jenkins.htm 13/02/2003

Johnson-Laird P (1983) Mental Models Towards a Cognitive Science of Language, Inference and Consciousness, Cambridge: Cambridge University Press

Johnson-Laird P, Girotto V and Legrenzi P (1998) Mental models: a gentle guide for outsiders Found at http://www.si.umich.edu/ICOS/gentleintro.html 12/03/2003

Johnston H (1985) Learning to Program, Englewood Cliffs, New Jersey: Prentice Hall International

Bibliography

Johnston J (1987) Electronic Learning: From Audio tape to Video disc, Hillsdale, New Jersey: Laurence Erlbaum

Jonassen D and Rohrer-Murphy L (1999) *Activity Theory as a Framework for Designing Constructivist Learning Environments* in Educational Technology Research and Development Volume 47 No 1

Jonassen D *Operationalizing Mental Models: Strategies for Assessing Mental Models to Support Meaningful Learning and Design- Supportive Learning Environments* Found at http://www.cica.indiana.edu/cscl95/jonassen.html 25/04/2001

Jonassen D, Mayes T and McAleese R (1993) *A Manifesto for a Constructivist approach to Uses of Technology in Higher Education* in Duffy et al (eds) (1993) Designing Environments for Constructivist Learning

Jones A and Scrimshaw P (1988) (eds) Computers in Education 5 - 13, Milton Keynes: Open University Press

Jones A et al. (1995) *Evaluating CAL at the Open University: 15 years on* in Computers in Education Volume No 26 April 1996 pp 5 - 15

Jones A, Scanlon E and Blake C (2000) *Conferencing in communities of learners: examples from social history and science communication* in Educational Technology and Society 3 (3) 2000

Jones D & Pritchard A (1999) *Realizing the Virtual University* in Educational Technology Volume XXXIX Number 5

Jones M and Winne P H (eds) (1990) Adaptive Learning Environments Foundations and Frontiers, Berlin: Springer-Verlag

Journal of Computer Assisted Learning Vol. 2 No 1 March/April 1986

Journal of Computer Assisted Learning Vol. 5 No 1 March 1989

Kagan J and Havemann E (1980) Psychology An Introduction, New York: Harcourt Brace Jovanich

Kahn H and Wiener A J (1968) The Year 2000 A Framework for Speculation on the Next Thirty-Three Years, New York: Macmillan

Kahney H (1983) *Problem solving by novice programmers* in Green et al (eds) (1983) The Psychology of Computer Use

Kant I (1979 edition) (Introduction by Lindsay A D Translated by Meiklejohn J M D) Critique of Pure Reason, London: Dent

Kaye A (ed.) (1991) Collaborative Learning Through Computer Conferencing The Najaden Papers, Berlin: Springer-Verlag

Bibliography

Keats J A and Keats D M (1978) *The role of language in the development of thinking - theoretical approaches* in Keats, Collis and Halford (eds) (1978) Cognitive Development Research based on a neo-Piagetian approach

Keats J A, Collis K F and Halford G S (eds) (1978) Cognitive Development: Research Based on a neo Piagetian approach, Chichester, Wiley

Kelly A, Maunder S and Cheng S (1996) *Does practice make perfect? Using TRANSMATH to assess mathematics coursework* in Active Learning Number 4 July 1996

Kemmis S, Atkin R and Wright E (1977) How do students learn? Working papers in computer assisted learning, UEA Norwich, Centre for Applied Research in Education

Kent T and McNergney R (1999) Will Technology Really Change Education? From Blackboard to Web, Thousand Oaks, California: Corwin Press

Kevles D (1997) *Pursuing the Unpopular: A History of Courage, Viruses, and Cancer* in Silvers (1997) Hidden Histories of Science

Khan T (1996) *Pedagogic Principles of Case-Based CAL* in Journal of Computer Assisted Learning Volume No 12 September 1996 pp 172-192

Kibby M and Hartley J (eds) (1993) CAL into the mainstream (1993) Computer Assisted Learning Selected contributions from the CAL 93 symposium,Oxford: Pergamon Press

Kidd M E and Holmes G (1983) CAL *evaluation: a cautionary word* in CAL 83 Selected papers from the Computer Assisted Learning symposium

Kimelman D Rosenburg B and Roth T (1998) *Visualization of Dynamics in Real World Software Systems* in Stasko et al (eds) (1998) Software Visualization

Klaus D J (1965) *An Analysis of Programing Techniques* in Glaser (ed.) (1965) Teaching Machines and Programed Learning II Data and Directions

Koch S and Leary D (eds) (1985) A Century of Psychology as Science, Worcester, Massachusetts: McGraw - Hill

Kolb D (1984) Experiential Learning Experience as The Source of Learning and Development, Englewood Cliffs, New Jersey: Prentice-Hall

Krems J (1995) *Expert Strategies in Debugging: Experimental results and a Computational Model* in Wender et al (eds) Cognition and Computer Programming

Kurland D, Clement C, Mawby R and Pea R (1987) *Mapping the Cognitive Demands of Learning to Program* in Pea and Sheingold (eds) Mirrors of Minds

Bibliography

Kurland D, Pea R, Clement C and Mawby R (1989) *A Study of The Development of Programming Ability and Thinking Skills* in High School Students in Soloway and Spohrer (eds) (1989) Studying the Novice Programmer

LaRose R, Gregg and Eastin M (1998) *Audiographic Telecourses for the Web: An experiment*, JCMC 4 (2) December 1998 Found at http://www.ascusc.org/jcmc/vol4/issue2/larose.html 13/01/2004

Laurillard D (1993) Rethinking University Teaching a framework for the effective use of educational technologies, London and New York: Routledge Kegan Paul

Leask M and Pachler N (1999) (eds) Learning to Teach Using ICT in the Secondary School, London: Routledge

Leatherdale W (1974) The Role of Analogy, Model and Metaphor in Science, Amsterdam, North Holland Publishing Company

Lecarme O (1973) *What Programming Language should we Use for Teaching Programming?* in Turski (ed.) (1973) Programming Teaching Techniques

Lefrançois G R (1989) Of children: an introduction to child development, Belmont, California: Wadsworth Publishing Company

Lemut A, du Boulay B and Dettori G (eds) (1993) Cognitive Models And Intelligent Environments for Learning Programming, Berlin: Springer-Verlag

Lerner R M (1984) On the nature of human plasticity, Cambridge: Cambridge University Press

Lesgold A (1988) *Intelligent Tutoring Systems* in Jones and Scrimshaw (1988) (eds) Computers in Education 5 - 13

Leventhal L (1993) *How Confirmation Bias Affects Novice Programmers in Testing and Debugging: Research Strategies and Implications for Tools* in Lemut et al (eds) (1993) Cognitive Models And Intelligent Environments for Learning Programming

Levin J, Rogers A, Waugh M and Smith K *Observations on educational electronic networks: The importance of appropriate activities for learning*, http://lrs.ed.uiuc.edu/guidelines/LRWS.html 14/09/2000

Lewis D (1978) The secret language of the child How children talk before they can speak, London: Souvenir Press

Lewis R (1989) *Information Technology in Education Research Programme 1988 - 1993* in Journal of Computer Assisted Learning Vol. 5 No 1 March 1989

Lewis R and Tagg E (eds) (1980) Computer Assisted Learning Scope, Progress and Limits, Amsterdam, North Holland Publishing Company

Bibliography

Lewis W (1981) Problem-Solving Principles for PASCAL Programmers Applied Logic, Psychology, and Grit, Rochelle Park, New Jersey: Hayden Book Company

