Event Driven Languages for Novice Programmers

Peter Shackleton
Department of Information Systems
Faculty of Business
Victoria University of Technology
1998

A thesis submitted in fulfilment of the requirements for the degree of Master of Business by Research (Business Computing) from Victoria University of Technology

Declaration

I certify that this thesis contains no material that has been submitted for the award of any other degree or diploma in any institute, college or university, and that it contains no material previously published or written by another person, except where due reference is made in the rest of the thesis.

[redacted]

Peter Shackleton

4th August, 1998

Date

Acknowledgements

Much of the work of this thesis could not have been done without the support and encouragement of a number of people.

Firstly, I would like to thank my supervisor Nick Billington for his support, guidance and, above all, his patience throughout the writing process. I would also like to thank my secondary supervisor Angela Scollary for the many hours she spent editing my work on her long trips to Wagga and for her invaluable advice.

My sincere thanks go to the many students and lecturers who gave freely of their time and who contributed so much rich data for this study.

To my colleagues in the Department of Information Systems at Victoria University I am also indebted.

Finally, I would like to thank my wife Kathy who has sacrificed far more than me during this long process and who has provided me with so much support and encouragement.

Summary

Most mature scientific disciplines have a sound and widely accepted foundation of basic concepts. This is not yet the case for the discipline of Information Systems. It is acknowledged that the majority of students will work in organisational settings where they will specify, develop, build, implement or manage information systems. Although there is consensus on the need for information systems education, the structure and content of courses of this type in each of Victoria's eight universities varies considerably. Computer programming, in different formats, is a part of each of these courses although the content and scope varies between universities.

This thesis investigates the use of event driven languages by novice programmers in information system courses. To establish the context for the study an extended literature analysis examines how novice programmers acquire programming skills. The current use of programming languages and program design methodologies in introductory programming subjects is also examined. The initial research for the thesis involves an investigation of the use of event driven languages at three Australian universities. Extended interviews identify the reasons why the event driven paradigm was adopted or rejected in these universities, and the advantages and disadvantages of its use in an introductory subject at one university. Changes to the method of teaching, and the aims and objectives of the introductory subject, together with the impact on the student learning from the use of object-based, visual programming environments is discussed.

A seven step program design methodology is developed for the teaching of event-driven languages to novice programmers. Using a case study approach, qualitative data is gathered on the use of the design methodology by students at an Australian university. Evaluation of the methodology indicates that students are heavily reliant upon the screen layout but are confused about events and the placement of code. It is found that students often fail to establish a clear mental model that depicts the operation of event driven systems. Reasons for this confusion and the strategies that can be developed to help novices gain a better understanding of event driven systems is discussed.

Table of Contents

List of Figures

List of Tables

Glossary of Acronyms

ACM	Association of Computer Machinery (United States of America)
ACS	Australian Computer Society
CAE	College of Advanced Education
CS	Computer Science
DEET	Department of Employment, Education and Training
IS	Information Systems
IT	Information Technology
TAFE	Technical and Further Education
VCE	Victorian Certificate of Education
VUT	Victoria University of Technology

Chapter 1 - The Problem and its Significance

1.1 Introduction

Australia was one of the first countries to establish courses in Business Computing, later to be known as Information Systems, when the Commonwealth Government required computing professionals for its growing administrative needs in the 1960's. Originally established in Colleges of Advanced Education (CAE), they eventually spread to all Victorian universities. As would be expected, the curriculum content of these courses has varied over the years in response to the changes in technology and the growing needs of business for information systems graduates. Programming, in some form and in varying degrees, has been an integral part of Information Systems courses throughout the years. Initially, procedural languages using structured programming techniques were the only choice of language but, in recent times, universities and colleges have been able to consider other languages from the object oriented and event-driven platform for their introductory programming subjects.

A change to a new programming paradigm, however, often necessitates changes beyond the subject content. Structured programming techniques is the popular design method for procedural languages such as Pascal, while object techniques can be adopted for object-oriented languages such as Smalltalk. Event-driven programming languages, although extremely popular in business, have not been used extensively as the platform in introductory programming subjects in information systems courses. Moreover, there is no recognised program design methodology for event-driven systems amongst systems developers and programmers nor is there a methodology for the teaching of these types of languages.

This study will, firstly, review the existing literature and, secondly, research the experiences of lecturers and students using event-driven programming languages in introductory programming subjects in Information Systems courses. It will examine the experiences of a number of lecturers from Australian universities where event-driven programming has been adopted. It will then explore the experiences of students who used a new program design methodology to develop event-driven programs.

Chapter 2 is the first of two chapters comprising an extended review of the literature. It investigates a range of programming tasks in the procedural paradigm and the current understanding of how novice programmers learn to program. Models of programmer behaviour and issues related to cognitive science, skills and processes are discussed. New visual programming environments and alternative programming paradigms are investigated, and changes to the programming tasks, program design and programming skills are discussed.

Chapter 3 reviews the literature on the area of teaching with object-oriented and event-driven programming. A framework for teaching and learning to program is established. The impact of object-based languages on teaching programming in introductory subjects in information systems courses is investigated. The small amount of literature available on event-driven programming in programming subjects is reviewed.

Chapter 4 explains the research rationale and methodology used in undertaking the study. The case study for the investigation of a new program design methodology is detailed. Each of the data collection methods is explained and the limitations of the methodology noted.

Chapter 5 outlines the findings from interviews with experts from universities who have adopted the event-driven paradigm into introductory programming subjects. Benefits and problems of the language are investigated in the light of the programming skills discussed in the literature. A clear need for a program design methodology is established and data from a pilot study is investigated. The chapter concludes by outlining a new program design methodology.

Chapter 6 analyses the results from the use of the program design methodology in an undergraduate and postgraduate introductory programming subject.

Finally, chapter 7 of the thesis reflects on the whole study by highlighting relevant issues that have arisen and pointing to future areas of research.

1.2 The Research Questions

In investigating the use of event-driven languages in introductory programming subjects a number of questions are considered in this thesis:

1. How do students in introductory programming subjects learn to program?
2. What changes need to be made to introductory programming subjects to accommodate event-driven programming platforms?
3. Can a program design methodology be effective for the learning of event-driven programming?

1.3 Information Systems as a Discipline

Over a period of 40 years, the Information Systems discipline has become an essential part of business and government organisations in Australia. As it has become more sophisticated the Information Systems discipline has moved from processing data to providing an information infrastructure with applications aligned to an organisational strategy (Longenecker, Feinstein et al., 1995: 4).

Although it would not be possible to obtain universal agreement on any one definition, information systems is generally associated with a broad group of subjects and methodologies in the information technology discipline. Figure 1.1, extracted from the Australian Computer Society (ACS) submission to Department of Employment, Education and Training (1992: 13), attempts to describe the boundaries of disciplines within computing.

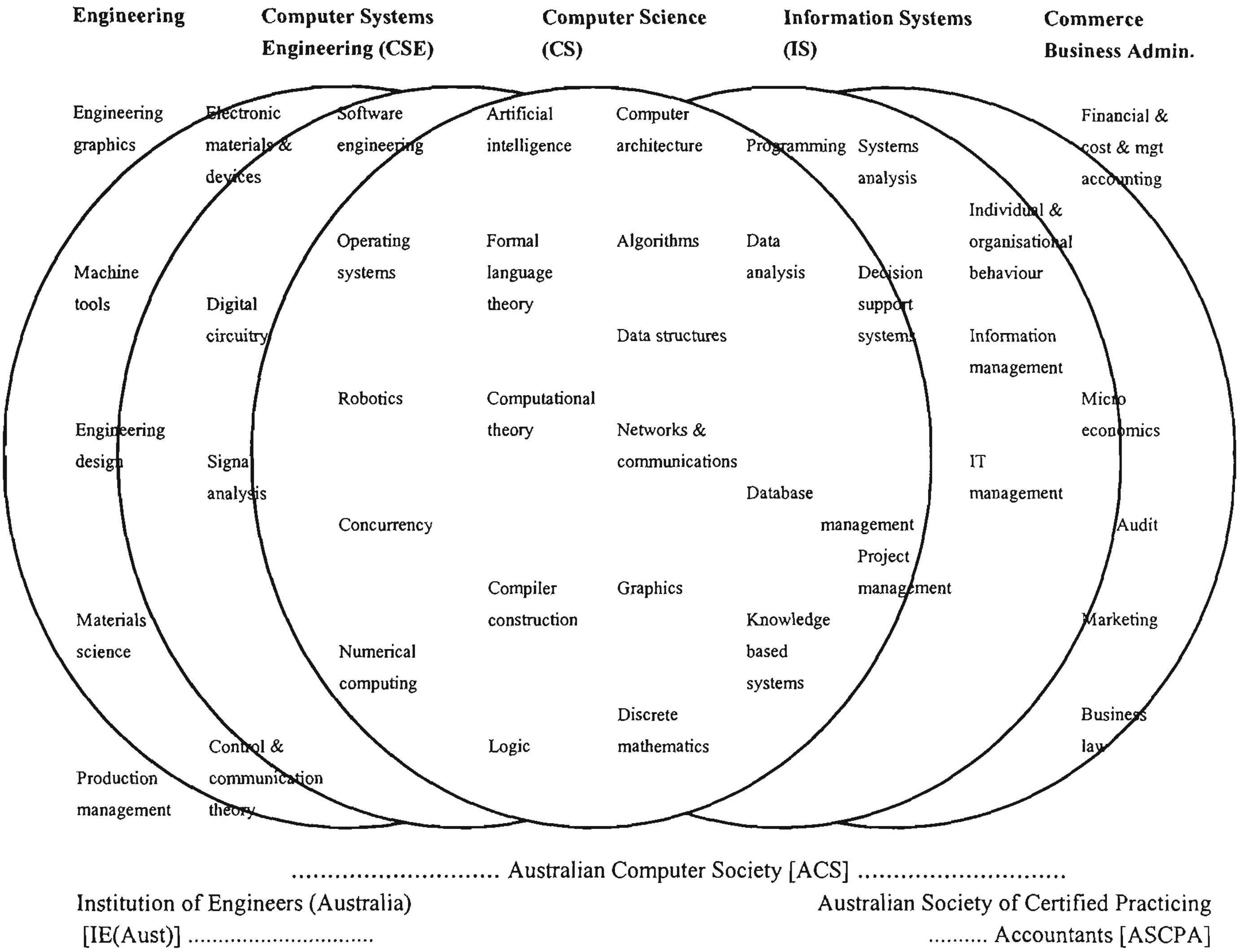


Figure 1.1: Scope of the Computing Studies and Information Science Disciplines

NB: The topic areas are indicative only and are not exhaustive. The orientation dimension, comprising basic research, applied research, product development and application areas (such as commerce, industry, government, libraries, land information and health) is orthogonal to this disciplinary dimension. Software Engineering is included as a subject area rather than a discipline.

Source: DEET et al, 1992: Figure 2.2

It is immediately apparent that there is considerable overlap between the various curriculum areas and the interests of the three professional bodies. Despite this overlap, there are important differences in the nature of the work performed, the types of systems developed and managed, and the application of the technology (Longenecker, Feinstein et al., 1995: 9). Information Systems is a discipline which is oriented towards business or commerce, that is, it involves matching information systems requirements to an organisation's objectives. Software engineering incorporates the principles of large-scale software systems, whereas the context for information systems is usually in smaller organisations. As a consequence, Information Systems courses are

> *'designed to prepare or develop further the abilities of students to analyse information needs, functions, operations, procedures, physical systems and technical problems of an organisation to establish the feasibility of, and develop procedures for, automatically processing data by use of computers and communications networks' (DEET, 1992: 16).*

Computer science, on the other hand, concentrates on algorithmic processes and system software, while computer systems engineering involves the design and development of computer and communication components and equipment, and computer-based systems.

Figure 1.2 further highlights the differences between the three areas of computer science engineering, computer science and information systems. The movement along the horizontal axis ('hard' to 'soft') represents a spectrum from hardware oriented to more human involvement in the development of applications. Given the growth in recent years of computer use and, in particular end-user computing, it is understandable how information systems has changed during that period of time.

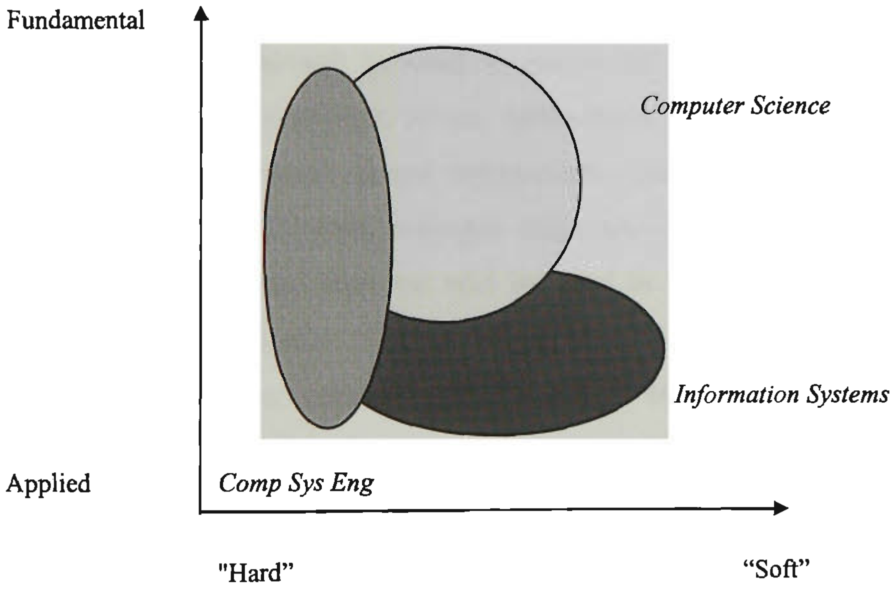


Figure 1.2: Information Technology - Principal Subject Clusters

Source: DEET et al, 1992: Figure 2.1

Information systems covers two broad areas: the acquisition, deployment, and management of information resources and services, and secondly, the development and augmentation of infrastructure and systems for information use in organisation processes (Longenecker, Feinstein et al., 1995: 9). The *information systems function* involves the deployment, implementation and management of information resources and services (such as computer hardware and communications), data and organisation-wide systems. This function includes the responsibility to monitor new information technology (IT) and to assist in incorporating IT into the organisation's strategy, planning and practices. The *systems development* component of information systems involves the systematic application of techniques and methodologies to construct systems used for a range of organisational tasks such as data acquisition and storage, communication, data analysis and decision support. Issues of innovation, human-machine systems and user interfaces are often relevant when creating information systems.

The nature and context of information system functions, however, has changed over the years as information technology has been moved into most functional areas of organisations. Today, the function of information systems is often one of supporting innovation, planning and coordinating resources and systems, rather than one where a specialist or specialists are doing all the work in isolation. Moreover, systems development concentrates heavily on individual and departmental application development in addition to the traditional role of supporting systems. This has required Information Systems curricula to train both specialists and to provide students from other disciplines with a knowledge of information systems functions and systems development.

Information systems will be used to *denote all operations and procedures involved in the design and implementation of an information processing system.* The term implies the integration of computer based components and manual operations to enable data to be collected, organised, saved, managed and retrieved as useful information (DEET, 1992: 16). Similarly, Information Systems will be used to denote a field of academic study in which curricula are *designed primarily to educate people in the efficient and effective application of computer hardware, software and systems to the solution of business and organisational problems* (Tatnall, 1993: 4).

1.4 Information Systems Curricula in Australian Universities

In response to business demand for graduates with an Information Systems qualification, Information Systems courses have gradually been introduced into Australian universities over the years. In Victoria as in other states, some of the newer universities, such as the Royal Melbourne Institute of Technology and the Victoria University of Technology offered Information Systems courses when they were Colleges of Advanced Education. Some established universities, such as Monash University adopted these courses when they amalgamated with smaller CAE's. Victoria's oldest university, the University of Melbourne, was the last to establish an undergraduate program in Information Systems[1]. Today, each Australian university offers a range of computing courses including Information Systems.

There has been significant growth, particularly in recent years, in the number of specialised undergraduate courses that lead to a formal qualification in information technology. These courses account for almost 70% of the number of undergraduate courses in the information technology field DEET (1992). At the same time, however, there has developed a greater flexibility in existing undergraduate programs, permitting new combinations of study leading to an increase in non-computing courses, that is, those with a computing content of at least 30% within a non-computing field of study. In 1991, 36% of information technology students were enrolled in courses of this type. These statistics reflect the extent to which Information Systems has permeated many other disciplines and the demand for graduates with some Information Systems expertise.

It is not surprising then that a large number of new computing professionals in Australia are qualifying through Information Systems courses. The most recent statistics from DEET (1992: 43) indicate that 2200 students are enrolled in computing courses at Australian universities that contain enough subject material to qualify graduates for Level 1 membership of the Australian Computer Society[2]. At least 80% of these students are enrolled in Information Systems courses (Tatnall, 1993: 4). Today, there is an increasing demand for graduates from Information Systems courses rather than computer science and engineering courses (DEET, 1992: 126).

Computer programming is viewed as a necessary skill for information technology professionals. In 1990, DEET released a report into the education and training needs of computing professionals in Australia (DEET, 1990). The report identified 15 groups of tasks undertaken by professionals and para-professionals within the computing profession. Their survey of each computing occupation indicated that only one group, System Trainers, were not involved in computer programming as part of their everyday work. Moreover, another study by DEET (1992) on the relevance of course content to employer needs for entry level information technology graduates indicated that a knowledge of programming, particularly procedural and object oriented programming, was highly desirable (1992: 108).

[1] The University of Melbourne has conducted IS courses through the Department of Librarianship and Information Systems (originally as part of the old Melbourne CAE) for a number of years. A Bachelor of IS was offered by the Department of Information Systems in the Faculty of Science for the first time in 1996.

[2] For ACS Level 1 accreditation, a degree or diploma from a University or CAE must contain at least a 30% information technology component. These guidelines are currently under review.

1.5 Programming in Information Systems Curricula

Computer programming, in some form or another is identified by the ACS as an important component of Information Systems education at the tertiary level. At Level 1, students must have a working knowledge of a programming language. Depending on how one defines computer programming, it is also part of the Level 2 and 3 requirements for ACS professional recognition.

There is, however, increasing debate on what constitutes a programming language. Prior to the development of powerful application programs, it was possible to identify computer programming as a skill where programmers carefully composed programs from symbolic expressions in arcane and esoteric programming languages that prescribed the tasks that we wanted our processes to perform (Abelson, Sussman et al., 1985: 1). Twenty years ago, according to Cooke (1990), the terms *computer user* and *programmer* were synonymous. In those times the owner of a computer almost certainly wrote programs. The developments in hardware and the increased sophistication of compilers, to mention just two factors, have made languages less procedural and, in recent years, more visual, enabling the end user to develop programs through individual applications (DEET, 1990: 209).

Today, with the widespread use of application languages, '*computer user* and *programmer* are again synonymous but this time, software developers are consciously trying to make languages more powerful and easier to use' (Cooke, 1990: 211). Languages for application users are as friendly as possible and the 'computer's resources ease you into the language and guide you through it'(Cooke, 1990: 211). Languages for professional programmers, on the other hand, are geared towards 'someone who needs to get the most power and flexibility out of an application, ... and tend toward the complex and difficult'(Cooke, 1990: 211). Nevertheless, a computer program is still a set of rules that outlines an *abstract being* comprising computational process which operate on abstract things called *data*. In the world of the abstract, novice and professional programmers must learn to understand and to anticipate the consequences of sets of instructions, its impact on the data and the computational process, as well as on other elements such as hardware devices (Abelson, Sussman et al., 1985: 1).

Glinert (1990) emphasises that a person is required to work at two distinct but highly interrelated levels when programming: at an abstract and at a concrete level. The *abstract* component determines a domain, such as data, along with sets of operations and constraints on the elements of that domain, while the *concrete* component provides a representation for the abstract language (ie. as strings of ASCII characters or other program syntax). If a programmer was coding in a low-level language then the distinction between the abstract and concrete components of a language may not be as evident as it would if a high-level language was being used. A similar analogy could be used between text-based languages and visual programming languages (which manipulate objects). This implies that a programmer may require a different set of skills depending upon the type of language being used and, in particular, the level of abstraction above the machine-level code. In programming subjects that use visual programming environments with object-based languages a different set of skills may need to be taught from those where traditional procedural languages are used.

According to Cooke (1990: 214) 'perhaps the most important development in applications languages is the recognition that they are languages. If they are designed and implemented as languages rather than as automated key presses, you can expect them to become much better.' It is now possible to expand the range of programming languages to include not only the traditional languages from the procedural and object-oriented paradigms, but applications languages such as macros in spreadsheets and coding tools in windows-based applications.

1.5.1 Aims of Introductory Programming Subjects

The aims of introductory programming subjects in Information Systems courses in different institutions are as wide and varied as the discipline itself. The aims of these subjects often vary according to the specific requirements of the overall course, the level at which the subject is taught and the need to cater for external demands. Accordingly, an introductory programming subject is often taught to satisfy one or more requirements: to train in a specific language, to teach broad programming concepts or methodologies, or to develop or enhance problem solving skills.

Where the subject focuses on training, programming is taught to develop skills in the syntax of the particular language. The major emphasis in a subject of this type is on the syntax of a computer language. Subjects aiming to teach programming concepts, on the other hand, would emphasise a particular process such as structured programming methods with language syntax used to implement a solution. Alternatively, the focus of a programming subject may be to formalise a student's problem solving skills. Using a procedural language such as Pascal or C, which are the most popular languages in IT courses (Reid, 1994), for training purposes, emphasis would be placed on the syntax of the language such as program architecture, declaration of variables and the use of functions. Subjects concentrating on programming concepts would place greater emphasis on structured programming concepts and the use of general procedures rather than on the syntax of the language. Lastly, where the emphasis was on the development of problem solving skills, the language would be secondary to the processes that the person used to develop a solution to a problem. The language, in this case, would be used to implement a solution.

1.5.1.1 Problem Solving Skills

Irrespective of the primary aim of an introductory programming language, learning to program and to develop an algorithm is, in itself, a form of problem solving development. According to Linn (1985)

> *'Structured programming concepts require a programmer to explicitly use some potentially powerful problem-solving skills. A programmer must manipulate data using only a small set of primitive instructions; they must decompose a large problem into a series of smaller manageable tasks; and finally they should test the programmed solution and make changes to the code'(p.14).*

When solving problems, Denenberg (1990) describes how many students have difficulty determining when or how to use an experimental approach versus a method of simple deductive reasoning. Many science and applied science courses develop the *a posteriori* or empirical approach (effect to cause) while mathematics courses develop the *a priori* or deductive (cause to effect) approach. He concludes that computer programming can be used as a vehicle to teach reasoning and problem solving skills and states

> *'there usually exist no courses to integrate the two methods or to develop the intuition of when it is appropriate to utilise one or the other, or both, in attacking a problem. Computer programming combines the methods of* a priori *and a* posteriori *reasoning and as such, can provide the necessary integrative vehicle' (p260).*

In response to industry demand, many Information Systems courses concentrate on developing or improving students' problem solving skills (DEET, 1992). Programming subjects are part of this process. This contrasts with programming subjects in computer science courses where the focus is on teaching specific *programming* skills through the *systematic study of algorithmic processes* (Longenecker, Feinstein et al., 1995: 10). Indeed, this emphasis on *computer knowledge* to the exclusion of other factors has been a source of major criticism of computer science courses from both academics and business (DEET, 1992: 186).

In the past, students coming to computer science courses have had a background in mathematics and/or science[3]. It has been argued that mathematics is not only a language but involves problem-solving techniques (DeGrace and Hulet Stahl, 1990: 13), although there is considerable research to challenge DeGrace's view (Pea and Kurland, 1983; Kurland, Clement

[3] With the decline in the numbers of VCE students, many of the entry restrictions have been removed from Computer Science courses

et al., 1986; Van Papstein and Frese, 1988; Linn and Songer, 1991; Turkle and Papert, 1992). This alternate view suggests that computer science students use their general cognitive skills in a programming subject to develop specific cognitive skills and processes in the programming domain (see Figure 1.3: Programming in a Computer Science Course).

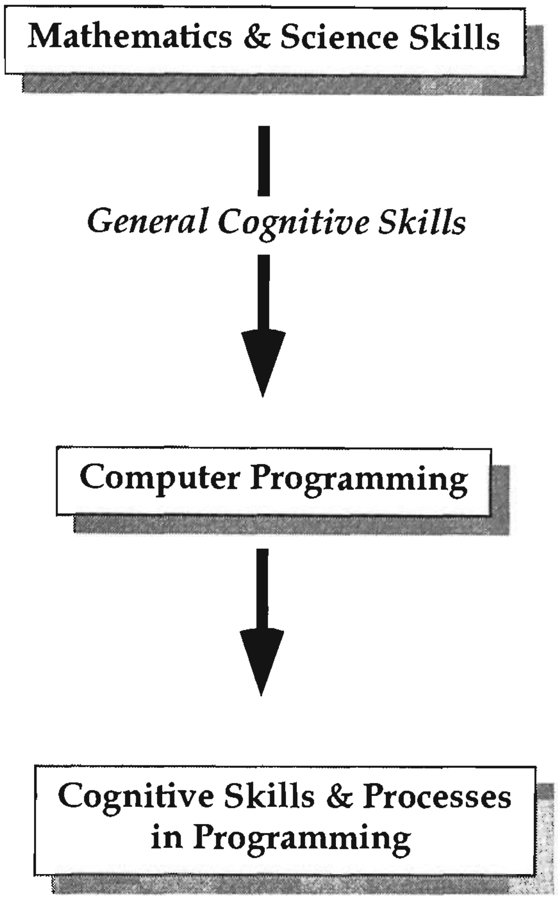

Mathematics & Science Skills
General Cognitive Skills
Computer Programming
Cognitive Skills & Processes in Programming

Figure 1.3: Programming in a Computer Science Course

Students bring *general* cognitive skills, developed in mathematics and science subjects to the computer programming subject through which they develop *specific* cognitive skills and processes in computer programming.

Traditionally, an information systems course has not attracted large numbers of students with mathematics or science backgrounds (Craig, 1996). Turkle (1992) found that many students who enter these types of courses posses a different set of problem solving skills and strategies from students with a mathematics background. Programming subjects in an Information Systems course, it is claimed, play an important role in developing a student's general cognitive skills (see Figure 1.4: Programming in Information Systems Courses).

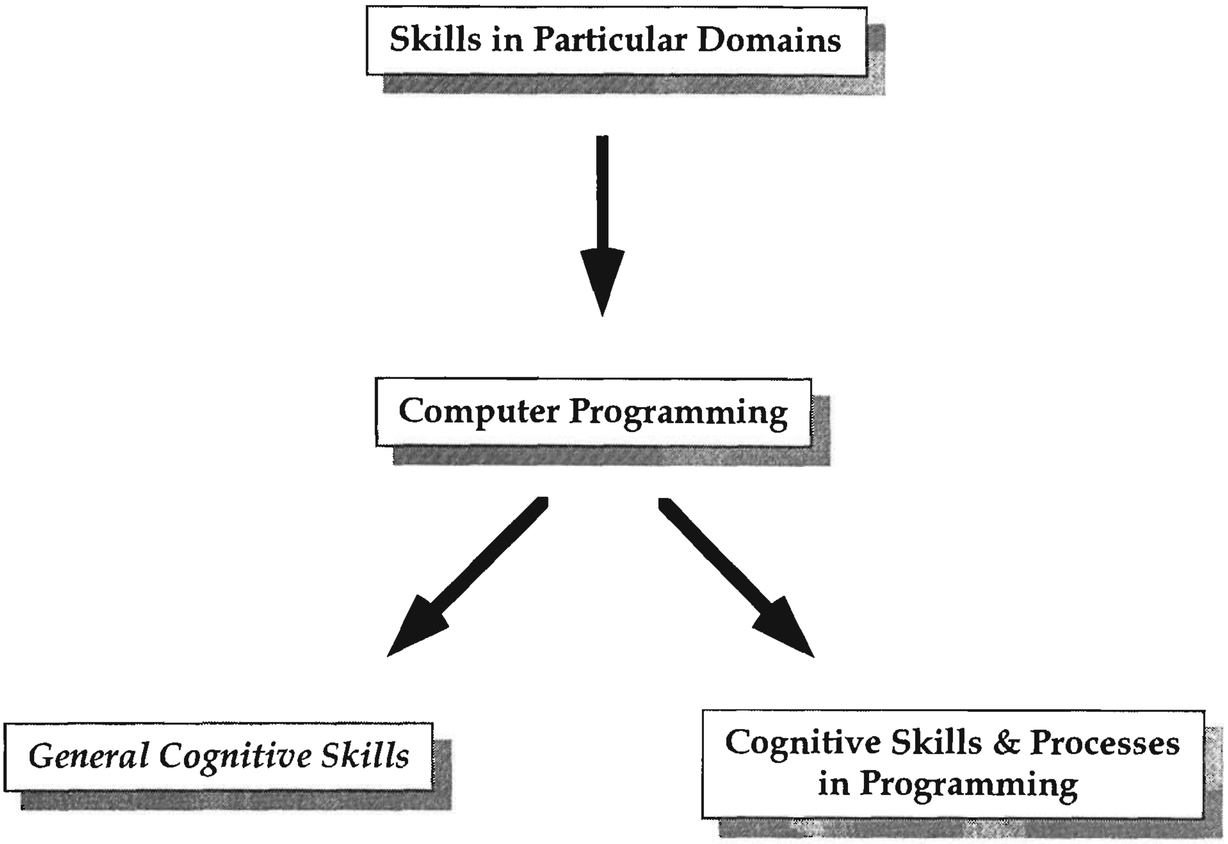

Skills in Particular Domains
Computer Programming
General Cognitive Skills
Cognitive Skills & Processes in Programming

Figure 1.4: Programming in Information Systems Courses

The programming subject aims to develop *general* cognitive skills **as well as** *specific* cognitive skills and processes in the programming domain.

1.5.1.2 Program Design

Traditionally, procedural languages have been used in introductory programming subjects and, almost by default, the design methodology has been *structured programming.* Procedural programs are constructed from statements that are precisely ordered and structured programming design not only acts as a *method* to ensure code is readable and maintainable but as a *process* for the development of a program solution. The main arguments expressed in favour of structured programming as a program design methodology are that it reduces the intellectual load when handling size-induced complexity, facilitates the maintenance and proving of design correctness, yields substantial improvements in program understandability and increased programmer productivity (Ratcliff and Siddiqi, 1985).

To obtain a solution to a procedural programming problem, a student is taught to adopt a classic or 'algorithmic' view of programming. Attention is focused on 'the smallest unit of a program (ie. the statement), on the sequential arrangement and performance of those statements, and on the required precision with which these are created and sequenced' (Yourdon and Constantine, 1979: 31). Moreover, Yourdon (1979) argues that this view of programming necessitates a strict order of teaching programming to novices which concentrates on finding a computational solution and on a detailed statement-by-statement translation of the algorithm. He states:

> *'Some function or task is given; an algorithm or "method of computation" is selected, discovered, or created; this algorithm is translated into a language which the computer will accept' (Yourdon and Constantine, 1979: 31).*

There has been, however, a steady movement away from the use of procedural languages in both industry and in introductory programming subjects. The steady movement away from text-based systems by both programmers and end-users has resulted in substantial productivity improvements in software development,

> *'...this is because the computer's ability to represent in a visual manner normally abstract and ephemeral aspects of the computing process such as recursion, concurrency, and the evolution of data structures through time, can have a remarkable and positive impact on both the productivity of programmers and their degree of satisfaction with the working environment' (Chang, 1990: 145).*

In recent years, tertiary institutions have adopted different programming paradigms into their programming subjects, in particular, object-oriented, object-based or event-driven languages. Moreover, most of these languages work within a visual programming environment. Although program design methodologies exist for object-oriented languages, the newer windows-based and event-driven languages do not have any recognised design methodology.

In the past, structured programming was the automatic selection for procedural languages and the institution only had 'to determine just the applications domain, which could consist of scientific *number crunching* or commercial *data processing*, and the language level' (Glinert, 1990: 1). Now, for the first time, many tertiary institutions that are adopting different programming paradigms are being forced to develop their own program design methodologies.

1.5.2 Other Programming Paradigms

Some Information Systems courses have introduced object-oriented methodologies in design and implementation subjects, in specialist programming subjects and of late, in first year programming subjects using a variety of languages from Smalltalk to C++. Where pure object-oriented languages are used in introductory programming subjects, the complex interface of the editor and interactive debugger has deterred many students from effective use of the language (Smith and Kelly, 1994). As Tatnall & Davey (1995) report, attempts to introduced object-oriented programming into other subject areas have not been wholly successful:

> *'Although students eventually come to grips with the concepts of encapsulation, inheritance and polymorphism, the ideas of reusable code and a closer correspondence with reality often do not happen any more than with loosely coupled programming libraries in any other language' (Tatnall and Davey, 1995).*

This phenomenon also seems to be mirrored in general practice and Jon Udell (1994) reports that

> *'The traditional OOP vision was, at best, vague on the subject of reuse: Objects would appear as by-products of software development, a market would emerge, and programmers would become producers and consumers of objects. There were two major roadblocks. Most OOP language systems , including C++, lack the means to package and distribute objects effectively in binary form. More subtly, the skills and disciplines needed to build components are often quite different from those needed to use them' (Udell, 1994: 46).*

Instead, event-driven programming languages have been advocated as an alternative to support applications development subjects. Students work within an object environment but without producing their own objects. Event-driven languages such as Microsoft Visual Basic and Borland Delphi, introduce aspects of the object-oriented paradigm but are not classified in the strictest sense as pure object-oriented languages. Applications are developed from objects which are driven or triggered by an event. The program code behind each event is procedurally based. For the application to function as intended by the programmer, communication between objects via messages is required. An object only responds to an event that has code placed behind it (Smith and Kelly, 1994).

Although the name may give the impression that these languages are visual programming languages, they are 'textual languages which use a graphical interface builder to make programming decent interfaces easier on the programmer. The user interface portion of the language is visual, the rest is not'[4]. The objects refer not to visual data but to traditional data types such as arrays, and to application-oriented data types such as forms, lists, text and databases. In an ideal world, it would be advantageous to have a visual programming language that could combine and manipulate these objects using visual aids. In reality, the user or developer must revert to some traditional procedural constructs to alter or combine these objects to develop an application.

The move to languages which have a visual component changes the processes people use to develop programs. When using a text based language inexperienced programmers or end users use the syntax of the language to prompt for higher level concepts or semantic knowledge. With experience, this semantic knowledge is gradually separated from the syntax of a particular language and can be transferred to other languages. By contrast visual programming enables

[4] This description was taken from http://www.cis.ohio-state.edu/hypertext/fraq/bugusernet/comp/lang/visual/_visual-lang%3afaq.html

the object to be selected and the operation to be performed immediately; there is no need for decomposition into multiple commands - '..each command produces a comprehensible action in the problem domain that is immediately visible' (Shneiderman, 1983: 326).

In the light of the increase in adoption of these new programming paradigms in tertiary programming subjects, it is important to examine the skills that are taught and the program design approach that is used. Currently, most ideas on what is important are based on the concepts of procedural programming, and this thesis addresses the changes of approach necessary when a procedural paradigm is replaced with an event-driven programming language. Moreover, a program design methodology for event-driven systems is developed, trialed and evaluated for application to introductory programming subjects.

1.6 Significance of a Study in this Area

A characteristic of information technology is rapid change. Today, information processing is more integrated into business operations rather than existing as a separate service function. Information systems is a growing discipline within Australian universities while demand for computer science declines along with the other more traditional science disciplines. Few other curriculum areas have developed so rapidly and there is a need to continually update curricula and program requirements. Computer programming has, for many years, been a part of Information Systems curricula in one form or another. Whether at an introductory level or in electives, students design, construct and test computer programs. Like information technology, the role of computer programming is changing in business as users obtain tools to develop and customise applications, and as a consequence it is likely that the demand for trained professional programmers will decline in the future.

> *'As the field of programming has matured over the years, attention has shifted from the program to the programmer - from the logical and computational structure of algorithms to the cognitive structures of the people who produce them. Innovations such as interactive programming environments, object-oriented programming, and visual programming have not been driven by considerations of algorithm efficiency or formal program verification, but by the ongoing drive to increase the programmer' effectiveness in understanding, generating, and modifying code' (Winograd, 1995: 65).*

It is clear that a study of the changing role of computer programming in information systems curricula is an important one.

Qualitative research methods will be used for this study; more specifically, a case study approach as part of interpretative inquiry. Tesch (1990: 67) classifies this type of qualitative research under the heading of *the comprehension of the meaning of text/action*. Qualitative research methods are more appropriate for a study of this type where there is a need to place 'emphasis on the human as instrument' (Tesch, 1990: 44). Moreover, in justifying qualitative research Lincoln and Guba (1985) state:

> *'Humans are the major form of data collection device'. As instruments, they 'can be developed and continuously redefined'... In fact, 'there is no reason to believe that humans cannot approach a level of trustworthiness similar to that of ordinary standardised tests'(p.195).*

Computer science has been taught in Australian tertiary institutions since the early 1960's. In the United States, attempts to formally outline a computer science curriculum began in 1965. Yet an Information Systems curriculum was not proposed in the United States until the early 1970's, with the publication of the Association for Computing Machinery's Graduate Professional Program in Information Systems. In Australia, the first Information Systems conference to discuss common ground in courses was not held until 1989. Identifying the key components of Information Systems curricula has not been as easy.

> *'Nailing down the fundamentals of this area has proved more difficult than computer science as, being more concerned with the* applications *of computing, the development of Information Systems curricula has reflected many changes caused by the rapid development of information technology' (Tatnall, 1993: 12).*

The increase in popularity of Visual Basic as a programming language is but one example of this rapid change in technology.

There will continue to be major changes in the programming profession in the very near future.

> *'The occupation so far contained by the single job description - computer programmer - is fast splitting into two divergent streams. One group comprises user-programmers, people primarily employed or trained to do something else, but expected to dabble in programming for* application-centric *tasks, using increasingly user-friendly tools such as Visual Basic to do so. Another group comprises traditional skilled and trained programmers. The job specs (sic) for these people are beginning to converge on* tool-centric *tasks, the invention and development of high performance, high quality, reusable software components' (Goschnick, 1993: 20).*

This study investigates the changing role of programming in Information Systems courses. Traditionally, procedural languages have formed the backbone of programming subjects in Information Systems courses. Structured programming was the traditional design methodology. Now, as new programming paradigms are adopted by business and government organisations, there is increased pressure on Information Systems courses to accommodate these new paradigms. The use of object-oriented languages in introductory programming subjects is relatively new and discussions on the merits or otherwise of these languages still appear in journals and conference proceedings. While there is evidence to suggest that object-oriented methods are being adopted by business, tools for their use are still cumbersome. Instead, businesses have adopted more practical languages such as object-based, event-driven programming languages which have offered productivity benefits for both the traditional programmer and the end-user. Although some may be quick to point out the lack of purity of these languages as object-oriented they have become very popular and will need support from the increasing number of Information Systems graduates. It is now estimated that there are 2 million corporate programmers using Visual Basic; this is second only to COBOL and far exceeds C++ (Van Kirk, 1995). As Information Systems courses adopt the event-driven programming paradigm into their programming subjects, lecturers are being forced to reconsider the skills they teach to students and to find a design methodology that can deliver the essential features of event-driven systems development.

In justifying the need for research into the study of programmer behaviour Soloway et al (1986) states

> *'Broadly speaking, the basic assumption of researchers who study programmers is this: By understanding how and why programmers do a task, we will be in a better position to make prescriptions that can aid programmers in their task. eg if we can understand how a maintainer goes about comprehending a program, we should be in a good position to recommend changes in documentation standards that would enable the maintainer to more effectively glean from the documentation the necessary information. Similarly, recommendations for software tools and education should also follow'(p.vii).*

1.7 Summary

Numerous studies have tried to laud one procedural programming language over another but, in most cases, the skills and concepts and the methodology taught in the subject were the same. Changes are occurring in programming environments and there are pressures placed on Information Systems graduates to have a different set of skills and concepts. As event-driven languages are now widely used in business, programming subjects in Information Systems courses are embracing this programming platform, often without considering the approach and methodology to be used.

This study investigates the use of event-driven programming languages in introductory programming subjects in Information Systems and evaluates the effectiveness of a program design methodology for learning event-driven programming.

Chapter 2 - Cognitive Skills and Program Design

As the field of programming has matured over the years, attention has shifted from the program to the programmer, that is 'from the logical and computational structure of algorithms to the cognitive structures of the people who produce them' (Winograd, 1995: 66). Any discussion of programming languages, processes and environments must examine our current understanding of the cognitive processes and skills which gradually evolve as a novice learns to program. This chapter reviews the literature on our current understanding of how novice programmers learn a range of programming tasks in the procedural paradigm. In addition, models of programmer behaviour and issues relating to cognitive science, skills and processes are discussed.

According to Dumas(1995), as programming tools and users become more sophisticated, large productivity gains are only accomplished through changes in the programming environment and in the ways programmers go about creating programs. The chapter concludes by investigating new visual programming environments and alternative programming paradigms and with a discussion of the changes to the programming tasks, program design and programming skills.

Starting with the cognitive aspects of well-understood programming tools, 'we can get insights into the demands for an environment that will support the ongoing development of computer software in a rapidly changing industry' (Winograd, 1995: 66).

2.1 Problem Solving and Design

Despite all the progress that has been achieved in programming languages and environments since the early days of computing:

> *'.. it is an undeniable fact that we remain unable to bridge the chasm that continues to separate the way we, as human beings conceive of solutions to problems from the way that we now must program these solutions for our computers. True, our computing engines become more powerful from year to year, and our knowledge of how to use them advances, too. But our expectations and demands rise even faster. The net result is that today, more than a quarter century since the dawn of programming, the central problem facing computer science is still to find a means by which we will be able to make these machines do what we really want them to. This is true whether the term programming is interpreted in the narrow, conventional sense or in the broader sense, as an endeavour that encompasses all aspects of human-computer interaction' (Glinert, 1990: 146).*

Malhotra et al (1980) describe the act of solving a problem as a process which moves a person from one state to another:

> *'A problem state is said to exist when a human, or other goal-oriented system, has a goal but no immediate procedure that will guarantee attainment of the goal. The goal may be a satisfaction to be achieved or a dissatisfaction to be alleviated. Problem-solving occurs in moving from a problem state to a non-problem state. In problem solving, then, a person begins in an initial state, uses transformations that move him from one state to another, and ends in a final state. Any of these states and transformations may be well-defined or ill-defined' (p.120).*

Algorithm design, which 'typically occurs after the decomposition of a large system into modules and before the more straightforward processes to be accomplished by programmers, ... involves transforming a declarative statement of what is to be done into a procedural specification of how to do it' (Kant and Newell, 1984: 97). Using problem space theory, Kant and Newell (1984) describe the process of designing an algorithm:

> *'A problem space contains partial knowledge about a problem and its solution (the* current state*). The subject has a set of* operators *that can be applied to the current state to produce a new state. The subject starts with an* initial state *(..the problem) and tries to discover a state that contains a solution (.. an algorithm). Behaviour in this space involves a* search, *since the subject usually does not have enough knowledge to proceed directly to the final desired state, especially if the problem is difficult. The subject does, of course, have some (often substantial)* search control knowledge *that guides the selection of which operators to apply. But in general the subject will try various paths and run out false leads into dead ends, causing a return to earlier states that can be remembered (or constructed), and eventually will proceed down more appropriate paths. The current state grows throughout the problem solving, as the subject gradually explores the space and acquires knowledge of its various aspects. (p.99)*

Palumbo and Reed (1991: 343) define the set of knowledge, information, facts, and relationships needed to solve a particular problem as the *task environment.* The mental representation of this task environment is included in the problem space. Thus

> *'one needs both an accurate and complete task environment, providing that all the information necessary to solve a particular problem is presented and then is accurately encoded into working memory for a successful solution of a problem' (Palumbo and Reed, 1991: 344).*

Often, however, the only *a priori* structure imposed on the processes that a programmer must follow is determined by the initial goal structure, that is, what the program should do:

> *'The problem in moving from the idea for a system to its final implementation is in transforming a linearly derived sequence of desired components into a hierarchical arrangement of functions or data transformations. Once the requirements have been delineated, they must be organised so that the inherent structure of the problem becomes visible' (Curtis, 1985: 209).*

Not all problems in the computing domain, however, are of the same type. Some problems may not have a clearly defined starting point from which the programmer or software developer can move. In other cases, the goal state may not be clearly defined, or there may exist a myriad of solutions open to the programmer. These types of problems are typical in software design even though the functional specifications may have outlined the broad aims of a system. As a consequence, a programmer is often forced to use a combination of *a priori* (deductive - cause to effect) and *a posteriori* (empirical - effect to cause) approaches (Denenberg, 1990).

Vitalari and Dickson (1983) outline the processes that an analyst must use to determine information requirements and to synthesise a solution that meets the needs of the user.

> *Specifically, the focus is on the frequency, ordering, and association with analyst performance of the clues, goals, strategies, heuristics, hypotheses, information, and knowledge manifested in the thought process of the analyst' (p.949)*

In contrast, algorithm design problem-solving belongs to an area of human problem solving which Reitman (1965) equates with 'ill-structured' problems. Many of the formal procedures, which may be used to solve well-defined problems, cannot be used by the programmer to solve design problems. 'Problem-solving behaviour in this domain characteristically cannot be specified minutely as a set of moves, selected from a small and finite initial state in order to derive a unique final or goal state' (Carroll, Thomas et al., 1985: 143). Moreover, 'a designer, typically, does not know in advance what the goal state will be, although he (sic) usually has criteria to evaluate potential goal states. Indeed, the designer often does not even have a definition of the initial problem state' (Carroll, Thomas et al., 1985: 143).

In the process of solving a problem a programmer is faced with two distinct tasks; comprehending the structure of the problem and constructing a solution structure.

> *'The solution structure is that which the problem solver must construct in order to solve the problem. Solution structure consists, on the one hand, of the process of solution, a sequence of steps or stages (eg., "incubation"), and on the other hand, of the more statically structured final product of the solution (ie., the "answer"). Problem structure may be decomposed into the inherent structure of the problem (formal relations between entities and an algebra for manipulating these relations) and the presentation of this inherent structure (the problem scenario, the sequence in which relevant facts are revealed to the problem solver, etc)' (Carroll, Thomas et al., 1980: 269).*

An empirical study by Carroll et al (1980: 269) found that the structuring of requirements has an impact on the problem solution; that is, greater structure in the original problem statement leads to a reduction in the number of iterations through design cycles. Similarly, Palumbo and Reed (1991) believe that

> *'problem situations become more complex as the relationships between elements in the problem space are more obscure, thus requiring more operations to construct an adequate problem solution. Problems where all the information is presented in the problem statement are, thus, much easier to solve than those with missing or obscure information' (p.344).*

This method of teaching concentrates on finding a computational solution and on the detailed statement-by-statement translation of the algorithm.

Over the years, tools have been developed to provide a structured approach to the analysis, design and construction of complex computer systems. Structured analysis and structured programming concepts have been used for procedural languages and various methodologies have been proposed for object-oriented systems development. These tools have outlined a set of procedures designed to assist humans solve complex problems. The need for these types of tools has been long recognised. In his famous letter to the ACM, Dijkastra (1968) wrote

> *'our intellectual powers are rather geared to master static relations and that our powers to visualise processes evolving in time are relatively poorly developed. For that reason we should do (as wise programmers aware of our limitations) our utmost to shorten the conceptual gap between the static program and the dynamic process, to make the correspondence between the program (spread out in text space) and the process (spread out in time) as trivial as possible'(p.147).*

For procedural languages, Kraft (1977) argues that structured programming is a way of thinking or 'attitude, one in which programs are written for people rather than machines' (p.130). This point is shared by Ratcliff (1985 :78) who argues that, while structured programming in such forms as step-wise refinement certainly embodies a framework for disciplined design, it provides no specific means of determining the *best* decomposition from a number of alternatives; 'whilst we have the knife, we do not know how to carve' (Ratcliff and Siddiqi, 1985 :78). Moreover, Waters (1993) believes that there is often too much emphasis placed on the final product rather than on the process of developing a program

> *'A good program (and by extension, a good system) is an articulate, lucid and elegant statement of the function it is to perform. Like a good report, the writing is simply the final recording of the information. Much work has preceded it. The ideas are clear, the intent is clear and it is easy to understand'(p.47)*

Although there are clear benefits from the use of a program design methodology, and they are advocated in the use and teaching of procedural and object-oriented languages, there does not exist any recognised program design methodology for event-driven systems.

2.2 Programmer Behaviour

According to Armstrong (1989), computer programming is a simulation of human controlled processing. It is easy to understand why such an analogy can be made as the processes that a human follows when solving a problem which involve designing, planning, monitoring, testing and evaluating, are similar to those used when programming a computer, that is, specifying an outcome, designing a solution structure for reaching the outcome, and evaluating the efficiency of a system in terms of the desired outcome. Thus, 'modelling the machine that models the human provides a structure within which control processes can be understood and provides a vocabulary which can be used to articulate them' (Armstrong, 1989: 71).

Cognitive research on programming, however, depends in part on the work by cognitive psychologists in modelling mental processes. Conceptual models are an essential component of cognitive science and they have been used to explain the cognitive processes in programming at different stages. The models presented in this section are based on cognitive theories that are continually being developed and modified. Although the models may vary, they have as their core short and long term-memory. Short term memory is a limited capacity workspace that processes information occupying the immediate attention of the individual. Long-term memory is an unlimited store and holds the programmer's permanent knowledge. Surprisingly,

> *'cognitive theories of programming have not been elaborated to the extent that they provide alternative explanations of programmer performance. In fact, many of the theories are interesting applications of psychological principles to programming, but they have not been sufficiently elaborated for consistent practical application at a technical level' (Curtis, 1985: 7).*

Some description of the rise in importance of cognitive science is pertinent as it provides a background to the explanation of mental models.

2.2.1 Cognitive Science

Prior to the development of a mathematical theory of computability, researchers believed there was a sharp distinction between the brain and the mind (Johnson-Laird, 1989); the brain was the physical object and the mind some ghostly nonentity. Succeeding the invention of the programmable digital computer, an analogy was made between the brain, which is controlled by the mind, and the computer, which is controlled by a program. As there was nothing particularly 'ghostly' about a program, *cognitive science*, the study of the mind, was born.

In time, cognitive scientists began to question the merits of Behaviourism (or Human Factors). Behaviourists viewed psychology as a natural science, one in which the goal was to predict behaviour in an attempt to control it. Behaviourists did not recognise the mind as a separate entity and believed that humans were incapable of conscious actions progressing through mental stages. Any study of mental process would be fruitless as they believed that any evaluation was possible using stimulus-response studies utilising strict empirical methods emanating from the natural sciences (Johnson-Laird, 1989).

In contrast, cognitive scientists

> *'... study how people perceive, organise, process and remember information. They also study how people later use this information in comprehending new information or solving problems. Finally, they study how different cognitive abilities are distributed across people (Curtis, 1985: 2)*

According to Johnson-Laird (1989), cognitive scientists believe it is possible to study mental processes as people process information or solve problems. In the computer, cognitive scientists had a strong ally:

> *'The return to the study of the mind had a new metaphor in it's language - the computer. In the past, the mind had been likened to a wax tablet, a hydraulic system, and to a telephone exchange. Now, there was a new reaction to Dualism: brain and mind are bound together as computer and program. Yet the full power of the metaphor eluded people. It seemed to reside in the existence of "machines that think". In fact, what the computer offered - or rather the theory lying behind the computer - was a new conception of psychological explanation. This conception is much more powerful than the earlier mechanical analogies. Indeed, ... it is yet to be exhausted' (Johnson-Laird, 1989: 23).*

The cognitive theories that have been developed attempt to illustrate how humans picture, store and retrieve information from their short and long-term memory. In addition, they attempt to explain how humans manipulate information to make decisions and solve problems, and how they implement their responses. A cognitive approach to solving a problem or making a decision assumes that human beings are adaptive and flexible in their actions, and constantly interacting with the environment. Eberts (1987) describes this cognitive approach in terms of a three cornered diagram including the actual product, a conceptual model and a mental model (see Figure 2.1).

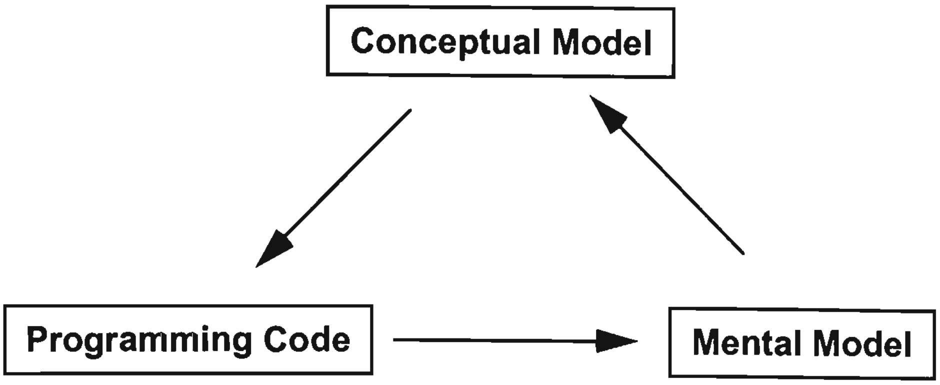


Figure 2.1: Goal of Cognitive Approach

Source: Adapted from Eberts (1987: 275)

In Figure 2.1, the *conceptual model* is a formal description of the computer hardware and operating system that is accurate, consistent and complete. This may be a virtual machine created by a high-level language. The *mental model* is the model that the programmer forms of how the computer system is manipulated by a program; this mental model determines the programmer's actions and behaviours. The programmers mental model may change over time through interactions with the *programming code* in the various programming phases. This process highlights the problems many inexperienced people have in developing computer programming skills. In the initial stages, it is often difficult for novice programmers to develop a mental model which accurately reflects the true conceptual model. As they develop clearer mental models their knowledge of programming improves.

2.2.2 Cognitive Style and Cognitive Ability

Cognitive style (or *processes*) refers to the manner in which individuals process information, that is, how they think (Foreman, 1990: 56). Over time individuals develop a preferred way of thinking, solving problems and interacting with the immediate environment. In terms of academic achievement, the various styles range on a continuum from field dependent to field independent learners. According to Foreman (1990), field independent learners are inclined to impose a structure on a problem if some logical pattern does not exist, whereas field dependent learners accept the status quo. Importantly, field independent learners have a greater degree of cognitive restructuring, a skill which enables them to interpret a problem and reformulate it into a series of structured concepts. Bostrom (1988), however, concludes that

> *'although individual difference variables have consistently accounted for much more of the variance in data than any experimental manipulation, hardly any one of the cognitive style factors has by itself emerged as a consistent predictor of effective decision-making performance' (p.230).*

Learning style is the only cognitive style variable that Bostrom et al (1988) believe could influence the learning of software.

Cognitive ability, on the other hand, is used to describe individual differences in performance (Foreman, 1990). Aptitude or intelligence is often used in reference to a persons cognitive ability. Intelligence includes, amongst other factors, the ability to learn, to adapt to new situations, and to solve problems but little is known about the mechanism that causes individual differences in basic cognitive ability (Foreman, 1990). Moreover, 'an attempt to define basic ability or aptitude is fraught with philosophical issues' (Bostrom, Olfman et al., 1988: 231). Nevertheless, a person's cognitive ability is enhanced by the cognitive skills they possess or come to possess, and the cognitive processes they use. Cattell (1987) divides cognitive ability into two categories based on cognitive function; crystallised ability and fluid ability. Crystallised ability is often used to describe the general ability of an individual whereas fluid ability is a specific ability. Crystallised ability is accumulated from previous experiences and involves applying that ability to new circumstances. Fluid ability is a skill that is needed to modify certain performance requirements. For example, general mathematical ability is acquired from previous experiences and, is thus a crystallised ability. On the other hand, the ability to follow instructions is an example of a fluid ability.

Computer programming demands a large range of cognitive skills and abilities in the process of developing an algorithm to solve a problem. The common method adopted by a programmer towards algorithm design involves reformulating the problem into to a series of structured processes matching the logic of the program (Turkle and Papert, 1992). Field independent learners, it is claimed, have a greater propensity to tackle problems in this way (Foreman, 1990) as they have greater skills in cognitive restructuring than field dependent learners. Programming a computer, however, calls on a large range of cognitive skills making it difficult to match them to one particular cognitive style (Shneiderman and Mayer, 1979; Pea and Kurland, 1983). As we will see later, research has shown that programmers do not necessarily use a structured, top-down approach to algorithm design in procedural languages (Ratcliff and Siddiqi, 1985; Turkle and Papert, 1992).

Moreover, some studies have suggested that there is a correlation between general mathematical ability (a crystallised ability) and cognitive skills required to learn programming (Koubek, Salvendy et al., 1989; Wilkes, 1991; Navrat, 1994).

> *'Caution must be used, however, when attempting to extrapolate these results to expert programmers. The skills used to learn programming and those used when actually performing the programming task are probably not identical' (Koubek, Salvendy et al., 1989: 180).*

Often, learning to program is a new experience for most people and fluid ability (eg. reasoning, spatial visualisation) may be equally, or even more important than general ability, particularly for non-procedural languages (DeGrace and Hulet Stahl, 1990; Foreman, 1990). The development of event-driven systems in visual programming environments is significantly different from the development of a main-line controlled procedural program. It cannot be assumed that the methods used to teach procedural programming to students or the previous mathematical knowledge possessed by a student, are applicable in an introductory subject using event-driven programming as the platform. Bostrom et al (1988) emphasises this point when he states

> *'a host of mental-aptitude constructs have been examined in a learning environment by numerous researchers. Only a handful have been shown to be consistent predictors of learning or problem solving proficiency' (p.231).*

2.3 General Cognitive Models

Any model of programmer behaviour must outline the cognitive processes and the stages of knowledge acquisition that occur for novice and expert programmers. Moreover, it must outline the basic tasks of programming in terms of the cognitive structures programmers possess or come to possess, and the cognitive processes involved in using their knowledge or in gaining new knowledge. It should be sufficiently general in nature to accommodate the fundamental tasks of the different programming paradigms whether they be structured procedural programming, object-oriented programming, event-driven programming, or some other programming paradigm.

The general cognitive models presented here were formulated when procedural programming languages, such as COBOL and Pascal, were the predominant languages. Despite some changes to accommodate new programming paradigms (such as Shneiderman (1983)) the basic models have remained intact and revolve around long-term and short-term memory. A greater number of novice and experienced programmers, however, are now programming computers at different virtual levels using a larger range of languages and processes than was available in the past. Some of these general cognitive models do not account for tasks which are now included in the programmer's domain such as screen design. Moreover, with the increased power of programming languages (particularly with event-driven languages) many of the traditional tasks such as system design, interface design and algorithm design are closely linked (Malhotra, Thomas et al., 1980; Cooke, 1990; Wilkes, 1991).

2.3.1 Syntactic/Semantic Model

One of the first models developed to explain programmer behaviour was the Syntactic/Semantic model developed by Shneiderman and Mayer (1979). Their general cognitive model was developed to explain five subtasks of programmer behaviour: composition, comprehension, debugging, modification and the acquisition of new programming concepts, skills and knowledge. Shneiderman and Mayer used the research of Greeno (1973), who outlined the components of memory used during problem solving, to develop a memory model of programmer behaviour. The model has three interrelated memory systems:

Short-term memory - a relatively limited capacity store taking up to seven 'chunks' or items

Long-term memory - an unlimited capacity store which contains the programmer's permanent knowledge and;

Working memory - which has a storage capacity between that of short-term memory and long-term memory, and allows for the interaction of information between short-term and long-term memory. New structures may be developed in this memory area.

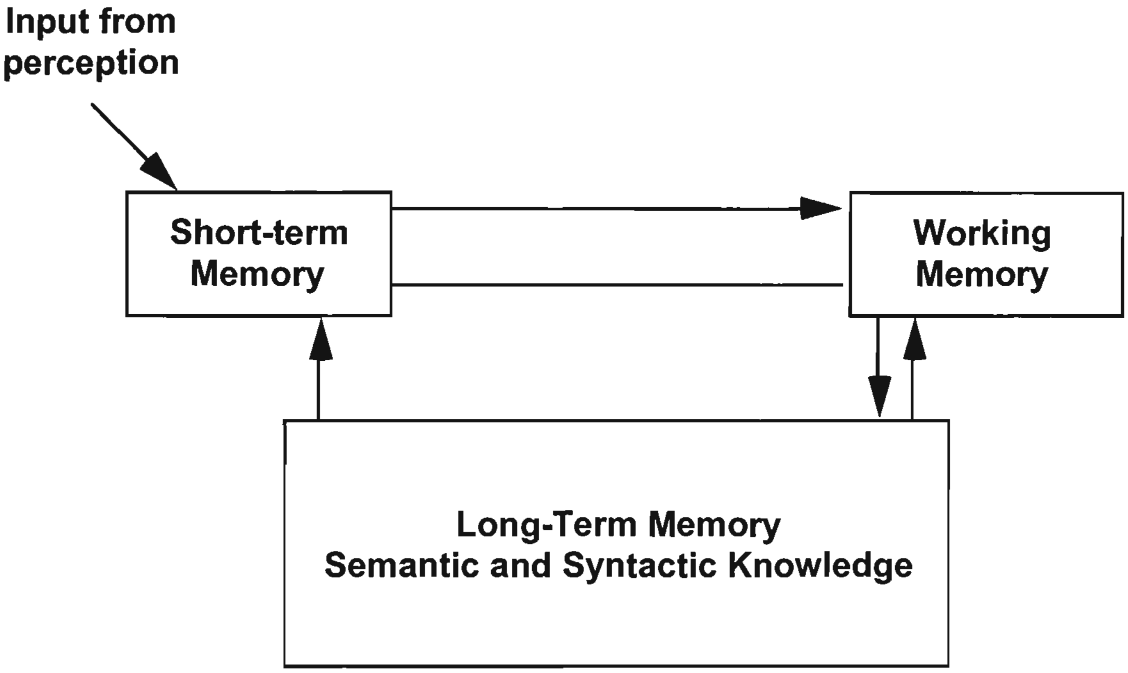


Figure 2.2: Syntactic/Semantic Model
Components of Memory in Problem Solving

Source: Shneiderman & Mayer (1979: 221)

When a programmer is required to develop code to solve a particular problem, information about the problem enters into short-term memory (see Figure 2.2). These perceptions of the problem are passed on to working memory where they are combined with existing concepts stored in long-term memory. Any new information or concept is stored back into long-term memory and action in the form of a solution is taken to solve the problem.

2.3.1.1 Types of Knowledge

Shneiderman and Mayer (Shneiderman and Mayer, 1979) distinguished between two forms of knowledge stored in the long term memory of an experienced programmer. Semantic knowledge is a complex multilevel body of knowledge that '..consists of general programming concepts that are independent of specific programming languages' (p.222). This type of knowledge takes the form of a hierarchical structure ranging from low-level notions, such as what an assignment statement means, to higher level notions such as recursion, and sorting and merging algorithms.

> *'Semantic knowledge is meaningfully acquired by reference to previous knowledge, by example, or by analogy. There is a logical structure to semantic knowledge that is independent of the specific syntax used to record it. Semantic knowledge is further decomposed into computer and task-related domains. Semantic knowledge has to do with the actions and objects in the computer domain. The low-level actions might be assignment, iteration, conditional execution, input, output, synchronisation , etc. Higher level actions are algorithms for sweeping through an array, for sorting items, or for recursive binary tree search. Computer objects include the low-level data types such as booleans, integers, strings, real numbers, etc. Higher level objects include arrays, records, stacks, or threaded trees' (Shneiderman and Mayer, 1979: 5).*

The semantics of the task domain can be decomposed into objects and actions. Actions and objects in the task domain are not necessarily related to a knowledge of computers or programming. The task domain may be a business area such as stock control or accounting, or some other discipline. In this case, a knowledge of the objects such as customers, stock or debtors is required along with a knowledge of the actions such as opening an account, or recording stock deliveries.

Novice programmers usually have a sound knowledge of the task but lack skills in the area of programming. Conversely, experienced programmers are knowledgeable in programming but may not be familiar with the task domain. The model implies that, as a programmer's experience increases, so does the quality and quantity of semantic knowledge (see Figure 2.3).

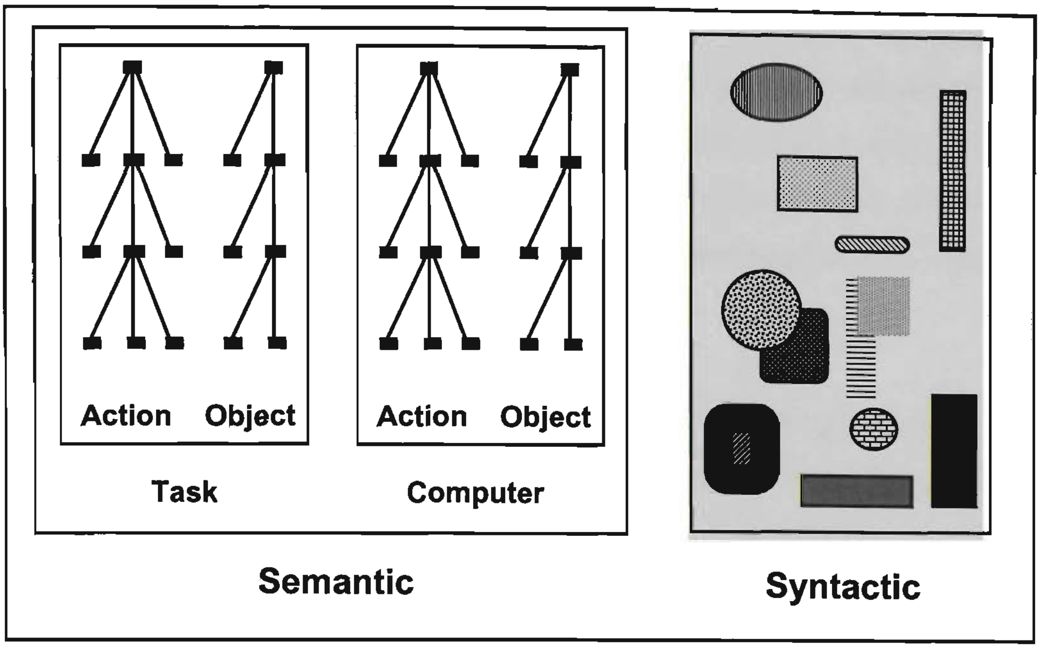


Figure 2.3: Long-Term Memory Components -Semantic versus Syntactic Knowledge

Source: Shneiderman and Mayer (1979: 6)

Syntactic knowledge, on the other hand, is more detailed, precise and arbitrary than semantic knowledge. Syntactic knowledge, like semantic knowledge, is stored in long-term memory and consists of detailed facts about a particular language. The model suggests that it is easier to learn a new syntactic representation of an existing semantic structure rather than learn a new semantic construct. Moreover, syntactic knowledge

> *'.. is arranged according to a particular programming language, with similar languages overlapping, and is used to express semantic knowledge in a programming language' (Koubek, Salvendy et al., 1989: 173).*

Bayman and Mayer (1988) divide semantic knowledge into two further sections, conceptual (or domain specific) knowledge and strategic knowledge (or transfer strategies). *Conceptual knowledge* refers to a person's understanding of programming concepts and the impact of these concepts on a computer system. This knowledge area includes the processing actions that can take place on a computer, the system's various components and the data structures and objects that can be processed. As an example, the Pascal statement `Count: = Count + 1` has a number of actions:

```
copy the value in memory location Count to a register
Increment the register by one
Write the register to the Count memory location
Increment the Program Counter
```

The person must understand the conceptual system and the actions of the program statement on this system if he/she is to have a sound conceptual knowledge of programming at that level. The statement indicates that he/she must identify between locations such as memory, registers, input and output devices; the concepts of moving, copying and incrementing, and the data structure (integer, floating point reals, and the program counter).

Strategic knowledge, on the other hand, is the ability of the programmer to use syntactic and conceptual knowledge to solve problems (Bayman and Mayer, 1988). This type of knowledge is used to generate, interpret and debug programs.

Through their review of domain specific and strategic knowledge in academic performance, Alexander and Judy (1988) further divide conceptual knowledge into *construct* knowledge, a knowledge of the programming constructs, and *algorithmic* knowledge, a knowledge of combinations of these constructs (Oliver, 1991). Figure 2.4 summarises the types of knowledge:

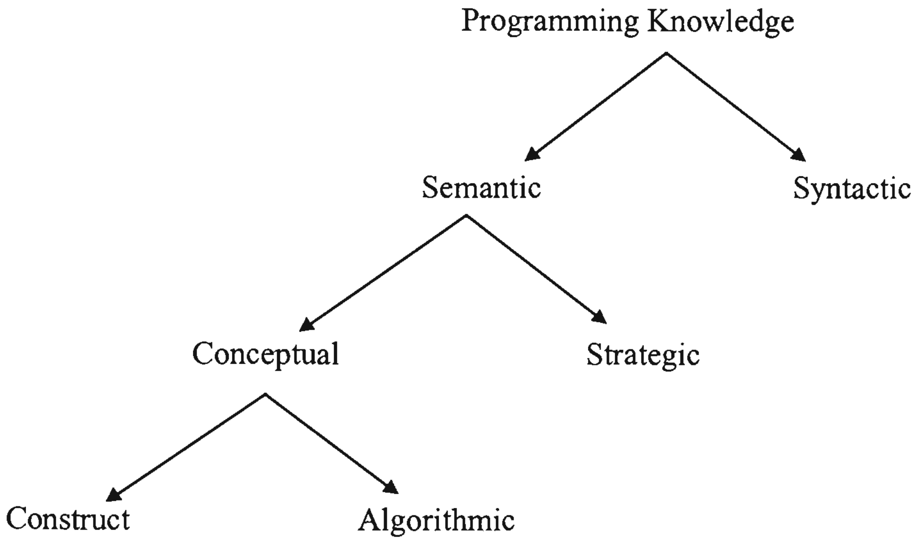


Figure 2.4: Types of Programming Knowledge

2.3.2 Production System Model (Coding/Comprehension Model)

In an effort to explain the order in which code is written by a programmer and the changes in the knowledge a programmer has about a program as writing progresses, Brooks (1977) developed the Production System model. The model uses a similar representation of memory to that outlined in the Syntactic/Semantic model.

Brooks (1977) asserted that programmer behaviour can be divided into 3 distinct states when programming a solution to a problem: understanding, method-finding and coding. The heart of the model is the application of production rules in the coding stage. As a programmer encounters a condition in the development of a solution, a specific course of action is undertaken. In the first stage, the programmer repeatedly scans resources such as program specifications and reports, or may ask questions to determine the exact nature of the problem. This information is stored in short-term memory which contains exact details on the nature of the problem as opposed to how it should be undertaken.

The second stage in the cycle is method-finding. Brooks (1977) defines a method to be a plan or outline of the program to be constructed and

> *'..consists of specifications of the way in which information from the real world is to be represented in the program (data structures) and of the operations to be performed on these representations to achieve the desired effect of the program (algorithms)' (p.740).*

The methods are stored in a hierarchical format in long-term memory, with smaller methods serving as components of larger methods. In addition, methods are independent of specific programming languages although some methods may be '.. specific to groups of programming languages which share common data control structures' (Brooks, 1977: 740).

The final stage involves coding the solution in an appropriate programming language. The problem-solving behaviour of a programmer is controlled by a production system. This production system is the heart of the model and comprises a series of production rules. The programmer takes elements or parts of the methods formulated in the previous stages, and determines the appropriate action, in the form of program code, according to production rules. The selection of the appropriate rule is determined by data stored in short-term memory and information about quantities and data structures which is stored in a long-term memory structure which Brooks (1977) calls *meanings*. The programmer can also use *external memory* or code which is the physical record of what the programmer has already written. Figure 2.5 shows the overall structure of the model.

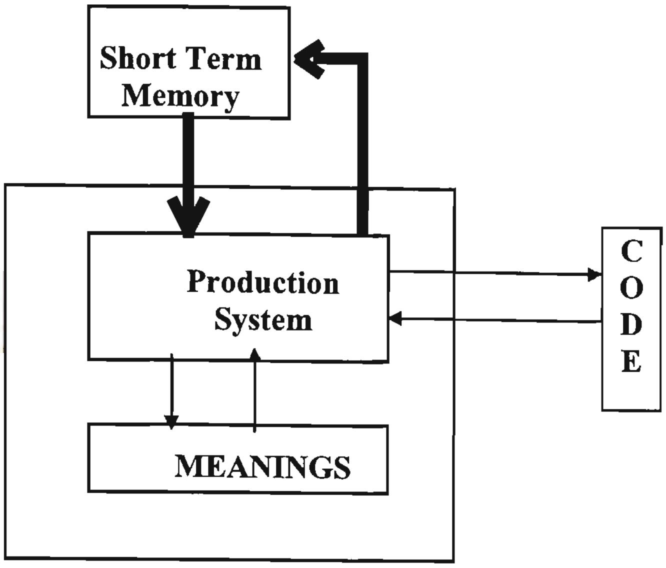


Figure 2.5: Generalisability of the Brooks Model

Source: Brooks (1977)

As the generation of code progresses, details in short-term memory may change or the effects from previous coding may require changes to be made. In this case, a new set of production rules may apply. The model suggests that, as a programmer's knowledge increases through experience, so too must the number of production rules for the myriad of conditions and combinations of conditions.

Brooks (1983) later extended his model to include the concept of beacons as a knowledge structure used in the comprehension of computer programs. Program comprehension, he argues, is important in many of the software development phases, such as code reviews, debugging and testing as well as an obvious use in program maintenance. The theory purports that:

> '1. *The programming process is one of constructing mappings from a problem domain, possibly through several intermediate domains, into the programming domain.*
> 2. *Comprehending a program involves reconstructing part or all of these mappings.*
> 3. *This reconstruction process is expectation driven by the creation, conformation, and refinement of hypotheses' (Brooks, 1983: 544).*

His theory of program comprehension is strongly top-down and hypothesis-driven '... in which an initially vague and general hypothesis is refined and elaborated based on information extracted from the program text and other documentation' (Brooks, 1983: 543). Central to his theory is the process of *verifying hypotheses.*

> *'The programmer does not study a program line-by-line but rather forms a series of increasingly specific hypotheses about program function. The highest-level hypothesis concerns overall program function and may be derived from*

> *documentation or from more scanty sources, such as program name. This overall hypothesis leads the programmer to expect to see certain objects and operations in the program, and these expectations then form a second level of more specific hypotheses about the program' (Wiedenbeck, 1991: 517).*

The process of hypothesis verification occurs at the point where the top-down hypothesis formation activity links up with the data of the program text (Wiedenbeck, 1991).

> *'Brooks believes that programmers verify hypotheses not by a minute study or simulation of every line of the code, but by searching for* ***beacons****, which are key features that typically indicate the presence of a particular structure or operation' (Wiedenbeck, 1991: 518).*

Brooks (1983) describes beacons as idiomatic or stereotypical elements which occur frequently in program code. During program comprehension, an experienced programmer uses beacons to identify key features of a program. Although Brooks described beacons in the context of top-down program comprehension, they are also consistent with studies observing both top-down and bottom-up strategies (Littman, Pinto et al., 1986). The programmer begins by systematically assigning meaning to sections of program code. As more knowledge of program functions is discovered it is added to the programmer's memory. 'At some stage in this process the programmer has enough information to make top-down hypotheses about program function. These hypotheses may then be verified by a search for beacons' (Wiedenbeck, 1991: 519).

2.3.3 Schemas and Rules of Discourse

Beacons are related to the concept of programming plans developed by Soloway et al (1984). Van Merrienboer (1990) observes that 'learning computer programming means both learning procedures to accomplish various goals and learning the information that is relevant to these procedures' (Van Merrienboer and Paas, 1990: 274). One of the main distinctions between experts and novice programmers can be made on the basis of a large store of organised knowledge called *schemata* or *programming plans* (McNutt and Lo, 1991: 406).

Soloway et al (Soloway and Elliott, 1984) argue that the expert programmers have at least two types of knowledge that novice programmers typically do not. Specifically,

Programming Plans:

Program fragments that represent stereotypical action sequences in programming

Rules of Programming Discourse:

Rules that specify the conventions in programming, eg., naming a variable so that it is compatible with its function.

A program which may have a complex goal can be thought of as a complex plan incorporating several simple plans. Kant (1985), however, argues that these programs have a kernel idea which 'serves as a beacon in program comprehension because it is central to the goal of the program and occurs in many variations of it' (Wiedenbeck, 1991: 519). Beacons, however, often include the kernel idea and other surface features of the program which may denote the program's function. 'This might in some cases include the control structure of the program (such as the looping structure in a sort), procedure names, and variable names' (Wiedenbeck, 1991: 519).

Soloway et al (1984) argue that

> *'programs are composed from programming plans that have been modified to fit the needs of the specific problem. The composition of those plans are governed by* rules of programming discourse. *Thus, a program can be correct from the perspective of the problem, but be difficult to write and/or read because it doesn't follow the rules of discourse, ie., the plans in the program are composed in ways that violate some discourse rule(s)' (p.595).*

Letovsky (1986), extending on the work of Soloway (1984) developed a knowledge based model using schema or programming plans. Their cognitive model views programmers as knowledge based understanders and covers comprehension, coding and design. 'The basic structure of the model is a set of knowledge assimilation processes which construct a mental model of the target program by combining information gathered through reading the code and documentation with the knowledge from a knowledge base of programming expertise' (Letovsky, 1986: 59).

The model provides for three components: a knowledge base, a mental model and the assimilation process.

The *Knowledge Base* is postulated to contain:

Programming language semantics - a knowledge of the language,

Goals - a set of common computational goals such as sort and search,

Programming plans - determine the development of programming expertise by transforming knowledge about mechanical processes in specific languages into correct plans which are language independent,

Rules of Programming Discourse - rules that specify the conventions in programming,

Domain Knowledge - knowledge of the application domain, and

Efficiency Knowledge - criteria to assess inefficiencies in code design.

A programmer is said to use an *Assimilation Process* to develop a mental model of a program. The assimilation process is the set of strategies that a programmer uses to search his/her knowledge base and to combine this with the documentation to construct a mental model of the final program (Letovsky, 1986). By assembling individual programming plans, a programmer develops a mental model representing an overall program plan and goal description. More experienced programmers have a larger knowledge base, a more extensive set of well defined plans, and a more refined set of cognitive skills and strategies to use when developing program code. Novice programmers have only a small set of plans that can be combined together to develop a program and, as a consequence, they often work at a syntactic level rather than by using programming plans. The resulting mental model of a program would include:

specifications - a complete description of the specification or goals of the program,

implementation - a complete description of the individual components of the program, and

annotation - an understanding of the role of each component and the relationship between the components.

In their work on the Programmer's Apprentice, Rich & Waters (1988) attempted to develop automated aids to program implementation, design, and requirements specification. They used familiar combinations or *clichés* as beacons or plans but their work did not extend to program comprehension.

2.4 Programming Tasks

Shneiderman (1979) divides the process of developing a program into a number of subtasks: program requirements, program design, coding, comprehension, program testing, debugging and documentation. Although there is no universal agreement on the precise tasks involved in developing a program, Shneiderman's categorisation serves as a useful classification for the review of literature in this area. It is interesting to note that,

> *'... although the sense of computer programming has not varied in nearly four decades, the set of activities involved in doing programming has undergone major qualitative transformations. In the early days of programming, for example, the programmer had to know the details of the computer hardware in order to write a program that worked; today this is no longer true. An important consequence of this evolution of the set of activities that constitutes programming is that the "cognitive demands" made by computer programming needs specification at the level of programming subtasks, or component activities' (Pea and Kurland, 1983: 5).*

2.4.1 Design/Requirements

In the case of design problems, the individual does not usually start from an initial state. In addition, there may be constraints placed on the tools and methods used, but the processes involved are usually limitless. 'In real-world design situations, the goals are, typically, fuzzy and poorly articulated and cannot be mapped directly into properties of the design. Thus, the exact configuration of the final state is not prescribed' (Malhotra, Thomas et al., 1980: 120). Further:

> *'A part of the design process consists of formalising and redefining the design goals into functional requirements that can be matched by properties of the design. Even so, it is usually difficult to tell how well a design meets a particular functional requirement. In addition, the functional requirements often cover different dimensions and the trade-offs between them are rarely well specified. The properties of a design arise from a combination of the properties of the design elements that constitute it and design organisation that specified their interaction. Design elements are available, indivisible units with given properties. Design consist of some number of design elements interacting with each other in design organisation' (Malhotra, Thomas et al., 1980: 120).*

Vitalari & Dickson (1983) outline a number of characteristics of the analysis and design phases which make problem solving different from other aspects of computing. They are:

- *Analysis problems have ill-defined boundaries, structure, and uncertainty about the nature and make-up of the solution.*
- *The solutions are artificial, that is they are designed and hence a number of different solutions exist for the same problem.*
- *Analysis problems are dynamic. The problem changes over time while they are being solved, given the complex nature of the organisations and the personnel involved*
- *Solutions to analysis problems require interdisciplinary knowledge and skill.*
- *The analyst must continually adapt to different technologies and end user demands*
- *The process of analysis, itself, is primarily cognitive in nature, requiring the analyst to structure an abstract problem, process diverse information, and develop a logical and internally consistent functional set of specification. All the other skills such as interpersonal interaction and organisational skill facilitate this cognitive process (p.949).*

In this area of computing, problem solving is the reasoning process that the analyst uses to analyse an information requirements determination problem and to synthesise a solution that meets the needs of the user and the organisation.

> *'Specifically, the focus is on the frequency, ordering and association with analyst performance of the clues, goals, strategies, heuristics, hypotheses, information, and knowledge manifested in the thought process of the analyst' (Vitalari and Dickson, 1983: 949).*

Winograd (1995) questions this 'outside-looking-in' perspective of software design where the focus is on the mechanisms and programs are developed in an operational sense '... relating a program to the operating systems, networks, programming interfaces and the like which will surround its operation' (Winograd, 1995: 68). Instead he believes designers should take an 'inside-looking-out' perspective with a focus '... on people and their situations: how people experience software; what they do with it; and the larger situation in which they encounter it' (Winograd, 1995: 68).

A novice programmer is intimately involved in the process of systems design when he or she are using an event-driven programming language such as Visual Basic. The tasks may range from interface design, identification of events and responses through to table design and data processing functions. At the interface development stage alone, the programmer is faced with numerous options (screen layout, how to enter data and the type of process method. It is clear that teaching students using event-driven languages compared to traditional procedural languages involves significantly different strategies and that it may not be possible to separate the once traditional programming tasks from other areas of systems development.

2.4.2 Coding

At the end of the design process the programmer may have developed a broad representation of the program in Structured English, a Nassi-Schneiderman diagram, a flow chart, a State-Transition diagram or some other design instrument. The coding process follows the design phase and involves the 'translation from the most refined version of the program design into the programming code'(Pea and Kurland, 1983: 19). The end result of this process is a programming language representation of the original design. Pennington (Pennington, 1982) describes the coding process as:

> *'a plan element triggers a series of steps through which a piece of code is generated and then the programmer "symbolically executes" that piece of code in order to assign an effect to it. This effect is compared with the intended effect of the plan element; any discrepancy causes more code to be generated until the intended effect is achieved. Then the next plan element is retrieved and the process continues. Throughout this process a representation of the program is built up, strong information about the objects (variables, data structures, etc), their meanings, and their properties.' (Pennington, 1982: 24)*

The general cognitive models described by Shneiderman and Mayer (1979), Brooks (1977) and Soloway et al (1984) describe the cognitive processes involved in coding a solution. Pea and Kurland (1983), however, believe that three general classes of cognitive demands exist for those successfully undertaking coding:

> '1. *ability to engage in hypothetical symbolic execution of code*
> 2. *ability to learn the coding templates that define the syntactical knowledge necessary for code generation; and*

> *3. ability to keep to the goal at hand, or program plan, unless deviations from it are required to generate the code; in such an event the plan would need to be revised, with consequences for the code then to be generated' (p.20).*

2.4.3 Comprehension

The study of general cognitive models and internal representations of a program '... is not a sufficient means for assessing program comprehension, since the external form of the program and the task being performed influence the mental representation' (Gilmore and Green, 1984: 46). Aaronson (1976) has shown that the comprehension of a program is related to aspects of the language, the specific programming task at hand and memory requirements.

On explaining the difference in levels of comprehension Gilmore & Green (1984) observe that some research has suggested that

> *'... every programming language is translated into the same mental representation, and that comprehension performance reflects the extent to which the external program conforms to the internal "knowledge language". Thus, if two programs in different languages specify identical algorithms, their representation will be identical, but it may be harder work to create the representation from one language' (Gilmore and Green, 1984: 33).*

Pea et al (1983) identified four different views of the program comprehension process: top-down, bottom-up, middle-out and transformational. Brooks (1983), in outlining his cognitive model, argues that program comprehension is a top-down hypothesis-driven process in which beacons in the source code are used to either verify or disprove the prevailing hypothesis about the program's functionality. Brooks (1983), however, fails to discuss '... how beacons are detected, or how differences in the comprehensibility of notations arise'(Gilmore and Green, 1984: 530). Shneiderman et al (1979) observe that expert programmers do recall more gist of a program than does a novice but ' ... argue for a bottom-up construction of meaningful units of program code, from operations to algorithms on up to the function of the program as a whole' (Pea and Kurland, 1983: 20).

Pea et al (1983), quoting from Pennington (1982), provide an outline of a multiple pass model developed by Attwood & Ramsay (1978) in which programmers utilise high-level and low-level information about a program structure in order to understand a program.

> *'On the first pass, some level of the [program] macrostructure is integrated, possibly as high as program function, possibly at some level of chunking. Successive passes lead to integration of lower level propositions (working down) and successive integrations of the macrostructure (working up).' (Pennington, 1982: 27)*

In contrast, the transformational approach (Pea and Kurland, 1983), implies that program comprehension is influenced by representations at three levels: deep plans (or the purpose), surface plans (in program structure) and the definitions of data objects, properties and input-output specifications for the program code

2.4.4 Debugging

Understanding the structural and functional components are essential ingredients in the task of maintaining or enhancing a computer program. For the programmer to efficiently debug a program, whether it be manually or using a debugging tool he or she must have

> *'... a clear understanding of the flow of execution in his program and of the data being manipulated, in order to select effective breakpoints and relevant variables to be examined or modified. Thus, the representation of the program from which the programmer is working is a critical factor in the debugging task, since it is from this point that knowledge of execution sequence and data usage is drawn.' (Brooke and Duncan, 1980: 1057)*

Littman et al (1986) identify two strategies for program maintenance, the systematic and the as-needed strategy. A programmer using the systematic strategy attempts to understand how the program behaves before attempting to modify it. A programmer traces data flow and control flow between components of the program such as subroutines. In contrast, a programmer using the as-needed strategy '... attempts, as soon as possible, to localise parts of the program to which changes can be made that will implement the modification' (Littman, Pinto et al., 1986: 81). In this case the programmer does not approach the maintenance task with a good knowledge of a program's functionality, and it becomes necessary for a programmer to gather additional information, often in an ad hoc manner, while the process of program modification is being performed. Littman et al (1986) found that programmers who adopted an as-needed strategy often failed to construct successful modifications and were 'unlikely to detect interactions in the program that might affect or be affected by the modification' (p.81)

Building on the general cognitive model using programming plans and rules of discourse, Gugerty and Olson (1986) found, not surprisingly, that experts with a greater knowledge of programming concepts were less likely to produce programs containing errors or were more successful at program maintenance than novice programmers

> *'... experts may know more particular strategies for focusing in on the bug, via topographic search; or they may use the ones they know more effectively ... [or] ... if experts are debugging in a familiar domain, they would be expected to have a larger mental "library" of symptom-bug associations and thus do better at symptomatic search' (Gugerty and Olson, 1986: 14).*

Students have always been tempted to circumvent systematic program design methods and to us as-needed strategies. Applications can be developed quickly using event-driven languages and there may be a tendency for students to be even more tempted by using as-needed strategies. Moreover, there is no recognised program design methodology for event-driven languages and little or no documentation.

Program code can be placed behind events in different objects. This fragmentation may hinder the testing and maintenance phases as there is no overall view of the system. On the other hand, as the code is often isolated behind events in objects, it may assist with the testing of different components of the system.

2.4.5 Documentation

Documentation can provide information to other persons enabling them to obtain a detailed understanding of the system. The structure, language and detail of the documentation will depend upon the intended audience of the documentation. In large systems, design and function specifications, operation manuals, source code listings, user manuals and on-line help are all items of documentation that help different users of the system. As applications have become more complex, responsibility for additional documentation such as design specifications and the user manual have become the responsibility of technical writers.

Nevertheless, one component of a good program is how well it can be maintained, and documentation of the source code and support material such as test data assists in that regard. Documentation, both of the source code and of software operation assists with program comprehension. Shneiderman (1982) found that well structured, high level comments in source code can assist programmers with the comprehension of program functionality.

> *'Competent programmers deal more with problem domain related units than with program domain related syntactic tokens. High level comments using problem domain terminology have been shown to be more effective in aiding comprehension than numerous low level comments using program domain terminology'(p.55).*

An important component of systems development is documentation for testing, comprehension and on-going maintenance of systems, yet no broad conventions have been established for external programmer documentation of event-driven systems.

2.4.6 Programming Tasks Revisited

The process of developing a computerised system rarely involves moving from the initial design phase through to the final implementation and documentation phases in a sequential fashion (Carroll, Thomas et al., 1979). Moreover, coding is a process that requires the programmer to revisit the output from various subtasks for verification, correction or modification. The program design phase, in particular, is closely linked to the task of program code generation (Pea and Kurland, 1983: 19). Although it may be possible to arbitrarily divide systems development into a series of subtasks, in reality the subtasks are interdependent. This is particularly so in the case of event-driven languages.

Moreover, the increased use of programming in applications has merged many of the previous clear and distinct programming tasks. The processes followed in different paradigms such as object-oriented and event-driven programming, may have shifted the emphasis from one task to another or changed the very processes themselves. It may not be possible to isolate a set of tasks which can be broadly associated with computer programming in totality. Instead, the purpose of a particular programming language or the role of the system that is produced may dictate different tasks for different activities in different contexts. As an example, programming in event-driven languages may require a different set of skills and strategies from those used in structured programming, or from those using direct manipulation in iconic environments. As Wilkes (1991) states:

> *'the development of modern workstations and programming environments has made it possible for immense power to be put at the disposal of one person. If that person is a programmer, he or she can handle every stage of a large programming project - taking a broad view of the work as well as working on details when necessary. The result is to make the modern programmer an autonomous professional with full responsibility for the work done' (p.23)*

Schurmann (1994) believes that the complexity of a program has been reduced through a combination of program standards, program architecture and the implementation technology. As languages have become more sophisticated 'the enforcement of constraints to limit complexity progressively changes from being a set of programming rules (standards), to favouring the architectural approach, to ultimately becoming part of the implementation technology itself' (Schurmann, 1994: 33). Figure 2.6 outlines this trend.

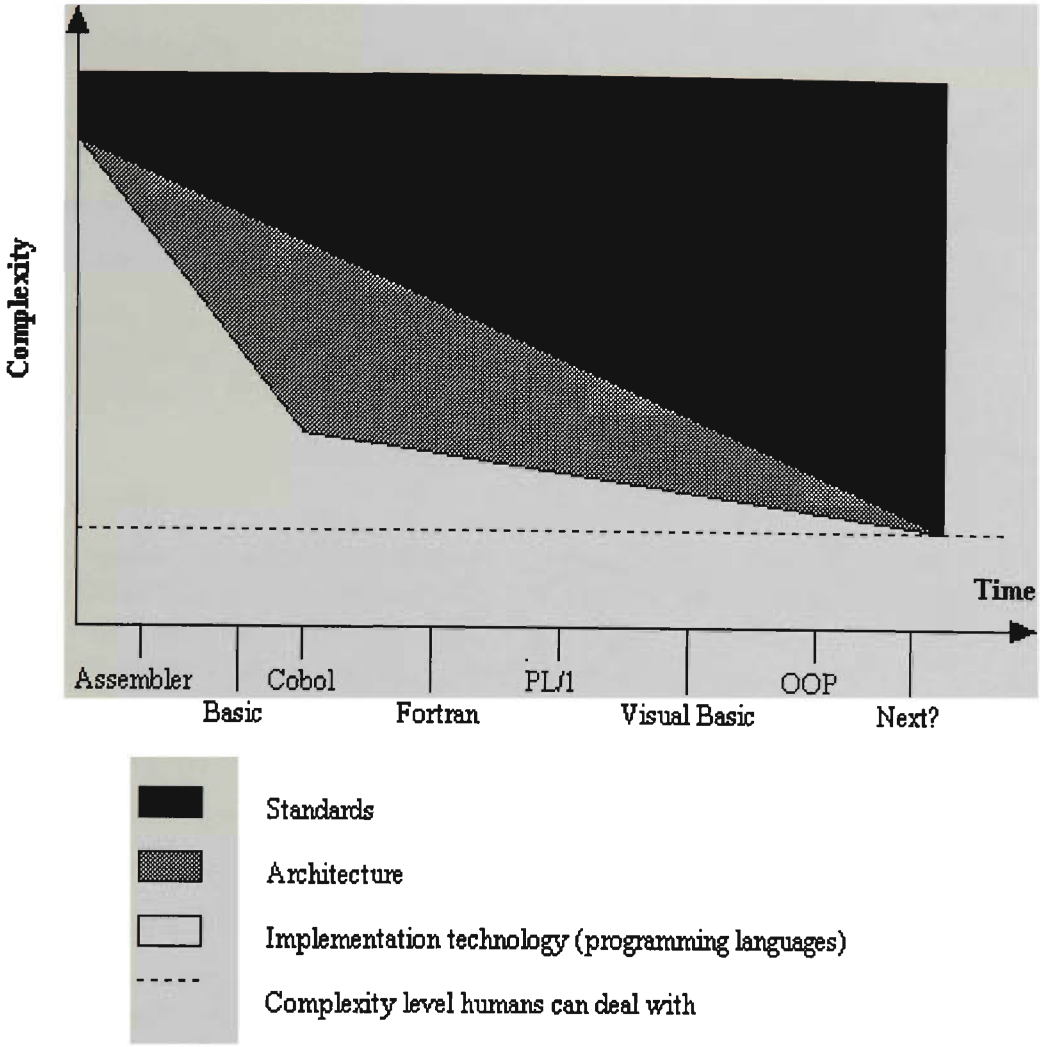


Figure 2.6: Program Complexity

Source: Schurmann (1994: 33)

As an example, the definition and management of data is inherent to Visual Basic and does not have to be provided by an architecture as it would with a procedural programming language. On the other hand, design issues are of great importance in Visual Basic, as it is the platform upon which code is built.

According to Chang (1990) this changing relationship between standards, architecture and implementation together with visual program environments, has changed what he refers to as the programming cycle. He believes that people don't conceive of solutions to problems in the same way that they must now program those solutions. Despite this, little work has been done to show how programmers go about developing solutions using event-driven languages in visual programming environments.

2.5 Programming Environments and Program Design

2.5.1 Program Environments

The development of programming environments where work is done in programming languages within the context of an integrated set of software support tools, especially input and output devices, has been an important step forward in software engineering.

> *'A good environment can embody and facilitate the principles and practices that make programming more productive. It brings the programmer's activities into focus along with the activity of the program being produced' (Winograd, 1995: 68).*

Chang (1990) believes that

> *'... issues related to graphics, alternative programming paradigms, and artificial intelligence, on the software side, and to multiple I/O device, on the hardware side, are but a few of those that need to be studied if such environments are to become a reality. (Chang, 1990: 147)*

In a traditional programming environment, '...the objects of interest are programs, and the programmer's tools are designed to operate on various representations of those programs' (Winograd, 1995: 68). Winograd builds on these traditional programming environments to develop a *software design environment*, one which

> *' ... works with a broader array of representations, including different kinds of conceptual models, mockups, scenarios, storyboards, and prototypes. The design methods reach outside of the workstation to include the setting and the thinking of the people who will use the software. The activities of a software designer include the traditional activities of software engineering and programming, such as specification writing, coding, and debugging, along with user-focused design activities '(p.68)*

A comparison of the two environments is made in Table 2.1

Table 2.1: Evironments for Programming and Software Design

In expanding from programming environments to environments for design, there are suggestive correspondences between current techniques and what is needed for user-oriented software design

Programming Environments	Environments for Software Design
• Interactive programming • Specifications • Reusable code • Interactive debugging	• Responsive prototyping media • User conceptual models • Design languages • Participatory design

Source: Winograd (1995: 68)

According to Dumas et al (1995) the tools provided with an environment are often as important as the language itself. Some of these tools such as interactive debuggers and coloured editors can assist the programmer and make the programming task easier. For example,

> *'the success of Visual Basic has been largely attributed to its programming tools, not the appeal of programming in the Basic language. Competition among programming environments has evolved from a focus on functionality alone, that is, offering more tools or more sophisticated tools, to competing on useability as well as functionality. While programmers are clearly computer literate, they, like any end users, want tools that provide both new capabilities and ease of use' (Dumas and Parsons, 1995: 47).*

These visual programming environments and support tools were not available in the early years of teaching with procedural languages. It is yet to be seen what role these tools can play in assisting novice programmers to understand programming concepts and their impact on the teaching strategies of lecturers.