Lieberman H (2000) *Programming by example* Association for Computing Machinery Communications of the ACM New York March 2000 Volume 43 Issue 3 pp 72-74

Linn M and Dalbey J (1989) *Cognitive Consequences of Programming Instruction* in Soloway and Spohrer (eds) (1989) Studying the Novice Programmer

Linn P (1987) *Microcomputers in Education: Living and Dead Labour* in Scanlon and O'Shea (eds) (1987) Educational Computing

Low J, Johnson J, Hall P, Hovenden F, Rachel J, Robinson H and Woolgar S (1996) *Read this and change the way you feel about software engineering*, Information and Software Technology 38 (1996) pp 77- 87

McGettrick A (1998) *Software Engineering Education in the UK – An Overview* Found at http://www.cs.utexas.edu/users/ethics/SE_education/se_edu_uk.html 13/11/2003

MacCunn J (1964) Six Radical Thinkers Bentham J S Mill Cobden Carlyle Mazzini T H Green, New York: Russell and Russell

Mackenzie J (1979) Simulations in Learning through computers An introduction to Computer Assisted Learning in Engineering, Mathematics and the Sciences at tertiary level

Maddison A (1982) Microcomputers in the classroom, London: Hodder and Stoughton

Magee B (1997) Confessions of a philosopher, London: Weidenfeld and Nicholson

Mahoney M (1977) *Publication Prejudices: An Experimental Study of Confirmatory Bias in the Peer Review System in Cognitive Therapy and Research*, Volume 1, No. 2, 1977, pp 161-175 Found at http://www.mang.canterbury.ac.nz/courseinfo/AcademicWriting/Prejud.htm 25/04/2001

Maier H W (1969) Three Theories of child development Contributions of Erik H Erikson, Jean Piaget and Robert R Sears and their applications, New York: Harper Row

Maier P, Barnett L, Warren A and Brunner D (1998) Using technology in teaching and learning, London: Kogan Page

Malim T and Birch A (1998) Introductory Psychology, London: Macmillan Press

Manktelow K (1999) Reasoning and Thinking, Hove: Psychology Press

Bibliography

Markle S M (1964) Good Frames and Bad A Grammar of Frame Writing, New York: Wiley and Sons

Marshall D (1988) CAL/CBT - The great debate, Sweden: Chartwell-Bratt

Martin J (1996) *Is Turing a better language for teaching programming than Pascal* Found at http://www.holtsoft.com/turing/essay.html 31/10/2000

Martin J and Beetham H (eds) (1998) Active Learning No 8 July 1998

Marton F (1988) *Phenomenography: A Research Approach to Investigating Different Understandings of Reality* in Sherman and Webb (eds) (1988) Qualitative Research in Education: Focus and Methods

Maslow A H (1970) Motivation and Personality, New York: Harper and Row

Mason R and Kaye T (1990) *Towards a New Paradigm for Distance Education* in Harasim (ed.) (1990) On Line Education Perspective on a New Environment

Matterson A (1981) Polytechnics and Colleges, Harlow: Longman

Matthews G (1972) Plato's epistemology and related logical problems, London: Faber and Faber

Maund C (1937) Hume's Theory of Knowledge A Critical Examination, London: Macmillan

Mayer R (1989) *The Psychology of How Novices Learn Computer Programming* in Soloway and Spohrer (eds) (1989) Studying the Novice Programmer

Mayer R E (1992 2nd Edition) Thinking, problem solving, cognition, New York: W H Freeman and Company

Mayer R, Dyck J and Vilberg W (1989) *Learning to Program and Learning to Think: What's the Connection?* in Soloway and Spohrer (eds) (1989) Studying the Novice Programmer

Mayes T (2001) *Learning technology and learning relationships* in Stephenson (ed.) (2001) Teaching and Learning Online Pedagogies for New Technologies

McCalla G (1990) *The search for Adaptability, Flexibility and Individualization: Approaches to Curriculum in Intelligent Tutoring Systems* in Jones and Winne (eds) (1990) Adaptive Learning Environments Foundations and Frontiers

McConnell D (2000) Implementing Computer Supported Cooperative Learning, London: Kogan Page

McCormick S (1985) *Software and television - a new approach* in CAL 85 Advances in computer-assisted learning: selected proceedings from the CAL 85 symposium

Bibliography

McDonald B, Atkin R, Jenkins D and Kemmis S (1977) *Computer assisted learning: its educational potential* in Final Report of the Director

McFarland T D and Parker R (1990) Expert Systems in Education and training, Englewood Cliffs, New Jersey: Educational Technology Publications

McGregor J and Watt A (1986) Simple Pascal, Avon: The Bath Press

McIver L (2000) *The Effect of Programming Language on Error Rates of Novice Programmers* in Blackwell and Bilotta (eds) (2000) 12th Workshop of the Psychology of Programming Interest Group

McKeown J and Farrell T *Why We Need to Develop Success in Introductory Programming Courses* Found at http://homepages.dsu.edu/mckeownj/CPCCCSCpaper.html 04/12/2000

Meadows S (1994) The child as thinker The development and acquisition of cognition in childhood, London: Routledge

Meadows S (1995) Understanding Child Development Psychological perspectives in an interdisciplinary field of enquiry, London: Routledge

Medcof J and Roth J (eds) (1979) Approaches to Psychology, Milton Keynes: Open University Press

Megarry J (ed.) (1983) World Yearbook of Education 1982/983, New York Kogan Page

Melrose M J (1996) *Got a Philosophical Match? Does it Matter?* in Zuber-Skerritt (ed.) (1996) New Directions in Action Research

Mendelsohn P, Green T and Brna P (1990) *Programming Languages in Education: The Search for an Easy Start* in Hoc et al (eds) (1990) The Psychology of Programming

MEP (1986) Topics in contemporary computer studies Book 3 The sensible use of computers, MEP

MEP (1986) Topics in contemporary computer studies Book 5 Teaching computing by applications, MEP

Merriënboer J, Krammer H, Maaswinkel R (1992) *Automating the Planning and Construction of Programming Assignments for Teaching Introductory Computer Programming* in Tennyson (ed.) (1992) Automating Instructional Design, Development and Delivery

Mill J S (1869 Edition) Analysis of the Phenomena of the Human Mind Volumes I and II, London: Longmans, Green, Reader and Dyer

Bibliography

Mill J S (1964 Edition) (Introduction by Lindsay A D) Utilitarianism Liberty Representative Government, London: Dent

Miller G A (1991) Psychology The Science of Mental Life, London: Penguin

Miller J (1988) *The role of human-computer interaction in intelligent tutoring systems* in Polson and Richardson (1988) Foundations of Intelligent Tutoring Systems

Miller L (1974) *Programming by Non-programmers* in International Journal of Man-Machine Studies 6 pp 237 - 260

Mitchell D (1997) *Making sense of computer aided learning research: A critique of the pseudo-scientific method*, Quebec Canada: Graduate Programme in Educational Technology, Concordia University

Mitchell P (1970) *Educational Technology: Panacea or Placebo?* in Aspects of Educational Technology IV, London: Pitman

Modgil S and Modgil C (eds) (1987 a) B F Skinner Consensus and controversy, New York: Falmer Press

Modgil S and Modgil C (eds) (1987 b) Noam Chomsky Consensus and controversy, New York: Falmer Press

Mogey N (1996) *Tenacious Lecturers Target Procurement: the problem of obtaining TLTP software* in Active Learning Number 4 July 1996