2.5.2 Program Design

According to Hanif (1996) one of the important steps in designing a solution to a given programming problem is program design. 'Program design is often described as the development of a solution algorithm, independent of language, which could be used for translation into a specific programming language' (Robertson, 1993: vii). Procedural application programs use a main controlling module which starts with the execution of the first line of code and follows a pre-defined path till it reaches the last line of code. Initially, the problem space may be large and ill-defined, however, program design is relatively straight forward and simple to document (Pea and Kurland, 1983).

In teaching programming, present day practice places importance on the notions of top-down design, procedural and data abstractions, and modular programming to assist with program design (Navrat, 1994). 'Design is the process of gradual concretisation' (Navrat, 1994: 19). Abstract concepts are implemented in terms of concrete concepts

> *'Frequently, this is accomplished in several steps, so abstract concepts are implemented in terms of less abstract (ie., more concrete) ones, which are in turn implemented in terms of even less abstract concepts etc.' (Navrat, 1994: 17)*

According to Navrat (1994) an important concretisation of any concept is the way it is represented. A programmer can use various design representation techniques which '... support the concept of a parent-child relationship between controlling routine and sub-routines' (Hanif, 1996: 2). Martin et al (1984) outlined a comprehensive guide for the representation of data and activities at various abstract levels for a procedural program (see Table 2.2).

Table 2.2: Representation of Data and Activities for Procedural Programs

	Data	Activities	
Entity-Relationship Diagram	Strategic Overview of Corporate Data	Strategic Overview of Corporate Function	Decomposition Diagram Action Diagram Warnier-Orr Diagram
Data Structure Diagram	Detailed Logical Data Model	Logical Relationship Among Processes	Decomposition Diagram Action Diagram Warnier-Orr Diagram Dependency Diagram Data Flow Diagram HIPO Diagram HOS Chart State-Transition Diagram
Jackson Diagram	Program-level View of Data	Overall program Structure	Action Diagram Data Navigation Diagram Warnier-Orr Diagram HIPO Diagram Structure Chart HOS Chart Jackson Diagram
Warnier-Orr Diagram	Program Usage of Data	Detailed Program logic	Action Diagram Data Navigation Diagram Flow Chart Pseudocode Nassi-Schneiderman Chart HOS Chart Decision Tree and Table

Diagrammatic representations of object-oriented environments have been developed by a number of authors ((Coad and Yourdon, 1990; Booch, 1991; Rumbaugh, Balha et al., 1991; Martin and Odell, 1992; Shaler and Mellor, 1992)) who have examined the analysis, design and coding of object-oriented systems. A number of these writers have used various representation techniques such as Class diagrams, Object diagrams, Module diagrams, Process diagrams and State Transition diagrams. According to Yourdon (1994)

> *'one of the important characteristics of an OO project is that* all *the life-cycle activities share common vocabulary, notation, and strategies - that is* everything *revolves around objects. ... This is in stark contrast to the development activities in projects using the older, conventional methodologies*

> *... where the programmers ignore all the modeling activities because it seems unrelated to their programming language' (Yourdon, 1994: 32)*

2.5.2.1 Program Design Tools for Event-driven Languages

Windows-based programming languages such as Visual Basic do not follow the procedural nor the object-oriented program development paradigm,

> *'... instead, miniprocedures and other program fragments are attached to the interface designed in the first step. For this reason, any attempt to design a Visual Basic application using traditional programming representation techniques is not applicable' (Hanif, 1996: 2).*

According to Cornell (1993)

> *'trying to force Visual Basic into the framework of old programming languages is ultimately self defeating - you can't take advantage of its power if you continue to think within an old paradigm'. (Cornell, 1993: 10)*

Braithwaite (1996) believes that, while Visual Basic makes event-driven programming easy, two major problems are evident from a program development focus:

> 1. *Procedural code is deterministic whereas event-driven code is asynchronous. Structure charts are hence unsuitable for designing event-driven programs. They do not incorporate the concept of events to control program flow. The sequential nature of structure charts is at odds with the asynchronous nature of event-driven programming.*
>
> 2. *Program structure and flow are difficult to document due to the asynchronous nature of the event-driven environment as well as the addition of objects to a procedural language (p.71)*

Braithwaite (1996) extended his research and found

> *'that state-transition diagrams have been used as the basis for visual programming languages, and it is possible to map Petri net constructs directly to flow chart equivalents. Unfortunately, neither of these techniques promote structure in transformational systems and hence the resultant designs may leave much to be desired in terms of legibility and maintainability. Both the program development focus and the user interface development focus present problems and hence cannot be regarded as effective approaches in isolation' (p.71).*

Separate research by Hanif (1996) and Tesch et al (1996) found little agreement on the processes that could be followed during the design of a Visual Basic program and an even less agreement on the notation used to represent functional tasks.

Petre (1995), however, believes

> *'the success of a representation, graphical or textual, depends on whether it makes accessible the particular information the user needs - and how well it*

> *copes with the different information requirements of the user's various tasks' (p.34).*

Visual Basic may benefit from what he describes as secondary notation, that is, the combination of graphical and textual notation. For Visual Basic there may be no single panacea. 'But if we can identify the particular strengths of different sorts of representation, we will be able to design appropriate solutions with a full repertoire of notational options' (Petre, 1995: 44).

2.6 Summary

This literature review has examined the skills and processes which make up the programming discipline. It has reviewed the cognitive models used to explain how novices learn to program, and it has established the need to re-examine these processes in the light of new programming environments and languages.

Research spanning several decades has investigated the difficulties novices face in learning to program. Over that period of time, several cognitive models have been developed that helps to explain the processes novices use when grappling with the concepts of computer programming. Moreover, these models have gone part of the way in assisting lecturers who are teaching programming. Most of the literature, however, is from era when procedural languages dominated introductory programming subjects. Top-down design and structured programming were the established methods for teaching these types of languages.

Although these models provide some insight into the cognitive processes and skills of learning to program, new environments, particularly visual programming environments, together with event-driven languages have reduced the complexity of many programming tasks. These cocktails of object-based languages within visual development environments have provided novice programmers with powerful tools but with little knowledge of fundamental programming concepts. Moreover, lecturers faced with teaching event-driven languages in introductory programming subjects have no established design methodology nor any clear direction on the mental models that are used by students to understand the operation of such languages. The next chapter will review the literature that has reported on the teaching of these types of languages in introductory programming subjects in information systems courses.

Chapter 3 Event Driven Programming and Curriculum Design

Academic programmes in Information Systems have undergone change throughout the years as faculties have fought for acceptance of this emerging discipline area in tertiary institutions. In recent years, business has found fault with computer-based programmes, particularly in the field of computer science, as computing has '... become more integrated into many businesses rather than a separate service function' (DEET, 1992: 126). Although the graduates from information systems courses appear to be more attuned to the needs of business, the complex interrelationships between information technology, human decision making, and the goals of those who seek to build and use information systems for competitive strategy have caused many researchers to view information systems from different perspectives (Seddon, 1991). Complicating the picture further is the need for information systems professionals to have formal technical training in order to be effective and to have a '... sound educational foundation in their discipline so they can keep abreast of new technologies and be responsive to change' (Richards and Sanford, 1992: 219). Programming continues to be an important part of tertiary information systems curriculum but has undergone significant change as institutions seek alternatives to COBOL and Pascal. This in part, matches the steady decline in text-based systems (for both end users and programmers) over recent years and the growth in 'off-the-shelf' software with the resultant devolution of information processing toward the end-user (DEET, 1992). According to Cooke (1990):

> *'The irony is that the more powerful and flexible an application becomes, the more options you have, and the more useful it becomes to be able to perform diverse functions based on system state. If you have commands that can alter system state, conditionals that choose alternate execution pathways, and a method of storing these things, you have a programming language'(p.211).*

This section of the literature review examines recent research on alternative approaches to the teaching of programming in Information Systems and computer science courses at tertiary institutions. Particular emphasis is given to the object-based and event-driven paradigms and the role they play in enhancing the programming skills of information systems graduates.

3.1 The Changing Information Systems Curriculum

Information Systems have always been significant in the management and operation of organisations. According to Longenecker (1994)

> *'Computer based IS are complex socio-technical entities that have taken on critical roles on local, national and global organisations. IS provide support for the goals of the organisation and its management - strategic, tactical and operational - in a timely and cost effective manner. The applied nature of the discipline suggests a critical link between education with the practicing professional community'(p.175).*

Since its inception as a discipline in tertiary institutions several decades ago, Information Systems curricula has undergone rapid change

> *'The reason for these changes are twofold: firstly the computing and communication technology, upon which the discipline is based, has made several quantum leaps during this period of time, and secondly the loci of applications, to which the technology has been applied, is widening at a tremendous pace. As a result, computing educators are still attempting to come to grips with an appropriate curricular and research framework for IS' (Ang and Lo, 1991: 386).*

Over this period the use of computers in organisations 'has evolved from machines which could calculate and produce simple reports to distributed multiprocessors with powerful individual work-stations for the end user' (Longenecker, Feinstein et al., 1994: 175)

Longenecker et al (1994) attempt to define information systems as an academic field by constraining it to two broad areas,

> 1. *acquisition, deployment, and management of information technology resources and services (Information Systems functions) and*
> 2. *development and evolution of infrastructure and systems for information use in organisations processes (systems development) (p.8).*

DEET (1992) uses information systems as a general term to denote all operations and procedures involved in the design and implementation of an information processing system. Keen(1987), however, argues that searching for an agreed definition of Information Systems is futile:

> *'There is no way in which the diversity of topics already addressed by people who see themselves as part of the Information Systems community can be summarised in some grand general theory. If the field is broad, eclectic and multi-disciplinary, coherence cannot be imposed by finding some common ground or theory or method' (p.2).*

According to Seddon (1991) the only common theme that is evident in any perspective of Information Systems is that all are concerned with the successful exploitation of information technology.

In a comparative study of information system curricula in the United States of America (U.S.A) and foreign universities, Goslar and Deans (1993) found that both U.S.A. and foreign universities tended to view their information systems programs as more managerial than technical. They reported that there was higher emphasis in U.S.A. Information systems curricula on programming and computer equipment although both groups viewed their courses as significantly different from computer science courses (p.12).

It is not surprising then, that courses in Information Systems in tertiary institutions place varying degrees of emphasis on different aspects of information technology and business concepts (DEET, 1992). Richards et al (1992) believe that the ultimate goal of any Information Systems course is to prepare students for working towards beneficial technological change. They believe that Information Systems courses have difficulty in adequately integrating technical skills within an organisational and social context. Moreover, they consider that the present approach to information systems education must be placed in a broader context which can be traced back to the mid-nineteenth century when a debate raged on the relative merits of practical and theoretical training for engineers. They state:

> *'The subsequent split between practical versus theoretical training has since been a pedagogical paradox faced by many instructors. The argument stems from the more inclusive notion of whether it is more appropriate to teach first principles instead of focusing on the practical, more concrete heuristic implementations which usually have attached to them a higher perceived utility. This dichotomy has been discussed in other forums, such as the tradeoffs between "deep" versus "shallow" knowledge and the importance of distinguishing between the abstract and concrete. ... The difficulties involved in imparting a solid foundation in theoretical knowledge to students is well known, and this enigma becomes even more pronounced for students who do not posses practical experience' (p.219).*

The Discipline Review of Computing Studies and Information Sciences Education Committee (1992) chose to extrapolate the findings of the Quality of Education Review Committee (Karmel, 1985) on primary and secondary education to the tertiary sector. The Quality of Education Review Committee identified five general competences (ie. *the ability to use knowledge and skills effectively to achieve a purpose*) which were adopted by the Discipline Review Committee:

- *Acquiring information and skills - skills, including the necessary technical knowledge in the relevant subject areas, which are acquired by reading, listening, recording, analysing and conducting experiments*
- *Conveying information (communication skills) - the competence of communication of information, orally and in writing*
- *Applying logical processes - the skill of effective analytical reasoning is fundamental for students to develop the competence of using technical knowledge and skills in practical situations*
- *The execution of practical tasks through the marshalling of ideas, materials and tools - this competence requires, inter ail, the ability to analyse problems, to exercise judgements about adequacy of results and to synthesise solutions, and is developed through the actual performance of practical tasks; and*
- *The development of interpersonal skills - this competence requires effective collaboration with others as a means of achieving particular purposes, and may require tact, sensitivity, tolerance, determination and timing. It*

> *requires A knowledge of people and how they interact, and is a competence often insufficiently recognised within our universities. It can be developed through group project work (p.153).*

The Discipline Review Committee (1992) added creative ability to the above list stating that, although creative ability is not a general competence achievable by all students they believed that a student's creative ability can be stimulated in educational programs by:

- *'subject areas that* do not *emphasise one fixed body of knowledge;*
- *stressing innovation through a process of subjecting Information Technology to continual change highlighting "lateral thinking" in students*
- *stressing the importance of ongoing professional education and the need for adaptable skills.*
- *highlighting the pervasiveness of Information Technology applications in almost all areas of endeavour' (p.155).*

The *IS '95* project was, in part, an attempt to draw together the different iterations of information systems curricula in the U.S.A. and to identify common ground and intended outcomes (Gorgone, Couger et al., 1994). Information systems curricula, however, have not been without their critics who have viewed them as lacking in mathematical and technical rigour. In criticising the earlier IS'90 model and advocating a computer science curriculum, Fabbri & Mann (1993) stated:

> *'It ignores the increasingly greater complex problem-solving skills demanded of the contemporary information system developer. It is too superficial and devoid of mathematical and technical content to prepare the graduate for the challenge of developing large complex integrated systems and of capitalising on rapid advances in technology' (p.78).*

Bonnici & Warkentin (1995), responding to Fabbri & Mann's criticism suggested that:

> *'effective problem solving requires a well-honed understanding of the problem -- and computer science graduates often lack a deep knowledge of traditional business systems components. Without a strong foundation in the study of such systems, it is impossible for an Information Systems professional to appreciate the intricacies of the problem and the problem's environment. Indeed, because today's Information Systems professionals often work closely with end users in support of their computing needs or during prototyping activities, the breadth provided by the DPMA curriculum ensures that the Information Systems professional can successfully work with financial analysts, managerial accountants, service operations managers, and a host of other decision makers with real problems to solve' (Bonnici and Warkentin, 1995: 97).*

Moreover, Bonnici & Warkentin believe courses in institutions 'which generate graduates primarily with programming skills are going to be training "buggy-whip" manufacturers' (p.96). They suggest

> *'... that instead, employers need graduates who can "bridge the gap" between technology and the real world of business. In fact, a number of universities with established schools of information technology have discovered that their graduates frequently excel at algorithmic thinking, but lack the necessary*

> *background and education to solve real business problems involving actual business systems and processes' (p.96)*

3.1.1 Programming in Information Systems Curriculum

Fifteen years ago, according to Cooke (1990), the terms computer user and programmer were synonymous. In those times the owner of a computer almost certainly wrote programs. Today, with the widespread use of application languages, computer user and programmer are again synonymous 'but this time, software developers are consciously trying to make languages more powerful and easier to use' (p.211).

The Department of Employment, Education and Training's report on the Education and Training Needs of Computing Professionals and Para-professionals in Australia found that two streams of programming are likely to emerge in the future

> *'Lower level programming will be hardware specific, requiring detailed technical skills and knowledge of multiple processing across a range of repositories. In contrast, higher level programming will be close to the end user, requiring detailed knowledge of business functions. The two streams will be reflected in the development of tool kits, utilities and generic system models on one hand and the configuration of these facilities into applications that meet specific user requirements on the other hand' (p.172).*

Cooke (1990) divides the higher level languages into two further divisions, user and power-user languages. Languages for users are as friendly as possible and the 'computer's resources ease you into the language and guide you through it' (p.211). Language for power-users, on the other hand, are geared towards 'someone who needs to get the most power and flexibility out of an application, ... and tend toward the complex and difficult' (p.211). An example of the former may be a macro wizard in a word processor and an example of the later would be programming in a database language or in Visual Basic. Moreover, Cooke (1990) states:

> *'Perhaps the most important development in applications languages is the recognition that they are languages. If they are designed and implemented as languages rather than as automated key presses, you can expect them to become much better. What constitutes "better" depends on what you want to do with the language. The split between user and power-user languages is going to become more pronounced. This split is less a matter of the power-user languages becoming more powerful (although they do with each new release of the applications) than it is one of the user languages becoming better fitted to their role. Innovations like iconic languages, object-oriented languages, and a general concern for the needs of the average user are showing up more strongly all the time' (p.214).*

Duntemann (1993) observes that, despite their many iterations throughout the years, traditional third generation languages have failed to incorporate relational database verbs into their syntax. Instead,

> *'the major databases are extending themselves toward the traditional programming languages by allowing users to write traditional structured code "behind" the usual interactive features for maintaining tables and generating queries and reports. The Big Blur between traditional languages and database managers has begun, and if it's coming from the direction opposite that which I expected' (Duntemann, 1993)*

In a study seeking the views of academics on the present and future mixture of subjects within an information systems curriculum, Ang & Lo (1991) found that:

> *'the ranking of "programming language" has dropped dramatically from 6th at present to 18th in the future, while 4GL's and applications generators and project management have moved up from 18th and 19th positions to a tied rank of 8.5. This indicates a rather strong view shared by IS academics that 3rd generation programming languages will gradually lose their role as the main tools for software systems development, while 4GL and applications generators will gain a more important role as systems development tools' (p.396).*

The study also showed an increasing importance for programming methodologies ranking this the second most important topic in future information systems curricula.

The DEET (1990) report into the Education and Training Needs of Computing Professionals and Para-professionals in Australia found that the current relationship between programming and application design (as depicted in Figure 3.1) will change as language technologies become more advanced so that application design and programming will overlap significantly.

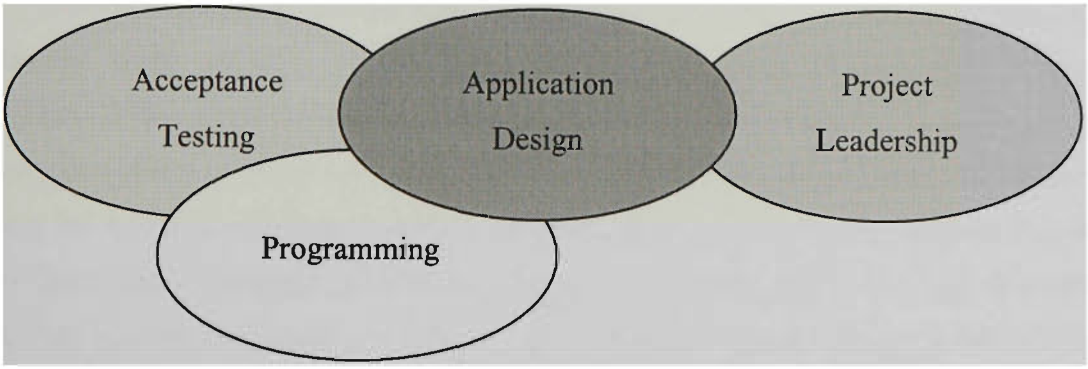


Figure 3.1: Overlap between analyst-programming tasks

Source: DEET (1990: 40)

They outline the impact on curricula by stating:

> *'these results clearly indicate that most areas of IT have a significant impact on skill and knowledge requirements in application design, both now and in the future. Both training providers and curriculum developers need to consider a broad range of technologies when addressing these training needs. ... Clearly, future programming curricula will need to cover emerging technologies in these areas' (p.196).*

Richards (1992) believes that the trend in computing towards a microcomputer environment with graphical user interfaces and non-procedural languages, '... points to the need to introduce more of this type of education into the curriculum' (p.225). Moreover, Bonnici & Warkentin (1995) state:

> *'... in today's computing environment, the knowledge of business-oriented 4GLs provides our students with more useful skills than ADA or Pascal. This is especially true when many of our graduates work in end user department environments or at information centres or help desks. So rather than providing our students with the theory of abstraction, modularization, encapsulation, or reusability, which may not be as important as communication skills, the DPMA model seeks to provide the students with a practical mix of computer and business skills which will allow him or her to succeed at the required tasks inherent in solving actual systems problems'(p.97).*

Fabbri & Mann (1993) are critical of the languages used in information systems model curricula as they believe the languages do not '... prepare the graduate for the challenge of developing large complex integrated systems and of capitalising on rapid advances in technology' (p.78). Bostrom et al (1988), however, believe that languages such as database and modelling languages, which can be used to model business problems, '... may in fact be more difficult to learn because of their problem-solving component' (p.224). In a study of languages taught to MBA students Bostrom et al found that many students had difficulty grasping the concepts of 4GL languages.

The widespread use of third generation procedural programming languages has often been a barrier to application development or modification by end users. An expert programmer has been able to adapt schema and strategies across languages. On the other hand, the characteristics of these types of programming languages, has often inhibited the end user from adequately describing a plan and strategy. If '... the programming language does not cognitively fit with the non-professionals' problem-solving skills, then a barrier has been erected to their use of computers'(Soloway, Bonar et al., 1983: 853). The use of a natural language may not be sufficient as it does not necessarily match the preferred cognitive strategies that an individual uses to solve a problem. The task for information system academics is to identify '... the strategy that individuals spontaneously use when solving a problem'(Soloway, Bonar et al., 1983: 853).

3.2 A Framework for the Learning Process

3.2.1 Introduction

Learning computer programming means both 'learning procedures to accomplish various goals and learning the information that is relevant to these procedures' (Van Merrienboer and Paas, 1990: 274). Expert programmers are able to perform a range of procedures without noticeable effort because '... they are able to respond in a highly reflective manner to abstract features of problems' (Van Merrienboer and Paas, 1990: 274). The skills of an expert programmer, however, include more than the automatic application of procedures. When confronted with a new programming problem for which no automatic procedures are available, expert programmers call on the large amount of specific programming knowledge they possess that can be used by more general problem solving methods to reach a solution. Thus:

> *'... besides the development of automatic procedures, the acquisition of highly structured knowledge, or schemata, plays a significant role in learning a skill like computer programming' (Van Merrienboer and Paas, 1990: 275).*

The ability of an expert programmer to apply the knowledge and skills they possess or come to possess does, in part, impact on the successful solution to a problem.

The sequence of steps that is used in a programming subject to instruct students and novice programmers plays an important role in developing schemata and determining a person's ability to solve problems (Linn and Songer, 1991). Turkle & Papert (1992) believe that the earlier image of the computer represented it as a logical machine and computer programming as a technical, mathematical activity that created barriers to many people. 'Both the popular and technical culture have constructed computation as the ultimate embodiment of the abstract and formal' (Turkle and Papert, 1992: 113). However when they observed programmers in action they saw 'they use concrete and personal approaches to knowledge that are far from the cultural stereotypes of formal mathematics' (p.113). Their research

> *'... points to discrimination in the computer culture that is determined not by rules that keep people out but by ways of thinking that make them reluctant to join in. Moreover, the existence of diverse styles of expert programming supports the idea that there can be different but equal voices even where the formal has traditionally appeared as almost definitionally supreme: in mathematics and science'(p.116).*

Instead they believe that:

> *'The conventional route into formal systems, through the manipulation of abstract symbols, closes doors that the computer can open. The computer, with its graphics, its sounds, its text and animation, can provide a port of entry for people whose chief ways of relating to the world are through movement, intuition, and visual representation'(p.115).*

A key objective of any programming course is to ensure that at the end of the subject, the student understands the operations of a system well enough to be able to program and use it (Bostrom, Olfman et al., 1988). This need not imply gaining a detailed knowledge of the computer's architectural structure or the operating system it uses, but more knowledge of the operation of a system at a notional machine level, that is

> *'... an idealized, conceptual computer whose properties are implied by the constructs on the programming language employed. That is, the properties of the notional machine are language, rather than hardware dependent' (Boulay and O'Shea, 1981: 237).*

Using the concept of a notional machine may be beneficial in developing the mental model of a computer in a novice programmer. As Bostrom(1988) states

> *'Understanding of the system implies a complete and accurate mental model of the system to be learned. The role of mental models is to facilitate an understanding of how the system works. They aid the user in making inferences about the system, reasoning about it, and guiding his/her actions. Although there is general agreement in the literature on the importance of mental models in human-computer interaction, opinion is divided on whether or not users build mental models spontaneously' (p.228).*

For a strategy of program instruction based on a notional machine to be effective, the notional machine must conform to two important principles. The notional machine must employ simple concepts (or simplicity), however the '... functional simplicity of a notional machine is not enough on its own to guarantee learnability or ease of use since the majority of machine's actions will usually be hidden, and will have to be inferred by the novice unless special steps are taken'(Boulay and O'Shea, 1981: 238). Novices often lose track of what state a computer has moved to during the implementation of an instruction. This is an important problem with event-driven languages, which comprise a reactive system (ie. a passive form waiting for an event to happen) which is transformed by events. It is easy to understand how a novice programmer can lose track of the system state when programming in event-driven languages. Bostrom et al (1988) outline a mental model formation process in which trainees or students are provided with a conceptual (or notional model) that will help them build mental models (see Figure 3.2).

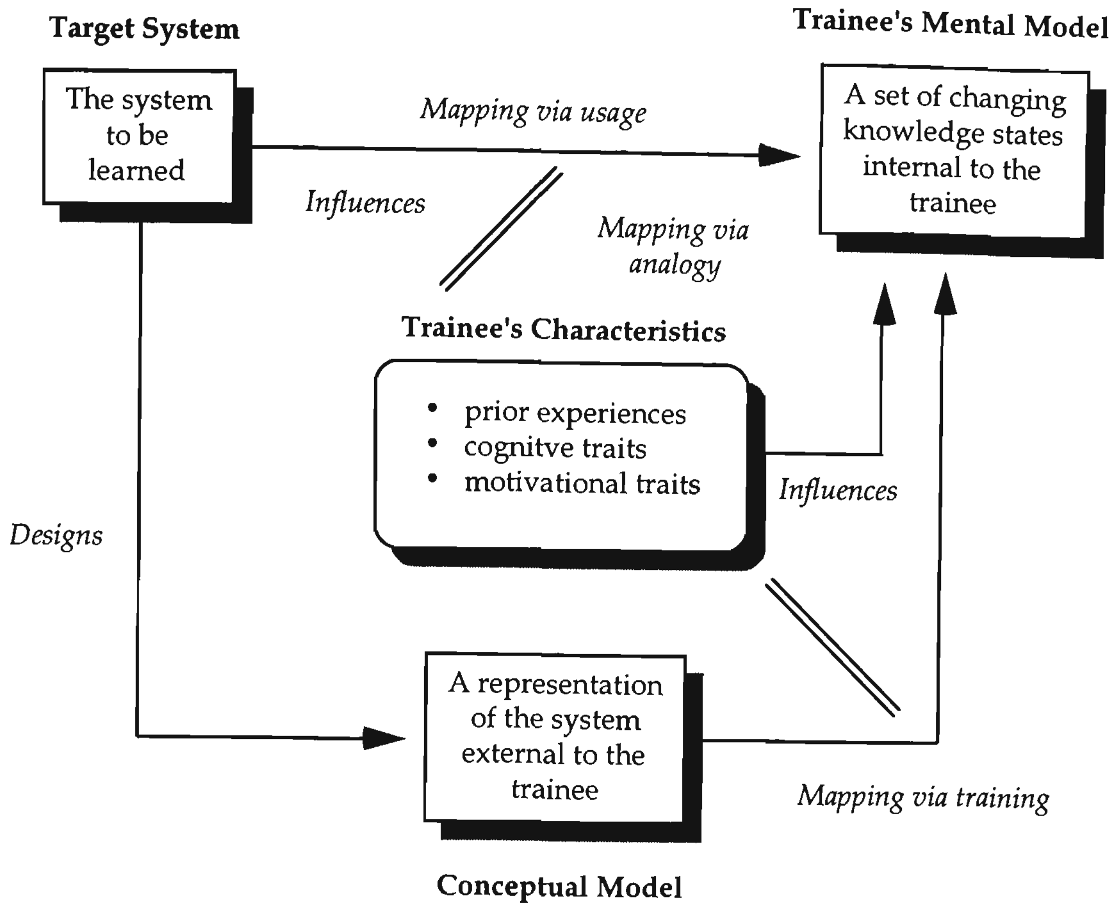


Figure 3.2: Mental Model Formation Process

Source: Bostrom et al (1988: 239)

Object-based, event-driven languages are relatively new and they are not supported by recognised conceptual models and tools that can help a student to understand how such a system operates. Moreover, there appears to be no research which has examined the mental models that a student develops to depict event-driven systems.

Although models have been devised to explain the learning process, it is doubtful that any one model can account for the many approaches, conditions and skills of students in a programming environment. Indeed, 'formal thinking, defined as synonymous with logical thinking, has been given a privileged status which can be challenged only by developing a respectful understanding of other styles, where logic is seen as a powerful instrument of thought but not as the *law of thought.* In this view, *logic is on tap not on top* (Turkle and Papert, 1992: 117). For some people

> *'... what is exciting about computers is working within a rule-driven system that can be mastered in a top-down, divide-and-conquer way. This is the "planner's" approach. ... This approach decrees that the right way to solve a programming problem is to dissect it into separate parts and design a set of modular solutions that will fit the parts into an intended whole. Some programmers work this way because their teachers or employers insist that*

> *they do. For others, it is a preferred approach; to them, it seems natural to make a plan, divide the task, use modules and subprocedures' (p.119).*

Event-driven programming languages may allow students a range of alternatives for systems development. As an example, Braithwaite (1996) identifies either a user interface development or a program development focus. For a program design methodology to be successful, it must assist a student in conceptualising an event-driven system.

3.2.2 Learning Outcomes

In recent years, the main focus of research on teaching and learning a computer programming language has centred on the cognitive demands on students, the cognitive consequences of teaching computer programming and the impact of instructional techniques (McNutt and Lo, 1991). Research by McNutt & Lo (1991), Ang & Lo (1991) and DEET (1990; 1992) has centred on extracting practical recommendations that impact upon existing practices. Although there is little agreement on the key subtasks required in learning to program (Brooks, 1977; Soloway and Elliott, 1984; Shneiderman, 1986; Vessy and Weber, 1986) there is broad agreement on the underlying knowledge that accompanies these tasks. McNutt & Lo (1991) define these as the Four S's Model comprising syntactic, semantic, schematic and strategic knowledge.

The Gagne taxonomy of learning objectives in the cognitive domain comprises the following sequence: facts, concepts, principles and finally problem solving. The sequence begins with the emphasis placed on simple factual learning. Next, students identify concepts behind the relationships between the facts, and this leads on to higher level problem solving skills. Sloan and Linn (1988) identified a chain of cognitive accomplishments from novice to expert programmer using the four types of underlying knowledge (see Figure 3.1).

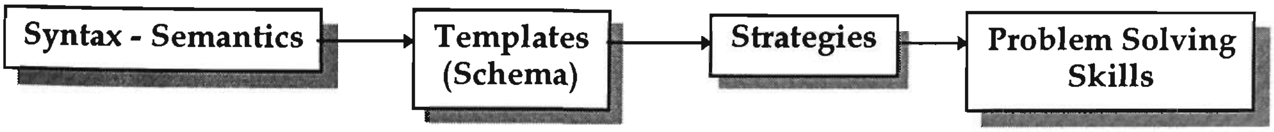


Figure 3.3: Sequential classification of programming knowledge

Source McNutt & Lo(1991: 408)

This sequential acquisition of programming knowledge implies that the novice programmer initially uses the syntax to become familiar with the semantics of a programming language.

Semantic structures are used to develop language independent templates or schema which are then used to develop a knowledge of different strategies and their appropriate uses. The programmer finally develops a range of generaliseable problem solving skills.

Recent research (McNutt and Lo, 1991; DEET, 1992) has found that a syntax immersion approach is used in introductory programming subjects in many tertiary Computer Science and Information Systems courses. The Discipline Review Committee (DEET, 1992) noted that '...the emphasis in computer science courses needs to be less on the teaching of computing knowledge for its own sake, and more on the design, development and application of systems in a practical context' (p.186).

Building on the work of Sime et al (1973), McNutt & Lo (1991) advocated a different approach. According to Sime et al (1973), two psychological processes appear to be used by programmers when they perform design and coding tasks:

i) taxonomising - identifying the conditions that evoke particular actions; and

ii) sequencing - converting the taxa (or actions) into a linear sequence of program code.

According to Vessy et al (1986) these two tasks evoke different psychological processes.

> *'The first involves classifying and sorting elements according to their attitudes; the second involves specifying the taxonomic criteria and associated actions within the syntactic and semantic constraints of the programming language used' (p.49).*

It is possible to match the underlying knowledge requirements to these two tasks (see Figure 3.4):

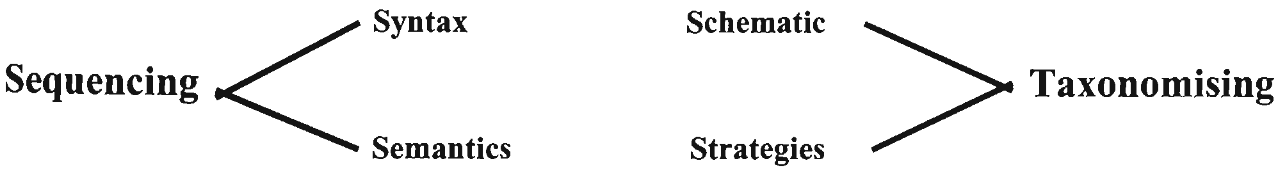


Figure 3.4: Programming Tasks and Underlying Knowledge Bases

Source: McNutt and Lo (1991: 409)

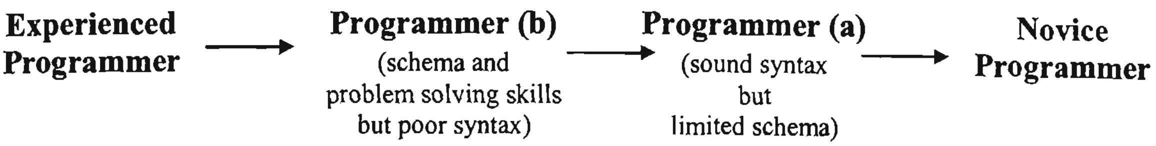
An experienced programmer usually possesses both sound sequencing and taxonomising knowledge. Often, they have developed a large array of cognitive skills and processes which enable them to examine problems and outline a solution in a number of programming languages. Novice programmers, on the other hand, have neither of these qualities. They have a sound knowledge of the syntax in one language, but have not had sufficient experience to develop semantic, schematic or strategic knowledge. The knowledge combinations of each group are outlined in Figure 3.5.

Taxonomising **Sequencing**	**Weak**	**Strong**
Weak	**Novice Programmer**	**Programmer (b)** (Good Problem Solving Skills Poor Knowledge of Syntax)
Strong	**Programmer (a)** (Good Syntax Knowledge Poor Problem Solving Skills	**Experienced Programmer**

Figure 3.5: Programming Knowledge Combinations

Source: McNutt and Lo (1991: 408)

In between these two groups are other types of programmers. Category (a) programmers have a sound knowledge of one particular language but have limited schema. They may be able to identify common routines but only in a particular language and probably through syntax rather than schema. Alternatively, category (b) programmers possess good problem solving skills but have a poor knowledge of programming, that is, they are unable to transfer the taxonomic criteria into a syntactic form. According to McNutt and Lo (1991), programmers can be ranked in the following order of priority according to the combination of skills they possess:


Experienced Programmer
Programmer (b)
(schema and problem solving skills but poor syntax)
Programmer (a)
(sound syntax but limited schema)
Novice Programmer

In terms of desirability of outcome, Information Systems courses should aim at producing students with good schema knowledge rather than syntax knowledge. In practice, however, a syntax immersion approach is unlikely to produce this outcome as students are unlikely to develop taxonomising skills without obtaining some long-term semantic and schematic knowledge.

Using these sequencing and taxonomising tasks as the basis for instructional design, it is possible to identify four didactic approaches to the teaching of programming (see Figure 3.6):

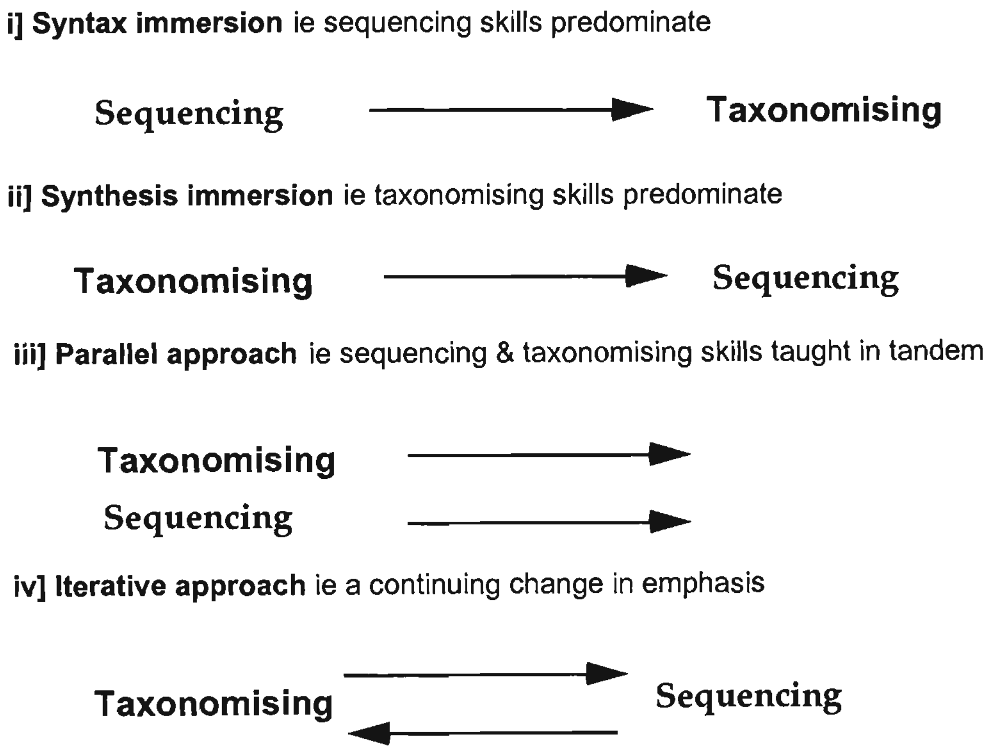

i] Syntax immersion ie sequencing skills predominate
Sequencing
Taxonomising
ii] Synthesis immersion ie taxonomising skills predominate
Taxonomising
Sequencing
iii] Parallel approach ie sequencing & taxonomising skills taught in tandem
Taxonomising
Sequencing
iv] Iterative approach ie a continuing change in emphasis
Taxonomising
Sequencing

Figure 3.6: Didactic Approaches

Source: McNutt & Lo (1991: 409)

The last three approaches are viable alternatives to a syntax immersion approach and, although no one has been identified as superior to the others, they emphasise the need for a broad-based approach to programming instruction. Until recently,

> *'one of the most important skills in programming ie. taxonomising which relies on schematic and strategic knowledge appears to have been neglected. A change in emphasis in relation to the teaching of the component skills*

> *considered to be important in acquiring expertise is needed' (McNutt and Lo, 1991: 412).*

Moreover, with the introduction of new programming paradigms in programming subjects, it is appropriate to examine how these new languages impact on the teaching approaches compared to the current practices used in teaching procedural languages.

3.3 Object-Oriented and Visual Programming Environments

3.3.1 Visual Programming Environments

The move to visual programming environments has resulted in a steady movement away from text-based systems and provided substantial productivity gains particularly for programmers.

> *'This is because the computer's ability to represent in a visual manner normally abstract and ephemeral aspects of the computing process such as recursion, concurrency, and the evolution of data structures through time, can have a remarkable and positive impact on both the productivity of programmers and their degree of satisfaction with the working environment' (Chang, 1990: 145).*

The move to graphic user interfaces from the traditional command/text based interfaces has made it easier for users to use computers and to manipulate them. Yet the interface is not, by itself, sufficiently powerful to enable the end-user to develop or modify applications to meet their own needs. Some knowledge of how a computer achieves a task is essential for the development of applications. Matsumura & Tayama (1990) researched the relationship between visual factors, textual commands and the need for formal study in programming. They found that there is an optimal level where a significant number of small applications could be developed by an end user using a combination of text and visual components (which they refer to as the 50 Step-Module). The more complicated the system, however, the greater the reliance on text (ie. code) which can only be obtained via formal study in programming (see Figure 3.7)

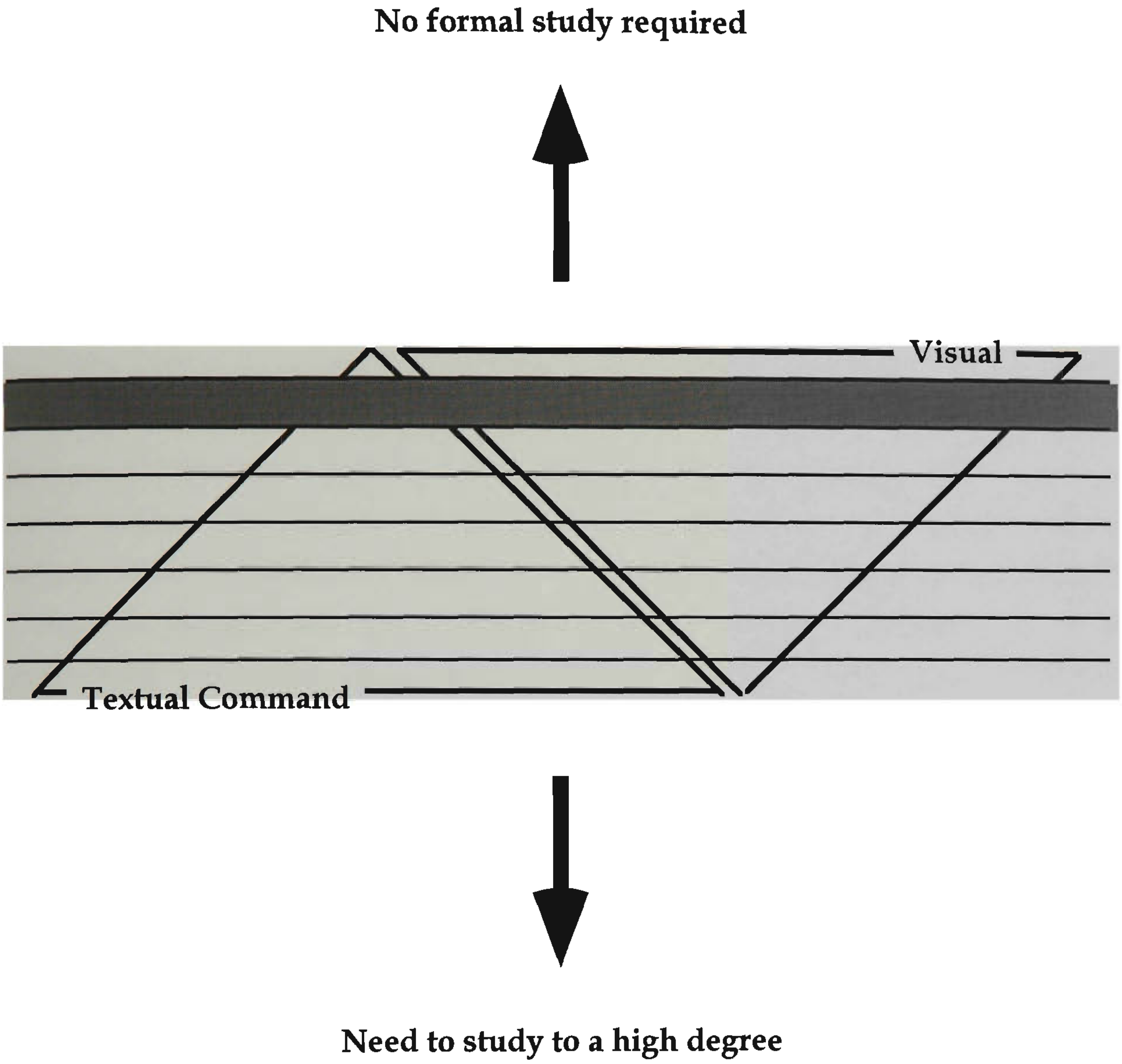


Figure 3.7: Level setting for textual command and visual factors

There is some optimal level for programming where the majority of programming can be undertaken by the end user using visual programming components. Some text/commands remain which provide for a higher degree of control.

Source: Matsumura(1990)

Visual programming environments 'provide visual ways of working with a programming language, whether the language is visual or textual' (Burnett and McIntyre, 1995). In such an environment, the user may revert to traditional procedural methods to alter objects or develop methods in an application. Visual programming environments have become popular in the development of Windows-based applications through languages such as Visual Basic and Delphi. Most popular object-oriented and event-driven programming languages operate within a visual programming environment. According to North (1994), visual programming environments promote a division of labour between specialists who write custom controls and those who build applications. Similarly, Beck (1994) divides the software development role into two parts: the abstractor, usually an experienced programmer creating reusable segments of code, and the elaborator, a person who combines and modifies these segments of code into an application to fit the needs of a user. In visual programming environments, he describes abstractors as component builders (for example, programmers who write DLLs or class libraries in C or C++), and elaborators as solution builders (those who use high-level tools such as Parts, Visual Basic, PowerBuilder, or an application framework in conjunction with low-level DLL components to construct application-level solutions for end users).

Duntemann (1993) believes that 'the language wars are over. The war now, if there is to be one, will be between visual development and traditional development' (p.3) He expresses concern, however, that the architecture and features of many visual programming environments impose a restrictive design vision upon applications written within those languages. In visual programming environments, he states, the highest-level design issues (and hence the shape of the created application) are defined and controlled by the shape of the tool being used.

> *'Templates, multiple inheritance, operator overloading, and things like that have an influence that permeates every corner of the eventual application. These are the sorts of things expressed by the nature of the underlying kernel within a visual language. I don't think they'll be easily changed. The design vision of a visual language will become the design vision of the applications it creates. The code that the programmer writes will almost invariably be local in nature'(p.3).*

Despite the restrictions on design, Beck (1994) believes that visual programming environments have enabled developers to keep pace with the increased demand for applications. When you

> *'introduce software reuse into the equation, however, it isn't one hapless programmer trying to figure out an obscure piece of code - it's a hundred' (p.15). ... 'As programmers we have to remember that creating a working, reliable application the fastest way possible is the goal of the game. Programming as we know it is indeed the most intoxicating kind of fun ... but if something better comes along, we're doing no one any favours by clinging to the old ways' (p.4).*

3.3.2 Object-Oriented Programming

The term object technology has been used widely and is very difficult to define because

> *'it involves a basic paradigm shift in computing and is involved in almost all aspects of computing. There are OO operating systems, OO middleware products, OO languages, OO methodologies, OO CASE tools, OO 4GL tools and OO databases. People talk about using objects to develop basic definitions of numbers and strings and they talk about encapsulating large COBOL applications as objects. Clearly this technology encompasses a wide variety of different techniques' (Harmon, 1995: 81).*

Many products are described as object-oriented or object-based but there are large differences in their level of functional abstraction and support for distribution development based on Common Object Request Broker Architecture (CORBA) (see Figure 3.8).

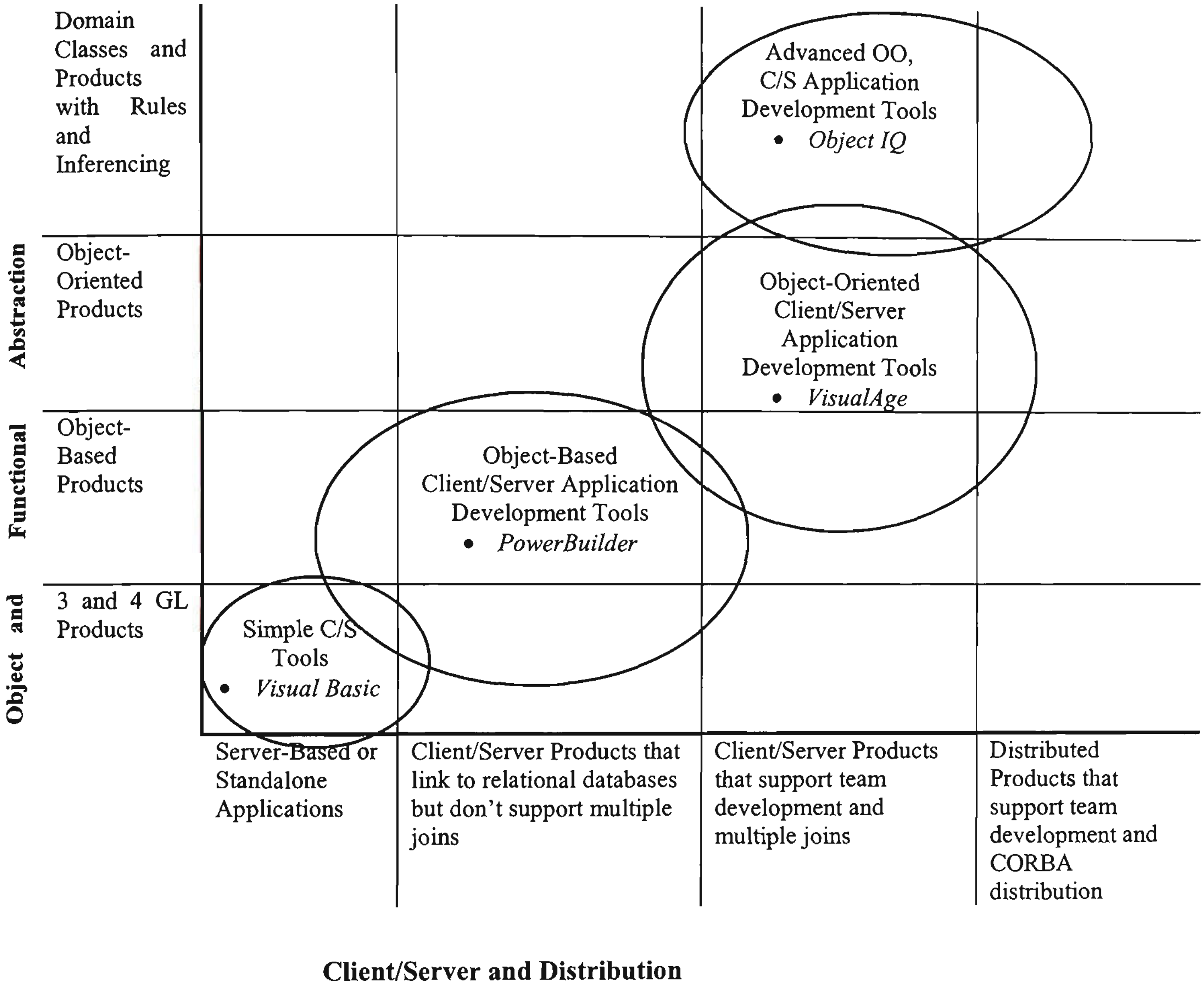

Object and Functional Abstraction
Domain Classes and Products with Rules and Inferencing
Object-Oriented Products
Object-Based Products
3 and 4 GL Products
Advanced OO, C/S Application Development Tools
• Object IQ
Object-Oriented Client/Server Application Development Tools
• VisualAge
Object-Based Client/Server Application Development Tools
• PowerBuilder
Simple C/S Tools
• Visual Basic
Server-Based or Standalone Applications
Client/Server Products that link to relational databases but don't support multiple joins
Client/Server Products that support team development and multiple joins
Distributed Products that support team development and CORBA distribution
Client/Server and Distribution

Figure 3.8: A matrix comparing object-oriented products based on abstraction and distribution

Source: Harmon (1995)

Yourdon (1994) defines a system built with object-oriented methods as one 'whose components are encapsulated chunks of data and function, which can inherit attributes and behaviour from other such components, and whose components communicate via messages with one another' (p.2). Moreover, he believes that object-oriented techniques are the most important development since the introduction of structured techniques during the 1970's and 1980's.

Object-oriented methodologies support the concepts of the object model, which consists of classes, their properties and relationships, and messages between objects. A *class* is any concept important to a problem, such as an employee or a bank account, and it localises in one place everything that is known about that concept, including its relationships with other classes. An *object* is an instance of a class.

The execution of an object-oriented program is based on the communication between objects via messages. A *message* is a request for an object to perform one of its operations. In order to do anything with an object's data, the program must send a message to that object. To process data in an object one must define a *method*, which is a definition of an object's behaviour.

This message-based approach encourages the notion of encapsulation, whereby different parts of a program are viewed as being relatively isolated from one another, resulting in a very modular system (Güvenir, 1995; Kennedy, 1995). This high degree of modularity is the basis for claims that 'object-oriented programming is a technique which has many benefits such as easy construction of complex systems, availability of inherently reusable objects [and] reduction in total life-cycle software cost' (Sudbury, 1995: 40).

Reuse of components is an attractive element of object-oriented programming but typically prepackaged components only provide rudimentary functionality '... so building a complex application still involves a great deal of new programming effort'(Pancake, 1995: 33). According to Sudbury (1995) '.. some developers of object-oriented systems become so infatuated with the idea of increased productivity through software reuse - which object technology promises to deliver - that they lose sight of the primary goal of useability' (p.18). Moreover,

> *'Reusability does not come without a price tag. Typically, it is only after an OO application is complete that the developers understand which objects might have broader use. Those objects then must be restructured through a process known as* generalisation, *that in turn may require revisiting several earlier stages of object design'(Pancake, 1995: 33).*

Navrat (1994) believes the ability of the object-oriented paradigm to support abstraction and generality makes it more attractive than procedural languages as a teaching platform in introductory subjects

> *'In procedural programming, only abstraction was originally supported, and this only partially, viz. the procedural one by the language construct for procedure declaration. ... Much of the apparent success of the object oriented paradigm can in fact be explained by its support to both abstraction and generality, surpassing the procedural paradigm by one entire dimension' (p.21).*

Sudbury (1995), however, believes that an abstract approach can cause problems for software developers as well as for novice programmers.

> *'An application created from general purpose objects can be difficult to maintain because it is harder for a maintenance programmer to see immediately how the objects relate to the business problem. Therefore, it can take the programmer longer to understand how to change the application. This can cause an extremely serious maintenance problem and result in a mess of the magnitude of the messes we sometimes inherit from previous generations of software(p.18).*

Udell (1994) suggests that object technology has failed to deliver on the promise of reuse, instead, off-the-shelf components that can be used to build real applications in a hurry have emerged in event-driven programming language such as Visual Basic. According to Sudbury (1995), there needs to be a balance between abstraction and generalisation, which fosters reuse on the one hand, and a focus on making the system simple and comprehensible so that it is reusable on the other hand.

> *'It might well be preferable to err in favour of the comprehensible. Software that is eminently useable tends by default to become reusable. The requirements of reusability and designing systems that are useable are not mutually exclusive and need not even be at cross-purposes' (p.18).*

Programs designed using Visual Basic are described as event-driven programs as the program responds to different events which are usually initiated by the user, although they can also be initiated by the program itself. In Visual Basic, a person builds up screens by defining forms containing Windows controls such as buttons, text boxes, scroll bars, labels, and other user-interface objects. The programmer sets the properties of each object and generates code (called methods) that execute in response to events. Typical events which a user might perform include a movement of the mouse, a single or a double click of the left (or primary) mouse button, or a keystroke (or a combination of keystrokes) from the keyboard.

Duntemann (1993) states that

> *'Visual Basic is a fully structured language, with procedures, functions, and every control-flow structure present in Pascal. They even write it now using the same standard indentation conventions that Pascal people have been using since Year One. What's significant in Visual Basic has nothing to do with any individual programming language at all: It has placed the bulk of the Windows UI machinery behind a curtain, from which it emerges in its naked terror only in dire need. The language behind Visual Basic's forms and controls could be any language at all'(p.5).*

Similarly, Schurmann (1994) believes that an

> *'event-driven programming language has a more-solid structure and clearer constraints than any procedural language. Much of the logic and structure that*

traditional systems enforce via an architecture are intrinsic to the event-driven approach and thus to the programming language itself' (p.33).

3.4 Object-based Languages in Introductory Programming Subjects

In the U.S.A., the Computer Science model curriculum (Tucker, 1991) and the Information Systems model curriculum, IS'95 (Longenecker, Feinstein et al., 1994), provide broad guidelines on the content of subjects taught within their respective disciplines, however they do not stipulate the language platform that colleges and universities should adopt. According to Levy (1995) Pascal was the most popular language in first year programming subjects in Computer Science (known as CS1) in the 1970's '... due to two pragmatic factors: the invention of the personal computer, and the availability of Pascal compilers' (p. 21). In the 1980's universities turned to other languages that provided students '... with an opportunity to learn to use a tool that will help them address problem solving issues, prepare them for Computer Science study, and offer them a taste of the Computer Science discipline' (Levy, 1995: 21). Evans (1996), borrowing from Gersting (1994), noted that CS1 subjects in universities have moved away from a depth first introduction to programming subject towards an emphasis on problem solving, specifically, with an '... emphasis on introductory software engineering mechanisms, procedural abstraction, and (some) data abstraction' (p.13). A recent survey of first year programming subjects in Computer Science (Reid, 1994) identified 139 colleges and universities in the United States of America that were not using Pascal. Levy (1995) conducted a follow up survey and found that most popular languages amongst this group of colleges and universities was C, Scheme, ADA and C++. Event-driven programming languages such as Visual Basic were not mentioned by name in the results of the survey.

In a more recent survey by Tesch et al (1996), 72 four year institutions were identified from the Microsoft Developer Curriculum Project Web site (http://biblio.isu.edu/ms/syllabi/vb.html) as using Visual Basic. The subjects, however, '... were not strictly programming or Information Systems program courses' (p.2), the most popular being beginning programming. Nevertheless, object-oriented programming and event-driven programming are rapidly being adopted by universities and institutions as part of their curriculum (Levy, 1995; Fienup, 1996; Kölling and Rosenberg, 1996; Tesch, Tesch et al., 1996) and some evaluations of these courses have been reported in journals and at conferences.

3.4.1 Reasons for Introducing Object-based Programming

Many universities and colleges appear to have introduced object-oriented or object-based programming in a response to the changing demands from business for information systems graduates, or in response to the perceived failings of a procedural language such as Pascal

(Decker and Hirshfield, 1994; Smith and Kelly, 1994; Levy, 1995; Fienup, 1996; Tesch, Tesch et al., 1996). At Purdue University, for example, the Computer Technology Department has 'always tried to pay attention to the new trends in business information systems and provide an education that will prepare graduates for the marketplace at the time of their graduation' (Lisack, 1996: 1). Visual Basic was introduced at that university because

> *'one of the big causes of change has been the overwhelming acceptance of the Windows environment, so that users have become accustomed to, and even grown to expect, a graphical user interface. In addition, the object-oriented paradigm has received widespread acceptance in various phases of the system development process, and data bases are regularly used, both on the PC and on mainframes' (p.1).*

DeClue (1996) believes that universities should adopt an object-oriented platform for reasons related more closely to the classroom, that is, for 'the intuitive, real-world problem solving aspect of the paradigm, the simplicity and organisational nature of the paradigm, and the problems caused by not switching to the object-oriented paradigm' (p.232). Osbourne & Johnson (1993) state, 'from an educational standpoint, one of the most useful benefits is that the model of the co-operating objects is intuitively close to human experience and expectation' (p.101). Moreover, Decker & Hirshfield (1994) point out that the real world modelling nature of the object-oriented paradigm raises the level of abstraction, thus enabling programmers to call upon some of their own real world experiences. They sum up the feelings of many writers who have sought alternatives to a procedural language:

> *'... all but our very best students appeared to us to lack what are currently regarded as basic software engineering skills. Of those students who produce working programs, relatively few wrote programs that could be described as modular, readable, traceable, testable, or maintainable. Still fewer were capable - even after our CS1 course - of analysing, specifying, designing, and managing even modest-sized programs of their own. Rhetoric and current programming texts notwithstanding, our students were not being trained as problem solvers, and were not developing skills that we regarded as essential to both their subsequent course work and to their careers. Our frustration with this situation led us to question not only how we had been teaching our majors, but also what we had been teaching them' (p.51).*

Similarly, Kölling et al (1995) believe there are several strong arguments in support of an object-oriented language in first year:

- *'Object-orientation encourages well structured programming, which is one of the most important lessons we try to convey to first year students.*
- *Re-using existing code can be taught in addition to the development of new code, leading to a more realistic perception of the tasks expected of a programmer.*
- *The ability for students to make use of ready-made objects in their applications opens a wide range of possibilities for real-world and interesting examples and exercises.*

- *Important software development concepts, such as evolution and re-use, can be introduced and experienced through object-oriented techniques at an early stage.*
- *Problems with the paradigm shift in moving between object-oriented and non-object-oriented environments seem to be reduced' (p.173)*

3.4.2 Evaluation of Object-based Languages in Introductory Programming Subjects

Cross & Philips (1996) believe that the object-oriented paradigm is too abstract for students at an introductory programming level and most of the features of an object-oriented programming language would have to be omitted. Moreover, Wallingford (1996) found that most popular object-oriented programming languages are not always appropriate for teaching purposes.

> *'These factors conspire to make learning problematic. Student attention is focused on a discomfiting language syntax, which distracts them from the already difficult task of learning a new problem-solving paradigm' (p.28).*

In seeking a language suited to data abstraction Cross and Philips (1996) believe there is 'a need for a methodology beyond procedural abstraction, but without encompassing the depth of object-oriented programming, even though one may elect to use a subset of an object-oriented language to implement it' (p.20).

In one of the few references where Visual Basic was employed as the platform for teaching introduction to programming, Neal & Lorents (1995) found that students had difficulty visualising the operation of a program due in part to the absence of design methodology. They state:

> *'A great difficulty arises in getting the student to visualise the entire program structure and to understand what portion of the code is executed at what time. An object hierarchy is used with lower level calls to sequential routines to explain the structure. Students have little difficulty understanding the objects and associated properties, events, and methods. However, understanding the possible scenarios of execution under the event model was hit and miss' (p. 58).*

At a conference forum on the experiences in teaching object-oriented design and programming in introductory programming subjects, Chi (1996) reported that

> *'although many believe that object-oriented is more than procedure orientation, there is still a gap between "objects" in programming languages and objects in the real world. Real world objects are more concurrent and dynamic. They are classified according to common properties as well as to haves and have-nots. Program objects are much more restricted' (p. 406).*

Biddle & Tempero (1996) found that students who learn the object-oriented paradigm quickly become familiar with two broad theoretical principles: that object-oriented programming is a natural way to design programs and that object-oriented programming provides good support for reuse, but have difficulty designing programs in practice. Their research found that, in particular, 'learners often have difficulty in understanding when to use the key object-oriented programming technique of *inheritance*, and even more difficulty with *multiple inheritance*' (p.217), one of the key components for software reuse.

Cross and Philips (1996) use the term *Abstract-Oriented Language* to describe languages which work within an object environment (object-based languages) and provide, according to them, the benefits of an object-oriented programming language although in a limited, static form. 'Abstract-Oriented Language reusability via generic packages is easier to understand, although clearly less powerful, than OOL reuse through inheritance. ... there is a vast supply of relevant, easy to understand abstractions which can be implemented in an Abstract-Oriented Language' (p.21). Similarly, Lisack (1996) found that 'by working with the built-in components to set properties at design-time or run-time, and using the methods already defined for those components, (students) became comfortable with the concept of objects' (p.5).

Despite these differing views on the most appropriate language or the depth of study possible, a number of authors (Smith and Kelly, 1994; DeClue, 1996; Kölling and Rosenberg, 1996; Lisack, 1996) remarked on the positive motivational nature of the object-oriented platform. This may be due in part, to the correlation between a person's problem-solving strategies in the real world and those required in object-oriented development (Decker and Hirshfield, 1994). DeClue (1996), however, believes the blend of object-oriented languages with a visual environment is 'a singularly motivational feature of object-oriented languages' (p.235). Moreover, Osbourne & Johnson (1993) reported that a 'windowing environment adds a certain excitement and an increased measure of entertainment to the programming process' (p.102). In one of the few accounts of the use of Visual Basic at an Australian university, Smith & Kelly (1994) found that students who were taught Visual Basic in elective units at the University of Ballarat were highly motivated.

> *'Students of both genders show strong enthusiasm to succeed. Tasks become achievable and not an arduous, laborious exercise as is perceived in C procedural programming classes. There is a desire to explore and enhance. Almost without exception students teach themselves more than the core concepts taught in lectures' (p.4).*

3.4.3 Program Design

On the important issue of program design Wick (1995) is critical of object-based and object-oriented platforms in the first programming subject.

> *'Students are not motivated through the use of concepts before they are asked to consider their implementation. This pitfall is not unique to using C++ or object orientation, but is intensified by their use. ... The end result is that by the time students are "ready" to study life cycle issues, they have become so deeply*

> *convinced they are not needed that they lack the motivation for serious study' (p.323).*

In a contrasting view, Decker and Hirshfield (1994) believe that object-oriented programming '... supports directly many of the software engineering concepts that are among the most difficult to convey in procedural terms: code reuse, encapsulation, incremental development and testing, and, of course, program design' (p.51).

According to Adams (1996) many colleges and universities that have adopted hybrid languages, ie. those languages with a combination of imperative and object-oriented features, have also been forced to address the issue of which software design methodology to employ. Adams (1996) believes neither an Object-Oriented Design-Early approach, that is, one which teaches object-oriented design from the outset, or an Object-Oriented Design-Late approach, that is one where structured design is taught in the early subjects and a switch is made to object-oriented design in latter subjects, are successful. Instead, Adams (1996) advocates an *object-centred* design approach - 'an approach that focuses on objects (without inheritance) at the outset, and expands gradually, culminating in object-oriented design' (p.79). In this approach, lecturers

> *'deliberately constrain the problem-space at the outset in order to introduce the students to object-oriented design using objects of the fundamental types and the operations that can be applied to objects of those types. Doing so gives our students experience successfully applying this software design to problems, building confidence'(Adams, 1996: 79).*

According to Wick (1995), the concept of software reusability is difficult for students to grasp in introductory programming courses, as object-orientation is a large-scale organisational strategy and cannot be taught within the time constraints. Instead, he advocates an approach where much of the design process is performed by the lecturer while students experience the powerful results of object-oriented design through the implementation of a system by only programming the methods.

On evaluating the use of Smalltalk in introductory programming subjects, Bellin (1992) states that 'Smalltalk is not just a language, but is really an environment. Deceptively simple syntax and concepts can be lost in a sea of classes and browsers' (p.135). Similarly, Osbourne & Johnson (1993) found that 'most students get hopelessly confused by the bewildering number of classes and methods' (p.104). Guzdial (1995) found that students learning to program in an object-oriented programming language such as Smalltalk tended to design programs with relatively small sections of reusable code despite efforts to teach an object-oriented strategy explicitly. He believes students have a centralised mindset which causes them to write programs that have:

- *'Centralised control: a single leader object makes all or most decisions*
- *Global or leader-centred communications: either all objects communicate with all other objects, or all messages in the student's classes have the leader object either as the receiver or sender' (p.182)*

Using work from Resnick (1992), Guzdial (1995) discovered that it is often easier for students to initially develop procedural-like programs with a central controlling object in an object-oriented language neglecting issues of code reuse. 'Centralised models are efficient, easy to understand, and are often accurate depictions of a problem domain - in many situations, there is central control and global or leader-centre communication' (p.183). According to Guzdial (1995), students tend to write decentralised programs when they truly understand the components of the model, something which is often neglected by lecturers when teaching object-oriented programming.

Some universities have moved to an object-oriented programming language enabling students to learn about event-driven programming concepts without sacrificing structured programming concepts and conventional program design (Lisack, 1996).

> *'The typical steps in program development (understanding the problem, designing the logic, coding the problem, testing and debugging the problem, and finalizing the documentation) were presented and emphasised throughout the course. Structured programming concepts were presented and enforced as much as practical. Modularization is a natural concept to introduce, because the very nature of an event-driven programming environment forces the programmer to break the program into modules coinciding with the events of the program' (p.4).*

Similarly, Decker & Hirshfield (1994) believe that an object-oriented platform provides a framework within which conventional algorithm design (functional decomposition, stepwise refinement, control structures) *must* still occur.

> *'One of the beauties (to us) of the OOP paradigm is that by merely describing classes one imposes an organisation on a problem that not only clearly reflects those aspects of the real world being modelled, but also serves as a first crack at an algorithmic description of those same aspects. By associating functions with classes, you have already taken a giant step, and for many students this is the toughest step by far, toward breaking a problem down into more manageable subproblems. At that point, much of what we have always taught our students about algorithm development comes into play' (p.53).*

Fienup (1996) found that, not only was there a need to emphasise object-oriented design instead of procedural (or top down) design with a switch to an object-oriented language in a CS2 subject, but there was a need to expose students to software engineering concepts at an earlier stage. He found that this change of emphasis brought with it the need to assign larger programming projects at an earlier stage in the course so that students could practice object-oriented design. In addition, Fienup (1996) believes there is a need to place greater emphasis on testing and debugging strategies at an earlier stage in programming subjects, a view supported by Lisack (1996) who found that students became 'more aware of the importance of program testing because of the interactive nature of most assignments' (p.5). Similarly, Bellin (1992) noted that 'testing and debugging become an integral part of the development process, not something to be done "after coding"' (p.134).

Kölling & Rosenberg (1996) contends that the program development environment is possibly more important than the object-oriented language but that many of these environments have created difficulties for teaching object-oriented technology. 'Despite the large emphasis placed on the choice of a particular object-oriented language, there seems to be little discussion of the programming environment, which may well have more of an influence on the learning experience than the choice of language' (p.84). Moreover, they believe that the graphical visualisation techniques used to model program structure are lacking in an object-oriented programming environment. 'The most advanced professional object-oriented development environments, such as Visual C++ or Delphi, use graphical support only to build the user interface of an application, but neglect the internal structure of the program itself' (p.85).

Chi's (1996) final remarks summarise the dilemma facing many lecturers evaluating their programming subjects - 'What kind of balance between object-oriented and procedure orientation is appropriate during the first two years remains an interesting question' (p.406).

3.5 Summary

This section of the literature review has highlighted the importance of programming in the information systems curriculum. It has reported on the use of event-driven and other object-based programming languages in introductory programming subjects, and it has established the importance of program design as part of these subjects.

The popularity of information systems courses has come about to a large extent, from the increase in demand from business for graduates who can implement and support computer information systems. Moreover, programming continues to be an important part of the tertiary Information Systems curriculum. As the popularity of non-procedural programming paradigms continues to increase in business, however, tertiary institutions have been forced to consider adopting similar programming platforms into their information systems courses. This chapter has reported on the experiences of those who have used event-driven languages in their introductory subjects. When a procedural language was used, there was general agreement of the subject content and design methodology that should be used which had been refined throughout the years. A switch from one programming paradigm to another, however, requires lecturers to review the subject material and the mode of delivery. Structured programming, long established as the programming method to use in the teaching of procedural languages, is not appropriate for event-driven programming languages. There is no recognised program design methodology for the development of event-driven systems and many lecturers are now faced with the problem of how to teach the concepts of programming using this platform. What is needed, therefore, is a program design methodology that can assist with the teaching of event driven programming in introductory programming subjects. Such a methodology is proposed and examined in practice as part of the following chapters.

Chapter 4 Research Rationale and Methodology

Information systems as a discipline may be viewed as incorporating multiple paradigms, as is found in other emerging disciplines from the social sciences. It is not surprising then, that a range of research methods have been used in the study of information systems. This chapter begins by outlining the reasons for choosing a case study approach using interpretive methods. Proceeding sections then outline the research design, details of the research setting and the methods and tools used to undertake the research. The benefits and limitations of each of the methods used in the study is discussed. The chapter concludes with a detailed description of the techniques used for data analysis.

4.1 Qualitative and Quantitative Research Methods

Traditionally, quantitative methods have dominated Information Systems research. There is, however, growing interest in the use of qualitative research methods brought about, in part, 'by a general dissatisfaction with the type of research information provided by quantitative techniques'(Benbasat, Goldstein et al., 1987: 369). Quantitative methods used in information systems research are entrenched in logical positivism derived from the scientific tradition and characterised by repeatability, reductionism and refutability. They are founded on a belief that values can be separated from facts through the use of scientific methods (Galliers, 1991). Understanding phenomena is thus primarily 'a problem of modeling and measurement, of constructing an appropriate set of constructs and an accurate set of instruments to capture the essence of the phenomenon' (Orlikowski and Bardoudi, 1991:9). Positivist researchers adopt an epistemological view that assumes it is possible to explain and predict phenomena within the social world by seeking regularities and causal relationships between various related elements. This view assumes that knowledge is infallible and can be proven, and allows empirical research methods to be adopted to test an hypothesis (Klein, Nissen et al., 1991 :1). Moreover, Klein, Nissen et al. (1991) argue that researchers adopting this perspective believe that scientific inquiry is 'value-free', enabling them to objectively predict outcomes but not partake in subjective opinion.

4.1.1 Qualitative Research Methods

Qualitative methods, on the other hand, hold that reasoning, ideas and spontaneous individual insights are the primary source of knowledge (Klein, Nissen et al., 1991 :5). Thus, in choosing a research approach,

> *'the researcher is in fact choosing which aspects of a phenomenon he wishes to focus on. The researcher constructs the form and nature of the phenomenon through the world view he adopts to do the research. So the researcher's 'assumptions and values are deeply embroiled in the phenomenon - even in the very selection of a research approach'(Orlikowski and Bardoudi, 1991:16).*

Moreover, 'like natural science, qualitative social research is pluralistic. A variety of models may be applied to the same object for different purposes' (Kirk and Miller, 1990: 12).

Most studies of computer systems based on quantitative methods produce either technical, economic, or performance measures (Kaplan and Duchon, 1988 :573). In contrast, qualitative research using interpretive methods:

> *'..assume that people create and associate their own subjective and intersubjective meanings as they interact with the world around them. Interpretive researchers thus attempt to understand phenomena through accessing the meanings that participants assign to them. ... Interpretive studies reject the possibility of an "objective" or "factual" account of events and situations, seeking instead a relativistic, albeit shared, understanding of phenomena. Generalising from the setting (usually only one or an handful of field sites) to a population is not sought; rather, the intent is to understand the deeper structure of a phenomenon, which it is believed can then be used to inform other settings' (Orlikowski and Bardoudi, 1991:5)*

Renata Tesch (1990: 3) describes any data that are not quantitative, and so cannot be expressed in numbers, as qualitative data. Qualitative methods collect data which are '..rich in description of people, places and conversations, and not easily handled by statistical procedures. Research questions are not framed by operational variables; rather they are formulated to investigate topics in all their complexity, in context' (Bogdan and Biklen, 1992 : 2). Unlike quantitative approaches to research in which there may be clearly laid down procedures to be followed, there is no single agreed set of established procedures for conducting qualitative research. 'The only agreement we would find among qualitative researchers is that analysis is the process of making sense of narrative data' (Tesch, 1990: 4).

Qualitative research places greater emphasis on the need to understand the why and how rather than the what of a problem or how much. Researchers adopting these methods are more concerned to understand individual perceptions of the world. They seek insight rather than statistical analysis. Qualitative researchers doubt whether social facts exist and question whether a scientific approach can be used when dealing with human beings (Bell, 1987).

In this thesis, it is argued that qualitative methods are more appropriate for a study of event-driven programming environments in information systems courses than quantitative methods. Moreover, it is argued, the methods that are most appropriate are those of the analysis of programs and documents, observation, and the collection of oral interviews from students and people teaching and using visual, event-driven languages. These methods will be used in a case study with a small sample size. Yin (1989 :20) states that

> *'The case study is preferred in examining contemporary events, but when the relevant behaviours cannot be manipulated. Thus, the case study relies on many of the same techniques as a history, but it adds two sources of evidence not usually included in the historian's repertoire: direct observation and systematic interviewing. ... the case study's unique strength is its ability to deal with a full variety of evidence - documents, artifacts, interviews and observations.*

4.1.2 Case Studies

Tesch (1990: 59) describes a *cognitive* map of four major qualitative research types ranging from the most structured to the least structured and holistic:

- the characteristics of language
- the discovery of regularities
- the comprehension of the meaning of text/action, and
- reflection.

She describes the comprehension of the meaning of text/action as research that seeks to discern meaning. She divides this category into:

- the discerning of themes (of commonalities and uniqueness), and
- interpretation.

Three methods she suggests in the interpretation category are the hermeneutics, case study and life history methods. Yin (1989: 23) defines a case study as

> *'an empirical inquiry that:*
> - *investigates a contemporary phenomenon within its real-life context; when*
> - *the boundaries between phenomenon and context are not clearly evident; and in which*
> - *multiple sources of evidence are used'.*

Tesch (1990) describes the characteristic mode of analysis of case studies as dialoguing with the data as this typically calls for interpretation of one piece of data and what it means, rather than dealing with regularities and patterns across many pieces of data of a similar kind. Case studies make sense of the data they collect, and the result of the analysis is 'an instructive portrait, the case study having been selected for its "typical" nature in the first place' (p.94). According to Yin (1989) a case study approach relies on many of the same techniques as a history, but it adds two sources of evidence not usually included in the historian's repertoire: direct observation and systematic interviewing. Case study analysis can be undertaken using qualitative or quantitative methods, however, 'the case study's unique strength is its ability to deal with a full variety of evidence - documents, artefacts, interviews and observations'(Yin, 1989: 20)

Yin (1989 :52) argues that the arguments in favour of a single experiment also justify a single-case study. Mintzberg (1983 :107) takes a similar view

> *'What, for example, is wrong with samples of one? Why should researchers have to apologise for them? Should Piaget apologise for studying his children, a physicist for splitting only one atom?'*

Using a small sample size can often yield more useful research results compared to results from a large sample particularly when the researcher attempts to study a large sample within limited time (Mintzberg, 1983 :108). Further, Mintzberg (1983 :113) states that

> *'we uncover all kinds of relationships in our "hard" data, but it is only through the use of this "soft" data that we are able to "explain" them, and explanation is, or (sic) course, the purpose of research'.*

The case study in this thesis will be used in the explanatory phase of the research. The essential part of any case study approach to qualitative analysis 'is for the researcher to seek connections and explanations, not just to describe' (Tatnall, 1993: 19). Tesch (1990: 85) argues that researchers in this field '...try to find out more than just *what is*, they also try to find out *why* it is'. Tatnall (1993: 19) suggests that 'what is important is the seeking of explanations, and that the analytical procedures used must be designed to facilitate such theorising'.

4.2 Interpretivism

Interpretivists view science as a

> *'... convention - related to societal norms, expectations, and values - engaged in the search for understanding. Science uses whatever tools, techniques and approaches which are considered appropriate for the particular subject matter under study' (Klein, Nissen et al., 1991:2).*

The world is not conceived of as a fixed body of objects, but rather as 'an emergent social process - as an extension of human consciousness and subjective experience' (Burrell and Morgan, 1979 :253). Walsham (1993: 5) states:

> *'Interpretive methods of research start from the position that our knowledge of reality, including the domain of human action, is a social construction by human actors and that this applies equally to researchers. Thus there is no objective reality which can be discovered by researchers and replicated by others, in contrast to the assumptions of positivist science. Our theories concerning reality are ways of making sense of the world and shared meanings are a form of intersubjectivity rather than objectivity. Interpretivism is thus an epistemological position, concerned with approaches to the understanding of*

> *reality and asserting that all such knowledge is necessarily a social construction and thus subjective'.*

The interpretive philosophy is premised on the epistemological belief that 'social process is not captured in hypothetical deductions, covariances, and degrees of freedom. Instead, understanding social process involves getting inside the world of those generating it' (Rosen, 1991). The underlying premise, therefore of interpretivism is that 'individuals act towards things on the basis of the meanings that things have for them, that meanings arise out of social interaction, and that meanings are developed and modified through an interpretive process' (Boland, 1979 :260).

Interpretive information systems research sees 'the pursuit of meaning and understanding as subjective and knowledge as a social construction' (Walsham, 1993: 21). This view of social constructionism does, of course, have an effect on research methodology. Tatnall (1993), borrowing from Goodson and Mangan (1991) suggest that there are two closely related sets of methodological implications.

> *'Firstly, ..that in order to adequately analyse an observed social process, the meanings ascribed to that process by the participants must be taken into account...[and].. that there are ethical imperatives, as human subjects of the research must be recognised as "fully engaged participants in the process", and that their dignity and autonomy as individuals should be respected'(Tatnall, 1993: 21).*

Orlikowski (1991: 11) believes 'while the results of natural science do not impinge on and change the nature of the phenomena studied, the results of social science do enter into the discourse of everyday human reality, and clearly can and do transform the nature of these phenomena'. Borrowing from Giddens (1987), Orlikowski (1991: 11) notes,

> *'in the social sciences, unlike in natural science, there is no way of keeping the concepts, theories, and findings of the researchers "free from appropriation by lay actors." Clearly, information systems research enters into the very constitution of the phenomena it studies'. Indeed, a major goal of information systems research is to have an impact on information systems practice; that is, the findings of information systems research are intended to inform and improve the development and use of information systems in organisations'.*

Interpretive researchers argue that, on a number of levels - conceptual, methodological, and substantive - all researchers are inherently implicated in the process of research. The researcher 'can never assume a value-neutral stance, and is always implicated in the phenomena being studied' (Orlikowski and Bardoudi, 1991:15). Researchers' prior assumptions, beliefs, values, and interests always intervene to shape investigations.

4.2.1 Case Study and Interpretivism

In arguing for case research, Walsham (1993: 247) believes

> *'if one adopts such a theoretical stance on the nature of knowledge, in-depth case studies are the only feasible method for empirical research'.*

Moreover, the need for context-dependent research in information systems necessitates interpretive research using case study methods (Kaplan and Duchon, 1988).

The need for case studies arises out of the desire to examine humans within their social settings. 'The case study allows an investigation to retain the holistic and meaningful characteristics of real-life events' (Yin, 1989: 14). Interpretive researchers avoid imposing externally defined categories on phenomena in a case study. Instead, they attempt to derive meaning by in-depth examination of and exposure to the phenomena of interest. Benbasat (1987: 369) states

> *'there are three reasons why case study research is a viable information systems research strategy. First, the researcher can study information systems in a natural setting, learn about the state of the art, and generate theories from practice. Second, the case method allows the researcher to answer "how" and "why" questions, that is, to understand the nature and complexity of the processes taking place. ... Third, a case approach is an appropriate way to research an area in which few previous studies have been carried out. With the rapid pace of change in the information systems field, many new topics emerge each year for which valuable insights can be gained through the use of case research'.*

Kaplan (1988: 573) quoting Lyytinen (1987) makes a similar appeal for case studies on the grounds that 'this research strategy seems to be the only means of obtaining sufficiently rich data' and because the validity of such methods 'is better than that of empirical studies'.

4.2.2 Interpretivism, Case Studies and Theory Development

In the search for causal relations, positivist research focuses on the reliability and validity of research procedures by adopting a 'predefined and circumscribed stance towards the phenomenon being investigated' (Orlikowski and Bardoudi, 1991:12). Often, the simplification and abstraction needed for good experimental design and the objective measurement of phenomena, can remove enough features from the subject of study that only obvious results are possible. Kaplan (1988: 572) notes that 'the stripping of context buys objectivity and testability at the cost of a deeper understanding of what actually is occurring'. In the search for universal laws, historical and contextual conditions, as possible triggers of events or influences on human action, may be over-looked. 'Neglecting these influences may reveal an incomplete picture of information systems phenomenon.' (Orlikowski and Bardoudi, 1991:12). Rowan (1973: 210) notes that 'research can only discover one-sided things if it insists on setting up one-sided relationships...You only get answers to those questions you are asking'.

Yin (1989) argues that it is a 'fatal flaw' in doing case studies to conceive of statistical generalisation as the method of generalising the results of the case.

> *'This is because cases are not "sampling units" and should not be chosen for this reason. Rather, individual case studies are to be selected as a laboratory investigator selects the topic of a new experiment....Under these circumstances, the method of generalisation is* analytic *generalisation, in which a previously developed theory is used as a template with which to compare the empirical results of the case study'(p.38).*

He argues that case studies, like experiments, are not generalisable to populations or universes. In these circumstances, the investigator's goal is to 'expand and generalise theories and not to enumerate frequencies' (p.21), that is, to generalise analytically not from statistical evidence. Moreover, Walsham(1993: 6) argues

> *'In the interpretive tradition, there are no correct and incorrect theories but there are interesting and less interesting ways to view the world. A reader might well ask "interesting to whom"? An author can only respond in the first instance that the theories which he or she presents are interesting to themselves and may be interesting to others. However, although the use by an individual author of a particular theoretical approach derives no doubt from his or her personal experience and insight, the testing of the value of these insights to others can be carried out by exposing the approach through verbal and written discourse to enable broader judgements of value to be made. Theory can be compared, evaluated and improved by this form of public testing; the result is not the generation of 'best' theory, but the creation of intersubjectivity tested theoretical approaches, considered of value to a broader group than a single individual'.*

Qualitative methods are characterised by an attempt to avoid prior commitment to theoretical constructs or to testable hypotheses formulated before gathering any data since the process of interpretive research is dialectical not linear (Kaplan and Duchon, 1988; Myers, 1994). Similarly, qualitative methods, by their very nature, use data to both pose and resolve research questions (Kaplan and Duchon, 1988).

4.3 Research Design

The research is divided into two phases. The first is exploratory research and the second is explanatory research using a case study. Figure 4.1 illustrates the relationship between the different methods used to accomplish the research.

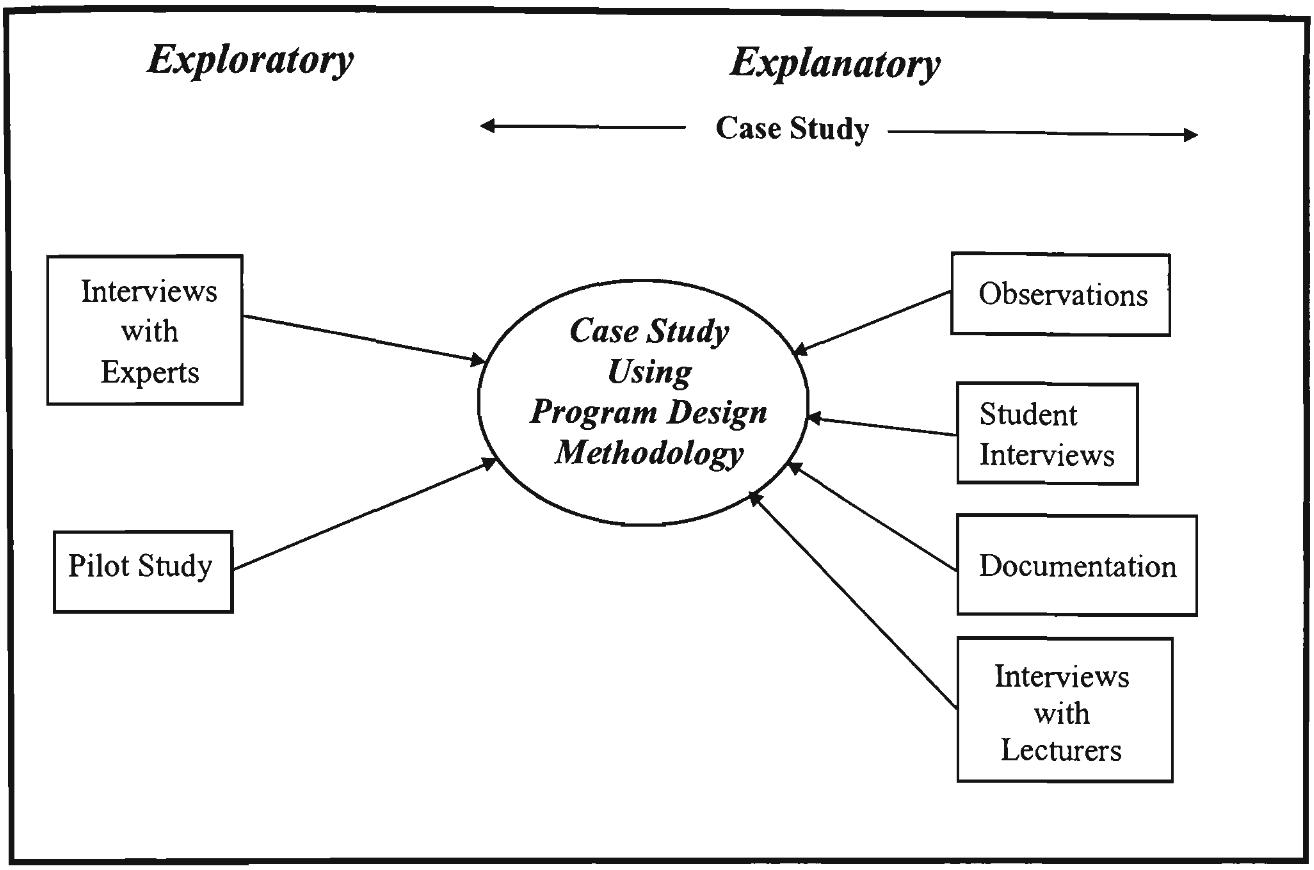


Figure 4.1: Structure of Research Design

4.3.1 Exploratory Phase

Research which is exploratory aims to formulate more precise questions for future research. Zikmund (1991: 32) believes exploratory research enables the researcher to obtain a better understanding of the dimensions of the problem. Moreover, a pilot case study can help with conceptual clarification, and refine data collection techniques and procedures (Yin, 1989 :80). In this research, two methods were used in the initial phase; interviews with lecturers from Australian institutions who had introduced or evaluated event-driven programming languages as the platform for introductory programming subjects in Information Systems, and a pilot study conducted on post-graduate students in an introductory programming subject at the Victoria University of Technology.

4.3.1.1 Pilot Study

In 1995, with permission from the Programming Committee of the Department of Business Computing at the Victoria University of Technology, Visual Basic was used for the first time as the implementation programming language in the Introduction to Programming subject for postgraduate students. In the pilot case study for this research, students were requested to complete an information sheet at the beginning of the semester and a survey-type questionaire at the end of the semester. In the pilot study, most of the emphasis was placed on observational data rather than on documentary material and interviews which, as expected, resulted in only a small amount of data.

4.3.1.2 Interviews with Experts

The exploratory research began by interviewing program co-ordinators from three different Australian universities who had investigated the possible use of an event-driven programming language in their respective introductory programming subject. Moreover, each co-ordinator had extensive experience in the teaching of programming in many institutions at various levels and with curriculum development. Only one university finally decided to use the event-driven paradigm. The pseudonym and background used for each expert is given below:

Lecturer D Lecturer D was well known for his work in computer education at many tertiary institutions. He has written books pioneering program design. Originally from a COBOL background, he introduced Visual Basic to first year students in 1995 and now conducts professional development programs in event-driven programming for members of the Australian Computer Society.

Lecturer B Lecturer B has extensive teaching experience and has widely researched the issue of women in computing particularly in programming. Lecturer B, evaluated Visual Basic as the platform for their introductory programming subject before settling on VisualAge.

Lecturer R Lecturer R had been actively involved in the introduction of computers into secondary education in Australia. He had been a computer education consultant for a state Department of Education. He is now a senior lecturer in a business computing department in an Australian university. He is the author of many books, several on Visual Basic.

Lecturer R, together with Lecturer T from VUT (who was interviewed later), had used evaluation sheets to refine the content of the materials they used and the processes that they had adopted to teach Visual Basic in respective second and third year subjects. Both lecturers reported on the various approaches they had used in their subjects and on the attitudes of students towards the use of event-driven programming languages based on results of these evaluation sheets.

4.3.2 Case Study

4.3.2.1 Research Context

The Victoria University of Technology was chosen for the case study for several reasons:

- It was one of only two universities in Victoria that had adopted an event-driven, visual programming language as the introductory language in an Information Systems course.
- Event-driven programming languages were being used in both undergraduate and postgraduate information systems courses
- Event-driven programming languages were being used at a number of year levels in the undergraduate course

The case study involved 80 students divided into five workshop groups over two semesters in 1996 for the introductory programming subject for undergraduates and post-graduates.

4.3.2.2 Research instruments

Four instruments were used in the case study for this research project:

- Documentation
- Interviews with lecturers
- Interviews with students
- Observations

4.3.2.3 Documentation

Personal documents that respondents used or wrote themselves played an important role in this case study. The documents acquired from informants included computer program listings, pre-program documentation (such as pseudocode and screen layouts), test plans and program documentation.

The most important use of documentary information in case studies is to support and supplement evidence from other sources (Yin, 1989 :86). Documentation helps to track the path of a person as they progress through the steps to produce a computer program. The documentation can provide clues as to how a person is thinking at some stage or how they view the process of developing a program. At times, inferences can be made from documents. For example, by observing the number and types of control buttons a person places on a screen layout, this may raise new questions about that person's understanding of program control. These inferences become clues worthy of further investigation .

Similarly, documentary data obtained from lecturers and course co-ordinators often gave clues to new directions or provided supporting evidence for information obtained from other sources.

4.3.2.4 Student Interviews

The interviews were firstly used to gather descriptive data, in the informants own words, which provided insights into how they worked with event-driven languages and the program design methodology. Secondly, the interviews were used to verify points gained from other data collection methods.

A semi-structured style of interview questioning was used. Students were given a prepared problem in the interview which they were asked to solve (see Appendix 1). Observations were made while a student attempted to solve the problem and the documentation was retained.

Table 4.1 provides a summary of the characteristics and background of the students selected for interviewing.

Table 4.1: Summary of Students Interviewed

	Bachelor of Business (Computing)	**Graduate Diploma of Business Computing**	**Bachelor of Business (Accounting)**	**Bachelor of Arts/Business Computing**	**Total**		
	Full-time	*Part-time*	*Part-time*	*Full-time*	*Part-time*	*Full-time*	*Overall*
Male	4	2	1		3	4	7
Female	5	2		1	2	6	8
Total	9	4	1	1	5	10	15

Care was taken to select students with a range of experiences and from different courses. Fifteen students were interviewed (seven male and eight female over two semesters. Appendix 2 outlines details of each student and their pseudonym.

4.3.2.5 Lecturer Interviews

Interviews were conducted with experienced lecturers who taught the program design methodology to students in workshops at the Victoria University of Technology. Three lecturers were interviewed and were given the pseudonyms Lecturer M, Lecturer O and Lecturer T respectively.

A small tape recorder was used when interviewing students and lecturers and brief notes were taken. A transcript of each interview was made and included interview notes where appropriate. Each transcript of an interview was entered as a new word-processor document, which was printed and stored ready for future analysis. A copy of the interview was sent to each lecturer for verification.

4.3.2.6 ***Observation***

The observation of individuals and groups in natural settings was another method used to obtain primary data. Students were observed in workshops and when they were attempting the prepared problem in the interview.

At the end of the research there was a large amount of data; interview transcriptions, documents and notes on observations from interviews and workshops.

4.3.2.7 ***Triangulation***

A major strength of case study research using qualitative methods is the opportunity to use many different sources of evidence (Yin, 1989). Indeed, the 'opportunity to use multiple sources of evidence far exceeds that in other research strategies, such as experiments, surveys and histories' (Yin, 1989 :96). Multiple methods can provide a richer, contextual basis for interpreting and validating results (Kaplan and Duchon, 1988 :575). Similarly, 'the use of multiple sources of evidence in case studies allows an investigator to address a broader range of historical, attitudinal, and observable issues (Yin, 1989 :97). 'Mixing methods can also lead to new insights and modes of analysis that are unlikely to occur if one method is used alone' (Kaplan and Duchon, 1988 :582).

By using multiple sources of evidence in a case study the researcher can reduce the problem of unreliable data and facilitate the development of converging lines of inquiry. Combining different methods introduces both testability and context into the research.

> *'Collecting different kinds of data by different methods from different sources provides a wider range of coverage that may result in a fuller picture of the unit under study than would have been achieved otherwise. Moreover, using multiple methods increases the robustness of results because findings can be strengthened through triangulation - the cross-validation achieved when different kinds and sources of data converge and are found congruent or when an explanation is developed to account for all the data when they diverge' (Kaplan and Duchon, 1988 :575).*

Triangulation is a process of checking inferences drawn from one set of data sources by collecting data from other sources (Trauth and O'Connor, 1991 :134). Triangulation '...is a term borrowed from land surveying, and means simply that you get a better view of things by looking at them from more than one direction'(McNeill, 1990 :123). Moreover, the 'triangulation of data from different sources can alert researchers to potential analytical errors and omissions (Kaplan and Duchon, 1988 :582).

In undertaking this study, triangulation has been used, where at all possible, to verify findings from respondents. In many cases, important issues raised by key respondents have been verified by asking similar questions of others, and in other cases it has been possible to triangulate these findings through interviews and observation of students in workshop settings. In some cases, it has been possible to triangulate from accounts published in journal articles. The reliability of data has been achieved by interviewing the key developers of visual, event-driven subjects in Information Systems education in Australian universities, observing and interviewing students in subjects, and collecting documentation.

4.3.3 Limitations of Methods

There are limitations to any method of research and some of the arguments against a case study approach have been noted earlier. The following points, however, are noted with regard to a case study approach and the instruments used for this research.