Montagnon P and Bennett R (eds) (1965) What is programmed learning? London: BBC Publications

Moore A (1993) *Siuli's Maths Lesson : Autonomy or Control?* in Beynon and Mackay (eds) (1993) Computers into classrooms More Questions than Answers

Moore T and Carling C (1987) *Chomsky: consensus and controversy - introduction* in Modgil and Modgil (eds) (1987 b) Noam Chomsky Consensus and controversy

Morgan C T and King R A (1975 5th Edition) Introduction to Psychology, New York: McGraw Hill

Mossner E (ed.) (1969) A Treatise of Human Nature, London: Penguin

Mouly G (1978) Educational Research The Art and Science of Investigation, Boston, Massachusetts: Allyn and Bacon

Muijs D (2004) Doing quantitative research in education with SPSS, London: Sage

Mukherjea S and Stasko J (1994) *Toward Visual Debugging: Integrating Algorithm Animation Capabilities within a Source-level Debugger* in ACM Transactions on Computer-Human Interaction September 1994 Volume 1 Number 3 pp 215 – 244

Bibliography

Mulholland P and Eisenstadt M (1998) *Using Software to Teach Computer Programming: Past, Present and Future* in Stasko et al (eds) (1998) Software Visualization

Munsinger H (1975 2nd Edition) Fundamentals of child development, New York: Holt Rinehart and Winston

Mussen P H (1983) Handbook of Child Psychology (eds Flavell J H and Markman E M) Volume III Cognitive Development, New York: John Wiley and Sons

Myers B (1998) *Natural Programming: Project Overview and Proposal* Found at http://reports-archive.adm.cs.cmu.edu/anon/usr/anon/home/ftp/1998/CMU-CS-98-101.html 24/04/2001

Nash A and Ball D (1983) An introduction to microcomputers in teaching, London: Hutchinson

Naur P (1983) *Program development studies based on diaries* in Green et al. (eds) (1983) The Psychology of Computer Use

NCET (1969 a) The report of the National Council for Educational Technology 1967 - 1968 Towards more Effective Learning, London: NCET

NCET (1969 b) Working Paper No1 Computers for Education Report of a Working Party into the potential applications and development of computer based learning systems, London: NCET

NCET (1969 c) Computer Based Learning: A Programme for Action, London: NCET

NCET (1969 d) Computer Based Learning Systems A programme for research and development Report of a Feasibility Study to outline an advanced programme of research and development to apply computers to education and training, London: NCET

NCET (1970) Journal of Educational Technology Vol. 1 No.1 Jan. 1970 London: NCET

NCET (1970) Journal of Educational Technology Vol. 1 No.2 May 1970 London: NCET

NCET (1973) Educational Technology: Progress and Promise The Report of the National Council for Educational Technology 1967 - 1973, London: NCET

NDPCAL (1977) Final Report of the Director Richard Hooper National Council for Educational Technology

Nelson G *On-line evaluation: Multiple choice, discussion questions, essay, and authentic projects* Found at http://wind.cc.whecn.edu/~gnelson/paper/TCC-98.html 14/09/2000

Bibliography

Nelson K (1996) Language in cognitive development Cambridge: Cambridge University Press

Newby T, Stepich D, Lehman J and Russell J (1996) Instructional technology for Teaching and Learning Designing Instruction, Integrating Computers and Using Media, Columbus, Ohio: Merrill

Nisbet J and Broadfoot P (1980) The Impact of Research on policy and practice in education, Aberdeen: The University Press

Noë A, Pessoa L and Thompson E (2000) *Beyond the grand illusion: what change blindness really teaches us about vision* Found at http://www2.ucsc.edu/people/anoe/GrandIllusion.html 22/02/2001

Norman D (1983) *Some Observations on Mental Models* in Gentner and Stevens (eds) (1983) Mental Models

Norris M and Rigby P (1992) Software Engineering Explained, Chichester, England: John Wiley and Son

North S (1998) *Visualizing Graph Models of Software* in Stasko et al (eds) (1998) Software Visualization

O'Connor DJ (1957) An Introduction to the Philosophy of education, London: Routledge and Kogan Page

O'Donohue W and Kitchener R F (eds) (1996) The philosophy of psychology, London: Sage

O'Shea T and Self J (1983) Learning and teaching with computers : artificial intelligence in education, Brighton: Harvester Press

Ohlsson S (1990) *Artificial Instruction: A Method for Relating Learning Theory to Instructional Design* in Jones and Winne (eds) (1990) Adaptive Learning Environments Foundations and Frontiers

Olson D R (1970) Cognitive development The child's acquisition of diagonality, New York: Academic Press

Opacic P and Roberts A (1985) *CAL implementation* in Reid and Rushton (eds) (1985) Teachers, computers and the classroom

Opie C (ed.) (2004) Doing Educational Research A Guide to First Time Researchers, London: Sage

Organick E (1973) *Information Structure Concepts for Beginners: An Informal Approach* in Turski (ed.) (1973) Programming Teaching Techniques

Bibliography

Ormerod T (1990) *Human Cognition and Programming* in Hoc et al (eds) (1990) The Psychology of Programming

Owen M (2000) *Structure and discourse in a telematic learning environment* Educational Technology and Society 3 (3) 2000

Pane J and Myers B (2000) *The Influence of Programming on a Language Design: Project Status Report* in Blackwell and Bilotta (eds) (2000) 12th Workshop of the Psychology of Programming Interest Group

Papert S (1980) Mindstorms: Children, Computers and Powerful Ideas, Brighton: Harvester Press

Papert S (1989) *Teaching Children Thinking* in Soloway and Spohrer (eds) (1989) Studying the Novice Programmer

Papert S and Soloman C (1989) *Twenty Things to do with a Computer* in Soloway and Spohrer (eds) (1989) Studying the Novice Programmer

Parekh B (ed.) (1974) Jeremy Bentham Ten critical essays, London: Frank Cass

Payne A, Hutchings B and Ayre P (1980) Computer Software for Schools, London: Pitman

Pea R and Kurland D (1983) On the Cognitive Prerequisites of Learning Computer Programming Technical Report No 18, New York: Center for Children and Technology

Pea R and Kurland D (1987) *On the Cognitive Effects of Learning Computer Programming* in Pea and Sheingold (eds) (1987) Mirrors of Minds

Pea R and Sheingold K (eds) (1987) Mirrors of Minds Patterns of Experience in Educational Computing, Norwood, New Jersey: Ablex Publishing Corporation

Pea R, Kurland D and Hawkins J (1987) *Logo and the Development of Thinking Skills* in Pea and Sheingold (eds) (1987) Mirrors of Minds

Peltonen M (ed.) (1996) The Cambridge Companion to Bacon, Cambridge: Cambridge University Press

Percival P and Ellington H (1988) A Handbook of Educational Technology, London: Kogan Page

Perkins D, Hancock C, Hobbs R, Martin F and Simmons R (1989) *Conditions of Learning in Novice Programmers* in Soloway and Spohrer (eds) (1989) Studying the Novice Programmer

Bibliography

Petre M and Blackwell A (1999) *Mental Imagery in Program Design and Visual Programming* International Journal of Human-Computer Studies (1999) Volume 57 (1) pp 7-30

Petre M, Blackwell A and Green T (1998) *Cognitive Questions in Software Visualization* in Stasko et al (eds) (1998) Software Visualization

Piaget J, Amann A, Bonnet C L, Graven M F, Henriques A, Labarthe M, Maier R, Moreau A, Othenin-Girard C, Stratz C, Uzan S and Vergopoulo T (1978) Success and understanding, London: Routledge Kogan Page

Pillsbury W B (1929) The history of psychology, London: George Allen and Unwin

Pinker S (2002) The Blank Slate, London: Penguin Books

Pocztar J (1972) The Theory and Practice of Programmed Instruction A Guide for Teachers, Paris, UNESCO

Polson M and Richardson J (1988) Foundations of Intelligent Tutoring Systems, Hillsdale, New Jersey: Lawrence Erlbaum Associates Publishers

Popper K (1969) Conjectures and Refutations, London: Routledge

Powell J (1985) The Teacher's Craft A Study of Teaching in the Primary School Edinburgh: The Scottish Council for Research in Education

Powney J and Watts M (1987) Interviewing in educational research, London: Routledge Kogan Page

Pressman R (1992) Software Engineering: A Practitioner's Approach, New York: McGraw-Hill

Price B, Baecker R and Small I (1998) *An introduction to Software Visualization* in Stasko et al (eds) (1998) Software Visualization

Price S and Hobbs P (1995) The Long-term Viability of ToolBook in CBL Development, UK ToolBook User Conference 1994, University of Bristol

Priest S (1990) Three British Empiricists Hobbes to Ayer, London: Penguin

Provenzo E F, Brett A and G McCloskey (1999) Computers, curriculum and cultural change An introduction for teachers, Mahwah, New Jersey: Lawrence Erlbaum Associates Publishers

Ramadhan H and du Boulay B (1993) *Programming Environments for Novices* in Lemut et al (eds) (1993) Cognitive Models And Intelligent Environments for Learning Programming

Bibliography

Rawlings G (1997) Slaves of the Machine the Quickening of Computer Technology, Cambridge, Massachusetts: MIT Press

Redish K and Smyth W (1986) Program *style analysis: a natural by-product of program compilation*, Communications of the ACM, Volume 29, Issue 2 February 1986 pp126 – 133

Reid I and Rushton J (1985) Teachers, computers and the classroom, Manchester, Manchester University Press

Reimann P and Spada H (1996) (eds) Learning in Humans and Machines Towards an Interdisciplinary Learning Science, Oxford: Pergamon

Reiss S (1998) *Visualization for Software Engineering – Programming Environments* in Stasko et al (eds) (1998) Software Visualization

Rensink, Ronald A (2000) *When Good Observers Go Bad: Change Blindness, Inattentional Blindness, and Visual Experience*, *Psyche* 6(9) Found at http://cogprints.soton.ac.uk/documents/disk0/2000/2000/10/50/index.html 22/02/2001

Repenning A and Perrone C (2000) *Programming by analogous examples* Association for Computing Machinery Communications of the ACM, New York March 2000 Volume 43 Issue 3 pp 90-97

Reynolds J (1981) The Craft of Programming, Englewood Cliffs, New Jersey: Prentice Hall

Richards G (1996) Putting psychology in its place An introduction from a critical historical perspective, London: Routledge

Richardson K (1988) Understanding Psychology, Milton Keynes: Open University Press

Richetti J (1983) Philosophical writing: Locke, Berkeley and Hume, Cambridge, Massachusetts: Harvard University Press

Richey R (1986) The Theoretical and Conceptual Bases of Instructional Design, London: Kogan Page

Ridgway J (1991) *Of Course ICAI is Impossible..., Worse, Though, it Might be Seditious* in Boyd-Barret and Scanlon (1991) (eds) Computers and Learning,

Ridgway J, Benzie D and Burkhardt H (1983) Investigating *CAL?* in CAL 83 Selected papers from the Computer Assisted Learning symposium

Rigney J W (1961) *Potential uses of computers as teaching machines* in Coulson (ed.) (1961) Programmed learning and computer-based instruction Proceedings of the conference on Application of Digital Computers to Automated Instruction October 10 - 12 1961

Bibliography

Rogalski J and Samurçay R (1993) *Task Analysis and Cognitive Model as a Framework to Analyse Environments for Learning Programming* in Lemut et al (eds) (1993) Cognitive Models And Intelligent Environments for Learning Programming

Rogers G A J and Ryan A (1988) Perspectives on Thomas Hobbes, Oxford: Clarendon Press

Romiszowski A (1987) *Artificial intelligence and expert systems in education: Progress, promise and problems* in Australian Journal of Educational Technology, 3(1), 6-24 Found at http://cleo.murdoch.edu.au/gen/aset/ajet/ajet3/wi87p6.html 12/09/2000

Rose E (1999) *Deconstructing Interactivity in Educational Computing* in Educational Technology Volume XXXIX No. 1

Rosin R (1973) *Teaching "About Programming*" in Turski (ed.) (1973) Programming Teaching Techniques

Ross W D (ed.) (1960) The Works of Aristotle translated into English Vol. VII Metaphysica, Oxford: Clarendon Press

Roth I (1990) (ed.) Introduction to Psychology Volume 2, Milton Keynes: Open University Press

Ruehr F and Orr G (2002) Interactive program demonstration as a form of student program assessment, Consortium for Computing Sciences in Colleges

Rushby N (1979) An introduction to Educational Computing, London: Croom Helm

Rushby N (ed.) (1977) Technical report No 16 Computer Managed Learning in the 1980s A future study report, London: CET

Rushby N (ed.) (1981) Selected Readings in Computer Based Learning, London: Kogan Page

Russell B (1998) (Introduction by Skorupski J) The problems of philosophy, Oxford: Oxford University Press

Russell J (1978) The acquisition of knowledge, Hong Kong: Macmillan Press

Sackman H, Erikson W J and Grant E E (1968) *Exploratory experimental studies comparing online nad offline programming performance*, Communications of the ACM, Volume 11, Number 1, January 1968, pp 3 - 11

Sacks O (1989) Seeing Voices, Berkeley: University of California Press

Saettler P (1968) A history of instructional technology, New York: McGraw-Hill

Bibliography

Samurçay R (1989) *The Concept of Variable in Programming: Its Meaning and Use By Novice Programmers* in Soloway and Spohrer (eds) (1989) Studying the Novice Programmer

Sanders S and Ayayee E (1997) *Engaging Learners in Computer Aided Learning: Putting the Horse before the Cart* Found at http://www.curtin.edu.au/conference/ASCILITE97/papers/Sanders/Sanders.html 15/02/2001

Sanger J (1988) *Programming Learners or Algorithm, who could ask for anything more?* in Schostak (ed.) (1988) The Impact of Educational Technology on Schooling

Sanger J, Wilson J, Davies B and Whitakker R (1997) Young Children Videos and Computer Games Issues for Teachers and Parents, London: Falmer Press

Sapolsky R (2003) *Bugs in the brain* Scientific American, March 2003

Sax G (1979) Foundations of Educational Research, Englewood Cliffs, New Jersey: Prentice Hall

Scaife J and Wellington J (1993) Information Technology in Science and Technology Education, Buckingham: Open University Press

Scanlon E and O'Shea T (eds) (1987) Educational Computing, Chichester: John Wiley

Scardamalia M and Bereiter C (1992) Surpassing Ourselves An Inquiry into the nature and implications of expertise, Chicago: Open Court

Schostak J (ed.) (1988) The Impact of Educational Technology on Schooling, London: Methuen

Schuell T J (1990) *Designing Instructional Computing Systems for Meaningful Learning* in Jones and Winne (eds) (1990) Adaptive Learning Environments Foundations and Frontiers

Scott D (1998 a) *Methods and data in educational research* in Scott and Usher (eds) (1998) Understanding Educational Research