A criticism of a case study approach is that it is usually restricted to a single setting or organisation and there is 'difficulty in acquiring similar data from a statistically meaningful number of similar organisations' (Galliers, 1992: p.67). Although this view is often championed by quantitative researchers advocating statistical generalisations, a researcher must be aware that the case study itself does not ensure that the results have a more generalisable applicability (Craig, 1996). As Walsham(1993: 15) argues

> *'from an interpretive position, the validity of an extrapolation from an individual case or cases depends not on the representativeness of such cases in a statistical sense, but on the plausibility and cogency of the logical reasoning used in describing the results from the cases, and in drawing conclusions from them'.*

One of the major difficulties for a researcher who adopts a case study approach is that the strategies and techniques for analysis are not well defined (Miles, 1983; Yin, 1989). As Tesch(1990) points out, the early case study researchers left few details of their analysis procedures. Although this enables a researcher to be flexible in seeking explanations, the success of the case study is dependent upon the skills of the researcher. According to Myers (1994), however, the question of the validity of case study data is a hermeneutical one: does the interpretation and explanation of the case study make sense? As Kirk and Miller (1990: 21) state,

> *'In the case of qualitative observations, the issue of validity is not a matter of methodological hair splitting about the fifth decimal point, but a question of whether the researcher sees what he or she thinks he or she sees'.*

A recognised problem when conducting interviews is interviewer effect, that is, the interviewer affecting the nature of the data he or she is trying to measure. A rigid interview structure may go part way to alleviate the problem but at the cost of obtaining appropriate data. As Tatnall (1993: 46) notes: '... all that the interviewer can do is be aware of this difficulty and take as much care as possible not to lead the subject into making particular responses'.

Direct observation has many advantages over other forms of data collection. Observations enable the researcher to:

- collect original data when it occurs
- obtain information that participants may ignore
- obtain information that participants believe is not relevant
- observe events in their entirety and in context with other activities

The process of observing participants, however, requires the researcher to be present and this may cause the individual to alter his or her actions for that period of time. Moreover, observations are limited to learning about present activities, not past or future ones, and should not seek the opinions, preferences, intentions or attitudes of an individual.

4.3.4 Ethical Issues

This research was undertaken within the boundaries established in Victoria University of Technology's Code of Conduct of Research (1993). Moreover, the research was conducted with the approval of the Programming Committee of the Department of Business Computing. All students in the introductory programming subject were informed of the research at the beginning of each semester. Potential interviewees were approached and given an explanation of the nature of the research. If they consented to be interviewed they were handed a letter explaining the nature of the research and asked for written consent (Appendix 3). Before the interview commenced interviewee's permission was sought to tape the interview. Photocopies of documents were only obtained from students who gave permission. Pseudonyms are used for students to protect their identity.

4.4 Data Analysis

4.4.1 Qualitative Analysis

Qualitative data analysis is a complex process. 'Qualitative data tend to overload the researcher badly at almost every point ... and ... there is no easy way to record qualitative data other than through running notes' (Miles, 1983 :118). However, raw notes ordinarily mean little or nothing to anyone other than the researcher and need to be segmented and organised into categories.

Tesch(1990: 113), along lines derived from Bogdan and Taylor (1975) defines data analysis as

> *'a process which entails an effort to formally identify themes and to construct hypotheses (ideas) as they are suggested by data and an attempt to demonstrate support for those themes and hypotheses'(Tesch, 1990 :113).*

Moreover, qualitative analysis is not the last phase in the research process but is done in almost constant interaction with data collection. Tesch (1990) suggests, in fact, that analysis begins as soon as a first set of data is gathered, resulting in a process where the collection of data and its analysis drive each other along. The analysis ends only after new data no longer generate new insights, that is, the process 'exhausts' the data. Moreover, 'manipulating qualitative data during analysis is seen as an "eclectic" activity, there is no one "right" way' (Tesch, 1990 :114).

While qualitative research may not set a standard format for the presentation of results, the analysis of data must, however, preserve the concrete details of case study data and present them in a coherent manner. The question of reliability of case study data does not depend on the replicability of data but 'on a process of data reduction and analysis that results in something that can be generally recognised as representing (or perhaps *re-presenting)* the data' (Tatnall, 1993 :18). Tesch (1990: 304) argues a similar position:

> *'... successful qualitative data reduction, while removing us from the freshness of the original, presents us instead with an image that we can grasp as the "essence", where we otherwise would have been flooded with detail and left hardly a perception of the phenomenon at all'.*

4.4.2 Analytic Strategy

Yin (1989) advocates a general analytic strategy detailing priorities on what to analyse and why. The role of a general analytic strategy is 'to assist the researcher to choose among different techniques and to complete the analytic phase of the research successfully' (Yin, 1989: 106). One of the two general strategies he outlines is developing a case description, that is, developing a descriptive framework for organising the case study. Without one of these strategies a case study analysis will proceed with difficulty or even place the case study in jeopardy (Yin, 1989 :109).

Tesch (1990 :98) too, advocates a systematic and comprehensive analytic process although not as rigid as Yin's approach. She outlines one general strategy, interpretive/descriptive, which, together with theory-building, are sub-groups of interpretational analysis. 'All types of interpretational analysis are descriptive ... and ... contribute to theory' (Tesch, 1990 :98). The purpose of interpretive/descriptive analysis is the identification of patterns or discerning meaning.

Mintzberg (1983), in what he defines as inductive research, identifies two essential steps in the analysis of qualitative data. The first is 'detective' work, that is, the tracking down of patterns and consistencies. The second step is the creative leap in relevant directions. Tesch (1990) describes the first step as data organisation, that is, the 'detailed examination' or 'the identification of themes', and the last step as data interpretation, that is, the 'determination of its essential features' or ' understanding' or 'construction of propositional statements'. Yin (1989 113), in discussing the analysis of case study data refers to this process as 'explanation building' - the gradual building of an explanation, usually in narrative form, through a process of refining a set of ideas.

All qualitative data analysis processes, however, operate on a conceptual and a concrete level simultaneously (Tesch, 1990 :99). Although all steps in the analytic process require intellectual involvement, manual tasks must be performed as data are segmented and organised into categories. This must be performed in some concrete way.

4.4.3 Data Analysis for this Research

In this study, a process of sorting or coding was used which Tesch (1990) defines as de-contextualizing and re-contextualizing. In the process of reading the data documents, smaller parts in the data documents - items, incidents, or meaning units were identified. Tesch defines this as 'a segment of text that is comprehensible by itself and contains one idea, episode, or piece of information'(p.116).

It was decided not to use text analysis software in handling the data from interviews and observations. Instead, a manual method of sorting and coding the printed documents of summary data was used. The analysis began by carefully reading through the documents to identify features and patterns in the data. In many cases, this was a slow process and involved several passes of the documents. Items in documents were then tagged for further processing. Using the cut-and-paste function of the word processor, relevant text segments were transferred to a new document. The tagged text was then assembled from this document into topics or categories. In some cases, it was necessary to contact the interviewee again for clarification of a particular point, or to ask an additional question. At the end of the process, there were a number of topic areas with supporting evidence from observations and interviews.

The next chapter presents the findings of the exploratory section of the research and outlines a program design methodology for teaching event-driven programming. Chapter 6 outlines the results from the case study where a program design was used by students in an introductory programming subject.

Chapter 5 Developing a Methodology for Event-Driven Programming

5.1 Introduction

This chapter will firstly present the analysis of the interviews with experts from three Australian tertiary institutions. The experts describe the outcomes from decisions to either adopt or reject an event-driven language in their respective introductory programming subject. One university adopted an event-driven language, one university retained a procedural language, and the third moved to an object oriented language.

The analysis of data from the experts identifies a number of key factors influencing the choice of language for first year programming subjects. Moreover, the changes to the content and teaching approach, and the need for a program design methodology is outlined. The chapter concludes by outlining a program design methodology for the teaching of event-driven languages to novice programmers.

5.2 Overview of Courses and Subject Characteristics

The data was gathered from experts at three tertiary institutions. In each institution, a single introductory programming subject is taught as part of their 24 unit degree, usually over three years of full time study. Two institutions offer degrees from within their respective Business faculty, while the third offers a degree from the School of Information Technology and Mathematical Sciences. In each of the degrees, emphasis is placed upon aspects of information systems and systems development, with subjects including systems analysis, systems design and database systems proceeding the introductory programming unit. Lecturer D outlines the overall aims of their subject and, in this case, justification for using Visual Basic:

> *reinforce all the other things we teach them in information systems - I mean a straight up the middle of wicket system subject. You might talk about user friendliness and screen interaction the sort of human computer interaction stuff. If you do that in terms of theory, you know in terms of week three of your systems analysis and design, they will sit there and nod their head and say yeah yeah yeah but it don't mean all that much to them. But if you get them on a PC and they actually have to put together a user interface and see how awful it is if it has 17 colours, you cannot read it properly etc. and all of that comes home much better to them if they are doing it than if you give them Chairman Mao's 47 rules on designing screens and it was something you cannot do with the COBOL's and the Pascals of the world because they have such dreadful character-based screen interaction. So one of the aims is to get them thinking about what computing systems do by getting them to write some software to make them do it and Visual Basic is such an easy way for them to write software (Lecturer D).*

Two of the institutions are highly reliant upon students from other departments to supplement numbers in their introductory programming subject and in other proceeding subjects. There is a resultant impact on the breadth of skills amongst students within the introductory programming subject:

> *We had a range of students in our first year programming courses, the pure computer science people and people who came from business, applied science, education, humanities who are doing the unit as an elective. So we had to step down to the middle of the path if we were going to use one language or we might have to stream two groups (Lecturer B).*

At these institutions, attracting and retaining students from other areas of their university is an important factor in the decisions about the language used in the programming subject.

5.3 Factors Influencing the Choice of Language

The literature analysis examined some of the deciding factors in the choice of an event-driven language for an introductory programming subject. It is relevant to briefly explore some of the factors that influenced the tertiary institutions analysed in this research before examining the impact of an event driven language on the subject aims, content and delivery. This section outlines three factors which are significant in the choice of the language: student and gender issues, technological change, employer perception and the need to update courses, and changes to the aims and objectives of the introductory subject.

5.3.1 Student Interest and Gender Issues

As mentioned above, the introductory programming unit at two of the institutions is viable only if students from other departments enrol in the subject. One of the experts was quite candid about the need to make the subject attractive to students:

> *So what we have to do is to make the computing bit interesting and sexy and attractive and keep them in there otherwise they leave. What has happened in the past I guess with the old COBOL and C syllabus, the die-hards have stuck in there but really a lot of them were pretty marginal and at best we would have had 30 students by the time we got through to the second year(Lecturer D).*

It is, however, important to note that this was only one of the reasons for changing to an event-driven language at this university. Most importantly Lecturer D was concerned that, without students completing both programming units they would gain the impression that the course was not a genuine information systems degree:

> *However for the kids who come in to do the Bachelor of Business Computing we get them to do that in the first year with a view of hooking them into the MIS major so they get some real computing to do in the first year otherwise they think we came along to be computing students and really there is only one mandatory computing unit in the first year (Lecturer D)*

It would be unrealistic to believe that the change to an event-driven language in itself could change student's perception of programming. Nevertheless, all of the experts held the view that if students believe that the language is modern and more relevant then they are more likely to enjoy the programming experience. This in turn, they believe, leads to better outcomes and helps attract and retain students in a course. Lecturers at one university found:

> *In terms of how it has worked out it has been a fantastic success. I think, partially perhaps to Visual Basic itself but partially I think just to get the students away from studying languages like COBOL. I haven't anything against COBOL, C, Pascal where kids look at a subject that is badged programming 101. I cannot do that, you have to be a rocket scientist to be a programmer and they all go off and do other things. ... you can put together applications that are like the applications that they use, so they have a bit of colour and menus and buttons to push and mouses to click - all the sort of things they see on normal software (Lecturer D)*

In a later interview with Lecturer M from Victoria University, he compared the attitudes of students who used Visual Basic in the introductory programming subject to students who used Pascal:

> *The biggest difference is that students like it more. They seem more excited about doing it. I had very few students grad dips or whatever who produced anything that they were really excited about in Pascal, whereas I think some of the ones, particularly doing the grad dip last semester, thought it was terrific and they could produce all sorts of stuff and whatever. So I think there is a motivating thing about it. I don't think it is more difficult than Pascal, I think once they get the hang of it and understand how the thing works that they get much more out of it (Lecturer M).*

The largest university of the three examined, does not appear to have as many problems attracting students to their course as do the two smaller universities. They retained PICK Basic as the platform for their introductory subject although they moved away from it in second and third year subjects

Research by Craig (1997) found that female students are more likely to leave Information Systems courses than male students. It was therefore encouraging to note the interests of female students was an important issue for some experts. Lecturer B noted a considerable change in the attitude of female students:

> *They see something they have developed very quickly, they see it happen and that is quite exciting - it is motivational, a sense of achievement and it was really one of the nice things that came out in the first semester last year was, instead of the females saying, 'I cannot cope, I don't understand this, it is all Chinese', to, 'Hey, this is easy.' It was a lovely change to observe and so there was a sense of achievement and one of the other reasons we swapped to VisualAge and Smalltalk was that we felt that all the students would come in on an equal level ... So here we had, for the first time, people on an equal level and the females and the weaker students able to achieve something very quickly very early on (Lecturer B).*

Moreover, Lecturer B believes that a visual programming environment is a vital component for motivating students:

> *It is the capacity to see a screen in the application functioning that I think allows the students to build confidence with using the product and using computers, because some have not used computers and so, from that point of view, the screen work is quite important in the visual approach (Lecturer B).*

Lecturer M observed that the enthusiasm for programming had changed amongst most students and especially female students:

> *The girls seem more interested in it than use to be. I know in the systems implementation, a lot of the girls said to me this is terrific, now I had work experience last year and I spent all my time doing RPG or the other and I come back and this is terrific. Because they had some programming skills because they had been out there doing that and they did not have to worry about the programming part of it, they could just see the presentation part of it was so much better. So I think it will motivate the students to do more programming. Pascal wasn't very exciting because you could never get it really look very good. It took you 50 lines of code to draw a box on the screen (Lecturer M).*

Lecturer D assessed the impact of the move to an event-driven language on student attitudes in the following way:

> *It has been good, they really do like it, and we have attracted a number of students from computing and maths who have come over do this. We have attracted students to this course with more than any other single thing that I can think of (Lecturer D).*

5.3.2 Technological Change, Employer Perception and the Need to Update Courses

The literature analysis highlighted the importance some institutions, particularly in the U.S.A., place upon adapting quickly to business needs. One of the universities in this study retained PICK Basic for their introductory programming subject believing it made their course more commercially attractive:

> *The department has a strong commitment to producing graduates who have mainstream commercial products under their belt. The students are going into industry, industry ready and because our research has shown there is no Pascal industry, we do not teach it. So we teach structured programming concepts but we do it with commercially real systems like Oracle and Visual Basic and Pick Basic. We were a niche marketer in the old days for PICK but now students could not get our degree without sound Oracle knowledge and sound Visual Basic knowledge and one other language and those are all chosen from the top dozen or so languages or environments in the computer based industry (Lecturer R).*

Lecturer B *recognised the need to introduce an object-oriented type approach* for similar reasons. Moreover, Lecturer D indicated that students do not see a subject with a procedural program as relevant in an Information Systems course and describes languages of this type as backroomish:

> *When you are teaching them programming with the Pascals, the C and the COBOL's of the world, whilst you might tell them that it is good for their soul, they sort of cannot see it because they write Pascal or COBOL with a bit of screen interaction, but they know that is not screen interaction like anybody really does and most of the work that you tend to do with them. They think, anyway, that it a bit boring and a bit backroomish and they cannot see any sort of relevance to other of the sort of software (Lecturer D).*

5.3.3 Aims and Objectives of Introductory Subject

As outlined above, two of the universities moved away from using a procedural language in their respective introductory programming subject without, they believed, compromising the original aims of the subject, and partially in recognition of making the subject more palatable for students and more commercially attractive. Lecturer D highlights this point in the following statement comparing programming in information systems curriculum to computer science:

> *It depends upon what you think is the programming task and to me the programming task is putting the do it instructions on whatever tool you have got. Now if the only tool you have got is a text file which is going to then be compiled by the compiler it seems if you followed their logic you might going back and saying well even that's not far enough old fellow. I mean how does the few lines of code that says hello world get converted back into binary strings and so on. I think all of us need to make use of contemporary tools without building them.. If you were producing kids who were going to work for Microsoft and build the next generation of Visual Basic then I might be on there side. These kids need to be builders of these things not just users so they better know how you build your own VBXS and your own forms. It is not what I believe is what Information systems course are in the business of doing. For me the kids are going to go out there, they are going to have these tools, then lets just roll up our sleeves and use the tools.*

It was interesting, therefore, to compare these views with those of Lecturer R. In a passionate response in support of procedural languages he stated:

> *It seems to me that you can learn algorithmic solutions or algorithmic representation of solutions, you can learn that without a third generation procedural language but why bother. And those languages are powerful teaching tools because they give students immediate reinforcement as to whether their algorithm is sound and I think it is just foolish to ignore one third of our discipline or to try to teach it other than by using what is a very powerful tool. Now if you think of procedural programming in a 3rd gen language as something where the syntax is important or some part that is not common to multiple languages or is unique to that language then I think that is equally stupid. But people will to the end of time gain control over their environment by learning how to convert problem solutions into an algorithmic form. It doesn't matter whether that solution is executed by a machine or a set of workers following a set of instructions it is still a powerful problem solving technique. Now the rest of it the aggregation type ideas of encapsulation in object-oriented, higher level statements and whatever, if you want to you can work at that level, that does not matter as long as you are still exposing students to algorithmic representations of problem solutions because if you ignore that you are ignoring a great big range of ways of looking at the world.*
>
> *If you ask me is 3GL programming important the answer is no. But it is a heck of a way, a really effective proven way of teaching algorithmic solutions.*

5.4 Changes to the Content, Teaching Approach and Design Methodology

This section reports on the impact of the adoption of an event-driven language on the content and delivery of the subject material. Moreover, it details the program design methodology used or, as will be seen, the ad hoc approach used in the development of event-driven systems amongst novice programmers. Although all of the experts expressed a desire for the introductory programming units to teach fundamental programming skills, they recognised that a move to the use an object-oriented or event-driven language would result in changes. Experts from two universities were comfortable with changes in the teaching approach and the movement to a visual programming environment, while the third university believed that event-driven languages were unable to adequately reinforce a professional approach to application development. Lecturer R stated:

> *The advantage is that you can show everything, and the disadvantage is that they show everything. You can show everything so that they can see all the elements of modern tools but the problem is that they are all there, all at once. And if you try to show that you can go from start to 100 mph in 2 seconds with this thing then you are losing sight of what you are suppose to be teaching which is professionalism. You have to have a strategy and the strategy is professionalism in application development.*

In contrast to this view, Lecturer D did not feel it necessary to retain a procedural language:

> *I don't really have any problems with that. I am always a little wary, a bit like the Irish situation, are you a Protestant or a Catholic, you cannot be both you have to be one or the other, I would prefer not to enforce that. If we can get some benefit out of Visual Basic or Delphi or those sorts of languages which are somewhat object oriented, I don't see that you then have to say Ah, well then you cannot use your procedural methods to do that, in the long run people will develop software the easiest way that they can see. People are not going to give up all of their procedural programming habits because someone's says they are now working in C++ and now you have to be object oriented, so forget about everything the Jesuits taught you back at school when you were doing procedural programming they don't apply any more.*

5.4.1 Teaching Approach

Many of the lecturers discovered, unexpectedly, that the move to an event-driven language often shifted the emphasis of their introductory programming subject. Lecturer R hinted to this point when discussing the use of Visual Basic in second and third year subjects:

> *In some cases it has replaced other languages. In some cases it is just allowing people to demonstrate something that they would not otherwise be able to demonstrate. In some cases it has replaced theory with theory and practice. It has made the subjects richer. As an example in an analysis and design subject, in the old days we had them separated, in design you could probably talk about screen design or connectivity now they do it. It has enriched the delivery.*

Lecturer D compared the teaching of the introductory programming subject when COBOL was the platform to the current use of Visual Basic:

> *It was a bit of a learning experience not so much learning the language but interesting how differently, if at all you would teach programming using Visual Basic to the way I have been teaching it for 30 years using COBOL. Because if we stick to COBOL what you would typically do there you would teach them not only the syntax of the language but you would also spend a lot of time teaching them programming techniques.... What I did with Visual Basic was to go the other way round and I adopted very much a hands on approach, to say, Now put in a form, good, now put on two labels right now put in three text boxes right now put in a command button right now behind this put a bit of code that does that and it was all done intellectually like that and I guess in retrospect it sort of ended up to be product based rather than programming based.*

He also observed that this approach emphasised *systems development* rather than just programming concepts:

> *You can still use it to teach them all they need to know about programming. So as a programmer's programming language you can still do all of that. But what you can do more, is that you can teach them a lot more about systems and how systems work or aspects of information systems and the learning is more palatable because you can get instant feedback. I mean you sit down and you write a COBOL program or a Pascal program probably for a week before it really does anything that you can really see, whereas with these sorts of language and you can sit the kids down in a two hour tutorial and they can take home a disk with a little miniature application on it - it may not do anything more that ask somebody for their name or address but it is all there and it has all of the components that they see with real live software have (Lecturer D).*

Although Lecturer R described this change as the *dynamics of the system*:

> *The dynamism, they have to understand the dynamics of the system in a way they did not have to before, screen design is an issue or interface design is an issue that didn't arise before.*

He is, unlike Lecturer D, concerned that this shift away from programming design methods may be detrimental:

> *There a lot of applications around that people pay big money to put in place and are mission critical that don't have one single error trapping routine in them. You wouldn't do that if you were doing old style programming, you just wouldn't do it. There are programs put in place where there are no back up procedures. You just shouldn't be doing that sort of thing - it is just unprofessional.*
>
> *It is possible to get a screen that works quickly that is what does it. So customers, as will always be the case, you deliver the product, the motor car, it looks as if it is finished, the fact that you didn't put the brakes on doesn't worry them in the first instance only when they try to come to an intersection*

Interestingly, Lecturer D modified their subject when it was taught the following year in an effort to place more of an emphasis on *programming concepts*:

> *With Visual Basic it was almost like we were doing exactly the other way round, look folks the real important thing is the product bugger the programming we will learn that by osmosis but the real important thing is the product and my current thinking is that that was wrong. So what we have done this year is to re organise*

> *the way we are teaching it. So I am running a two hour lecture just in an ordinary lecture theatre and then 2 hours worth of tutorials in the labs.*

Moreover, Lecturer B observed how aspects of *software application engineering* were accentuated more so than had previously been the case when C was used:

> *We hoped to cover more than we did, and when you look at C, the things you cover are file input or output or standard input output and you are getting into pointers where as we talked about pointers but never actually used pointers as such. So in terms of content it is obviously different so makes it difficult to compare how far you get because in C you don't have to cover things like objects and events and whatever. But in C you never get to do anything like User interfaces, help systems etc. Where you lose in one area you gain in another. And you can probably start developing the concept of software application design or engineering rather than the procedural syntax which is what you tend to concentrate on there. There are some aspects of software engineering coming into the unit which were never covered in a procedural language.*

5.4.2 Problems for Lecturing Staff

Lecturer R noted that inexperienced lecturers are often unaware of the pitfalls of teaching an event-driven language for the first time and the need for a different approach to teaching concepts:

> *It is probably inevitable that Visual Basic will be displacing other things and I would say that the staff who are doing that are not ready for it because they have not written something commercially in Visual Basic. And til you try to write something of a decent size and have the pressure that it has to work with people other than you running it, you don't realise just what sorts of traps you can get into. And the difference, the things that you were not taught in your initial degree, we are coming, we always do in computing , across that problem again, that we were trained in ordinary linear environments and this is a long way from ordinary and it is a long way away from linear (Lecturer R)*

This view was supported by Lecturer D who describes the impact of the move from the notion of a procedural *main-line loop* to events in event-driven systems:

> *That has been the single biggest leap I think. Before I did that I guess it hadn't impressed itself upon me all that much but I remember reading a remark that somebody wrote, I don't know a year or two ago, and this guy was saying that he was in a Visual Basic type environment and he said the trouble with getting our old COBOL programmers into the new environment is that they are always looking for the main line loop and I sort of read it but the penny didn't drop until I got into this myself Do until while end of file which is what drives all our COBOL programs but that doesn't happen any more.*

Moreover, Lecturer D discovered the problems that other inexperienced lecturers faced teaching Visual Basic for the first time:

> *It came home to one of my lecturers here who had been tutoring for me in the Systems Implementation A unit and I drew up an exam paper and she said to me, 'You haven't got any exercises in there for them to do that involve loops'. And I said, 'Yes, you're right.' She said, 'You should do that, that's something they need*

> *to think about.' OK, what will we put on and she started coming up with all the things that we would have put in a COBOL program. But I said that wouldn't happen would it. And she said, 'Oh no it wouldn't.' And what about. 'That would happen either, no. Yeah, you're right.' And there is almost nothing that you use loops for is there.*

Lecturer B also found that their extensive knowledge of procedural languages sometimes hindered their transition to an object-oriented language although the students do not find the concepts difficult:

> *We have also found in lectures, and particularly for Paul who is overseas at the moment, we have a tendency because we are procedurally based, our backgrounds, to try to instil a problem into the lectures when really there is not one. And we spend 20 minutes trying to go around in circles explaining a problem when it isn't in there in the first place. The students have understood it 20 minutes ago lets get going.*

5.4.3 Skills Development

When asked if students use a different set of skills when programming an event-driven systems, Lecturer R stated *No, there are just more that is all. There are new additional ones.* Lecturer R stated:

> *In the past it was sort of menu and where are we going to put the status bar, now it is more complicated. Where in the heck are we going to place the logic, and where is the display going to be and all that sort of stuff has become more important ... and the connectivity sort of arises out of that, doesn't it. We have got a database that can do things for us automatically and we'll call up those aspects of the database that we want to call at a given time or will we write a bit of logic ourselves so that connectivity thing is new that has to be thought about. Where does choosing the appropriate.... In a 3GL your instruction set is really extremely limited whereas one of these environments we are talking about there doesn't seem to be any limit at all for the instruction set.*

Lecturer B noted that in object-based *languages there is a lot of the refinement going on as the developmental work is coming on.* Moreover,

> *There is an element that perhaps the division between the analysis and design and the implementation is being broken down with the object-oriented stuff whereas with procedural languages it was very clearly defined. Whereas that is now starting to merge together.*

This point is supported by Lecturer T from VUT who found that students in Systems Implementation, a final year subject, need to *experiment* as they developed an event-driven system:

> *If you don't quite understand what you are trying to do ... you can have a sort of over view of what the basic aim of the thing but it is pretty hard to have that top down approach unless you know everything that is possible and knowing what is possible is a bit of a problem with something like Visual Basic because they almost need a sort of play stage to see what works and what doesn't work and step back from it and say well... for instance if a combo box is going to be best for this, it is*

> *pretty hard to know that in advance until you have tried to put one on and have a play and then you think that is OK.*

A further point identified by Lecturer D and Lecturer R was the inability of many students to determine where to place event code, despite their experience as users of Windows programs:

> *The main problem that the kids have is that they sort of know what has to happen but they don't know what events to tack it onto. You say, somewhere along the way here fellas you have got to calculate the cost of that thing, well yeah what event are you going to hang that on - gee I don't know. In one way is to put a button on the screen that says calculate cost that does the job - yeah but there are other things that might happen aren't there - are there - lost focus(Lecturer D).*

> *It is a conceptual barrier because it is not just a practical thing it is something where these environments have all this richness and it is possible to move between all of the rich places that you are in ... Students can get lost not because any particular part of Visual Basic is difficult but because they don't know where they are(Lecturer R).*

5.4.4 Program Design Methodology

Lecturer D has been a strong advocate for teaching program design particularly during the period when procedural languages were popular in programming subjects. He believes that students still require a systematic approach to the development of an event-driven system and this starts with the screen:

> *It has come to imprint itself on me that really the skills that you need for your C and your COBOL's programs you also need for a Visual Basic program but at a different level, that is, you don't tackle the whole problem and put in your main line loop driving your first level manager routine and each of them driving the second level routine and so on but your structure is more around the structure of the screen in many ways, what are the activities that are going on in the screen and then behind each of those what are the subtasks. So I still think you need to have the skills to be able to break a more complex task down into smaller tasks but it seems to me the level you think about these is a bit different, you never have to, I guess I am trying to say, think of the whole problem, if someone gives you a 6 page spec for a COBOL program the first thing you have to do is work out what the main bits and pieces of that are, but you sort of don't have to do that much with Visual Basic, you know the first thing you have to do on looking at the spec is create the screen with the way you are going to interact with the user and it is only, I think, with what you finish up with on the screen that you start to think well there has to be a calculation here, there has to be a check done there, there has to be a write to file done here. Now when we go behind those activities then what are the sub activities involved and that sort of thing.*

At one of the other universities, similar emphasis was placed upon screen development where students do very little coding for the first 6 weeks. Lecturer B recognised, however, that they had not yet adopted a formal methodology:

> *We start by approaching it in a visual programming sense. What we do is there is a composition editor in VisualAge and they can develop the user interface and they do not need to do any form of coding to start with in any way, it is all menu*

> *driven. For those first 5 to 6 weeks what we are doing is very much approaching the development of applications through this visual programming approach. ... I don't know if we actually have a methodology - and I suspect that that comes back to our inexperience. And this is something that we are going to learn a lot more about in that software engineering unit this coming semester. Much more about what are the correct methodologies to use from a much more formal approach to analysis and design and implementing.*

Moreover, at that same institution, it would appear that the approach differs significantly between the first and second subjects.

> *There are two different approaches depending upon what you want to achieve. The first one is quite visual the second is not quite so visual and the strategies would be quite different in how students would approach that design. ... We have relied particularly on the visual nature of the software and tried to build concepts from there into language concepts, but talking in terms of application development we still have quite a lot of work to do (Lecturer B).*

This emphasis on screen design and an apparent lack of design methodology in the introductory subject, however, is of concern to Lecturer R:

> *I think there is some dangers in that if you to suggest that that is a good way of doing things. It is hard, I do agree with prototyping but students do get yelled at if they spend too much time fiddling with the screen. One of the first natural human reactions to Visual Basic environment is to go and put as many twiddly things on a screen as you can.*

5.4.5 Summary

Procedural languages have been used in introductory programming subjects for decades. The aims, content and the teaching approach of these subjects had been well documented and it was often only necessary to consider which of the procedural language had to be used. Object-based, event-driven languages working within visual environments are modern development tools and it was logical to assume that they would eventually be adopted by some universities as the language in introductory programming subjects. The experiences of three Australian universities has been outlined. It highlighted the problems information systems departments face attempting to evaluate and, sometimes, change to a radically different paradigm. For some institutions the change, they believe, will compromise the original aims and objectives of their introductory subject. For other institutions, the event-driven paradigm has been embraced and, despite some problems, has been accepted by both students and staff. Lecturer D sums up their view:

> *My sort of reaction to using Visual Basic is that it has been absolutely terrific. And I come to it as an old die hard programmer so it is not like I have a gee whiz syndrome. I have found that it is much easier to get kids interested in the programming activity than it ever used to be because you can get some instant gratification out of it, it is a bit of fun. In programming before it was a bit like castor oil, you had to say this is good for you, believe me it is good for you believe old Uncle Peter.*

The lack of a recognised design methodology is of concern to all of the experts interviewed. Most have adopted an ad hoc approach choosing to encourage systems development starting with the screen, although one of the experts appears to use elements of object modelling as well. The next section describes a program design methodology for the teaching of event-driven languages to novice programmers.

5.5 The Need for a Design Methodology for Event-Driven Programming

The literature review revealed that Visual Basic has not been adopted as a programming platform in introductory programming subjects in part because it incorporates multiple paradigms and encourages ad hoc programming through rapid application development techniques. Moreover, the interviews with experts who have introduced Visual Basic into Information Systems courses reveal the need for a program design methodology when teaching with an event-driven programming language. The issues of structure, rigour, user requirements and maintenance are clear aims in an introductory programming subject in Information Systems courses regardless of the programming language adopted.

The proposed program design methodology incorporates design tools from a range of programming paradigms yet emphasises the heart of event driven systems, that is, that program flow is controlled by the triggering of events. Moreover, although Visual Basic is not a visual programming language, Visual Basic programs are developed within an visual programming environment where the programmer places controls on a form. The design of the user interface is therefore important as both a means of interaction between the user and the finished program and as a development platform for the programmer, and therefore should be incorporated into a program design methodology. Finally, the proposed design methodology was developed to meet the requirements for *teaching* event driven programming and although it may be used in the development of larger commercial systems the primary aim is to construct a framework within which students can develop, document and test event driven systems.

5.6 A Program Design Methodology for Event-Driven Systems

A program design methodology which addresses the concerns and requirements outlined by lecturers above has been developed by Peter Shackleton and Douglas McConville (1997) and is outlined in this section. Moreover, the program design methodology has been developed to assist novice programmers with the seven programming tasks identified by Shneiderman (1979) as outlined in the literature review. The program design methodology contains 7 components

Problem Statement - a concise expression of the overall aim of the system.

Defining Diagram: - shows the inputs which need to be processed to produced the required outputs.

Screen Layout: - the screen layout or user interface and shows the placement of the objects with their names.

Event/Response List - a list of the events and their responses.

Test Data: - a table of test data together with expected results for specific inputs.

Program Variable Table: - lists and describes the variables in methods used in the program.

Detailed Algorithm: - a group of pseudocode algorithms for each method

5.6.1 Program Design Methodology and Cognitive Style

The literature review revealed a novice programmer will use one or more cognitive styles in the early stages of program design, coding and comprehension. An algorithmic problem solving approach is frequently preferred in the teaching of procedural languages as it is highly structured and cognitively fits with the procedural paradigm although some novice programmers may not be comfortable with this style. Nevertheless, as revealed in the literature review, novice programmers may place emphasis on one or more tools such as structure charts, hierarchy diagrams, Nassi-Schneiderman diagrams or pseudocode to help them design and code a system. Moreover, a lecturer is often forced to make a decision for one methodology over another in the teaching of programming fundamentals.

Although the programming language behind methods in event-driven languages is procedural, it is clearly evident that the design and construction of event-driven systems is not as structure as procedural systems, and this may encourage other cognitive styles and processes. The program design methodology outlined below, by its very nature, will encourage students to develop event-driven systems in a preferred sequence. Nevertheless, while recognising this constraint, it is sufficiently flexible that it enables novice programmers to place emphasis on one or more steps depending upon their own cognitive style.

5.6.2 Problem Statement and Defining Diagram

The *problem statement* is the first stage in the process and is designed to encourage students to give a concise outline of the problem and a brief task description. Students often fail to read the problem documentation and cannot articulate what is required. Students are encouraged to read through the documentation, whether this be formal specifications, a description of the case study or interview notes, several times until they believe they have a good understanding of the exact problem. Moreover, they are encouraged to note any points that they believe are ambiguous and note any areas that require more details.

As an example, if students were told that a business wanted to establish a computerised payroll system, they would first need to determine exactly what it was that was actually required. For example, is there a requirement to print out cheques? Is a summary of the payroll needed? What data is to be entered - all the details of each employee or just the hours worked?

Recognising that it is difficult at times to determine the *exact* nature of the problem, students are encourage to match the problem requirements to a *system* of inputs, outputs and processes (or tasks) that work on the inputs to produce the outputs. This *systems method* would produce:

Inputs - *raw data* required for the program to act upon. These could be keyboard data inputs (Note: this does not include events such as pressing the Enter key) or records from files.

Processes - *actions* performed by the program. This is not concerned with how these are performed just that they need to be done. The processing component of a system shows the sequence of events that need to be performed in order to produce a solution.

Outputs - the *information* produced by the processes in the program.

A *defining diagram* is used to assist in clarifying the problem and formalising the task of identifying inputs, processes and outputs. It may also assist with the development of an algorithm. At the end of the second stage, students should have a clear understanding of the overall system aims and have determined what needs to be obtained (the **input** section of the defining diagram), what is to be produced (the **output** section of the diagram) and the sequence of processing steps (the **tasks** section of the defining diagram).

As an example, suppose a business needs a program that would allow the user to input the hours and minutes an employee worked during the week together with the rate of pay, so that the program could then calculate the decimal value of the hours worked and produce a pay slip for that employee. We could write the Problem Statement as:

> *To create a system which will produce a pay slip for an employee showing the time worked and the gross pay*

The next task is to develop a defining diagram. The first step in completing the defining diagram *may be* to treat the processing component as a *black box* and identify any inputs that would be required. In this case three inputs are needed: the hours worked, the minutes worked and the pay rate of the employee. These things could be called HoursWorked, MinutesWorked and PayRate although it is not necessary to use variable names at this stage. The next step could be to determine what will be produced as a result of the process. In this case, there are two outputs, the decimal value of the hours which could be call TimeWorked, and the gross pay called GrossPay. The defining diagram (below) would show all the inputs and outputs.

Input	Tasks	Output
HoursWorked		TimeWorked
MinutesWorked		GrossPay
PayRate		

The next step could be to work logically through the steps involved in obtaining the inputs, the tasks necessary to process these inputs, and in producing an output. Firstly the three inputs HoursWorked, MinutesWorked and PayRate would need to be established (note details of what is displayed on the screen are not required, only the processes need to be identified). Next, these inputs are converted to a decimal value: TimeWorked, and then the GrossPay is calculated. Finally the TimeWorked and the GrossPay are displayed on the screen. The completed defining diagram would show:

Input	Tasks	Output
HoursWorked	Get HoursWorked	TimeWorked
MinutesWorked	Get MinutesWorked	GrossPay
PayRate	Get PayRate	
	Calculate TimeWorked	
	Calculate GrossPay	
	Display TimeWorked	
	Display GrossPay	

In this example, inputs and outputs were identified and then the tasks were determined, however there was no specified order for the completion of the defining diagram. Note also that the defining diagram does not show details of how a calculation is performed nor details of how the data is displayed.

If there were several options such as printing of cheques or calculating note and coin requirements, then a separate defining diagram may be required.

5.6.3 The screen layout

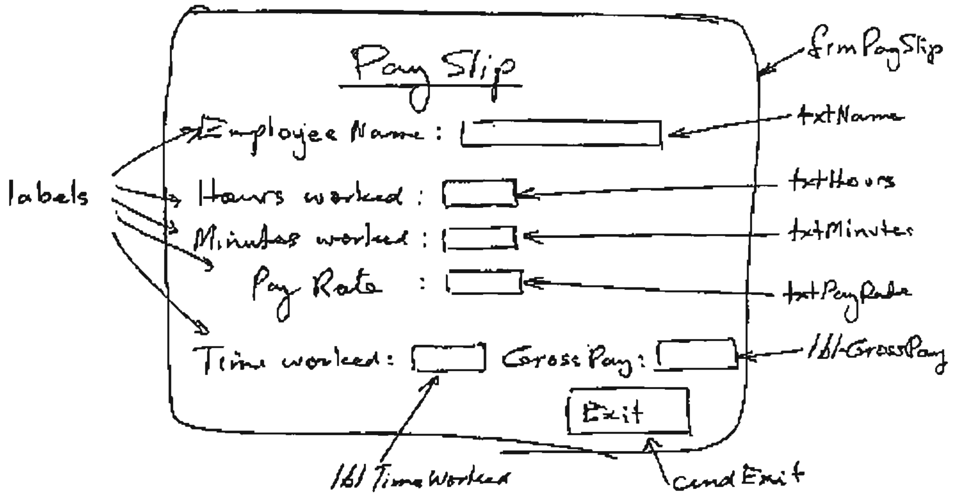
This next step in the process is extremely important in the design of event-driven programs because it enables the students to consolidate their understanding of the problem so far and to more closely consider all of the inputs and outputs required. It also gives the student a concrete object from which to work on the more detailed design of what actions are to be taken in response to the various events that the user may invoke.

In the methodology, students are encouraged to think about the screen layout prior to placing it on a computer. This process, however, could be viewed as inefficient and discouraging adjustments to the screen as they become necessary. This may be viewed as either one of the strengths or one of the weaknesses of visual programming environments, that is, that changes to the processes of data entry or output on the screen should be encouraged as the system is developed. While not precluding the need for changes as efficiencies are recognise, such as an option button rather than text entry, at this stage students are encouraged to have some screen plan from which the rest of the system can be developed.

To design the screen, students are encouraged to represent the screen or form on a sheet of paper rather than directly on the computer. The student then builds up a form by progressively adding objects using the defining diagram as a guide. As an example, they would indicate the main heading for the screen and draw an Exit button if appropriate. Next, using the defining diagram, they may indicate the position of label boxes to contain the input prompts and the position of the text boxes which will contain the users' input values. The next stage might involve drawing label boxes to display the output values (obtained from the output section of the defining diagram) and next to these they may place the labels that will inform the user what each output represents. Further command buttons and other controls may be added that will be required for the tasks section of the defining diagram.

The student then allocates a *name* to each of the label boxes, text boxes and command buttons that will be used for input, output and task control. They are encouraged to use descriptive names with a conventional prefix to describe the type of control required (Prompt labels could also be named if required but generally the default is used).

Students may now have a paper screen design like this:



Note that the calculation of total time worked and gross pay is done in the lost focus method on the text boxes but students may chose to enter a separate command button.

5.6.4 The Event/Response List

The fourth step of this program design methodology is the heart of the methodology as it identifies all of the possible events which the student will allow the user to perform on the screen and the responses which they want the program to activate. These response will include the anticipation of any user errors and the display of relevant error messages. Moreover, this stage forms the basis for writing the algorithms (and the program code or *methods* as the last step in the process).

Students are encouraged to look in turn at each of the objects (controls) on the form, and decide which *controls* they want to respond, to which *event* they should respond, and what they want that *response* to be. Typical events would be those listed in Table 5.1.

Table 5.1

Types of Events

Object Type	**Event**	**How activated**
Text Box	Click	The user presses the left (primary) mouse button once.
	Change	The user changes the text (or numerical value) in the text box or the user enters any single character.
	Lost Focus	The user 'Tabs' to another object on the screen or clicks on another object (- ie the cursor is moved from the box).
Button	Click	The user presses the left (primary) mouse button once.
Form (window)	Load	A form is made active for the first time, either at the beginning of the program or due to some other event.
	Activate	A form is activated again, it having been loaded by a previous event.

Students are asked to write the description of each event as briefly, but also as concisely as possible, so as to isolate each probable event. The response portion should also be concise, but most importantly, it should convey a good description of what actions need to be performed. In this section students are still concentrating on *what* is to happen, not *how* it is to happen. For example:

Event: The user types a character into text box 'txtName' (ie. *changes* its value).
Response: Check to see if it is the Enter key, if it is then Tab to the next text box.

Event: The user tabs out of txtName (ie. it *loses the focus*).
Response: Check to see if a name has been entered, if not display message and request user to re-enter.

As students become better acquainted with all of the possible events, their *event* portions would include the actual words used to describe the event. eg. the above might become:

Event: Text box txtName loses focus.

Other events may be:

Event: Text box txtHours loses focus

Response: Check to see that less than 65 hours, if not display message and request user to re-enter otherwise calculate time worked and gross pay and display

Event: Text box txtMinutes loses focus

Response: Check to see that less than 60, if not display message and request user to re-enter otherwise calculate time worked and gross pay and display

Event: Text box txtPayRate loses focus

Response: Calculate time worked and gross pay and display

Event: Click cmdExit

Response: End

5.6.5 Test Data

The literature review and interviews with experts revealed that students often fail to test their programs against adequate test data. Moreover, test data developed by the student often lacks breadth to test functionality or is incorrect. The importance of test data in the development of a program is reflected in this methodology where it occupies the fifth stage in the process. Students are required to develop a table showing values for each input and the expected results for output. Using our example the test data may be:

Test Data			Expected Results	
Hours Worked	Minutes Worked	Pay Rate	Time Worked	Gross Pay
0	30	100	0.5	50
1	0	50	1.0	50
2	45	100	2.75	275

Note that the table could include incorrect data to test for error trapping.

5.6.6 Program Variable Table

The sixth step in the methodology describes in tabular form the global and form variables that are used in the event methods. Students construct a *Program Variable Table*, a table used to provide a formal description of each global and form variable. A Program Variable Table is designed to assist with the development of a program and as a documentation tool. It equals in part the data dictionary used in larger systems.

The Program Variable Table contains information about each of the variables (and constants) that are used in the program. The structure of the table should be easy to read and the table should be constructed in the following manner:

Name	Description	Type	Size	Comment

Name: the names used for variables and constants in the program.

Description: what the variable is and how it relates to the program.

Type: the data type of the variable.

Size: where appropriate it may be necessary to specify the size of the variable (eg it is a string of 25 characters)

Comment: if there are any additional pieces of information that are required, they can be placed here. These comments may be useful when further developing the problem or for future maintenance. This section may also be used to show the assignment of an object property to a variable or vice versa.

Although described as the sixth step in the process, it is unlikely that students could complete this table directly from the information provided so far. The defining diagram may possibly provide some information on input and output variables and provide some guide to the variables that would be needed in an algorithm but it is likely that the program variable table would be completed as the algorithms are developed.

Using the running example on the calculation of gross pay, a defining diagram was developed for a program that would allow the user to input the hours and minutes that an employee worked during the week, together with the rate of pay. The program was then required to calculate the decimal value of the hours worked and print a pay slip for the employee. The completed defining diagram was:

Input	Tasks	Output
HoursWorked MinutesWorked PayRate	Get HoursWorked Get MinutesWorked Get PayRate Calculate TimeWorked Calculate GrossPay Display TimeWorked Display GrossPay	TimeWorked GrossPay

From this defining diagram a Program Variable Table can be developed

Name	Description	Type	Size	Comments
Hours	Number of whole hours worked	Integer		
Minutes	Number of minutes worked in last part of hour	Integer		
TimeWorked	Convert time to real number format. Decimal hours calculated from hours and minutes worked	Decimal		
PayRate	Hourly rate of pay	Decimal		
GrossPay	Gross pay for an employee	Decimal		

The algorithm may not place the value of an input text box, say txtHours, into a variable. If this was the case then those variables would not exist and they would not be included in the program variable table. The following variables would be declared in the General section of a form

```
Dim Hours As Integer
Dim Minutes As Integer
Dim TimeWorked As Single
Dim PayRate As Single
Dim GrossPay As Single
```

5.6.7 A Detailed Pseudocode Algorithm

Event driven languages use procedural code at the method level to implement the process describe in the response to an event eg. calculate the total time worked, tax and gross pay, or assign a value in a label box to a variable, or switch from one form to another. Pseudocode is still a very popular tool for presenting an algorithm prior to its conversion into the final program code. Pseudocode enables the writer to easily express the steps in an algorithm in a way which often matches the processes one uses to solve a problem. Moreover, the writer is not restricted by complex programming language syntax - pseudocode is language independent. Pseudocode has benefits too for the teaching of programming as it assists with the development and description of an algorithm *prior* to the conversion into a programming language. Students are required to give some thought to the design of an algorithm before conversion into code rather than developing the system from scratch directly in the language.

Students are encouraged to write algorithms in a form of pseudocode that matched the structure and components of programming languages. Although there are no set rules that precisely define pseudocode the important thing is that it should be able to be easily understood and, in particular, be able to be easily translated into a high-level programming language. Although pseudocode combines English words with symbols, terms and functions like those used in most high-level programming languages students are encouraged to avoid using specific computer programming syntax allowing for the possibility that it could be translated into most event-driven programming languages.

Students are provided with some guidelines.

- Each line or operation should be described briefly and precisely. Avoid long-winded descriptions. For example, 'Get FirstNumber' conveys the same meaning as 'Get the value of FirstNumber from the keyboard'.
- Use standard symbols for arithmetic operations: + for addition, - for subtraction, * for multiplication, / for division. eg Average = Total/Count
- Pay particular attention to readability. The pseudocode should be able to easily translate into a computer language. Use a separate line for each instruction, indent blocks of instructions under the control of selection or iteration statements and add comments if necessary.
- Use key-words to describe operations common in high-level languages. Get or Enter usually describe an input from the keyboard; Read often refers to something obtained from a file. Display refers to output onto the screen; Print to output onto a printer and Write refers to writing back to the file.
- Avoid using statements that describe display formats. If something is to be displayed or printed, such as an invoice or receipt, use a statement like: Print Invoice as per specifications. When the pseudocode is translated into a programming language, the programmer will refer back to the detailed specifications to program the form layout.
- Assignment is performed when it is necessary to change the value of a variable, and in pseudocode this is indicated with a statement like:

  ```
  ItemPrice = 12.95
  ```

  This presents no difficulties. When the value of a variable must be increased by 1 a pseudocode statement like any of the following could be used:

  ```
  Add 1 to Count
  Increment Count by 1
  Count = Count + 1
  ```

  The last (and most common) of these however, presents an apparent 'mathematical contradiction'. If the statement is taken for a mathematical equation, then it is a nonsense - there is no value of Count for which it is true. The point is that it is not an *equation*, but an *assignment* and so should be read as: take the value of Count and assign it the new value Count + 1

Using the example outlined, the algorithm could be:

```
Module Calculate[1]
    Hours = value of txtHours
    Minutes = value of txtMinutes
    TimeWorked = Hours + (Minutes/60)
    PayRate = value of txtPayRate
    GrossPay = TimeWorked * PayRate
    Assign GrossPay to lblGrossPay
End module

Module Exit (ie. cmdExit_click () event)
    End
End module
```

The Visual Basic code would be:

```
Private Sub txtName_KeyPress (KeyAscii As Integer)
  If KeyAscii = 13 Then
       txtHours.SetFocus
    End If
End Sub

Private Sub txtName_LostFocus ()
    If Len(txtName.Text) = 0 Then
        MsgBox "You must enter a Name first"
        txtName.SetFocus
    End If
End Sub
```

[1] This would be attached to the txtHours_LostFocus, txtMinutes_LostFocus and txtPayRate_LostFocus events.

```
Private Sub txtHours_LostFocus ()
    Hours = Val(txtHours.Text)
    Minutes = Val(txtMinutes.Text)
    PayRate = Val(txtPayRate.Text)
    TimeWorked = Hours + (Minutes / 60)
    GrossPay = TimeWorked * PayRate
    lblTimeWorked.Caption = TimeWorked
    'Display Gross Pay formatted as currency
    lblGrossPay.Caption = Format(GrossPay, "Currency")
End Sub

Private Sub txtMinutes_LostFocus ()
    Hours = Val(txtHours.Text)
    Minutes = Val(txtMinutes.Text)
    PayRate = Val(txtPayRate.Text)
    TimeWorked = Hours + (Minutes / 60)
    GrossPay = TimeWorked * PayRate
    lblTimeWorked.Caption = TimeWorked
    'Display Gross Pay formatted as currency
    lblGrossPay.Caption = Format(GrossPay, "Currency")
End Sub

Private Sub txtPayRate_LostFocus ()
    Hours = Val(txtHours.Text)
    Minutes = Val(txtMinutes.Text)
    PayRate = Val(txtPayRate.Text)
    TimeWorked = Hours + (Minutes / 60)
    GrossPay = TimeWorked * PayRate
    lblTimeWorked.Caption = TimeWorked
    'Display Gross Pay formatted as currency
    lblGrossPay.Caption = Format(GrossPay, "Currency")
End Sub

Private Sub cmdExit_Click ()
    End
End Sub
```

5.6.7.1.1 A Complete Example

The following example uses all of the seven steps in the program design methodology.

Case study description: Peter's Pizza Parlour requires a program to determine the prices of their pizzas. A Small pizza with no extra toppings is $7. The cost of extra toppings on the Small size are: Ham 35c, Salami 40c and Prawns 50c. A Large pizza costs 1.7 times the price of a Small pizza.

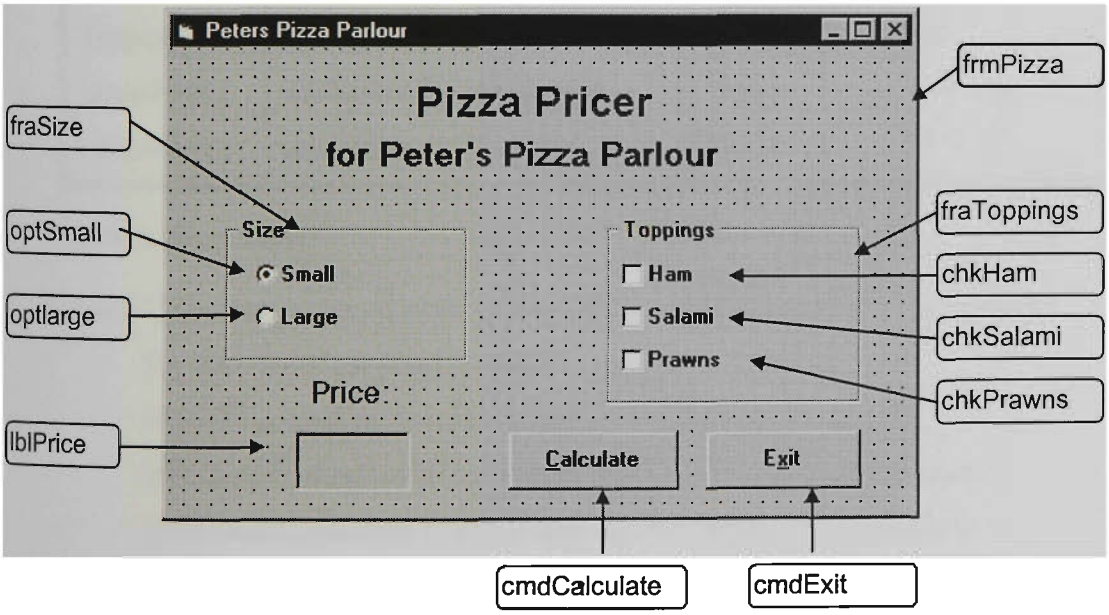
Problem Statement

Peters Pizza Parlour requires a program to determine the prices of their pizzas.

Defining Diagram

Input	Tasks	Output
The size of the pizza The toppings for the pizza	Display the form with the small size selected Get the size and the toppings required. Calculate the price of pizza + toppings Display the price	The price of the pizza

Screen Layout



Event/Response List

Event: The user clicks on a pizza size option button

Response: The button is marked on and the other is marked off.

Event: The user clicks on pizza topping check box

Response: The box is checked or unchecked

Event: The user clicks on the Calculate button

Response: The price is calculated and displayed.

Event: The user clicks the Exit button

Response: The program terminates.

Test Data

Size	**Ham**	**Salami**	**Prawns**	**Expected result**
Small	no	no	no	7.00
Small	yes	no	yes	7.85
Large	yes	no	yes	13.35

Program Variable Table

Name	Description	Type	Size	Comments
BasePrice	price of small pizza, no extras	Const		= 7.00
Price	price of selected pizza	Dec		
Toppings	total price of toppings	Dec		
SmallFactor	multiplier for small size	Const		= 1.0
LargeFactor	multiplier for large size	Const		= 1.7

Detailed Algorithm

```
Module Calculate (to be attached to the 'click' event of cmdCalculate button)
        Set the cost of toppings to zero
        If Ham is checked Then set toppings = toppings + ham cost
        If Salami is checked Then set toppings = toppings + salami cost
        If Prawns is checked Then set toppings = toppings + prawns cost
        Set price = Base price plus toppings cost
        If Small size is selected Then multiply price by SmallFactor
        If Large selected Then multiply price by LargeFactor
        Display the price
End Module
```

```
Module Exit (attach to the 'click' event)
    terminate the program
End Module
```

Visual Basic Code

```
Private Sub cmdCalculate_Click()
    'declare constants and variables
    Const BasePrice = 7
    Const SmallFactor = 1.0
    Const LargeFactor = 1.7      'ie. 1.7 times base price
    Dim Toppings As Single, Price As Single

    'set the initial toppings value to zero
    Toppings = 0

    'for check boxes: 0 = not checked, 1 = checked
    If chkHam.Value = 1 Then Toppings = Toppings + 0.35
    If chkSalami.Value = 1 Then Toppings = Toppings + 0.4
    If chkPrawns.Value = 1 Then Toppings = Toppings + 0.5

    'add the toppings to the base price
    Price = BasePrice + Toppings
    If optSmall.Value = True Then
        Price = Price * SmallFactor
    Else
        Price = Price * LargeFactor
    End If
    'assign price to caption property and format to currency
    lblPrice.Caption = Format(Price, "Currency")
End Sub

Private Sub cmdExit_Click()
    End
End Sub
```

5.7 Summary

This program design methodology provides a formal basis for the teaching of event-driven languages. In the next chapter it will be evaluated in a case study of students who were enrolled in an introductory programming subject in the Department of Business Computing at the Victoria University of Technology.

Chapter 6 Analysis of a Program Design Methodology

The experiences of the students and lecturers in a subject utilising the Program Design Methodology for Event-Driven Programming will be described in this chapter. As outlined earlier in this thesis, the data includes interviews with undergraduate and post-graduate students, interviews with experts from other institutions where event-driven programming is taught, observations of students in workshops, materials collected from students throughout the semester, and interviews with lecturers who trialed the design methodology.

The overall purpose of the design methodology was to support the teaching and learning of event-driven programming. The analysis commences by reviewing the mental models developed by students as a consequence of working in this paradigm. This is followed by consideration in turn of the seven steps incorporated within the design methodology.

Amongst its major findings, the analysis firstly reveals that many students fail to construct a clear mental picture of the internal operation of an event driven system. This, in part, is caused by lecturers and workshop leaders who, although having experience with procedural languages, do not have sufficient examples and metaphors to describe the concepts of event-driven systems to novice programmers. The absence of a conceptual model to describe an event-driven system, however, is particularly telling at this point.

Another major finding from the analysis is that an event-response list, which is instrumental in determining the structure and operation of a system, is often difficult to construct, even for moderately simple systems. The concept of an event, although apparently easy to describe, in reality is very difficult for most students to understand. Moreover, lecturers often contribute to this confusion by modifying the definition of an event or placing a different interpretation on an event at different stages in a course. The analysis found that most students were confused about:

The boundaries of an event - that is, whether to include prior actions in an event such as an input to another object

The context of the event - whether events should be viewed from a programmer's or from a user's perspective, or from both

The status of an event - the distinction between active and passive events, and

The level of detail in a response - whether to include multiple paths such as for validation.

The full analysis which follows also discovered the differing degrees of importance students place on each step in the design process and the problems of coding and testing event-driven systems.

6.1 Mental Models

The literature review suggested that a person will develop a mental model enabling them to gain some understanding of the operation of a program on a computer. The mental model may not be accurate in technical terms but embraces the concepts of the particular programming paradigm. The model enables a person to understand how a program or program instruction is interpreted and implemented. In many respects, the conventional main line procedural program is comparatively easy to comprehend, although a student may find it difficult to convert a solution to a series of ordered instructions. In the procedural paradigm, control is surrendered to the computer, and each line is executed one after the other. Selection and iteration constructs separate the logical from the physical structure of the program and increases complexity, nevertheless, the program is executed from the first line and works its way down line by line. This start-at-the-top and work-your-way-down approach is still relatively easy to comprehend.

Microsoft Windows is one of the most popular operating systems in the world. Most students in Australian tertiary institutions would work within a graphic user interface environment for word processing and on other applications, and a major argument in favour of an teaching of event-driven language such as Visual Basic, is that it works within a Windows environment. It was therefore surprising to observe that many students had problems understanding the fundamental concepts of event-driven programming. Event-driven languages, such as Visual Basic, are object-based and use a mixture of procedural and object-oriented programming concepts and, this in part, may explain some of the confusion. Nevertheless, the basic concept of an event-driven programming language is that events are the sole way of triggering code and a large number of students had trouble understanding this fundamental concept.

Most students could describe an external event, such as a click. External events are commonly required for applications developed by students in other subjects. Postgraduate students, for example, are required to develop macros triggered from buttons in their first spreadsheet assignment in the first year Management Information Systems subject. As an example, when asked to describe what was meant by an event, Dianne, a postgraduate student, stated:

Like a click yes. It triggers something off.
We have been doing this all along with Arthur (Dianne).

Moreover, most students appeared to understand, or came to understand, internal events such as the loading of a form although they did not appear to see this as a real event. A large number of students, however, had problems comprehending what will be referred to in this thesis as *external-related events*, such as lost focus or change. A change event is an external action, such as the keying in of data, which executes some code each time the value of a text box changes. As an example, a change event would be executed three times if the number 100 was entered into a text box when a change event existed. A lost focus event, on the other hand, triggers a response only when the user moves off the object, that is, the response would be executed only once after the user typed in 100 and moved to another object. Depending on the type of active event that is present, a student may view keyboard entry as the event, such as in the case of change but not for lost focus. This is very confusing for many students. In reality, the event is either a change of text in an object (which may or may not be done via the keyboard) or movement off an`object (such as by using the mouse to click onto another object) and not the keyboard entry itself. In some case, students appear to be relating an external event, keyboard entry, to an event of another kind.

When asked how easy he found the concept of event-driven programming, Clinton indicated the problem he has with the change event:

OK - even with change I understand it but it takes a bit of time (Clinton).

Chigden and Elizabeth lamented that they did not use a calculate button for their major assignment:

I wish I did use it. A calculate button is clearer for me, on change isn't (Chigden).

It is not easy. If you take away the calculate button I would be lost (Elizabeth).

Windows-based applications often require a strict order of external events to be performed before a function can be performed successfully. As an example, in a word processor, to turn an existing piece of text to bold requires the operator to highlight the text and then press the bold button. Students appear to have a clearer understanding of this ordered process which may explain the easier transition to an all-embracing Calculate type button on a form.

It was difficult to glean information that hinted at the mental framework which students use to understand the concepts of event-driven programming. Some students recognised the importance of gaining some knowledge of how a computer operates, at least in broad terms:

> *Everyday, it opens up, it is so wonderful. It is like when people talk about creation and how wonderful it is. It is like that. I am staggered how wonderful, how clever it is. I have been reading a lot. I am aware of the structure of computer. I have some vague idea. It is not like there is a fairy godmother who takes all these things (Gerald).*

Others appeared to pay little importance in trying to understand the operation of a program at a level lower than the code:

> *I haven't thought of it. I have no idea. Just a big box (Dianne).*
> *I have no idea. I have never thought of that (Tom).*
> *You put up the screen layout. You have bits behind it that make it work. That's all (Mark).*

A common technique used by lecturers to help explain a particular concept is to liken it to an everyday experience which is easily understood by students. In a procedural program, it may be sufficient to describe the workings of the three constructs, sequence, selection and iteration, which are the components which describe the entire operation of the system. To describe the sequence construct, we could use an example of making a cup of tea - it is necessary to have a cup before you can pour water from a kettle.

In contrast to procedural languages, it becomes necessary to describe the operation of the whole system as well as the Structure Theorem in event-driven programming languages. One could liken the operation of a car to an event-driven system where a car responds to the events of the driver; turning off the key (clicking on the Exit button) or pressing the accelerator (clicking on the Calculate button). One student, Fred, recalled an example used by a lecturer who used an automatic teller machine as an real example of an event-driven system. However, these analogies often describe external (and some internal) events but rarely can they be used to describe external-related events such as change and lost focus. When asked to liken event-driven programming to everyday experiences, most students opted for control based systems:

> *Probably like some filing system. It makes it way down. It somehow sifts through it (Jaranka).*
>
> *They are not much help. In the MIS subject and databases, whether it was because it was work related, I don't know, I could understand it a lot easier. I cannot with this subject (Dianne).*
>
> *Well, a line of Code. Well I think I have some idea and with the writing background it is just a matter of getting the correct constructions, is what I see, and once the instructions are clear and precise there is usually no problems and it is a matter of going back and checking and picking up. Obviously there are going to be mistakes along the way but usually that is just me and having to find out clearer and more precise ways of achieving a certain objective. But I see it as just like writing an essay or reading a book, you start at the beginning and you go through and certain information or descriptions will be able to provide certain things.(Mary-Beth).*

> *When you enter it into that box you have commanded it to go to a database or something. To me it is like you are commanding it to do something (Sylvia).*

Other students, however, were more precise:

> *To me it goes top down. I understand modules. It starts at the top line and moves down. If it gets to a loop it just goes around and around. It looks for the first event and then it waits for the next event. Triggers first line of procedure and runs down to the bottom (Fred).*
>
> *I think it makes it a lot easier. If you know that only on an event you are going to get a certain reaction. It breaks it down. Only when you get a click or only when there is change will something occur (Billy).*

The comments from Mary-Beth reveal the importance she puts on establishing links between everyday activities and concepts in event-driven programming.

> *I think sometimes it would be useful to have not one example but two examples. So I think people can draw from two examples their own understanding of what you are trying to elicit with what is ever being talked about. And that is only because it gives people two counter points to come up with their own things sometimes. The other things is that I don't know, I understand what you are saying about jacking up the car but I don't know how many people in the class would have done that. I was thinking about it this morning coming to class and I really don't think I understand or I have a clear understanding or own the knowledge of this stuff and I think that is really important not only for me but for other people in the class and I am not clear why that is (Mary-Beth).*

It is evident that most students interviewed do not have a clear mental picture of the operation of an event-driven system on a computer and that this is impeding their ability to master more difficult concepts. Although a program design methodology might provide some structure to teaching the concepts of programming using an event-driven language, it is clear that lecturers need to do a significant amount of work at a conceptual level with students before the benefits of any methodology can be fully harvested.

6.2 The Problem Statement

The literature review revealed that, irrespective of the language, some students are prone to developing a system which does not match the specifications. The major purpose of the first step in the program methodology was to encourage students to determine the nature of the problem and to gain some broad understanding of what is required from the system. The first step requires students to write out the problem statement from a broad description of what is required. No specific directions were given to students on the amount of details that were required although it was expected to be accurate and concise. In the interview, it was important to discover if students wrote out the statement, and the process they followed ie. whether they identified key word or phrases. A problem statement was required as part of submissions for assignments. Fred outlined what we hoped students would appreciate from this process:

> *When you can actually write the problem statement you can write you have understood what the program is about. Even if you don't write it down you mentally have to do it otherwise you don't go anywhere (Fred).*

Some students put great importance on writing the problem statement in their own words:

> *I re-write it. I like to re-write it in my own words that way I get an understanding of it (Billy).*
>
> *I try to put it into my own words and ideas and write it as I interpret it (Jaranka).*

In contrast, other students believed that is was not necessary to write a problem statement Nevertheless, they appear to place importance on the process of developing a problem statement and move directly to another stage, often the screen.

> *For the problem statement I would underline what the problem is and what it is asking for. The problem is there and I don't need to write it down (Tom).*
>
> *Read what is required and rewrite it. I get rid of some of the unnecessary detail in the problem that has been written out ... Read through everything again. Then work out in your mind what the screen would look like (Clinton).*

It was evident that the mature age students placed greater emphasis on this first step than most of the younger undergraduates. Sylvia believes that:

> *With the problem statement, I know what needs to be done (Sylvia).*

In part this may have been due to their previous work experiences. Fred, on describing an experience he had with a programmer at his work, outlined the process he now follows:

> *White-board. We would start with the problem and what we want. I would hand out a whole lot of forms from the organisation and put beside it what it should be. number, minimum length and so on. I would write out an organisation structure - a paper flow. The first time I didn't write down the problem, he and I had different understandings of what were trying to do. It was in building. And what he was doing was re-writing the actual building materials that I supplied off the shelf what they actually wanted was building material where the bill can be altered by the operator at two points. This option was not understood as we did not write it down (Fred).*

Mary-Beth, a mature age student who had returned to full-time study observed that:

> *Look I think there were even a lot of problems with people being able to define input output and the steps in between and that goes back to people defining the problem. And if people are not defining the problem correctly then they are not going to be able to do that input output stuff (Mary-Beth).*

The introductory programming subject at Victoria University has a one hour workshop per week which is limits the material a workshop leader can cover. Mary-Beth was critical of the process many workshop leaders follow in trying to skip over earlier steps and move onto coding:

> *I think it is underestimated a little bit. You just skip over it. Now you can do that because of your background. And people say "Well how do you get there" I don't know. They don't pick it up at that level they pick it up at the input output level, which seems to be a little more hands on in terms of OK they understand inputs and outputs because they have seen you do that before but they don't necessarily direct it back to definition of the problem (Mary-Beth).*

It is possible that in the process of skipping many of these steps in workshops students believe that a system can be produced just as easily without going through these steps as it can by following the steps. The end result is that they become confused:

> *Maybe they are not clear about what is required of them or they're not clear of that whole scene. I feel a lot from the class that there is quite a bit of frustration there, I still don't know and I don't seem to ask the right questions or don't seem to be able to elicit the information to get me any closer to a solution, and that is not really a good thing (Mary-Beth).*

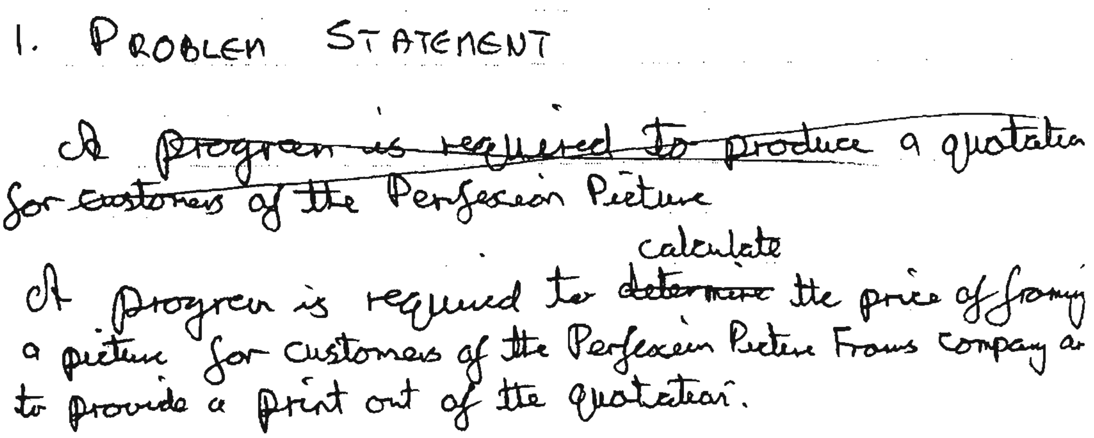
The process of formulating a problem statement appears to help in determining inputs and outputs and, possibly, in the formulation of a screen. Fred states:

> *If I have read it 4 or 5 times but still don't understand it, I won't do anything until I understand it. When I get a picture, I will start going through the 7 steps which I definitely follow (Fred).*

Peter wrote out a problem statement for the first assignment in the semester but now prefers to highlight key words or phrases

> *I did in the first assignment but not in this assignment. I organise the key words. But what needs to go on the form that is what they require so that is what the problem statement would be (Peter).*

Peter provided copies of the work he had done on the first assignment over a 4 week period. The first iteration of the problem statement below read:

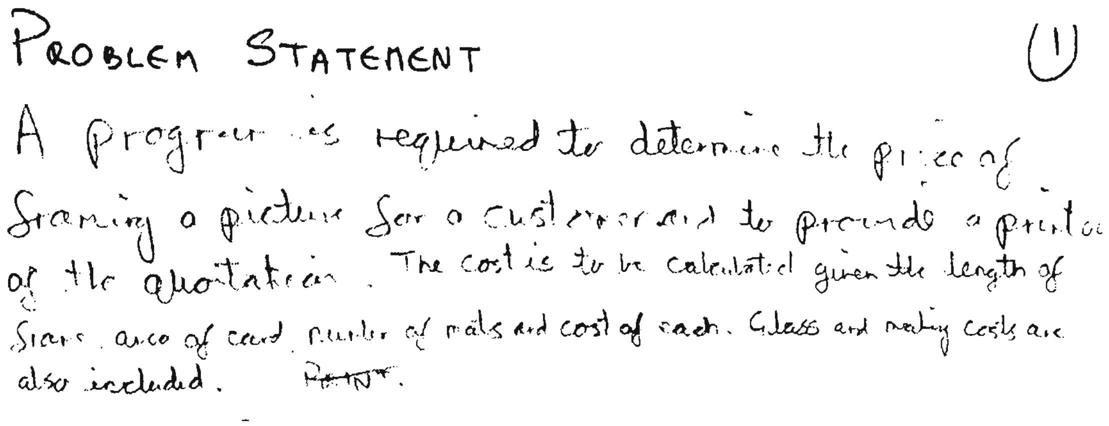


1. PROBLEM STATEMENT

~~A program is required to produce a quotation for customers of the Perfection Picture~~

A program is required to ~~determine~~ calculate the price of framing a picture for customers of the Perfection Picture Frames company and to provide a print out of the quotation.

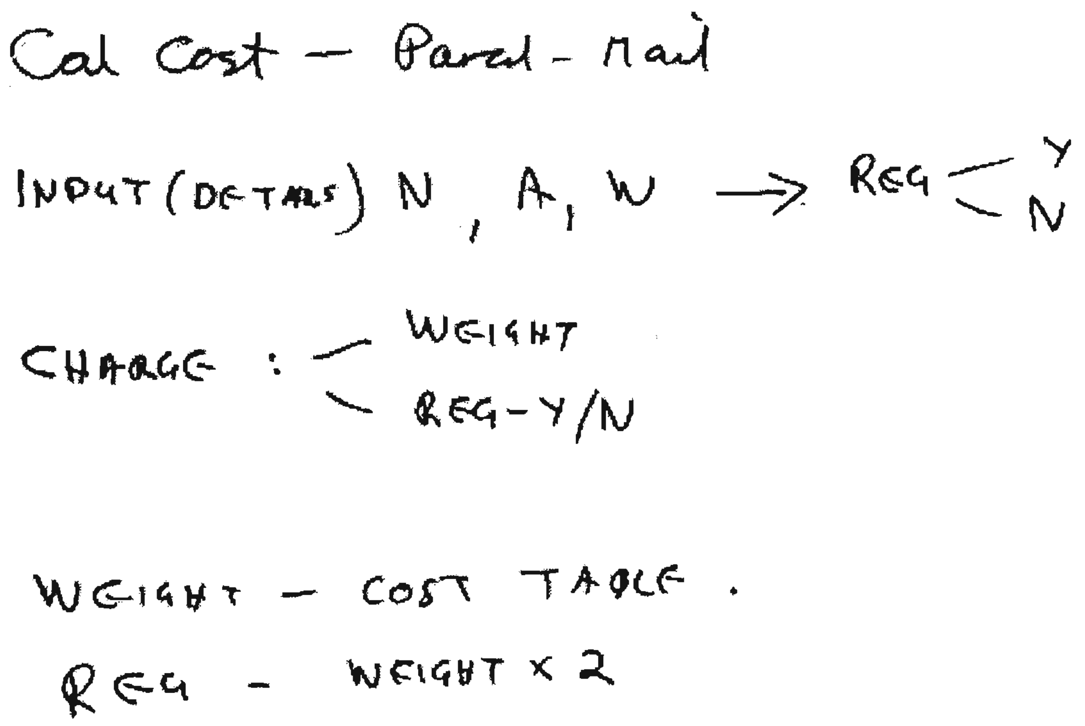
and the final problem statement read:



PROBLEM STATEMENT (1)

A program is required to determine the price of framing a picture for a customer and to provide a printout of the quotation. The cost is to be calculated given the length of frame, area of card, number of mats and cost of each. Glass and mating costs are also included. ~~PRINT~~.

This detailed problem statement starts to reveal some of the inputs and outputs and alludes to the process of calculation. In the interview, Peter underlined key words in the sample problem and re-wrote the problem using symbols describing inputs and outputs and a brief outline of the steps in the calculation of the cost:



Cal Cost – Parcel - mail

INPUT (DETAILS) N, A, W → REG < Y / N

CHARGE : < WEIGHT / REG-Y/N

WEIGHT – COST TABLE.

REG – WEIGHT × 2

It is interesting to observe how students may use an example in the sample problem as a method of testing a more detailed problem statement. Although no specific calculations are required at this early stage, some students are obviously trying to determine how an output is determined. This is actually the first steps towards determining the processing part of the defining diagram.

> *Try to use the example as a guide. Then create my own example (Jaranka).*

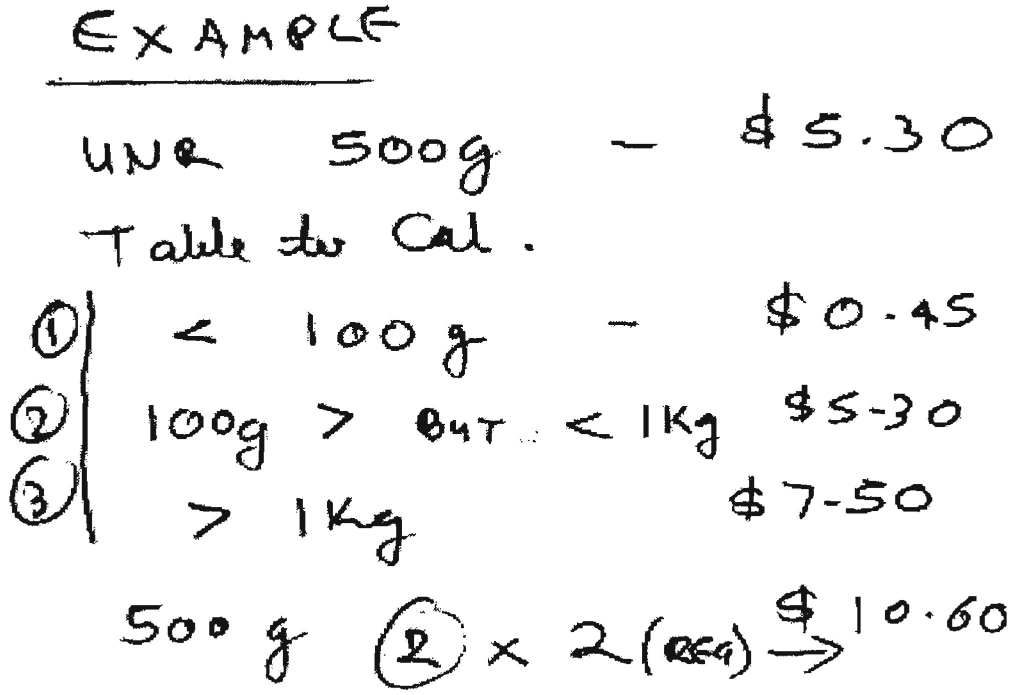
In the interview, Peter used the example and the table of costs. It was observed that Peter

> *underline points on the sample problem*
> *write out points onto a separate piece of paper*

He then stated

> *I go back to that (cost table) to find out if I have understood it And then go to the example to see how it is calculated.*

He then writes down details of calculation:



EXAMPLE

UNR 500g — $5.30

Table to Cal.

① < 100g — $0.45

② 100g > BUT < 1Kg $5-30

③ > 1Kg $7-50

500g ② × 2 (REG) → $10.60

He was asked, "What would you do next?", and he replied:

> *I would see what the outputs would be - cost of the parcel, unreg or registered*

Lecturer T, a lecturer who taught the program methodology at Victoria University of Technology, believes that students are confused about what to include in the problem statement and that it is too early for students to determine what processes need to be undertaken in the system. He believes that the problem statement would be more beneficial if it was written in a way that it included inputs and outputs:

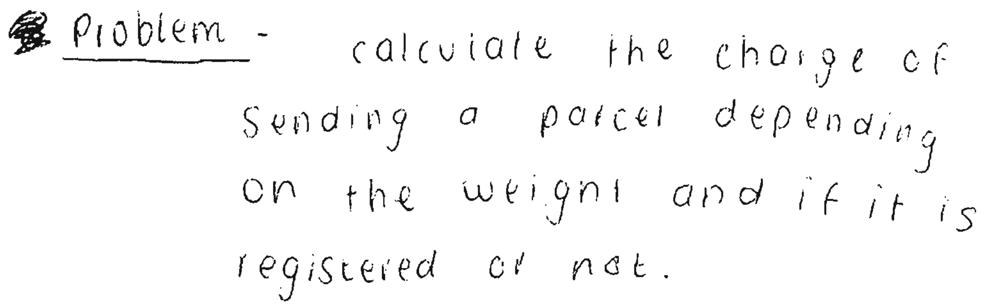
> *I think it would be good if it could lead more into the next step in other words it could concentrate on inputs and outputs - what the program is doing couched in terms of output.(Lecturer T).*

The problem statement appears to be trivialised by some students as a formal step but for those students it does appear to be an informal method of leading into the next stages in the development of a system. On the other hand, a number of students, particularly mature age students, appear to place great importance on this step as a method of determining the boundaries of the system.

6.3 Defining Diagram

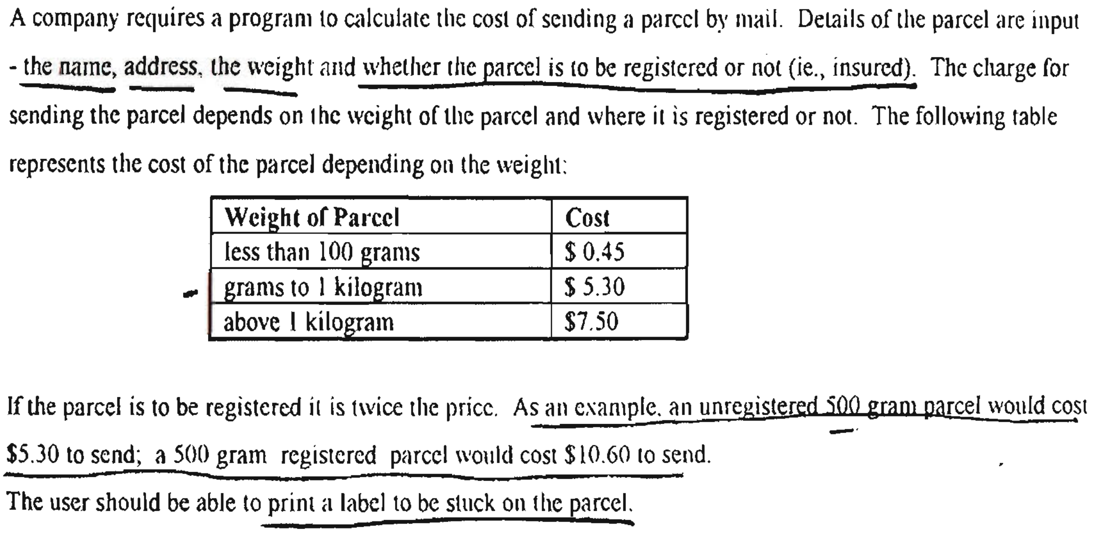
6.3.1 Introduction

The first step in the design process, developing a problem statement, may assist in identifying the data that has to be input and the information to be output from the system. We have already seen that most students identify the output or the form of the output in the problem statement (ie. print a quotation) and some students identify at least some of the processes (ie calculate) and data inputs. As an example, Seeana wrote the problem statement:



Problem - calculate the charge of sending a parcel depending on the weight and if it is registered or not.

Nevertheless, the process of underlying items can also assist with the identification of data and processes as indicated by Tom's work



A company requires a program to calculate the cost of sending a parcel by mail. Details of the parcel are input - the name, address, the weight and whether the parcel is to be registered or not (ie., insured). The charge for sending the parcel depends on the weight of the parcel and where it is registered or not. The following table represents the cost of the parcel depending on the weight:

Weight of Parcel	Cost
less than 100 grams	$ 0.45
grams to 1 kilogram	$ 5.30
above 1 kilogram	$7.50

If the parcel is to be registered it is twice the price. As an example, an unregistered 500 gram parcel would cost $5.30 to send; a 500 gram registered parcel would cost $10.60 to send.

The user should be able to print a label to be stuck on the parcel.

As described earlier, a defining diagram is the first formal step in identifying inputs and outputs as well as providing a broad outline of the intervening processes. A defining diagram can be used in a variety of programming paradigms as a method of identifying inputs and outputs. The processing column, however, implies that an ordered series of operations is performed to produce an output and probably fits more comfortably in a procedural language. An event-driven program may allow a user to by-pass steps in a solution to a problem, say getting all the inputs, and may allow the user to go directly to the calculation component, say clicking on a

Calculate button. Nevertheless, the processing section of the defining diagram identifies the processes that must be followed to obtain the correct output and indirectly implies that error trapping is required.

The input for a procedural program is via the keyboard or a file but this is not the case in languages such as Visual Basic where input can be made easier by the selection of option buttons or check boxes via the click of a mouse button. In this case an event may change the value of a property in the object and it may trigger an event. In a procedural language, this input would have been assigned to a data variable. It was found that some students failed to recognise the value in option buttons or check boxes as inputs because they were often used directly as a conditional in a selection or iteration construct, eg. If optFolding.value = True Then ..., and these values were rarely assigned to a data variable. Moreover, some students appear to confuse events and with data outputs. Events, such as the click of a command button may trigger a process, say the printing of a form, rather than the calculation of a data output.

6.3.2 Identification of inputs and outputs

Some students placed a large amount of importance on the defining diagram such as Mary-Beth:

> *To me the defining diagram, it is the only handle that I can use to get through to a solution it does not matter how much code I write if I don't know where I am going with it (Mary-Beth).*
>
> *Yes I would have to do something like that clear in my head (Dianne).*
>
> *It helps clarify everything that needs to be input into the program and what tasks the program needs to do (Clinton).*

Others chose to either ignore it or placed little importance upon it within the overall design methodology:

> *I realise it makes it easier but I have not used it. I have a rough idea of what I want to do, it is just a matter of writing in the code to do it (Mark).*
>
> *I usually do that last. You know how you are suppose to sit down and do the algorithm and all that first. I try and do it on the screen and later on an hour before ... I think most of us do it backwards. We don't do it the correct way (Elizabeth).*

Some students appear to place a greater emphasis on the input and output section of the defining diagram, particularly for screen design, while other students use the processing section to gain an understanding of the tasks in a coded solution. Chigden chooses to by-pass the defining diagram and moves directly to the event response list. Asked if developing a defining diagram made analysing the problem easier, Chigden stated:

> *I disagree. I usually work out my problem first and then I would go and do the rest. I go back and fill in the defining diagram after everything is done (Chigden).*

In a small program with few inputs and outputs, Tom ignores the defining diagram. Even with a larger program he is usually confused about the processing section of the defining diagram at this stage:

> *In a larger program I use it. I probably leave the task part of the table out because I am not sure what is in the middle yet (Tom).*

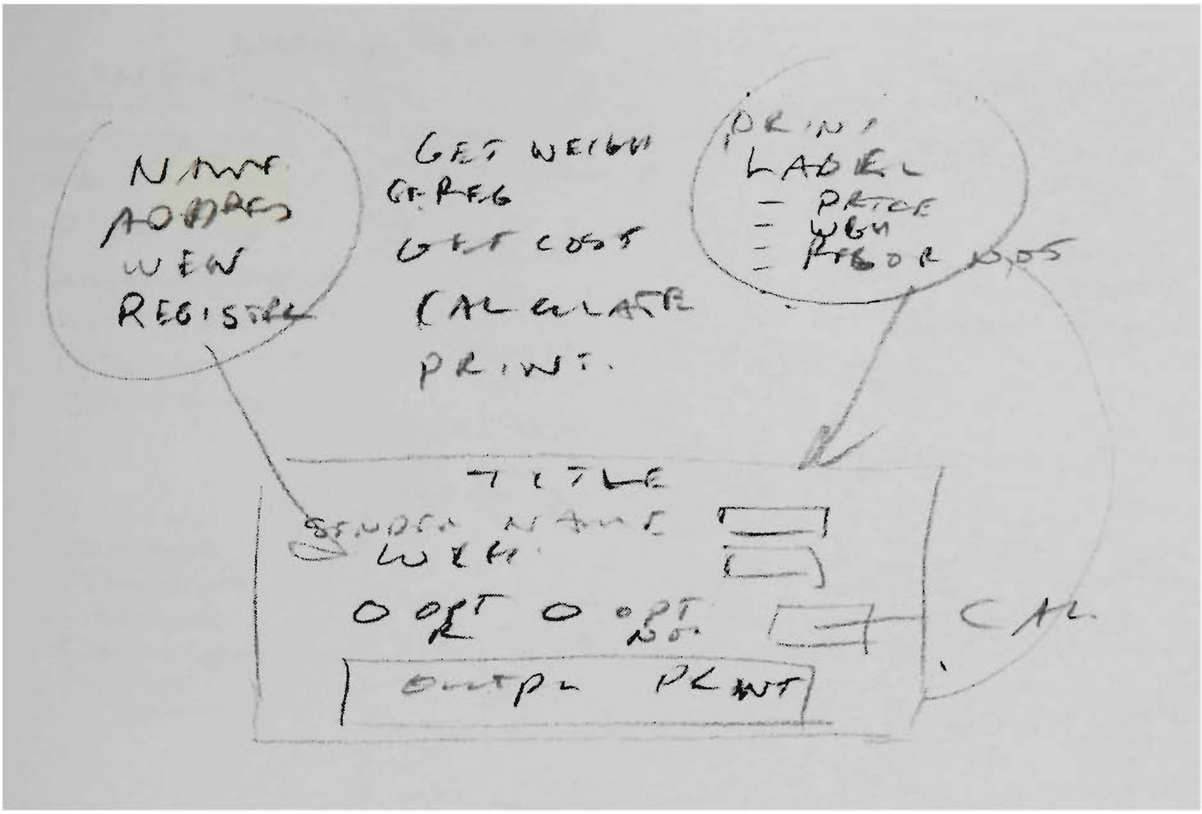
Many of the students appear to use the input and output sections of the defining diagram as a cross-reference to controls on the screen design. In response to the question "Does developing a defining diagram make analysing the problem easier", some students believe:

> *It does in a sense knowing what is coming in and out is more important. And this is it* (points to her screen design) *what is coming in and going out. This is what I am thinking of when I am doing the screen. I am visualising inputs and outputs through the screen (Sylvia).*
>
> *Yes it does. You have the inputs, tasks and outputs. Once that is done, I go about the event response and screen especially. You can go back and say, especially for the screen, I need so many text boxes and so many labels and it helps out with the event response just having the basic thing of what is going to happen (Billy).*
>
> *I would go back to the defining diagram. I work out the inputs I would have to put in - name, address - the outputs - total cost. Then the tasks - calculate, parcel cost and registration cost (doubled or not). Labels have to come out as well. Yes it sort of did, but I would have to have the screen layout before hand and then I would go back to it (Jaranka).*

Similarly, Fred believes that:

> *I would do the defining diagram first. Most definitely because those input and outputs are going to have a relationship on the screen. ... I tend to do the defining diagram and then the screen layout and go back and review it (defining diagram) and find a few things that I have not thought of. Its a bit of a circle (Fred).*

This relationship is clearly evident in the work by Fred on the sample problem in the interview where he draws lines between the inputs of the defining diagram and the screen design.

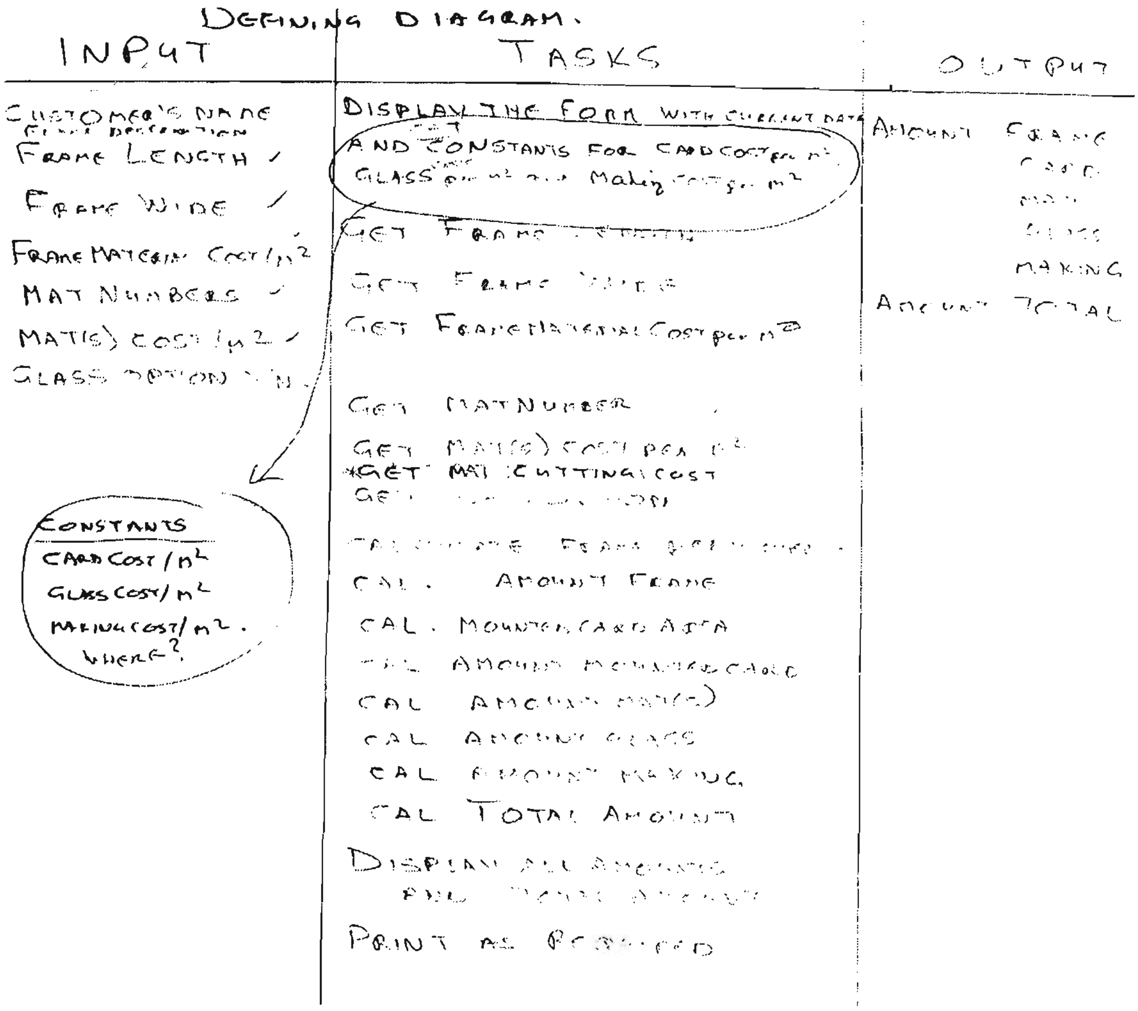


The defining diagram appears to serve a useful purpose in determining the inputs and outputs and cross-referencing these against the inputs and outputs outlined on the screen design.

Others appeared to concentrate on the processing section of the defining diagram.

> *What I try to do, because of my maths background, I look at how things are calculated and then I try to work out how the example is calculated. So I understand the guts of what is going on in a mathematical sense.(Peter).*
>
> *Yes, it involves getting the information from the assignment and putting it into here so I know exactly what I am calculating (Seeana).*
>
> *I tend to do the output first which is like the defining diagram in a phrase more or less. And then I start to put in the inputs and then I look at the inputs then I write the tasks.(Gerald).*

Peter's draft defining diagram for the first assignment in Introduction to Programming shows how he has used the processing section of the defining diagram as a way of identifying tasks in calculating the Total Cost.



DEFINING DIAGRAM.

INPUT

CUSTOMER'S NAME
[illegible] DESCRIPTION
FRAME LENGTH ✓
FRAME WIDE ✓
FRAME MATERIAL COST/m²
MAT NUMBERS ✓
MAT(S) COST/m² ✓
GLASS OPTION Y/N ✓

CONSTANTS
CARD COST/m²
GLASS COST/m²
MAKING COST/m².
WHERE?