Scott D (1998 b) *Ethnography and Education* in Scott and Usher (eds) (1998) Understanding Educational Research

Scott D and Usher R (1998) (eds) Understanding Educational Research, London: Routledge

Scott-Kakures D, Castagnetto S, Benson H, Taschek W and Hurley P (1993) History of philosophy, New York: Harper Perennial

Bibliography

Self J (1985) Microcomputers in Education A Critical Evaluation of Educational Software, Brighton, Harvester Press

Sengler H (1983) *A model of the understanding of a program and its impact on the design of the programming language grade* in Green et al (eds) (1983) The Psychology of Computer Use

Sewell D (1990) New Tools for New Minds A Cognitive Perspective on the Use of Computers with Young Children, New York: Harvester Wheatsheaf

Shaffer D R (1985 2nd Edition) Developmental psychology Childhood and adolescence, Pacific Grove: Brooks Cole Publishing Company

Shapin S (1994) A Social History of Truth Civility and Science in Seventeenth Century England, Chicago and London: The University of Chicago Press

Shaw M (1990) Prospects for Engineering Discipline of Software, IEEE software November 1990 pp 15-24

Sherman R and Webb R (eds) (1988) Qualitative Research in Education: Focus and Methods, London: The Falmer Press

Shermer M (2004) *The Enchanted Glass* in Scientific American, May 2004 p 27

Shields B (1991) *Universities Funding Council Information Technology Training Initiative (ITTI)* in The CTISS File No 12 September 1991

Shimahara N (1988) *Anthroethnography: A Methodological Consideration* in Sherman and Webb (eds) (1988) Qualitative Research in Education: Focus and Methods

Shneiderman B (1980) Software Psychology Human Factors in Computer and Information Systems, Cambridge, Massachusetts: Winthrop Publishers

Shneiderman B (1985) *A Model Programming Environment* in Hartson (ed.) (1985) Advances in Human-Computer Interaction

Sibley M (1985) Computer Assisted Learning, London: Century Communications

Siemer-Matravers J (1999) *Intelligent Tutoring Systems and Learning as a Social Activity* in Educational Technology Volume XXXIX No 5

SIGSCE Bulletin Conference Proceedings The 4th annual SIGSCE/SIGCUE Conference on Innovation and Technology in Computer Science Education ITiCSE'99 June 27 –July 1, 1999, Cracow, Poland

Silverman D (1994) Interpreting Qualitative Data Methods for Analysing Talk, Text and Interaction, London: Sage

Bibliography

Silvers R B (ed.) (1997) Hidden Histories of Science, London: Granta Books

Sime H, Arblaster A and Green T (1977) *Reducing programmer errors in nested conditionals by prescribing a writing procedure* in International Journal of Man-Machine Studies 9 pp 119 – 126

Sinclair B and Bacon R (1997) *Learning with SToMP: flexibility in a complete environment* in CAL 97 International Conference WWW Proceedings

Skinner B F (1953) Science and Human Behaviour, New York: Macmillan

Skinner B F (1962) *The Science of Learning and the Art of Teaching* in Smith and Moore (eds) (1962) Programmed Learning Theory and Research An Enduring Problem in Psychology Selected Readings

Skinner B F (1978) Reflections on behaviorism and society, New Jersey: Prentice Hall

Skyrme D J (1981) *The evolution of graphics in CAL* in Computer Assisted Learning Selected papers from the CAL 81 symposium

Slater J (1996) *The Impact of TLTP* in Active Learning Number 4 July 1996

Sleeman D and Brown J (eds) (1982) Intelligent Tutoring Systems, London: Academic Press

Sleeman D, Baxter J and Kuspa L (1989) *A Summary of Misconceptions of High School Basic Programmers* in Soloway and Spohrer (eds) (1989) Studying the Novice Programmer

Smith F and Miller G A (eds) (1966) The genesis of language A psycholinguistic approach, Cambridge, Massachusetts: MIT Press

Smith H and Green T (eds) (1980) Human Interaction with Computers, London: Academic Press

Smith J (2002) *Learning Styles: Fashion Fad or Lever for Change? The Application of Learning Style Theory to Inclusive Curriculum Development*, Innovations in Education and Teaching International, February 2002, Volume 39 Number 1

Smith P and Webb G (2000) *The Efficacy of a Low-Level Program Visualization Tool for Teaching Programming Concepts to Novice C Programmers* in Journal of Educational Computing Research, Volume 22 No 2, 2000, pp 187-216

Smith PR (ed.) (1981) Computer Assisted Learning Selected papers from the CAL 81 symposium, Oxford: Pergamon Press

Smith PR (ed.) (1983) CAL 83 Selected papers from the Computer Assisted Learning symposium, Oxford: Pergamon Press

Bibliography

Smith P R (ed.) (1985) CAL 85 Advances in computer-assisted learning: selected proceedings from the CAL 85 symposium, Oxford: Pergamon Press

Smith W I and Moore J W (eds) (1962) Programmed learning Theory and Research An enduring problem in psychology Selected Readings, Princeton, New Jersey: D Van Nostrand

Solomon C (1986) Computer Environments for Children A reflection on Theories of Learning and Education, Cambridge, Massachusetts: MIT Press

Soloway E and Spohrer J (eds) (1989) Studying the Novice Programmer, Hillsdale, New Jersey: Erlbaum Associates

Somekh B and Davis N (eds) (1997) Using Information Technology Effectively in Teaching and Learning Studies in pre-service and in-service Teacher Education, London: Routledge

Sorley W R (1920) A History of English Philosophy, Cambridge: Cambridge University Press

Spohrer J and Soloway E (1989) *Novice Mistakes: Are the Folk Wisdoms Correct?* in Soloway and Spohrer (eds) (1989) Studying the Novice Programmer

Spooner C (1986) *The ML Approach to the Readable All-Purpose Language* in ACM Transactions on Programming Languages and Systems Volume 3 No 2 April 1986

Squires D (1999) Educational Software for Constructive Learning Environments: Subversive Use and Volatile Design in Educational Technology Volume XXXIX No. 3

St Amant R, Lieberman H, Potter R and Zettlemayer L (2000) *Visual generalization in programming by example* in Association for Computing Machinery Communications of the ACM New York March 2000 Volume 43 Issue 3 pp107-114

Stainback S and Stainback W (1988) Understanding and Conducting Qualitative Research, Dubuque, Iowa: Kendall Hunt

Stansifer R (1995) The study of programming languages, Englewood Cliffs, New Jersey: Prentice-Hall

Stasko J and Lawrence A (1998) *Empirically Assessing Algorithm Animations as Learning Aids* in Stasko et al (eds) (1998) Software Visualization

Stasko J, Domingue J, Brown M and Price B (eds) (1998) Software Visualization Programming as a Multimedia Experience, Cambridge, Massachusetts: MIT Press

Steffe L P and Gale J (eds) (1994) Constructivism in Education, Hillsdale, New Jersey: Erlebaum

Bibliography

Steier F (1994) *From Universing to Conversing: An Ecological Constructionist Approach to Learning and Multiple Description* in Steffe and Gale (eds) (1994) Constructivism in Education

Stephenson J (ed.) (2001) Teaching and learning Online Pedagogies for New Technologies, London: Kogan Page

Sternberg J (ed.) (1994) Thinking and Problem solving, San Diego: Academic Press

Sternberg R J (1996) Cognitive Psychology, Fort Worth: Harcourt Brace College Publishers