TASKS

DISPLAY THE FORM WITH CURRENT DATE
AND CONSTANTS FOR CARD COST per m²
GLASS COST per m² and Making COST per m²
GET FRAME LENGTH
GET FRAME WIDTH
GET FRAME MATERIAL COST per m²
GET MAT NUMBER
GET MAT(S) COST PER m²
*GET MAT CUTTING COST
GET [illegible] OPTION
CAL [illegible] FRAME [illegible]
CAL. AMOUNT FRAME
CAL. MOUNTED CARD AREA
CAL AMOUNT MOUNTED CARD
CAL AMOUNT MAT(S)
CAL AMOUNT GLASS
CAL AMOUNT MAKING
CAL TOTAL AMOUNT
DISPLAY ALL AMOUNTS
AND TOTAL AMOUNT
PRINT AS REQUIRED

OUTPUT

AMOUNT FRAME
CARD
MAT
GLASS
MAKING
AMOUNT TOTAL

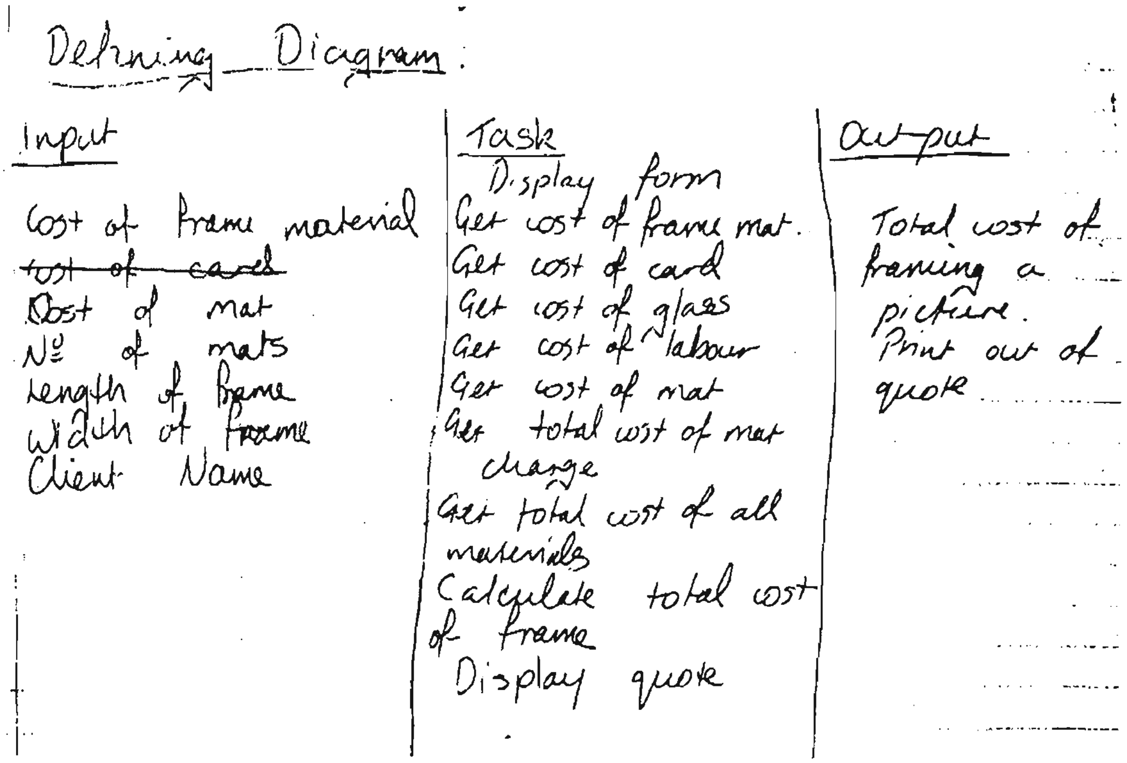
Lecturer O observed that some students had trouble developing a defining diagram.

> *The defining diagram was not done well. I think there was a lot of inconsistency between how people approached them. That may be in part due to the way that I taught it. I tend to get a bit confused myself going on about the distinction between the defining diagram and the event response list in terms of what actually goes into the defining diagram and I tended to have events listed in there as well as data. The defining diagram is almost, might work better perhaps, as a top level DFD where you have the inputs coming in from the outer environment, you have the major processes represented and the major outputs represented. I think it is suppose to represent the same type of information as a level zero DFD.(Lecturer O).*

Lecturer T his undecided on the amount of detail that should be included in the defining diagram:

> *I wondered about the middle part of it. The inputs and outputs are clearly useful but I have been wondering about the tasks. The processes may be useful or not, and I have sort of come to the conclusion in the end that they probably are but they should be very broad statements like using this this and this calculate that rather than ... what rather than how. With the inputs again, I am not total sure, say, you have got to get someone's name and address. Maybe I suspect it would be better to put down in the input name and address but maybe we need to put down surname, first name, address, postcode. I mean how much detail. I don't know what the answer to that is (Lecturer T).*

It was common to have assignment submissions which included printing a form (a *process*). Dianne wrote:



Defining Diagram :

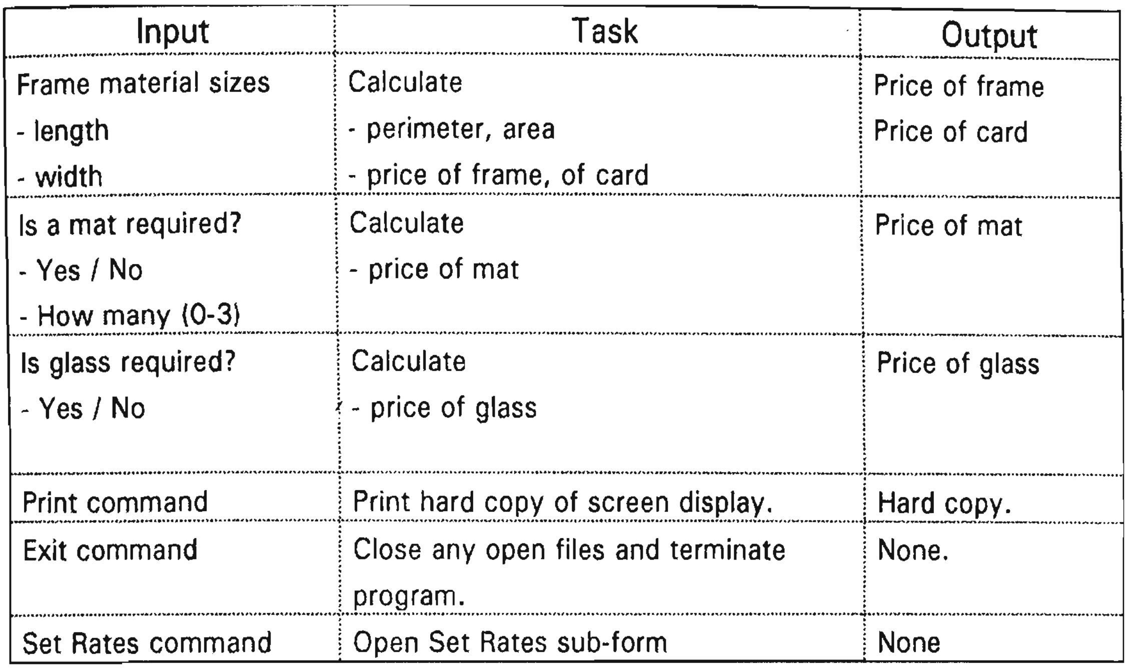
Input	Task	Output
Cost of frame material	Display form	Total cost of framing a picture.
~~cost of card~~	Get cost of frame mat.	Print out of quote
Cost of mat	Get cost of card	
No of mats	Get cost of glass	
Length of frame	Get cost of labour	
Width of frame	Get cost of mat	
Client Name	Get total cost of mat charge	
	Get total cost of all materials	
	Calculate total cost of frame	
	Display quote	

Surprisingly, students tend to be able to identify inputs, processes (such as calculating information) and outputs but tend to see a physical output, such as a printout, as a data item rather than a process.

Consistent with her observation that students tend to underestimate the importance of the problem statement, Mary-Beth adopted a quite different method for the identifying inputs and outputs.

> *I do the whole input output table and I do it in the way that you do it. I actually do it differently to you in terms of, I have an input and I have an output for that input and then I have to fill in the middle. I break it down from one big problem into a whole series of little problems, and this is the way that I have been taught, and then work it from there. And so to do the framing say, you need to get all these details before you can move onto how you calculate it. You have to have this input to get this output and then you work from there. And so it is one step after another so I actually break it down into a series of problems rather than inputs outputs but they are actually physical steps I suppose.*
>
> *I don't define output as absolute final. And the thing that I have found is that it is actually easier for me to do that because I don't necessarily know what the final output is going to be because I am not familiar enough with the code to be able to determine the best. And those sequence of steps I will return to and go back constantly (Mary-Beth).*

This process was adopted by some other students in the classes. One student, George, submitted the following defining diagram for his first assignment:



Input	Task	Output
Frame material sizes - length - width	Calculate - perimeter, area - price of frame, of card	Price of frame Price of card
Is a mat required? - Yes / No - How many (0-3)	Calculate - price of mat	Price of mat
Is glass required? - Yes / No	Calculate - price of glass	Price of glass
Print command	Print hard copy of screen display.	Hard copy.
Exit command	Close any open files and terminate program.	None.
Set Rates command	Open Set Rates sub-form	None

The inputs, which include both input data and events, are matched to a specific process which is in turn matched to an output.

The defining diagram assists with the identification of inputs and outputs prior to moving onto the next stages, or is used by some students as a tool to verify inputs and outputs on the screen layout. The problem of modelling an event driven programming system as an ordered series of steps, as would be the case for a procedural language, are evident at this stage. Some students appear to be confused between a data item and an event, and that an event, such as the selection of an option button, can result in a data input. The processing section of the defining diagram implies that a sequence of operations will lead to the desired output, however, event-driven programming often enables events to occur out of sequence and may lead to a different output from that which is desired. An alternative tool, such as a Data Flow Diagram as hinted at by Lecturer O, may be more advantageous in identifying system requirements than a defining diagram, which is founded on the premise that processes take valid inputs and produce outputs.

6.4 Screen Layout

The design of the screen involves two steps; creating a screen layout and naming objects. Decisions about which objects to include on the screen layout may have been determined, at least partially, by the identification of inputs and outputs in the defining diagram. Moreover, the mode used to process data inputs, outlined in the processing section of the defining diagram, may necessitate additional controls such as command buttons for calculation or lost-focus on objects. The screen layout is the first concrete manifestation of the finished system and one of the aims of this section of the methodology is to provide students with a view of how the front end of the system will look to the user and to identify key controls on the screen. Lecturer M stated the importance he places on the screen layout:

> *I think they have got to draw it at some stage and they have to draw something. But it's probably a bit to do with the way I always think and whenever I need to do something I always draw a sketch or whatever. I get it sorted out in my mind what size and shape it is going to be. Admittedly, people can get away without doing that. ... It is what they start from, it is something concrete, and they should be able, from their inputs and outputs and everything in between, what is going to happen. I think they do see it that way. It is a bit hard to work out whether they have or haven't seen it that way (Lecturer M).*

Lecturer T believes, however, that it is important that the screen layout be done (whether on paper or directly onto a computer) but that it is likely that it will be modified through various iterations as students learn more about *what* can be done:

> *... the sense of knowing what can be done. I mean knowing what you want to do has to be in some way couched in terms of what you can do. If you didn't know anything about computers, for instance, and you think you could have voice input, for instance, you might be in for a rude shock when you discover the current state of voice input. ... What are my range of choices? If I want to get some input, how can I get input? I could use a text box or I could use some combination of buttons, option buttons or check boxes. You have to know there is a list of things possible first to make a choice from them, the problem then is to what extent do you teach that before the methodology (Lecturer T).*

In the subject evaluation sheets completed at the end of the semester, students identified the development of a screen as the most enjoyable aspect of the design methodology. This view was supported by evidence from the interviews:

> *It was fun it was the only thing I enjoy about it. Designing something is great. Do all the screen first (Chigden).*

> *It is the easiest part. I like to do that first on the computer. I am fussy about that (Seeana).*

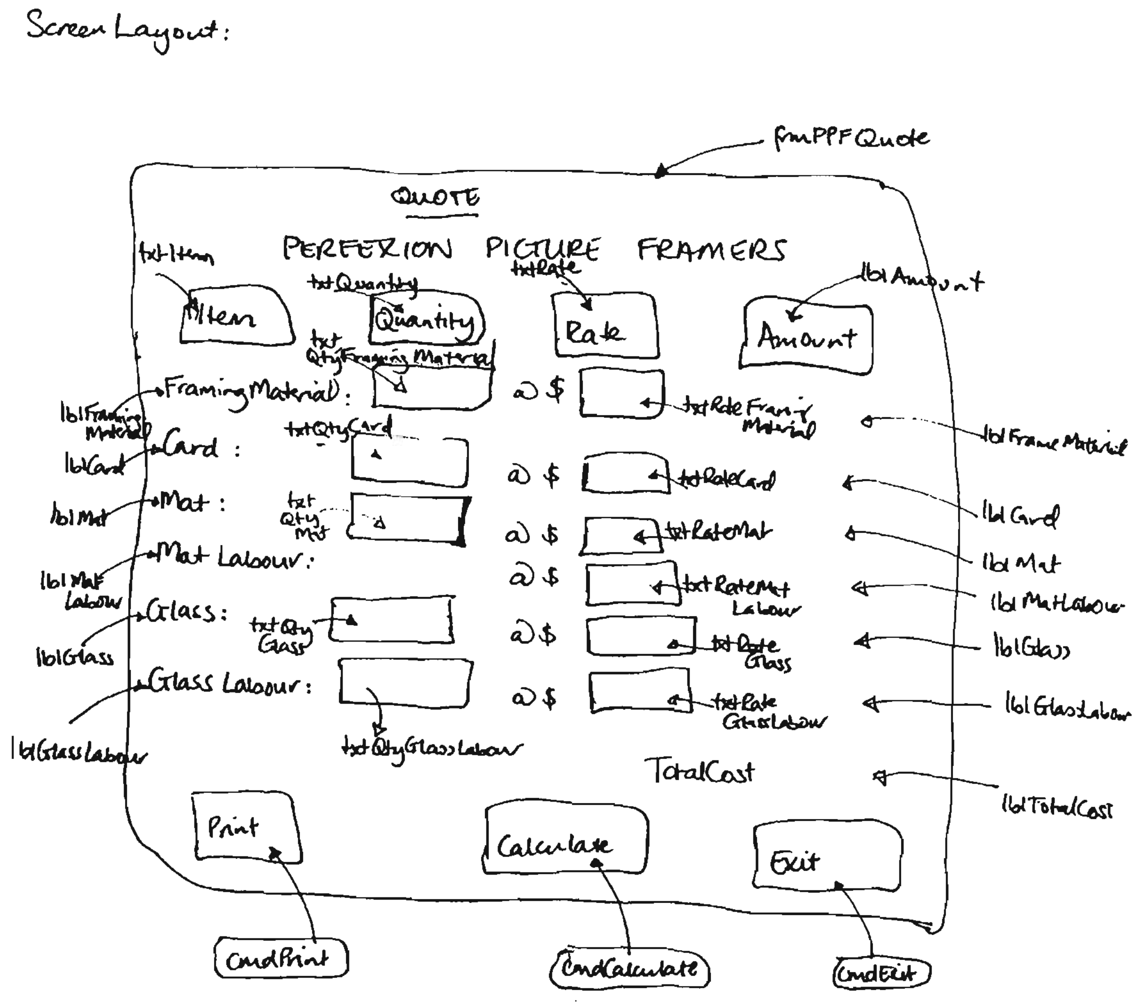
> *I love it. You have done your programming and you get a screen exactly how you want it. With Visual Basic it is so exact whereas with Pascal it was difficult to get it where you wanted. With Visual Basic you can virtually do what you want to (Billy).*

Lecturer O revealed in his interview that his research had found that students in the third year systems implementation subject enjoyed this aspect of event-driven program development as well. He stated:

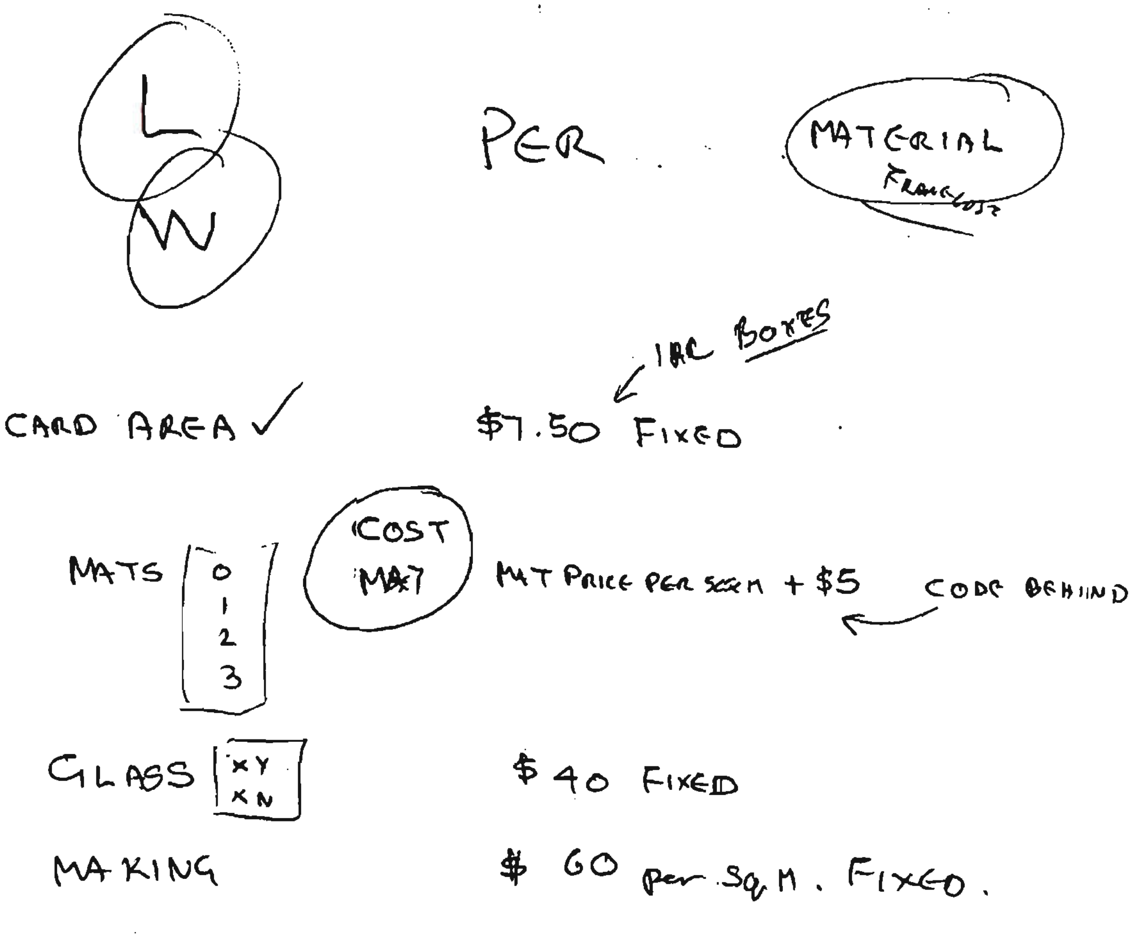
> *They enjoyed producing the user interface stuff and some people might criticise that as just doing pretty colours on the screen. It is a bit more than that. Sure it is playing with pretty colours, I think it is being able to actually construct something that looks good. They get some pride out of that whereas no matter how hard you try it is awfully hard to get a nice looking result in Pascal, for example, unless you were an expert at it.(Lecturer T).*

6.4.1 Development of the Screen Layout

Most students submitted extremely detailed screen layouts in their assignments. A typical screen layout is shown in the figure below.

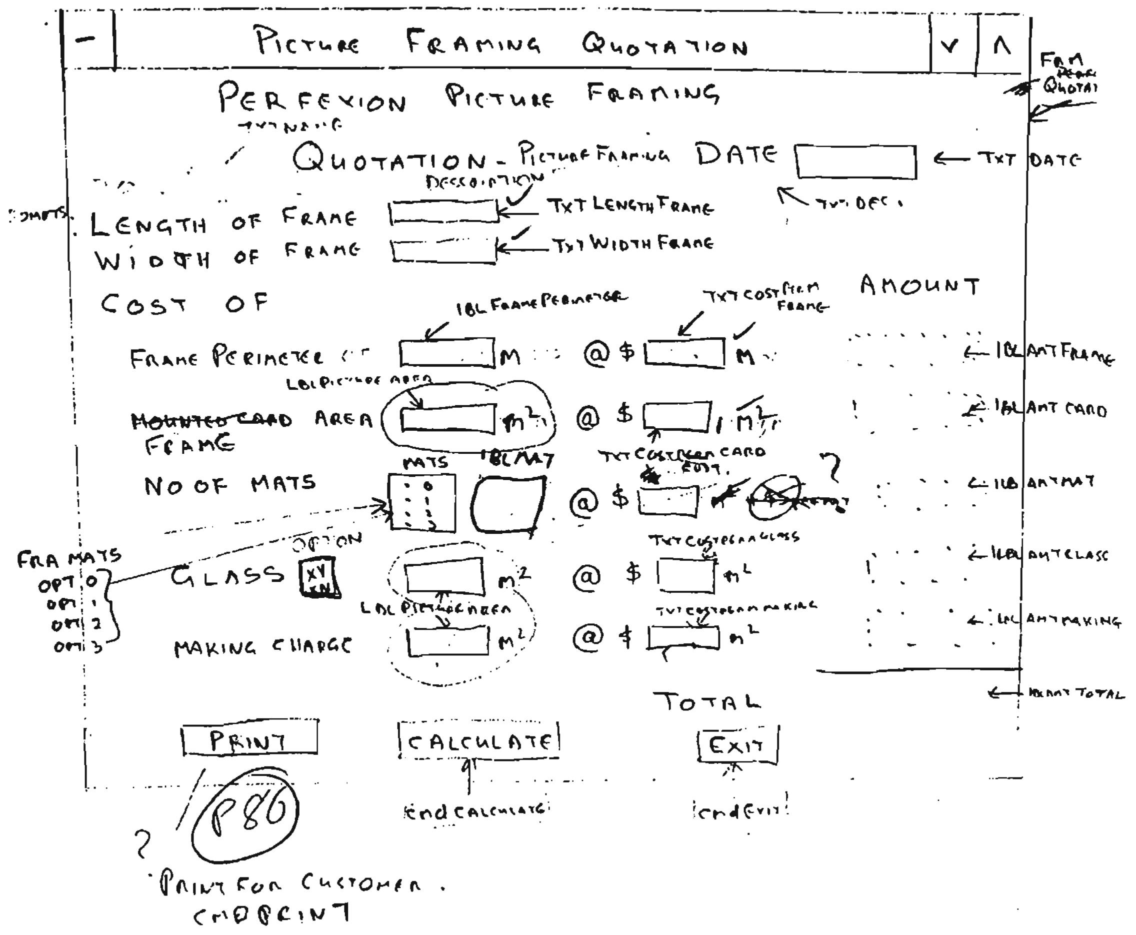


These final screen layouts, however, are often rewritten and hide the amount of editing that has been done by a student to arrive at the final draft. Work by Peter reveals the changes that he made at the various stage. The screen layout below is his first attempt at the screen and denoting inputs and their location on the screen.



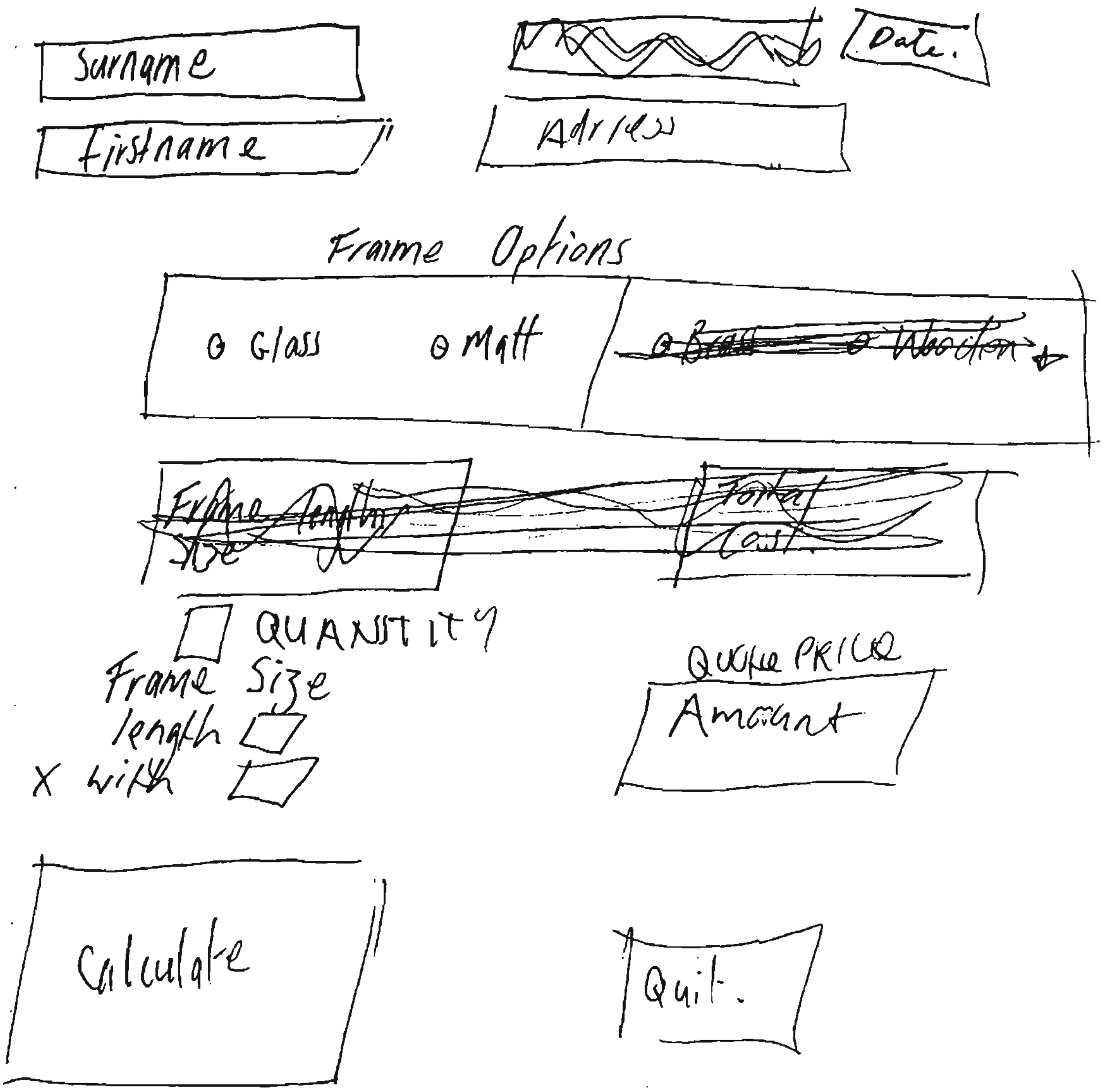
His second iteration builds on these to develop a more detailed screen layout:

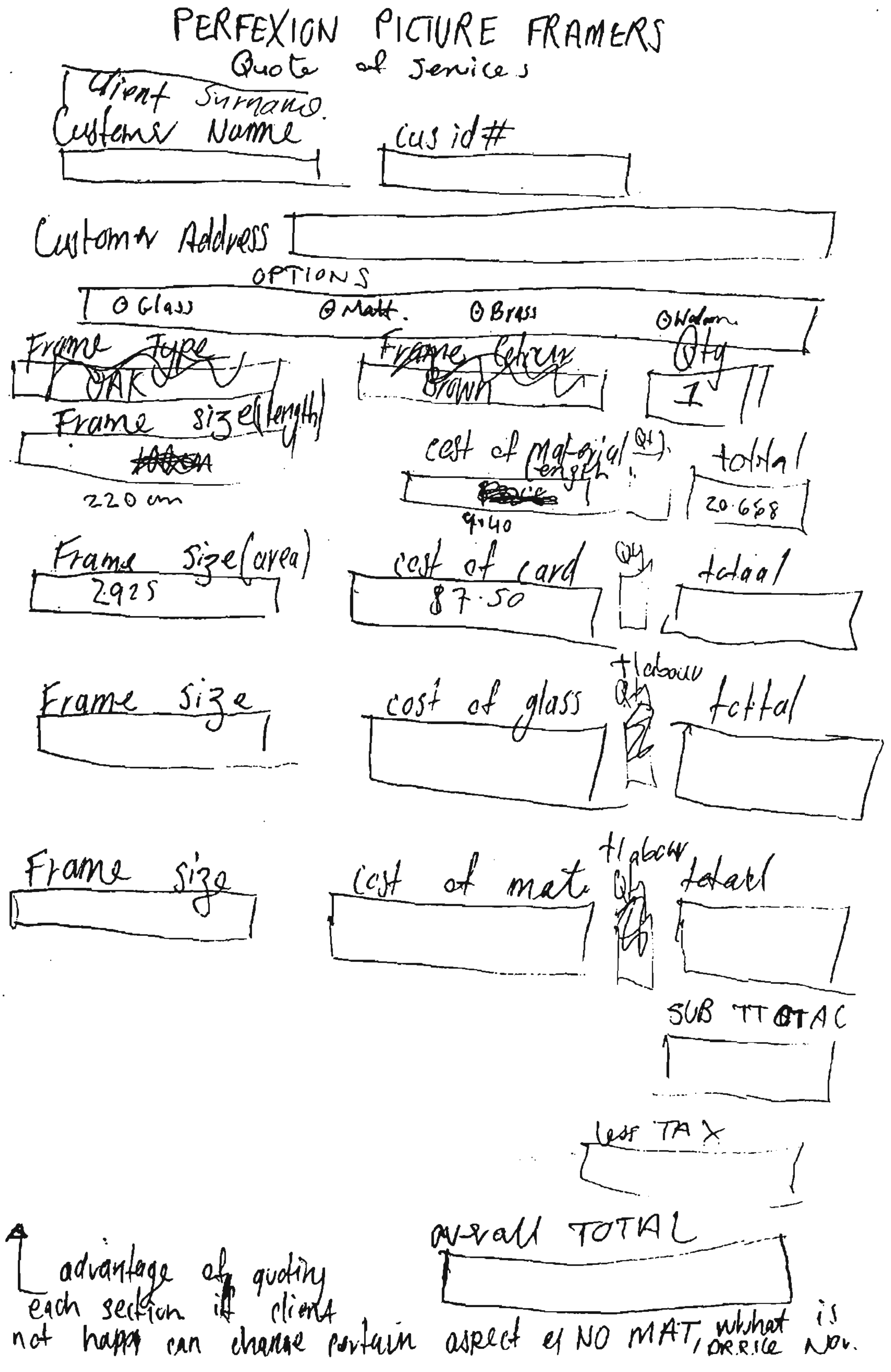
SCREEN LAYOUT



Work by another post graduate student shows a similar amount of editing between iterations:

Brass Wooden Sand. Const.





PERFEXION PICTURE FRAMERS

Quote of Services

Client Surname.

Customer Name

cus id#

Customer Address

OPTIONS

○ Glass ○ Matt. ○ Brass ○ Wedom.

~~Frame Type~~

OAK

~~Frame Colour~~

Brown

Qty

1

Frame size(length)

220 cm

cost of material (Qty) length

9.40

total

20.668

Frame Size(area)

2925

cost of card

87.50

Qty

total

Frame Size

cost of glass

+ labour

Qty

total

Frame size

cost of mat

+ labour

Qty

total

SUB TOTAL

less TAX

over-all TOTAL

advantage of quoting each section if client not happy can change certain aspect e.g. NO MAT, what is price now.

In this latter case, the student has changed the method of input, expanded the output section to include more details, and moved from a calculate button to on change or lost focus. The comments at the bottom of the last screen gives some idea on the reason for the change.

6.4.2 Editing the Screen Layout

It would appear that, once the screen has been developed, students tend not to make major changes to the screen layout, indicating that they prefer to get an accurate picture of the system, as depicted by the screen layout, in this stage of the design methodology.

> *It stays in the corner only because I understand all the processes and how they fit in but I always find once I have done the algorithm it changes it all apart from the problem statement and the defining diagram. I probably wouldn't change the screen because I would think oh my God it would confuse myself even more. Try and make it balance.(Dianne).*
>
> *Do the screen layout. Do it in pencil and then stick to it. although I can change it (Chigden).*
>
> *What I tend to do is draw up boxes. I haven't yet labelled them as text boxes. Some I know straight away, they are text boxes or label boxes. Some are coming to me as I am writing it down, but as I tend to not so much move the boxes around I tend relabel them.(Gerald).*

Lecturer M also observed this to be the case:

> *I found that once they made their screen they stuck with it.*

Lecturer T found that a number of students in his class chose to develop the screen layout directly on the computer. In justifying this approach he states:

> *I don't think there is a major problem with that. You would like to think they use paper first but it is a bit like saying do you sketch your essays out on paper first, some people say they do but.. . The mere fact that you can put something on the screen and move it around to where you want I think is important and one of the differences is that means you don't necessarily have to sketch it out on paper firstly. Another issue, take that example of folded or not, where do you make a decision that that is not a text box, you make a decision that that is an option button rather than a text box, I am not sure. I think in that first step they have to identify that there is some input that will determine that, then there has to be this leap of faith that this sort of input comes a text box and this one becomes an option button.(Lecturer T).*

Moreover, a completed screen design, however, does not necessarily lead to the development of a fully operational system or one that matches the system requirements. A screen is the front end to a system but appears to give some students the illusion that the system is almost complete. Asked if students identified more with the design of a screen than other steps in program design, Lecturer O responded:

> *Oh yes because it was so much fun to do. Many of the programs I had submitted didn't actually work at all, most of them did not work properly and only a handful gave up the statistics accurately. But every one of them had two forms.(Lecturer O).*

One could liken the development of parts of an event-driven system to the building of a house. Once developed the screen appears to give a novice programmer a false impression that the program is nearly finished and the rest can be done in an equal amount of time. In the building industry, the frame of a house can be built very quickly, often giving customers the view that the house can be completed quickly. There were numerous times in class, particularly in the early weeks of the semester, when student underestimated the work on their assignments after the development of the screen. Statements such as "I have done the screen, I am nearly finished" or "I just have the code to do" were common responses to queries about their progress.

A number of students found that developing a screen helped guide them through the next steps of the design process.

> *Very important. It helps to understand the whole thing. I go back to it all the time even when I write down the program variables.(Gerald).*
>
> *Very important. Once I saw it on the screen and played around with it a bit I knew exactly what to do - well not exactly - I knew what to do.(Dianne).*
>
> *Yes you get to see what you want to do (Tom).*
>
> *Fairly important to be able to work out the other parts (Gerald).*

Elizabeth, a student who had completed subjects in Pascal and COBOL at a previous institution found that the screen helped give her a picture of what was required:

> *Because it is visual, you can see it. You create what you want to see first. You have to see. You need a vision. When it comes to cutting code, I think you do need to know what you are doing before you actually do something. When you do Visual Basic you do get nice boxes, your layout looks nice whereas with Pascal I'd say, 'Why are you doing this, why are you doing that?' At the end it hits you what you have done. 'Oh, is that why we have done it' You need to know what your objectives are before you can cut code. Because if you don't understand something you are not going to put your best into it. You are not going to really do your best because you don't understand it ... If I can see it I can do it.(Elizabeth).*

Joanne and Billy, other students with prior experience in a procedural programming language found that the screen layout helped as well. Joanne appeared to have put great importance on the screen layout:

> *I found it pretty good. You had more idea of the problem, what you were meant to be doing with the problem, so you were not just writing and not realising what it has got to do with. Whereas if you have got the screen layout you know what you are actually doing - it is more realistic I find for the problem (Joanne).*

whereas Billy only chose to do the screen layout at times:

> *I have a mental picture of the screen and then do the event response list and then you can go off and draw it if you like.(Billy).*

Fred, a part-time student who works for a company developing computer-based accounting systems, related some of his work experience to the importance of the screen layout stage:

> *We don't go anywhere unless we get the screen layout right 'cause that, from my point of view, says a lot about what is going to be happening from there on. So much so, that in that case, I spent 2 days in overalls talking with every person who would have involvement with it. From that we built up a screen layout and that tells you a lot more about what is required from the program. Usually do all the planning first. I am reluctant to enter the screen on the computer just in case it needs to change (Fred).*

Mary-Beth, a mature-age student, identified the dangers of placing too much importance upon the screen design at the expense of other steps in the process.

> *Yes I see it as important but I don't see it as more important than anything else, say to have the screen and work from the screen. The other thing is that the screen is the very last stage of the process and working from that it might be able to be a visual incentive for people to be able to do the other steps but it isn't really necessarily going to help them in the long run. You can make it look nice and really pretty. And I put it up in the last class in Arts just to show people that there are options that they have got to understand; that they are given the base knowledge but they have got to be able to understand that there are a whole lot of options out there that they need to investigate in terms of trying to bring this stuff together. Put up different buttons in different locations - people interact with it in different ways (Mary-Beth).*

Event-driven languages, such as Visual Basic, require systems to be built using forms. The user interface components of a system under development are pre-programmed within the environment. To create an acceptable user interface using a procedural language, a person would often be required to program a number of trivial and repetitive segments of code. Visual Basic frees the programmer from this task allowing them to concentrate on other, arguably more important programming tasks. Screen design, although important in introductory subjects using procedural programming languages, has been elevated to a greater level of importance in windows-based systems. Lecturer O believes that:

> *Effective user interface design is essential if a system is to be successful. Visual programming environments do not so much hijack the User Interface design process as facilitate the use of different interface controls best suited to the task at hand, while eliminating the need for the programmer to develop trivial code.(Lecturer O).*

Although the main aim of developing a screen layout in this design methodology is give students a concrete picture of how the front end of the system will look, other screen design issues may need to be considered in introductory programming subjects. As Lecturer O points out, students are empowered with power design tools in these early subjects without having been formally trained in user interface design.

> *One of the great benefits of Visual Basic and also one of the short comings. One of the good things about the old fashioned systems we use to use was that they were*

> *so limited in what you could do on the screen design. You would teach students good screen design in Pascal in three and half minutes because they were so limited in what they could do, you know, don't use to many colours, nothing flashes anywhere. ....You can teach them screen design in Pascal very quickly in Visual Basic what I did was walk around behind them and say "Oh, that is ugly and point out what was wrong you are going to lose marks for that suggesting that if your not too sure where to put things have a look at a few programs. OK button is at the bottom right hand corner. Try and have the inputs starting at the top left and finishing at the bottom right hand cause that is the way that people tend to work. The last thing that they do should be on the right so don't have the close button on the left. It enables them to do all the very bad things. It enables students to develop a whole range of bad habits - we don't have time to teach them interface design but here we have all the tools that enable them to build really woeful interfaces (Lecturer O).*

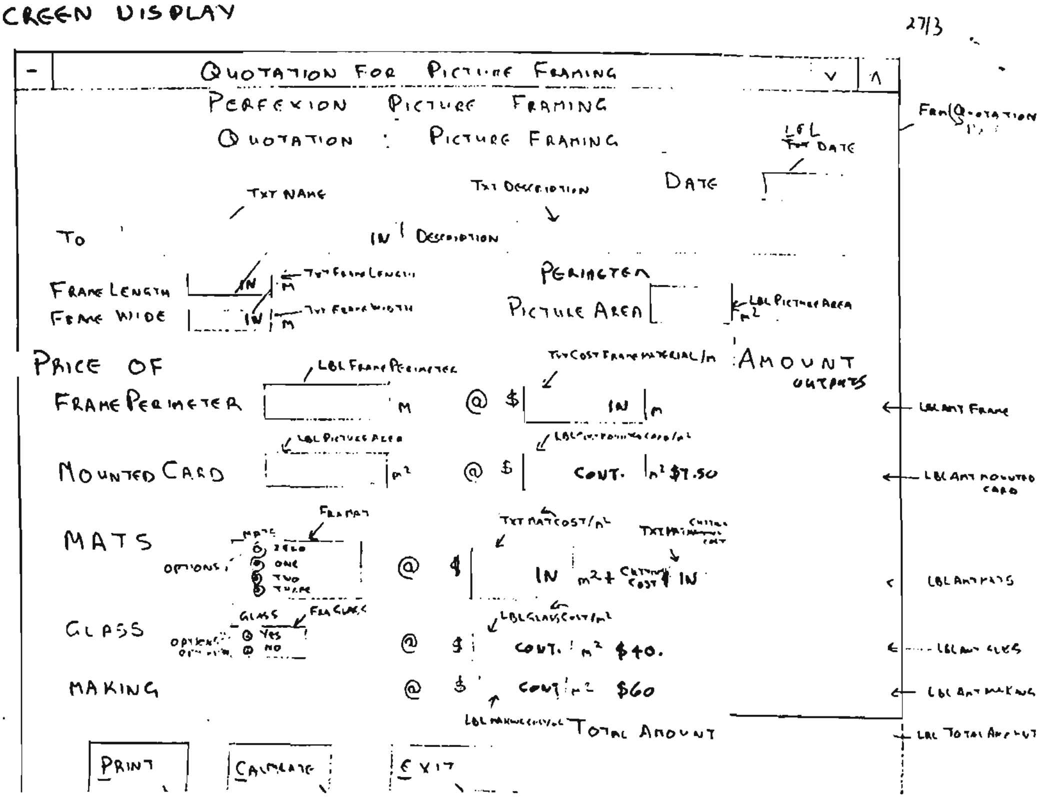
6.4.3 Relationship to Other Steps in the Design Methodology

The chicken and egg argument applies in the case of screen design and program structure; Does the screen design determine the structure of system or the structure of the system determine the screen design? Or do they evolve together? It is clear that aspects of the screen are almost predetermined by the inputs and outputs and some students go from the defining diagram to the screen:

> *That is the way to go. Once I have done the defining diagram this needs to go into a text box, this into a label box. Because I don't know much about the code I find it good (Peter).*

An early version of a screen layout for an assignment by Peter shows how he matches elements of the defining diagram, such as inputs (denoted by 'IN') and constants (denoted by 'CONT') to the screen controls to:

3. SCREEN DISPLAY



Moreover, observations reveal that many students initially opted for their preferred method of processing data which determined whether the system would have an all embracing calculate button or process data based on change or lost focus in controls. As mentioned in other sections, most students were comfortable with a command button which would have a method that would validate and calculate the output, or they started with a command button and then eventually eliminated it by transferring code to lost-focus or change events. Elizabeth outlined a typical approach:

> *You draw the screen nice and pretty and then you try do the calculate or click button and then try to do the program, the writing behind it. I probably jot a few notes down like this equal this where I put everything down but mainly I try to write the code out without doing the algorithm or pseudocode.(Elizabeth).*

Many students preferred to do the screen layout either on a computer or on paper before or while developing a defining diagram:

> *Screen layout, actually I do it just before or at the same time as the defining diagram. Once you know what is going to be on the screen it helps to be able to see what input is required by setting up the screen (Gerald).*
>
> *I tend to figure out the screen first and I figure out what text boxes and all that and then I go back to the defining diagram and try and figure out what actually has to be put in or calculated or whatever.(Jaranka).*
>
> *The task part of the defining diagram would not be done until I did the screen layout. So in the screen I know we have done a calculate so I would go back to the task part of the defining diagram and write it down.(Tom).*

> *Once I have a screen layout, the programmer and I sit together and we scratch out on the board which now I call the defining diagram, these are the thing I want to put into it and these are the things I want to get out of it - the middle column you, the programmer, work it out (Fred).*

In the interviews, the majority of students did the screen layout after the problem statement and before the defining diagram (if at all). This may have been due to the fact that the problem was quite small and because of time constraints, nevertheless, it appears that the screen is both a check on system inputs and outputs and/or helps formulate these requirements.

Only one student Billy appeared to develop a formal event response list (the next stage in the design methodology) before building a screen.

> *I like to do the screen after the event response list. Once you have gone through you event response list you can say this and this is going to happen and you might remember that you need a command button to print and so go back and event clicks print, program prints screen and that is something you can draw in. You don't have to draw it do your event response oh forgot this and then go back and draw it. So it is more systematic.(Billy).*

Observations did not find any students in classes who developed an event response list prior to the screen although it is likely that some students tend to develop a mental picture of some events and their responses particularly towards the latter part of designing the screen. The screen layout, it would appear, is not sufficient for most students to enable them to move onto developing program code. Observations reveal that, when given a system to modify, most students in classes became confused on how to make changes. In the final assignment of the semester, students were required to develop a booking system for a pleasure cruiser where label boxes represented each cabin. The system used random-access files to store booked data. In the past, files and file manipulation have proved to be difficult for students to understand, so a skeleton version of the system including one cabin label box and some commands, was handed out to students. No documentation was handed out although the system requirements were outlined in the assignment. Students were to add label boxes and additional features such as saving and modifying data. The fully operational system was demonstrated to students and they were then required to proceed. It was observed that nearly all the students were unsure how to add features to the system beyond adding additional label boxes. After a lengthy discussion, it was found that most students were trying to use the screen as a hint to what modifications to make rather than matching the system functions to events. We then re-engineered an event response list which helped them identify how to implement the required changes.

Although the screen provides students with a good guide on how the system will appear to the user and assist then in further steps, it is grossly inadequate in determining system structure.

Sadly, it would appear that students place too much emphasis on screen design at the expense of other stages in the design process, in particular the event response list. The event response list, in some formal sense, appears to be critical to the documentation and development of these types of systems.

6.5 Event Response List

In event-driven programming, a system moves from one static state to another by responding to an event. On an event occurring, a system follows a predetermined path according to programmed instructions included under an event procedure of a particular object. Thus, although two screens may look identical on different systems, there may be a significant difference between the two systems depending upon the events and their responses. The event-response list, therefore, is instrumental in determining the structure and operation of a system. In this design methodology, the event-response list has the added aims of assisting students identify where to place the code and to provide a systematic method for dividing up the programming task and developing the code in event and general procedures.

The analysis found that the concept of an event, although apparently easy to describe, in reality is very difficult for most students to understand. Moreover, lecturers often contribute to this confusion by modifying the definition of an event or placing a different interpretation on an event at different stages in a course. Students were confused about the following four areas:

The *boundaries* of an event - that is, whether to include prior actions in an event such as an input to another object

The *context* of the event - whether events should be viewed from a *programmer's* or a user's perspective or both

The *status* of an event - the distinction between active and passive events

The *level of detail* in a response - whether to include multiple paths such as for validation.

6.5.1 Overview

There was a mixed reaction from students to the event response list:

> *I have some problems sometimes with the events. What happens when and what triggers off this reaction. Not quite clear all the time about it. Therefore I wonder where to put the code. ... To me, not overwhelm but I would like to say whelming. I am bowled over completely by it. But it is not just getting to grips with Visual Basic with me I have to get my mind and my fingers round the computer itself. Clicking pressing things those basic things I have to get. For me it is two things, or more than two things, it is learning all that and learning the language. So it is a lot for me. I am not crushed by it yet. I will stay with it. (Gerald)*

> *The theory is OK but the hands on is not. In front of a computer I spend heaps of time. ... Theory and practise does not go hand in hand (Dianne)*
>
> *I never do the event response list because to me, maybe that is something to do with the input output to because maybe that is where I do a lot of the event stuff. (Mary-Beth)*
>
> *Not really useful on a small scale but for larger ones it is. It sort of maps it out exactly what is going to happen when you do this and what the response will be and makes sure it is doing what it should be doing (Seana).*
>
> *Probably the easiest part apart from the problem statement. Cause there a lot of examples in the book and they tend to become quite similar like pressing the exit button. When you have to input data and the response, it should come out calculated I can follow that quite easily(Jaranka).*
>
> *It is quite good ... Once you have worked out all the possible events it makes it a lot easier, because you know exactly what is going to happen at a particular stage of the program. (Billy)*
>
> *The event - response I kind of understand a little bit. The algorithm is OK. (Elizabeth)*
>
> *It is beneficial. I feel though as if I have done it for nothing unless I get the code done first? (Sylvia)*

The analysis of the data from the event-response list revealed that students appeared to benefit in latter steps of the process from identification of the events but they had trouble interpreting the exact nature of an event. The text book used by students in the introductory programming subject at Victoria University of Technology sometimes varies the description of an event to include prior inputs or *passive* events such as turning one option button off as another one is turned on. Moreover, lecturers and workshop leaders appear to vary their interpretation of an event depending on the detail of the system ie. whether validation of inputs is to be included or to reinforce the ordered nature of inputs to obtain an output. Although, students had trouble with some types of events ie. external-related events, the analysis revealed that the problems students had interpreting and explaining events and responses can be broken into four main categories:

The *boundaries* of an event
The *context* of the event
The *status* of an event
and the *level of detail* in a response.

The following program will be used as a running example to help explain each of these categories .

> On the click of a command button, an event-driven system takes inputs from two text boxes, Hours Worked and Pay Rate, and the true value from a pair of option buttons, Part-time or Full-time, and calculates the Gross Wage for an employee and places the value in a label box. On changing the value of the Gross Pay label box, that method calculates the Tax and Net Pay and displays each of these in a label boxes.