Stillings N, Weisler S, Chase C, Feinstein M, Garfield J and Rissland E (1995) Cognitive Science An introduction Cambridge, Massachusetts: Harvard Press

Stokes R J S (1994) Computer Assisted Learning in Engineering at the University of Humberside, Open Learning Foundation

Stolurow L M and Davis D (1965) *Teaching Machines and Computer-Based Systems* in Glaser (1965) Teaching Machines and Programed Learning II Data and Directions

Stone L (1977) The Family, Sex and Marriage, 1500 - 1800, London: Weidenfeld and Nicholson

Straker A (1989) Children using computers, Oxford: Blackwell

Satratzemi M, Dagdilelis V and Evangelidis G (2001) *Telemachus: A system for the submission and assessment of students programs*, *Learning Technology*, Volume 3 Issue 3 July 2001, IEEE Computer Society

Stronach I and MacLure M (1997) Educational Research Undone The Post Modern Embrace, Buckingham: OU Press

Suppes P Jerman M and Dow B (1968) Computer Assisted Instruction Stanford's 1965-1966 Arithmetic Program, New York: Academic Press

Sutcliffe F E (ed.) (1968) Discourse on Method and The Meditations, London: Penguin Books

Tansey P J (ed.) (1971) Educational Aspects of Simulation, London: McGraw-Hill

Tawney D (ed.) (1979 a) Learning through computers An introduction to Computer Assisted Learning in Engineering, Mathematics and the Sciences at tertiary level, London: Macmillan

Tawney D (1979 b*) CAL in different contexts* in Learning through computers An introduction to Computer Assisted Learning in Engineering, Mathematics and the Sciences at tertiary level

Bibliography

Tawney D (1979 c) *CAL and learning* in Learning through computers An introduction to Computer Assisted Learning in Engineering, Mathematics and the Sciences at tertiary level

Taylor A E (1970) Thomas Hobbes, Port Washington: Kennikat Press

Taylor J and Walford R (1978) Learning and the Simulation Game, Milton Keynes: Open University Press

Taylor M (1997) *Learning Styles* in Inquiry, Volume 1, Number 1, Spring 1997, 45-48 1997 http://www.br.cc.va.us/vcca/i11tayl.html 09/12/2000

Teasley B (1993) *Program Comprehension Skills and Their Acquisition: A Call for an Ecological Paradigm* in Lemut et al (eds) (1993) Cognitive Models And Intelligent Environments for Learning Programming

Tennyson R (ed.) (1992) Automating Instructional Design, Development and Delivery, Berlin: Springer-Verlag

Thagard P (1996) Mind Introduction to cognitive science, Cambridge, Massachusetts: MIT Press

The Proceedings of the Loughborough Programmed Learning Conference April 1966, London: Methuen

Thomas J E (1980) Musings on The Meno A new translation with commentary, The Hague: Martinus Nijhoff

Thomas P and Paine C (2000) *Tools for Observing Study Behaviour* in Blackwell and Bilotta (eds) (2000) 12th Workshop of the Psychology of Programming Interest Group

Thompson J (1996) *Embedding technology into language examinations: a case study* in Active Learning Number 4 July 1996

Thornbury H, Elder M, Crowe D, Bennett P and Belton V (1996) *Suggestions for successful integration* in Active Learning Number 4 July 1996

Thorndike E (1931) Human Learning, Cambridge, Massachusetts: MIT Press

TLTP (1994) Newsletter No 1 July 1994, TLTP

TLTP (1994) Newsletter No 2 Autumn 1994, TLTP

TLTP (1998) Newsletter No 11 Summer 1998, TLTP

TLTP A report on a workshop on the implementation and distribution of courseware across a university campus, TLTP

TLTP Institutional case studies, TLTP

Bibliography

TLTP Science case studies, TLTP

TLTP What has been achieved? CTI

Tuck R (1989) Hobbes, Oxford: Oxford University Press

Tuckman B (1978) Conducting Educational Research, New York: Harcourt Brace Jovanovich Inc

Tufte E (1997) Visual Explanations Images and Quantities, Evidence and Narrative, Cheshire, Connecticut: Graphics Press

Turner J (1975) Cognitive development, London: Methuen and Co

Turpin S (ed.) (1995) Catalogue Phase 1 Spring 1995, TLTP

Turpin S (ed.) (1996) Catalogue Phase 2 Spring 1996, TLTP

Turpin S (1997) Association for Learning Technology Newsletter No 18 July 1997

Turski W (ed.) (1973) Programming Teaching Techniques Proceedings of the IFIP TC-2 Working Conference on Programming Teaching Techniques Zakopane, Poland September 18-22, 1972, Amsterdam: North-Holland Publishing Company

UNCAL (1975) The Programme at Two An UNCAL Companion to Two Years On University of East Anglia: Centre for Applied Research

Underwood J D M and Underwood G (1990) Computers and Learning Helping children acquire thinking skills, Oxford: Blackwell

Unwin D and Leedham J (eds) (1967) Aspects of Educational Technology, London: Methuen

Urban-Lurain M (1997) *An Instructional Theory for Introductory Computer Science for Non-Computer Science Students* Found at http://aral.cps.msu.edu/CPS101SS99/CPS101Visitor/InstructionalTheory.htm

Usher R (1998 a) *A critique of the neglected epistemological assumptions of educational research* in Scott and Usher (eds) (1998) Understanding Educational Research

Usher R (1998 b) *Textuality and reflexivity in educational research* in Scott and Usher (eds) (1998) Understanding Educational Research

Van Dalen D (1979) Understanding Educational Research An Introduction, New York: McGraw-Hill

van de Veer G and van de Wolde G (1983) *Individual differences and aspects of control flow notations* in Green et al. (eds) (1983) The Psychology of Computer Use

Bibliography

van de Veer G (1993) *Mental Representations of Computer Languages – A Lesson from Practice* in Lemut et al (eds) (1993) Cognitive Models And Intelligent Environments for Learning Programming

Van Lehn (1988) *Student modelling components of ITS* in Polson and Richardson (1988) Foundations of Intelligent Tutoring Systems

VanLengen C and Maddux C (1990) *Does Instruction in Computer Programming Improve Problem Solving Ability?* in Journal of Information Systems 12/90 Found at http://www.gise.org/JISE/Vol1-5/DOESINST.htm 24/04/2001

Vasta R (ed.) (1992) Six theories of child development Revised formulations and current issues, London: Jessica Kingsley Publications

Venables A and Haywood L (2001) *Programming students NEED instant feedback!*, 5th Australasian Computing Education Conference, Adelaide, Australia

Verma G and Beard R (1981) What is educational research? Perspectives on the Techniques of Research, Aldershot, Hants: Gower Publishing Company

Verma G and Mallick K (1999) Researching Education Perspectives and Techniques, London: Falmer Press

Vygotsky L S with Cole M, John - Steiner V, Scribner S, Souberman E (eds) (1978) Mind in Society The Development of Higher Psychological Processes,

Walford G (ed.) (1994) Researching the Powerful in Education, London: UCL Press Limited

Wartenburg T E (1992) *Reason and the practice of science* in Guyer (ed.) (1992) The Cambridge Companion to Kant

Watkins M, Glutting J and Youngstrom E (2002) *Cross-Battery Cognitive Assessment: Still Concerned* Found at http://www.nasponline.org/publications/cq312cbattery.html 24/04/2003

Watson D (1987) Developing CAL: Computers in the Curriculum, London: Harper and Row

Watson G (1966) The Stoic theory of knowledge, Belfast: Queen's University

Watson J (2003) *Egg shells out £12m in development restyle*, Computing, 13 November 2003

Watson R I and Evans R B (1991 5th Edition) The Great Psychologists A History of Psychological Thought, London: Harper Collins

Webb G (1996) *Becoming Critical of Action Research for Development* in Zuber-Skerritt (ed.) (1996) New Directions in Action Research

Bibliography

Webster R (1996) Why Freud was wrong, New York: Harper Collins

Weinberg G (1971) Psychology of Computer Programming, New York: Van Nostrand Reinhold

Weizenbaum J (1984) Computer Power and Human Reason From Judgment to Calculation, Middlesex, England: Pelican Books

Wellington J J (1996) Methods and Issues in Educational Research, Sheffield: University of Sheffield Division of Education

Wells H K (1960) Sigmund Freud a Pavlovian critique, London: Lawrence Wishart

Welsh T (1999) *Implications of Distributed Learning for Instructive Designs: How Will the Future Affect the Practice?* in Educational Technology Volume XXXIX No. 2

Wender K F, Schmalhofer F and Bocker H (eds) (1995) Cognition and Computer Programming, New Jersey: Ablex Publishing Corporation

Wertheimer M (1987 3rd Edition) A brief history of psychology, Fort Worth: Harcourt Brace and Jovanovich

Wertsch J V (ed.) (1995) Culture Communication and Cognition: Vygotskyian Perspectives, Cambridge: Cambridge University Press

Wertsch J V and Stone C A (1995) *The concept of internalization in Vygotsky's account of the genesis of higher mental functions* in Wertsch (ed.) (1995) Culture Communication and Cognition

White B and Watts J C (1973) Experience and environment Major influences on the development of the young child, Volume I, New Jersey: Prentice Hall

Wiedenbeck S and Fix V (1993) *Characteristics of the mental representations of novice and expert programmers: an empirical study* in International Journal of Man Machine Studies (1993) 39 pp 793 – 812

Wilkes M V (1995) Computing Perspectives, San Francisco, California: Morgan Kaufman

Wilkinson A (ed.) (1983) Classroom Computers and Cognitive Science, New York: Academic Press

Williams L and Kessler R (2000) *All I really need to know about pair programming I learned in Kindergarten* Association for Computing Machinery Communications of the ACM, New York: May 2000 Volume 43 Issue 5 pp108-114

Willig C (2001) Introducing Qualitative Research in Psychology Adventures in Theory and Method, Buckingham: Open University Press

Bibliography

Wilson B (1999) *Evolution of Learning Technologies: From Instructional Design to Performance Support to Network Systems* in Educational Technology Volume XXXIX No. 2

Winslow L (1996) *Programming Pedagogy – A Psychological Overview* in SIGSCE Bulletin Volume 28 No 3 Sept 1996

Winter R (1996) *Some Principles and Procedures for the Conduct of Action Research* in Zuber-Skerritt (ed.)(1996) New Directions in Action Research

Wittich W A and C F (1973) Instructional Technology Its Nature and Use, New York: Harper and Row

Wolcott H (1990) *On Seeking – and Rejecting – Validity in Qualitative Research* in Eisner E and Peshkin A (eds) (1990)

Wood A (1979) *CAL in biology* in Learning through computers An introduction to Computer Assisted Learning in Engineering, Mathematics and the Sciences at tertiary level

Wood D L (1985) *Designing microcomputer programs for disabled students* in CAL 85 Advances in computer-assisted learning: selected proceedings from the CAL 85 symposium

Wood D, Bruner J S and Ross G (1976) *The Role of Tutoring in Problem Solving* in Journal of Child Psychology and Psychiatry Volume 17 1976 pp 89 - 100

Wood T (1994) *From Alternative Epistemologies to Practice in Education: Rethinking What it Means to Teach and Learn* in Steffe and Gale (eds) (1994) Constructivism in Education

Woolf B P (1990) *Towards a computational model of Tutoring* in Jones and Winne (eds) (1990) Adaptive Learning Environments Foundations and Frontiers

Woolgar S (1988) (ed.) Knowledge and Reflexivity New Frontiers in the Sociology of Knowledge, London: SAGE Publications

Woolgar S (1988) *Reflexivity is the Ethnographer of the* Text in Woolgar (1988) (ed.)

Woolhouse R (1971) Locke's philosophy of Science and Knowledge A consideration of some aspects of An Essay Concerning Human Understanding, Oxford: Blackwell

Woolhouse R (ed.) (1988) Principles of Human Knowledge/Three Dialogues, London: Penguin

Wright J P (1983) The Sceptical Realism of David Hume Manchester: Manchester University Press

Bibliography

Wright T (1999) *Towards a Method for Evaluating Computer-Assisted Learning Software* Found at http://www.fed.qut.edu.au/projects/asera/asera99_abstracts.htm#Authors%20S-Z 21/04/2004

Wu Q and Anderson J (1993) *Strategy choice and change in programming* in International Journal of Man-Machine Studies (1993) 39 pp 579-598

Yazdani M *Artificial intelligence, Powerful Ideas and Children's Learning* http://www.media.uwe.ac.uk/masoud/author/ideas.htm 24/08/2000

Yolton J W (ed.) (1974) An Essay concerning Human Understanding Volume I, London: Dent

Yolton J W (ed.) (1974) An Essay concerning Human Understanding Volume II, London: Dent

Yourdon E (1976) *How to be a superprogrammer*, Infosystems, February 1976

Zabeeh F (1960) Hume Precursor of modern empiricism An analysis of his opinions on Meaning, Metaphysics, Logic and Mathematics, The Hague: Martinus Nijhoff

Zuber - Skerritt O (ed.) (1996) New Directions in Action Research, London: Falmer Press

Zusne L (1975) Names in the history of psychology A biographical sourcebook, Washington: Hemisphere

Index

A

accidental sampling 40
action research 21 – 27
Ada 149
Ada Programming Support Environment *See APSE*
affective issues in programming 127
AI 134, 190, 104 – 105
Alexander, C 19, 58–61, 94, 242, 244
algorithm animation 138, 140, 141 – 142
algorithmic toolkit 73
algorithms 74
ALICE *See syntax editor*
Alvey Committee 354
anthropology 311
APSE 149
aptitude tests 102 – 103
artificial intelligence *See AI*
associationism 157
ASSYST 266
author 7
  background 17
  own CAL 218
automated grading of student programs 263 – 270
axiomatic semantics 133

B

Bacon, F 295
BALSA 141
BASIC 80
behaviourism 157 – 166
benign introspection 51
benign reflection 52
Bergin, J 62, 63
Bruner, J 169 – 171
big Q 30 – 33

Index

C

CAI 178
CAL 8, 10, 11, 12, 146, 196 – 201, 286
  by the author 55 – 58 , 218 – 221
  by the School of Engineering 202 – 213
  cost of 349
  drill and practice 185
  for collaborative and active learning 196 – 201
  intelligent 190 – 193
  tutorial 53, 57, 178, 189
  types of 178 – 184
CAN 8, 205, 286
CASE 142
CBCA 101
CBT 178
Ceilidh 264
change blindness 110 – 114
CMC 198
CMI 178
COBOL 81
code layout 142
cognitive psychology 104, 166 – 167
cohort studies 41
Controlled Area Network *See CAN*
Computer Aided Instruction *See CAI*
Computer Aided Learning *See CAL*
Computer Aided Software Engineering. *See CASE*
Computer Assisted Learning *See CAL*
Computer Environment for Interactive Learning in Diverse Habitats. *See Ceilidh*
Computer Based Training *See CBT*
Computer Programming for All *See CP4E*
Computer Managed Instruction *See CMI*
Computer Mediated Communication. *See CMC*
Computers in Teaching Initiative. *See CTI*
computing curriculum 60 – 68
conditioned reinforcers 164
confirmatory bias 114 – 115
connectionism 158
constructionism 175 – 176
constructivism 171 – 175
correlational research 33
CP4E 137
Cross-Battery Cognitive Assessment. *See CBCA*
criterion referencing 261 – 264
CTI 355 – 360

Index

D

data gathering 249
Dearing Report 358
De Bono, E 101
debugging 123 – 131
Department of Engineering 8
dependent variables 44
design patterns 62
designing new programming languages 135
diaries 38
Dijkstra, E 71
drill and practice CAL 185
Dynalab 138

E

Economic and Social Research Council *See ESRC*
Educational paradigms 180 – 181
  instructional 180 – 181
  revelatory 180 – 181
  emancipatory 180 – 181
  conjectural 180 – 181
educational research 29
  methodologies 20 – 27
  tools and methods 27 – 47
  value of 47 – 49
ego–less programming 129
epistemology 156
ESRC 356
ethnography 53, 300, 308, 311 – 312
ethnomethodology 311
Euphoria 136
evaluating student programs 259 – 270
ex post facto 33
exit questionnaire 280
expert programmers 97, 106, 124,125, 322 – 330
external validity 42
  threats to 43
extraneous factors 46
extreme programming 153

Index

F

first programming language 79 – 92
focus groups 38, 275
formalism 37
FORTRAN 81
FORTRAN 77 264
FPL 80

G

Games Computing degree 271
games 187 – 189
Gardner, H 118 – 119
gate, the 59
goals of the research effort 17
goto 71, 72
GRAIL 84, 135
Gregorc, A F 119–121

H

Hawthorne effect 43
hermeneutics 309
Higher Education Funding Council 9
higher education 8
Hillside group 246
human aspects of learning to program 96
Humberside Polytechnic 8

Index

I

ICAI 190 – 193
IconAuthor 286
Idols of the Tribe 295
ILT 366 – 367
inattention blindness 110 – 114
independent variables 54
Information Technology Training Initiative *See ITTI*
Institute for Learning and Teaching, *See ILT*
Intelligent Computer Assisted Instruction. *See ICAI*
Intelligent Tutoring System. *See ITS*
intelligences 118
interactive program demonstration 267
internal validity 42, 46
introductory programming 13
introspection 52
ISO9000–3 5
ITS 134,150
ITTI 356

J

Java 139
Jeliot 139

K

Keirsey Temperament and Character tests 100
Kolb, D 119
Kruskal Wallis 251

L

Language List 79
Law of Effect 164
learning styles 117 – 121
learning to program 1, 65, 220
Lens 138
Lewin, K 23
Likert Scale 251
literate programming 143 – 146
little q 30 – 33
LOGO 7, 82, 87 – 89, 128
longitudinal studies 41

Index

M

Mars Climate orbiter 3
MBTI 100, 101
Memento pattern 244
MENO-II 134
mental models 103, 107–108
MEP 353 – 355
metaphor and analogy 115 – 116
Microelectronics Project *See MEP*
microworlds 193 – 195
ML 136
modelling 187 – 189
modes of presentation 170
    enactive 170
    iconic 170
    symbolic 170
modularity 75 – 76
MPE 149
MSc 7, 8, 283
Myers-Briggs Type Indicator *See MBTI*

N

National Council for Educational Technology. *See NCET*
National Development Programme in CAL. *See NDPCAL*
NCET 342 – 347
    Work up to 1969 342 – 346
    1969 – 1973 346 – 347
NDPCAL 347–352
novice programmers 97, 124,125,130, 221, 238, 299, 322 – 330

O

observation 253 – 256, 276
ontology 156
OOPSLA87 62, 243
operant conditioning 164
Open University 348
outline of author's research 55

Index

P

pair programming 154
Papert, S 7, 88
parameters 75–76
parsing 327
participant observation 36, 37
Pascal 7, 8, 90 – 92, 283, 315 – 321, 434
Pascal Help system 223 – 226
pattern elements 63
pattern languages 88
Pattern Languages in Programming *See PLoP*
pattern syntax 63
pattern template 248, 314
patterns and software 61
patterns 88, 242, 245
Pavlov 159
PBD 151 – 152
PBE 152
pedagogical patterns 62, 93, 94
Perl script 423, 424, 426 – 429
personality tests 99 – 101
PGCE 7, 282
phenomenography 308, 310
phenomenology 308, 310
PL/I 81
PLATO 337 – 340
PLoP 243
positivism 308 – 309
Postgraduate Certificate in Education *See PGCE*
pragmatism 37
pretests 43
probability sampling 39
procedure 74
program visualisation 137, 139
programmed instruction 165
Programmed Logic for Automatic Teaching Operation *See PLATO*
programmer profiles 259
Programming by Demonstration *See PBD*
Programming by Example *See PBE*
programming concepts 68
  curriculum 66 – 68
  environments 148
  semantics 132
  syllabus 68
programming to learn 65
Prolog 17, 82, 86, 139, 284
psychology of programming 96
Python 137

Index

Q

QA 5
qualitative research tools 34
qualitative versus qualitative research 27 – 28
Quality Assurance *See QA*
questionnaires 34, 250 – 254, 447 – 457
questionnaires on CAL use 213 – 218

R

random sampling 39
reading 13
reflexivity
    concept of 53
    constitutive 51
    disengaged 52
    epistemological 53
    role of language 54
repetition 74
research focus 18,19
research questions 13, 290
sampling strategies 39

Index

S

scaffolding 171
School of Engineering 201
SDE 133
selection 73
semantics 323
semi–structured interviews 35
sensitisation 45
sequence 73
Sheffield University 282
Skinner, B.F 162
SLE 268, 442
SmallTalk 61, 83
Specialised Learning Environment *See SLE*
software 1
    crisis 4
    development 3
    engineering 4, 68
    metrics 260
    program visualisation 137
    Quality Assurance *See SQA*
    visualisation and teaching programming 146 – 148
    visualisation systems 142
SQA 5
stages in action research 26 – 27
Stanford project 311 – 337
statistical regression 47
Stroop effect 112
structured programming 2, 69, 70, 71
submit 265
suggestions for the teaching of programming 302
survey methods 33
syntax 323
syntax editor 132

Index

T

taxonomy for software visualisation 140
Teaching and Learning Technology Programme. *See TLTP*
teaching of programming 92
teaching programming using software 131
TekniCAL 202
Telemachus 266
Thorndike, E 157
timeless way 59
Time-shared Interactive Computer Controlled Television. *See TICCIT*
TICCIT 337 – 340
TLTP 300, 360 – 365
Total Quality Management *See TQM*
TQM 5
triangulation 39
Turing language 136
tutorial CAL 53, 57, 178, 189
types of CAL 178 – 184
typography 143

U

University of Lincoln 256 – 258
UNCAL 349
Understanding Computer Assisted Learning *See UNCAL*

V

variables 75
videoconferencing 197
vision *See visual cognition*
visual cognition 109 – 110
Vygotsky, L 167 – 169

W

Web-based Center for Automated Testing *See Web-CAT Grader*
Web-CAT Grader 268
Whorf, B 91

X

XP *See extreme programming*

Index

Z

Zone of Proximal Development *See ZPD*
ZPD 168