6.5.2 Event Boundaries

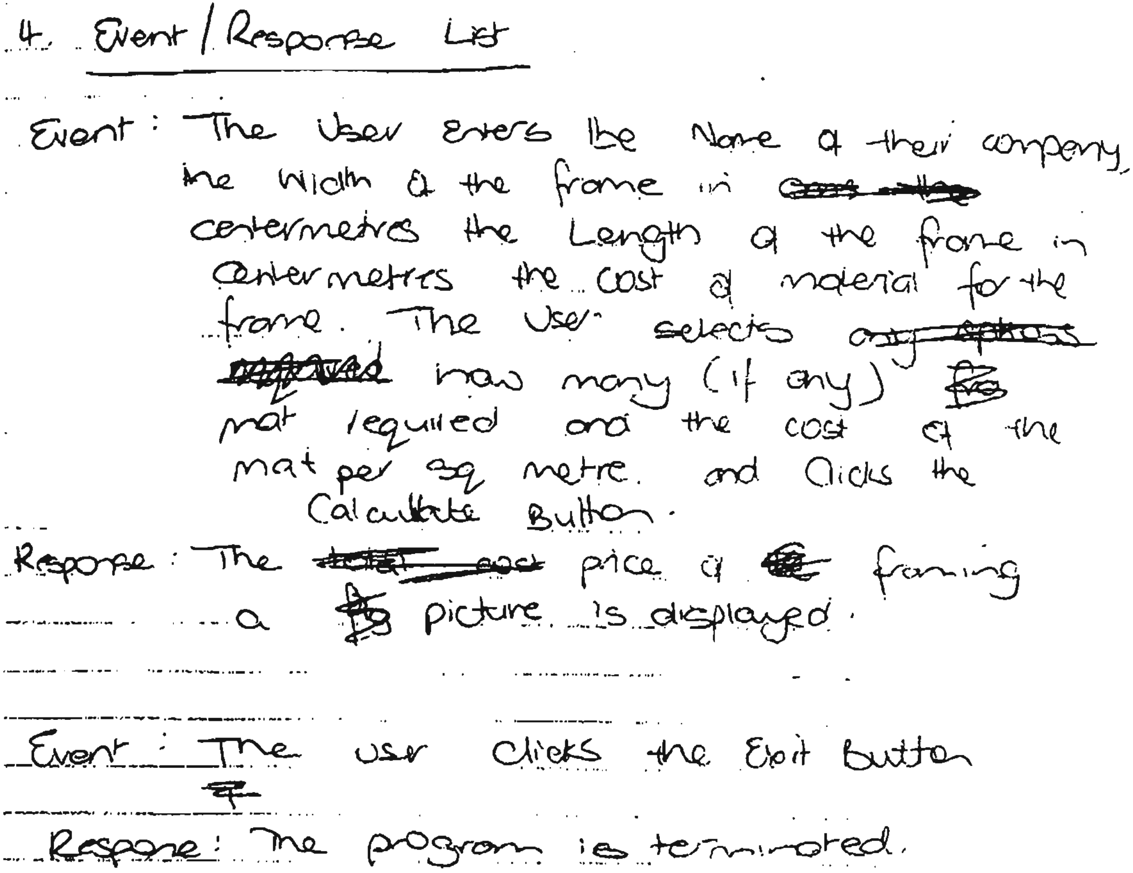
Observations reveal that a major problem faced by most students was understanding what was meant by an event. In theory, although an event is relatively easy to define (ie. an action that results in the execution of a method or section of code), in practice an event appears to take on different meanings at various stages throughout a programming subject. Analysis of the data reveals that these different interpretations of what is meant by an event creates confusion in the minds of some students and their understanding of the concept of event-driven programming. As mentioned earlier, students appear to understand external events such as the click of a mouse on a button and most internal events such as form load. The concept of an external-related event, such as on-change however, can be difficult for some students to identify and to code. This is clearly evident as a significant number of students opted for a control button to do most of the validation and calculations rather than use on-change or lost-focus events in their assignments.

> *Yes, I haven't worked out how to do the other ones. I was going to do different for the last assignment but I am not 100% sure.(Elizabeth)*
>
> *I would have anything other than the Calculate button. That is all I have been shown. (Dianne)*
>
> *I am having trouble understanding where the code will go. If this one changes it automatically changes the other. (Mark)*
>
> *It is a lot more difficult than the cmdCalculate. (Billy)*

Some students started with a Calculate button and then transferred code to on-change or lost-focus events:

> *What I do is put it through a calculate button firstly, simplify the problem then make sure everything is working and then take it out and get rid of the calculate button. ... What I did in the second assignment was have a calculate button there and then created all the code and then removed it. (Clinton)*

The most common variation on the strict definition of an event was to extend the boundaries to include a number of prior steps or inputs. The first draft of Sylvia's event-response list included the following events:



4. Event / Response List

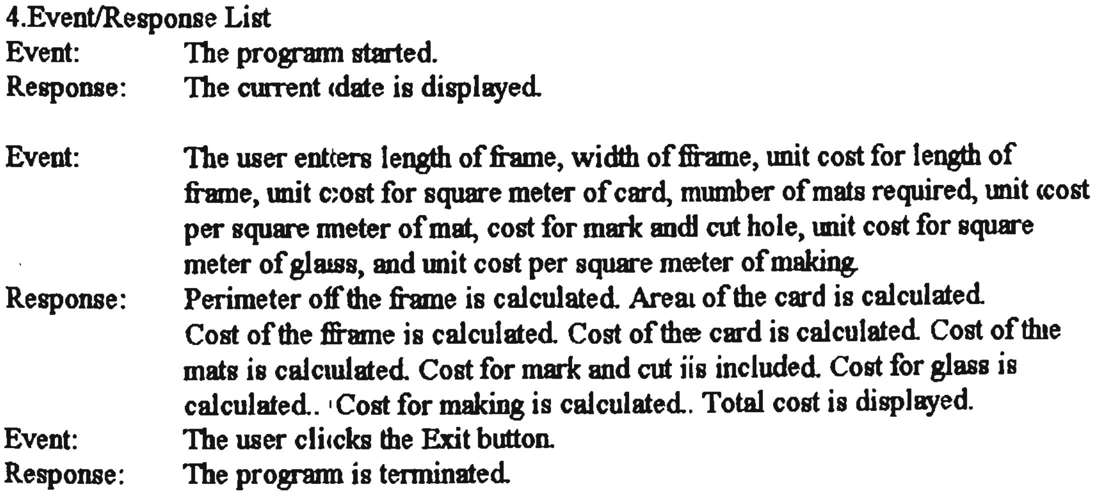
Event : The User enters the Name of their company the Width of the frame in centermetres the Length of the frame in Centermetres the cost of material for the frame. The User selects ~~any options~~ ~~required~~ how many (if any) mat required and the cost of the mat per sq metre. and Clicks the Calculate Button.

Response : The ~~total cost~~ price of framing a picture is displayed.

Event : The user clicks the Exit button

Response : The program is terminated.

Similarly, Ruby, another post-graduate student, submitted the following event-response list:



4.Event/Response List

Event:	**The program started.**
Response:	**The current ‹date is displayed.**
Event:	**The user entters length of frame, width of ffiame, unit cost for length of frame, unit c:ost for square meter of card, mumber of mats required, unit ‹cost per square mmeter of mat, cost for mark andl cut hole, unit cost for square meter of glasss, and unit cost per square mæter of making**
Response:	**Perimeter off the frame is calculated. Areaı of the card is calculated. Cost of the ffiame is calculated. Cost of thæ card is calculated. Cost of tlie mats is calcıulated. Cost for mark and cut iis included. Cost for glass is calculated.. ˈCost for making is calculated.. Total cost is displayed.**
Event:	**The user cliıcks the Exit button.**
Response:	**The program is terminated.**

Note in both cases the large number of inputs prior to the actual event of clicking a command button.

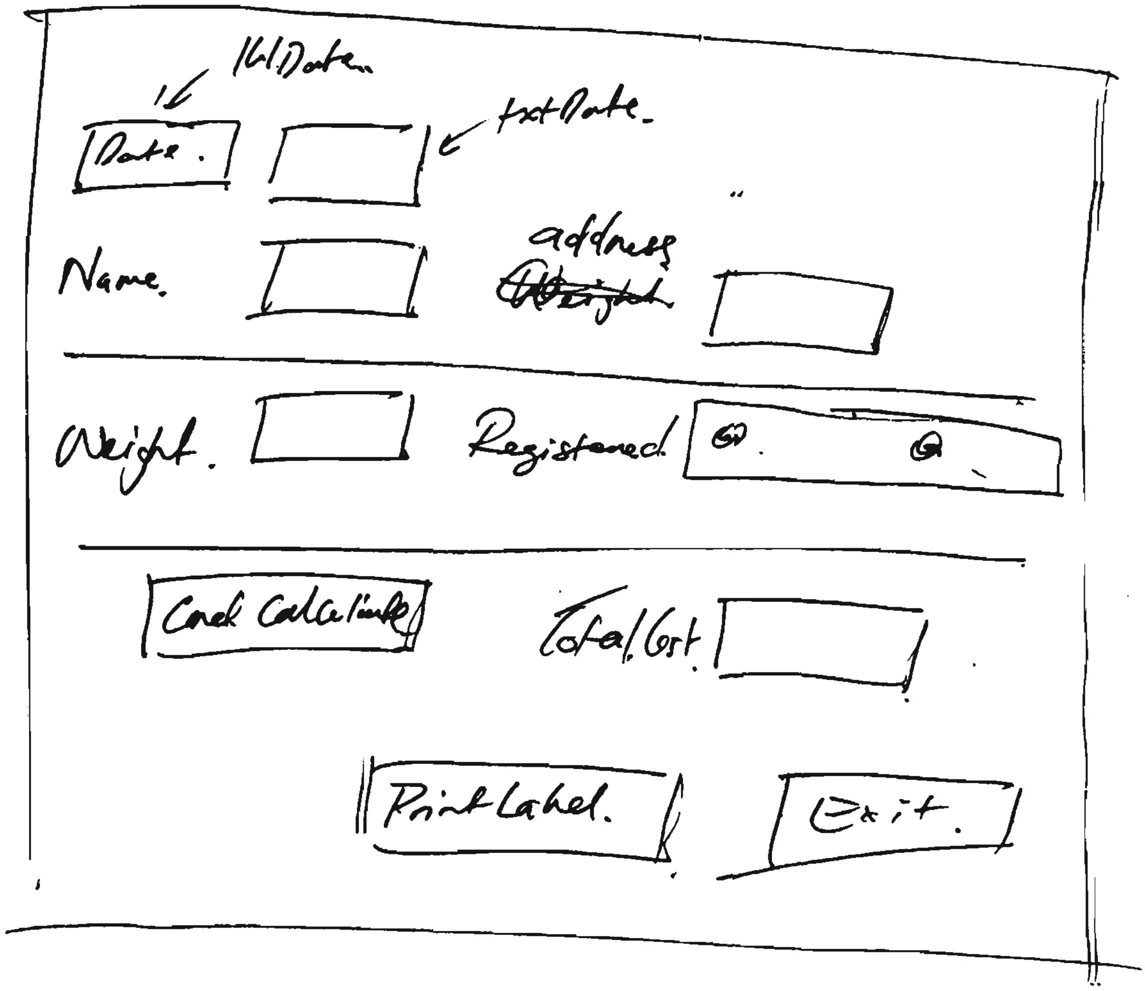
In the example above, it may be possible to state the first event as:

Event:	The user inputs the hours worked, the pay rate, selects part-time or full-time and clicks the Calculate button
Response:	Display Gross Pay

In this case the event is modified to include three inputs but does not necessarily check that they are present or correct. In theory, the event is not a series of ordered steps but some students appear to benefit from outlining these steps in their event response list. In the early stages of the introduction to programming subject, students appear to build a mental model which has the computer program as controlling the system not, as is the case in event-driven systems, where the user can determine the order of events. Students have a better understanding of the concept of an event-driven system in the latter part of the subject where they tend to include validation of inputs and disabling of controls at that time. Their events drop references to prior inputs. Georgina was the only person interviewed who specifically mention this anomaly:

> *When you come to the event response list there seems to be a difference between what you do next and what happens.*
>
> Question: To you then, the event-response list is confusing?
>
> *Yes, to me it simply means it is like explaining what do you do next and what happens. It is clear on what the events are and their responses. I understand that. For example you say inputting data itself is not an event but when you read in the book they say event input name, response is this.(Georgina)*

The defining diagram outlines the inputs required for calculation and this may be one of the reasons why students quote inputs as part of their events. The analysis, however, suggests that it reflects the problems students have understanding the nature of control in event-driven systems. In the example in the interviews, only five students did not use a Calculate button. Although the example may have lent itself to this particular structure, most students had difficultly describing the events when the calculate button was removed. Moreover, it was interesting to observe how two students in the interviews intended to manipulate their screen designs with the intention of forcing input in a particular order. Billy's screen layout was:

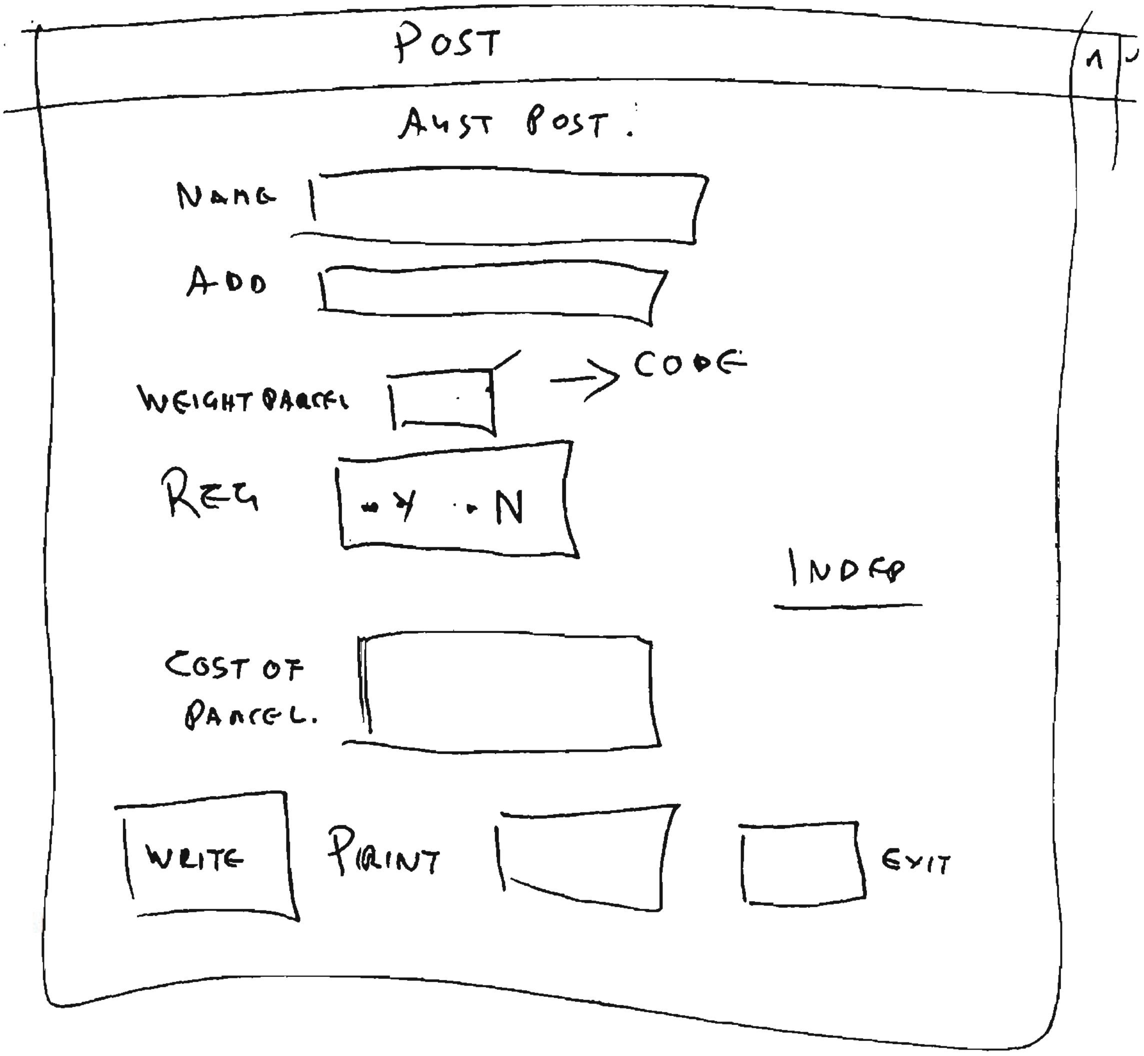


When he was asked to identify the events if the calculate button was removed, he stated:

> *Put in the weight. Move the option reg or not around so that appears before the weight - the weight is the last input.*
> Question: What happens if they had to change the registered status, where would the code go?
> *You could link it to weight and the option button I suppose?*
> *It is a lot more difficult than the cmdCalculate.(Billy)*

In this case, Billy appears to believe that the inputs will be entered in order from top of the screen to the bottom and he fails to understand that the user determines the order of events.

In the interview, Peter developed a screen layout without a calculate button:



and identified the on change of weight as the event for the calculation of the charge. Asked if the code should go anywhere else, Peter responded:

> *Should be behind the option boxes - No I think I can get around it*

Although Peter has opted to use the on-change event, he appears to require a strict order of inputs in his system and hints that he would change the screen layout to alter the order of inputs and *get around* adding another event.

Lecturer M believes that students were confused about events:

> *It was probably the confusing part of it. It wasn't to me but I think it might have been to them. I think that might have been just too much because it was quite obvious to us that you type in the telephone no etc then do that. I don't think they really understood that most of those things did not matter. But maybe we should have picked out the ones that really did cause the event, I don't know how you really put it in. ... I think we are trying to put too much into the event. We are trying to say to them this is how you will operate the program, this is like a set of user instructions because you have to fill in this and this and this and then click the button. Now this is not what we are really after, the event is just click it. This might give them a better idea of*

> *assigning a particular bit of code to a particular event on a particular object.*

Similarly, Lecturer O believes students were confused about what to include in an event:

> *I don't know - I didn't get really narked if they left out the exit button they tended to do either simple or too complex. Some of them would go through and detail every text box or they chose too long complex events where they meant to say when you press the OK button, they weren't too sure how to handle things like exceptions like validation events for example and there were a variety, some would say when you enter the data into it if this and if that, they would have all the ifs in the response, others would have the data entry as one event, the fact that it was wrong was another event.*

From observations, the failure to adhere to describing systems in terms of true events causes immense problems for students particularly those who are struggling to understand the concept of event- driven programming. Including a sequence of inputs or steps in the description of an event may help some students understand how to derive an output but it creates the illusion that the system will automatically follow these steps and emphasises a procedural approach in an event-driven environment. This appears to be very confusing for some students.

6.5.3 Event Context

An event can be viewed from two different perspectives or *contexts*, a user or a programmer perspective. The response to an event from a *user* perspective is the change or combined set of changes to the observable parts of the system; it may be a new form on the screen, an output added to a form, or intentionally saving a file. Using our example above, the event response list could be written as:

Event:	Click Calculate button
Response:	Display Gross Pay, Tax and Net Pay

In this case, the user observes one event changing all the outputs. The system, however, has two events; one that calculates Gross Pay and another that calculates Tax and Net Pay:

Event:	Click Calculate button
Response:	Display Gross Pay

Event:	On change of Gross Pay
Response:	Display Tax and Net Pay

The first event, the click of the calculate button, takes the three inputs and uses these to calculate the Gross Pay. The second event, changing the Gross Pay, calculates and displays the Tax and Net Pay. In this case, it is only possible to identify both events if a person, usually a programmer, has a knowledge of the structure of the system. Moreover, some events may lead to changes in the system state which are hidden from the user.

Throughout the semesters, students were told to describe events for a system from a programmers perspective. Nevertheless, in the early weeks of the subject in particular, most students who up until that point in time had only been users of applications, had difficulty identifying events from a programmer's perspective. The difference in events and responses from a user and programmer perspective was most evident when moving between forms on a multi-form system, when one event changed the value of a variable prior to the calculation of an output by another event or in *system* events such as on form load. When moving from one form to another, changes are often made to the new form prior to its display - there are often many options open to the programmer to make these changes. Similarly, the user is often oblivious to the operations that can be done on the initial loading of a system such as opening files and setting restrictions. Finally, while operating a system, the user may input data or change options values which in turn can change the value of variables or trigger the calculation of some interim values prior to calculation of the final outputs. As an example Irena wrote:

Event: The user enters an even number of pages and selects double side option button.
Response: The number of pages to be copied is divided by 2.

Event: The user enters an odd number of pages and selects double side option button.
Response: The number changes to be copied is divided by 2 and on the result is added 0.5.

It is unlikely that all the events of a moderately complex system could be identified by a student in the first pass of the event-response list. In particular, if a student took a user perspective, it is highly likely that additional changes to the original list would have to be made. Chigden highlighted this point:

> *I just sit back and think of it but not straight away I am not. There would be more tasks but I would put them in as I go. (Chigden)*

Observations reveal that many students initially took a user perspective when identifying events and added additional events, more closely linked to a programmer's perspective, at the coding stage. This is understandable as students were encouraged to simulate the running of a system from their form design ie. the user interface, as a way of identifying events. Sadly, they often failed to link the specific code in methods to events, and to move back and forwards between these two stages of the design process.

6.5.4 Event Status

To simplify teaching event-driven programming, we restrict most of the discussion in lectures and workshops to a few events ie. load, click, lost focus and change. A control object, however, can have multiple events and it is only when code is attached to an event that it is deemed to exist. Moreover, some objects are created with the intention of having an attached event while others are only created to help with display, take inputs or receive outputs. The purpose of an object frequently depends on the existence of an event.

The structure of an event-driven system is determined by the status of events and this appears to cause problems for students. As an example, if a command button is used instead of a series of on-change events, then a text box may change from being an active object in a system to one which only takes input and does not trigger an event. Similarly, if validation is assigned to an object, then this change in event status changes the role of the object from one of accepting any input to only accepting valid inputs. In our example, if validation was included on the lost focus event of each text box then the event response list would be expanded to include:

Event:	On lost focus of Hours worked
Response:	If invalid amount then display message box, clear entry and set focus to Hours worked
Event:	On lost focus of Pay rate
Response:	If invalid amount then display message box, clear entry and set focus to Hours worked

Lecturer M suggested that the event response list be expanded to include compulsory validation to reinforce the need for inputs prior to calculation of an output:

> *I was wondering whether in that event response list we should have error type things, like the event is they fill in cost and press the click button if they haven't filled in the cost then do we get in there to write in they trap the error and say 'you must fill in a cost first'?... May be there should be a set of preconditions before the event response list. We say, the preconditions are that they have filled in this, this and this box and they have entered this data and then a certain event happens. I don't know whether that is getting too prescriptive at this stage. We are trying to make it relatively simplified at this stage.*

Although, events are deemed to exist only if code is present, students are not so clear on this issue. In the early stages of the course, it is sometimes difficult to describe to students the difference between non-existent and existent events. Many students seem to believe that every keyboard entry is an event. Irena, an undergraduate student submitted the following event-response list for her first assignment:

Event/Response List

Event: The program is started.
Response: The current date is displayed.

Event: User enters the number of copies required and the number of pages to be copied.
Response: As the values are entered, they are displayed.

Event: The user chooses and clicks on the paper size, cover type, paper colour, binding and copy side option buttons
Response: The buttons are marked on and others are marked off.

Event: The user clicks on the folding check box.
Response: The folding is selected and binding and covers type frames disappear.

Event: The user selects the binding type option button.
Response: The binding option button is marked on and others are marked off.

Event: The user enters an even number of pages and selects double side option button.
Response: The number of pages to be copied is divided by 2.

Event: The user enters an odd number of pages and selects double side option button.
Response: The number changes to be copied is divided by 2 and on the result is added 0.5.

Event: The user enters 2 as a number of pages and the double side option button.
Response: The number changes to 1 and the folding check box appears.

Event: The user chooses and clicks on the binding option button.
Response: The button is marked on and others are marked off.

Event: The user clicks the Print button if he/she wishes to print the price.
Response: The form as displayed is sent to the printer.

Event: The user clicks the Exit button.
Response: The program is terminated.

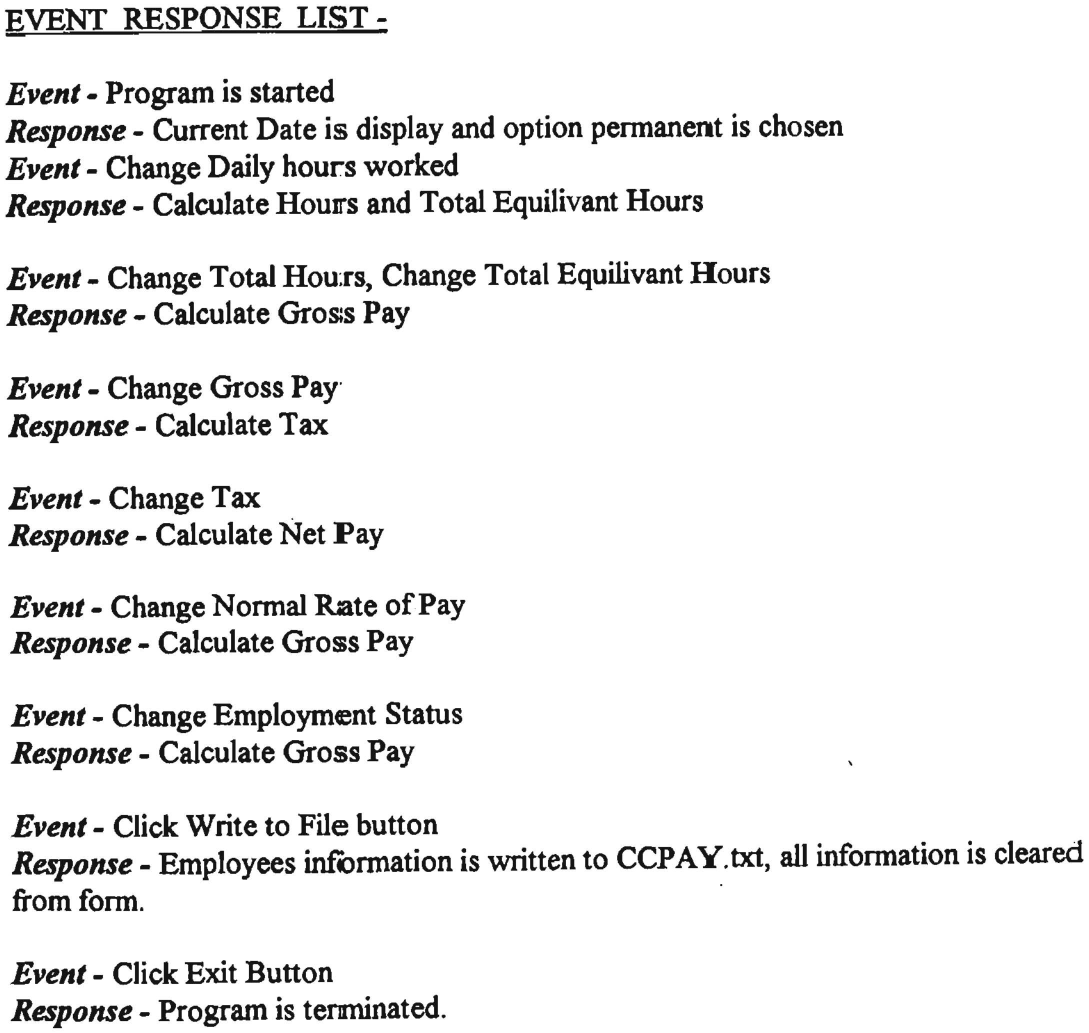
The second event, User enters the number of copies required and the number of pages to be copied, is not an event. However, in her second assignment when she included validation on inputs there was an event associated with the object.

Moreover, some events are part of the program architecture and happen automatically which often leads to further confusion. An option button in a frame or on form will automatically turn other buttons to false if it is chosen - no code is required - yet Irena has identified many of these as events in her system ie. The user selects the binding type option button - The binding option button is marked on and the others are marked off. It is important to note that unless option buttons are in a frame or are exclusive to a form, event code would have to be included behind each option button.

6.5.5 Response Detail

The intention of the response section of the event-response list was to define what was required of the method. In our running example the response to the event Click on the Calculate button could be Calculate and display the Gross Pay. It is possible, however, to place more detail into the response thus it could read; Multiply the HoursWorked by the PayRate for either Part-time or Full-time employees and display the Gross Pay. In the latter case, the response is used as a tool for the initial identification of steps in the code while the former only identifies outputs.

It was surprising to observe that, almost without exception, students described responses in terms of the output with little or no mention of how the output was to be determined. Seana's event response list shows the lack of detail in the description of the response:



<u>EVENT RESPONSE LIST -</u>

Event **- Program is started**
Response **- Current Date is display and option permanent is chosen**
Event **- Change Daily hours worked**
Response **- Calculate Hours and Total Equilivant Hours**

Event **- Change Total Hours, Change Total Equilivant Hours**
Response **- Calculate Gross Pay**

Event **- Change Gross Pay**
Response **- Calculate Tax**

Event **- Change Tax**
Response **- Calculate Net Pay**

Event **- Change Normal Rate of Pay**
Response **- Calculate Gross Pay**

Event **- Change Employment Status**
Response **- Calculate Gross Pay**

Event **- Click Write to File button**
Response **- Employees information is written to CCPAY.txt, all information is cleared from form.**

Event **- Click Exit Button**
Response **- Program is terminated.**

When details were included in a response it was usually due to the need to include additional outputs or to describe system functions. In the example below, in the first event Sylvia lists the outputs that are to be calculated and in the second event describes the outputs that are saved to file and refreshing the screen.

EVENT: The File Button is clicked.
RESPONSE: The Date, Pay Week, Employee Name, Employee Number, Gross Pay, Tax Pay and Net Pay is written to the CCPAY Text File. The cleared screen is then returned for the next Employee details.

If the level of detail in the response section to events reflects the amount of thought students give to the proceeding stages in the design process, then it would appear that they give very little consideration to how an output is going to be determined at this stage. Moreover, the observations of students in class would reinforce this view.

6.5.6 Placement of Code

Students appeared to prefer designing systems that use a command button to calculate the output. The code to do these calculations is placed in only one location. In these types of systems, the students are often using mental models developed around the procedural paradigm assuming the entry of inputs first followed by the click of a button leading to the calculation of an output. So far we have only analysed *individual* events but it is equally important to investigate how students developed and presented their event response list. The examples so far show that most students present their events in the order one would expect a user to follow when using a system, that is, inputs, processes and outputs. This is a logical order to present events, after all a system is made up of inputs processes and outputs, but the context of some of the events (particularly passive events such as putting in inputs) is further evidence that a large number of students appear to look at systems in a procedural manner. As Lecturer O observed it is sometimes difficult for students to take a holistic view of event-driven systems. Other observations of students would support this view as it would appear that most students opt for a top-down systems approach to the development of event-driven systems. Moreover, this may account for why so many students place importance upon the screen layout seeing this as an overview of the system. Lecturer O identifies the problem many students have moving from the screen layout to the event response list:

> *... some way of tying in more directly the event response to the screen layouts of forms, as it is when students are building an event response list they are thinking what has it got to do, oh it has to add up all the numbers right we'll make an event for that, whereas what they should be doing is thinking of it visually, a visual programming systems, they should be looking at their layout which presumably if they are doing it properly has got spots for all the input stuff that they need and they will add to it as they go along. And will work out what has to happen. I tend to sit there with a big pencil drawing of the screen layout and press the buttons with the fingers and say what should happen when I press that button, what should happen when I type in, and to think specifically in terms of text entry, validation, different ways of doing it, whether you do it on change or lost focus and thinking in those terms and then work from there to the event response list so that*

> *anytime that something is suppose to happen in response to something that you do it is put in the event response list. It occurred to me correcting the last set of assignments that putting the event response list on the same piece of paper might be a useful way of approaching it to tie it in more directly and you can see at a glance that this particular control is a passive object, it is there as a display or as passive data entry with no data validation.*

When students are required to move to automatic calculation via lost focus or on change events many are confused on where to place the resultant code. Lecturer M observed that event-response list did not help these students:

> *Well I don't think it worked that well last semester, I think they really didn't know exactly what was meant what we wanted to put in there. I don't think they were sophisticated enough probably to say "I want to be able to click on this button and this happen", I don't think they could think as simply as that. ... They know what an object is. I think we have got to have it there. I think it is one of the main parts. You have got to have that with the screen because it is no use having the screen if you can't tell people what is going to happen when they do it. ... it wasn't clear what an event was. The students did not understand an event. They did not realise there were only a few particular events they should be working on - loading, click. They didn't really know where to put the code. It should have been obvious from that event response list but it wasn't. It should have told them what object and under what event. And we don't worry about whether they filled in anything else unless it was critical . I mean if they filled in name, address and telephone number it doesn't really matter, they're not events which are going to cause anything.*

Although there is an implicit link between an event in the event-response list and the placement of code in the matching object it is not explicitly stated. Lecturer O highlights this point:

> *They did their screen designs early, they often stick with that, later on they would discover while they were writing the code that they would need something extra, they would plonk it in but it didn't go back on the form design. They forgot to go back and fix the documentation when you finished doing the code. There were a few whose detailed algorithm was an edited version of the code (replace code with generic pseudocode). There was obviously not much design there. I think what they tended to do that they do that event response list by thinking what happened here that I suppose that and that, and right, a detailed algorithm is something which that you sit down and a very simple detailed algorithm a sort of top level module design that basically says you need an input module and you need a processing module and an output module and there identified and sit down and write the code and input all the code for all the bits and pieces and there will be no connection between what they actually did and what was in the event response list. I don't think they used the event response list as a device to help them make sure they had done everything. Obviously some of them didn't because they hadn't done some of the things.*

Lecturer T believes it may be beneficial to have an additional column in the event-response list which specifically mentions the matching object.

> *Instead of event 1, this is the event that occurs on cmdExit you actually list the name of the object there. You probably don't need to say it is a command button because that naming convention tells you that and what does it do. ... I am suggesting it probably make more sense if you had the object is cmdExit the event is click and then put the response in words in other words put the action in symbolic form and the response in words (LecturerT).*

6.6 Test Data

This methodology explicitly requires test data to be produced. There is an implicit assumption that this data will be used to test a system. Sadly, the analysis reveals little change in the attitudes of students to the testing event-driven systems from students in previous semesters where a procedural language was used. Despite the importance placed upon the need to develop a system that is functionally correct, students often used ad hoc procedures and a limited range of data to test their systems. No student was observed testing their program from prepared data in classes. One of the experts (Lecturer D) observed a similar situation amongst his students:

> *They do not more or less than they have done. That is, they do the minimal amount possible but generally do what programmers and programming students have always done is that they put the data in that they know it is going to accept and answers come out that they expect and therefore they think everything is OK.*

Two issues surfaced from the analysis; firstly, how and when students developed their test data, and secondly, the strategies they use to test and debug an event-driven system

6.6.1 Development of Test Data

6.6.1.1 When test data is prepared

Students submitted test data with their assignments. Irena's test data is typical of many submissions:

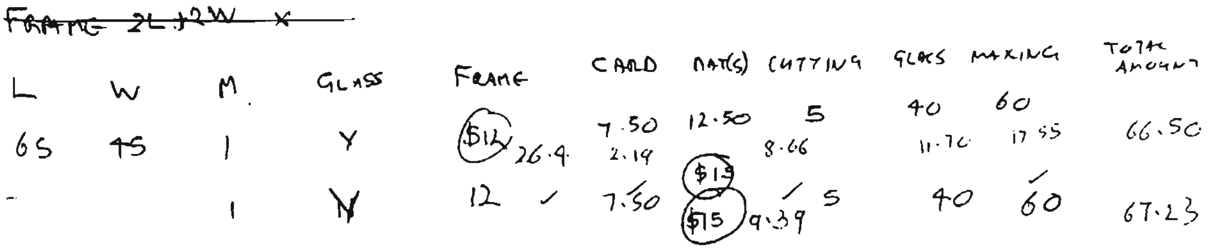
Test Data For A 4 paper size

Copies	Pages	Paper size	Covers type	Binding type	Paper Colour	Copies type	Folding	Expected results ($)
100	13	A 4	No	Tape	Colour	Single	No	317.00
17	3	A 4	No	No	Colour	Single	No	4.59
60	17	A 4	Light	No	Colour	Single	No	103.80
10	61	A 4	Clear	No	White	Single	No	44.20
150	88	A 4	Light	No	White	Single	No	954.00
200	31	A 4	No	Spiral	White	Single	No	1134.00
100	25	A 4	No	Spiral	Colour	Single	No	575.00
90	5	A4	Light	Tape	White	Single	No	229.50
95	10	A4	Clear	Tape	White	Single	No	270.75
45	7	A4	Clear	Spiral	White	Single	No	186.30
62	8	A4	Light	Spiral	White	Single	No	264.12
10	22	A4	Light	Tape	Colour	Single	No	41.80
5	45	A4	Clear	Tape	Colour	Single	No	31.00
60	47	A4	Clear	Spiral	Colour	Single	No	472.80
300	90	A4	Light	Spiral	Colour	Single	No	3540.00
50	30	A4	Light	Tape	White	Double	No	200.00
60	61	A4	Clear	Tape	White	Double	No	352.20
85	70	A4	Clear	Spiral	White	Double	No	667.25
55	52	A4	Light	Spiral	White	Double	No	375.10
65	125	A4	Light	Tape	Colour	Double	No	716.30
89	13	A4	Clear	Tape	Colour	Double	No	278.57
60	112	A4	Clear	Spiral	Colour	Double	No	689.40
25	51	A4	Light	Spiral	Colour	Double	No	183.50
30	350	A 4	Light	Tape	White	Double	No	696.00

The interviews, however, revealed when most students developed test data:

> *Last. It would be as soon as I am ready to hand it in. It is almost last. I don't like doing the paper work behind I just like nutting it out. (Elizabeth)*
>
> *Once you have worked through the algorithm and the program variables I think it is just logical to do the test data. (Billy)*
>
> *Pretty much at the end.(Clinton)*
>
> *I pretty much follow the steps. I do develop some test data using a calculator and write it down. I develop my own calculations and try to run it through the code and see whether it matches. (Jaranka)*
>
> *It is quite easy and I generally leave this to last. (Sylvia)*

Some students developed small amounts of test data in the early stages, often prior to any validation, to test that the main algorithm was correct. Peter developed the following test data prior to coding:



~~FRAME 2L+2W x~~

L	W	M	GLASS	FRAME	CARD	MAT(S)	CUTTING	GLASS	MAKING	TOTAL AMOUNT
					7.50	12.50	5	40	60	
65	45	1	Y	$12 26.4	2.19		8.66	11.70	17.55	66.50
-		1	N	12 ✓	7.50	$13 $15	9.39 ✓ 5	40	✓ 60	67.23

No other test data, however, appears to have been developed until the very end of the development process.

It was clear that most students developed test data at the very end of the design process and viewed it more as a requirement for assignment submission than as an important tool for the testing of their systems. On the other hand, some students prepared parts of their test data as a method of understanding the system requirements:

> *What I try to do because of my maths background I look at how things are calculated and then I try to work out how the example is calculated. So I understand the guts of what is going on in a mathematical sense.(Peter)*

6.6.1.2 How test data is prepared

Students appear to follow a black box method, in the preparation of formal test data for assignment submission and for the use of data for the immediate testing of an event. As an example, in the first assignment students are required to develop a system to calculate the cost of manufacturing an item, say a picture frame, from a range of options. The test data that is produced covers all the possible paths based on the range of options . Where an all embracing calculate button is used this often equates to *white box* testing of an entire event procedure. In this case, students are often testing that a set of specific inputs produce the expected results.

Lecturer D, however, identified a second set of test requirements in event-driven systems, which he calls *interface testing*, that is often overlooked by students:

> *So when you test their program and you click on a button, perhaps it opens a file and then you click on it again and the program crashes, they say "What did you do that for?". And you say, "Well, why did you let me do it twice if it was a silly thing to do?". So I guess what we try to impress upon them is that there are two dimensions to testing - one is the testing of the fundamental operations that the program has to do, like multiply 2 by 2 to get 4; if it gets 6 something is wrong. And the other testing which you really haven't had to do in the past much is the interface testing, and that is it now possible for the user to do some combination of dumb things which causes the program to crash or behave incorrectly or whatever like will you allow them to open a file that is already opened; will you allow them to write to a file that has not been opened and that sort of stuff.*

Moreover, Lecturer O indirectly refers to this problem:

> *Either obviously they didn't test or they ran out of time and thought heck I still have to hand it in and see what I can get for it. I assume the latter although some of them had not obviously tested the disk they were handing up. OK you may get it working at home but bring it in try it on one of the machines here. And some of them obviously didn't because they had references to drive b.*

It is probably only when students are conversant with the concepts of event-driven programming and have included validation in their systems that interface testing is fully addressed. This was clearly evident in the latter assignments. A greater emphasis placed on test cases at an early stage would provide a better structure for the development of test data for both algorithm and interface testing. Lecturer T shares a similar view:

> *Test data is something they should definitely do, I am not sure where that should go and the form of it. I think I would prefer test cases rather than test data, even a diagram. If it said test for the situation where you have got negative input on any of the available places, test for folding multiple copies where they cannot do it, you know more in generalities than trying to list the actual data for each. The trouble with listing the data for each it is the wood for the trees thing, you get lost in the detail. I said to students, they don't seem to take that first step what things in principle do I want to test and then go to the practice of putting in the specific items. I think they jump to the specifics.*

It is clear that most students develop formal test data after the system is complete, use a black box method to develop test data based on the range of options depicted on the screen, and often fail to recognise the need to test user interaction with the interface such as for the validation of inputs.

6.6.2 Testing Strategies

The literature review revealed that object oriented and event-driven programming, which enables code to be distributed amongst events in different objects, can facilitate testing as the system is gradually developed. Lecturer B, one of the experts from another university, shares this view:

> *And that is one of the features that is not there in procedural languages, is that you have to go so much further in developing an application before you can test it where as with this you have that ability to get immediate feedback live while you are developing stuff, you don't have to go to a compiler and go through that process, whereas this happens almost instantaneously. You can get that interactive development strategy and test more often because it is live. Put something in, test, prove it doesn't work and go back and correct it. So you have this one problem at a time to solve rather than 15 errors that come out in a C program when you are compiling to test.*

As an object performs a function on a unique set of data, the programmer has a large amount of flexibility when testing and debugging. In a procedural language it is often cumbersome to isolate functions in a programs. When Pascal was used in the Introduction to Programming subject, students had to enter a substantial amount of the program code before the program could be run. As a consequence, it was found that students often had trouble identifying which section of the code was causing the error particularly in a program without procedures. When event-driven programming was adopted, it was observed that students had a tendency to add a function to a system, test it immediately and then move onto the next aspect of the system. On the other hand, a small number of students would develop a basic system and progressively add functions but they usually tested each function immediately. The interviews with students revealed some insight into their testing strategies:

> *I like to correct the error. As soon as you fix that error you go to another one. I like doing that way. I don't like having the code all in front of me and fixing up.(Elizabeth)*
>
> *I have done a few things as I go along. (Dianne)*
>
> *I like to test after each module if I can.(Billy)*
>
> *I did it in small chunks, I tested parts as I went and sent output to the top left hand corner which I delete later. Visual Basic is so easy to alter quickly. Unless I tested at that stage later on you would be scratching your head trying to figure out what was wrong. (Fred)*

Despite the importance given to the development of test data in this methodology, in reality nearly all students do not return to this test data to undertake a comprehensive test of their finished systems. Although event-driven programming encourages testing on components of a system as functions are progressively added, it appears to discourage the testing of a system against formal test data on its completion. The ability to fragment the code into different events enables students to develop test data for that function which may not be exhaustive particularly when a system is complete. Given the problems of identifying and rectifying errors in a program irrespective of the programming paradigm, it is clear that testing is still viewed by most students as one of the least desirable tasks in programming.

6.7 Program Variable Table

Although the program variable table is described as the sixth step in the design methodology for event-driven systems, students quickly come to realise that it is completed as the algorithm is constructed. Practical examples in workshops and lectures reinforces this fact. The program variable table names and describes each global and form variable. Local variables, whose values cannot be transferred to other procedures or forms are not entered into the table. The program variable table is designed to assist with the development of an algorithm, to assist with declarations at the coding stage, and to be a reference for a programmer.

Lecturer O observed that the program variable table was not used by most students:

> *Some of them worked one out after they had finished and they had a long list of variables. Others basically put down a variable for all the major pieces of data that they knew they were going to use. I don't think it is something they use, and for myself, a program variable table is something I think I should maintain but I wouldn't. But if I did, it is something I would build as I go. And there is not much point in doing it when you are finished. If you do it on the way, the number of times I saw people using a different variable for the same thing in different spots and the number who used last number* [referring to an example in the text book] *without needing it.*

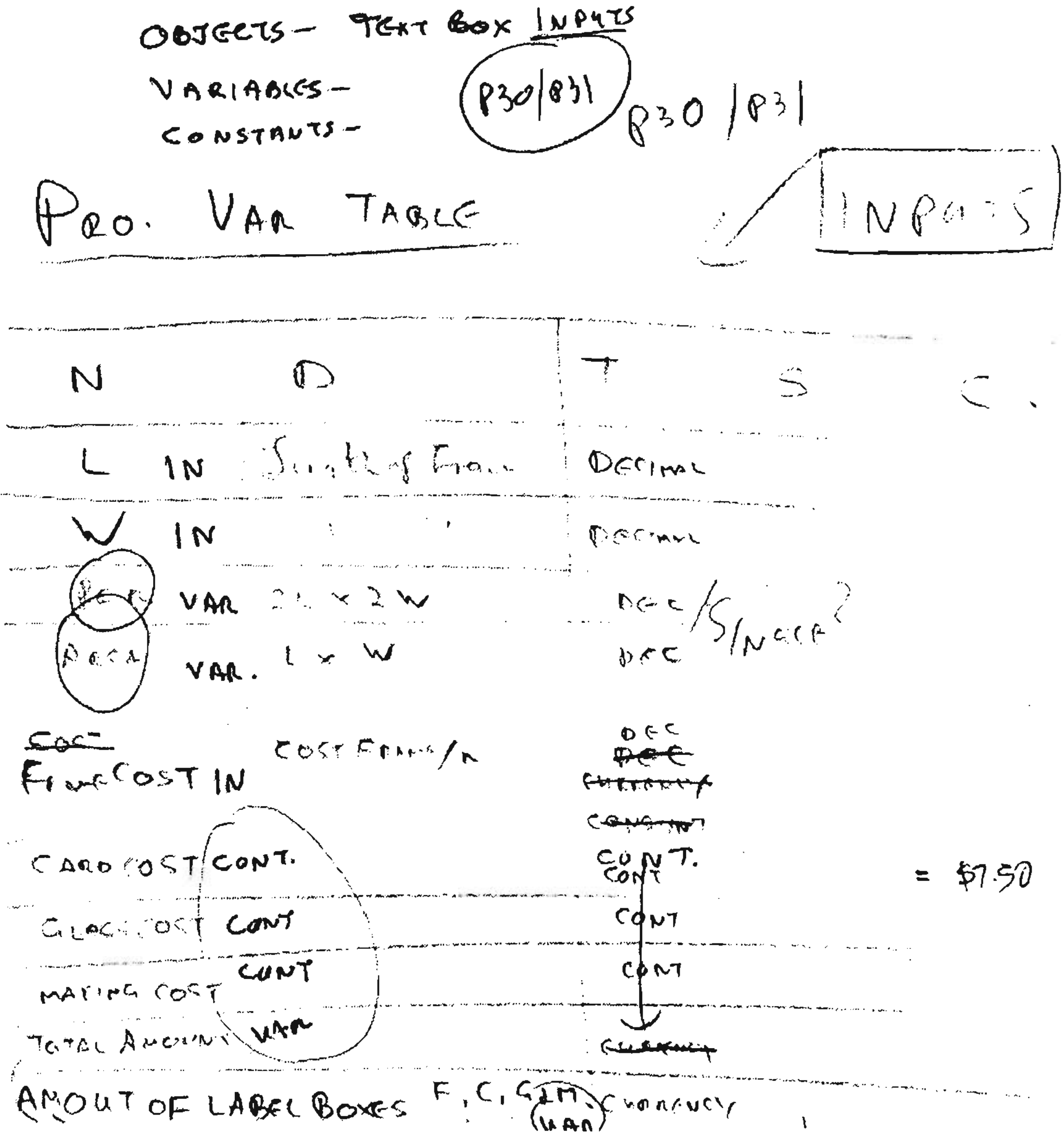
Some students felt that the program variable table was difficult to use.

> *The defining diagram and the screen layout is OK. The one where you have to put the variables in I don't understand. I try to put anything down and I look at the book and say this is what they do and I look at another page and say ' Oh, that is calculate'. (Elizabeth)*
>
> *The program variable table is very difficult. When I think I have it down pat, when I go to use it again it is difficult. ... I usually jump into the algorithm, into the programming because I think I would be better off that way.(Sylvia)*

Only Clinton and Dianne were able to give some hint to why the program variable table seemed to be difficult for students to use:

> *Your variables are not necessarily on the screen and there are a lot of variables that are needed in the calculation so you have got to work out - a knowledge of what the program is going to consist of before you can have a good understanding of all your program variables. (Clinton)*
>
> *I would probably do the code and go back and do the others. Only because it would make more sense because I have got the code then I will know to define the program variable table or I might do part of the program variable table, define a few things and add to it.(Dianne)*

The program variable table appears to be used in different ways by different groups of students. Peter matched the entries in the program variable table with those in the defining diagram. As a rough draft of a program variable table for the first assignment, Peter wrote:



OBJECTS – TEXT BOX INPUTS

VARIABLES –

CONSTANTS –

P30/P31 P30 /P31

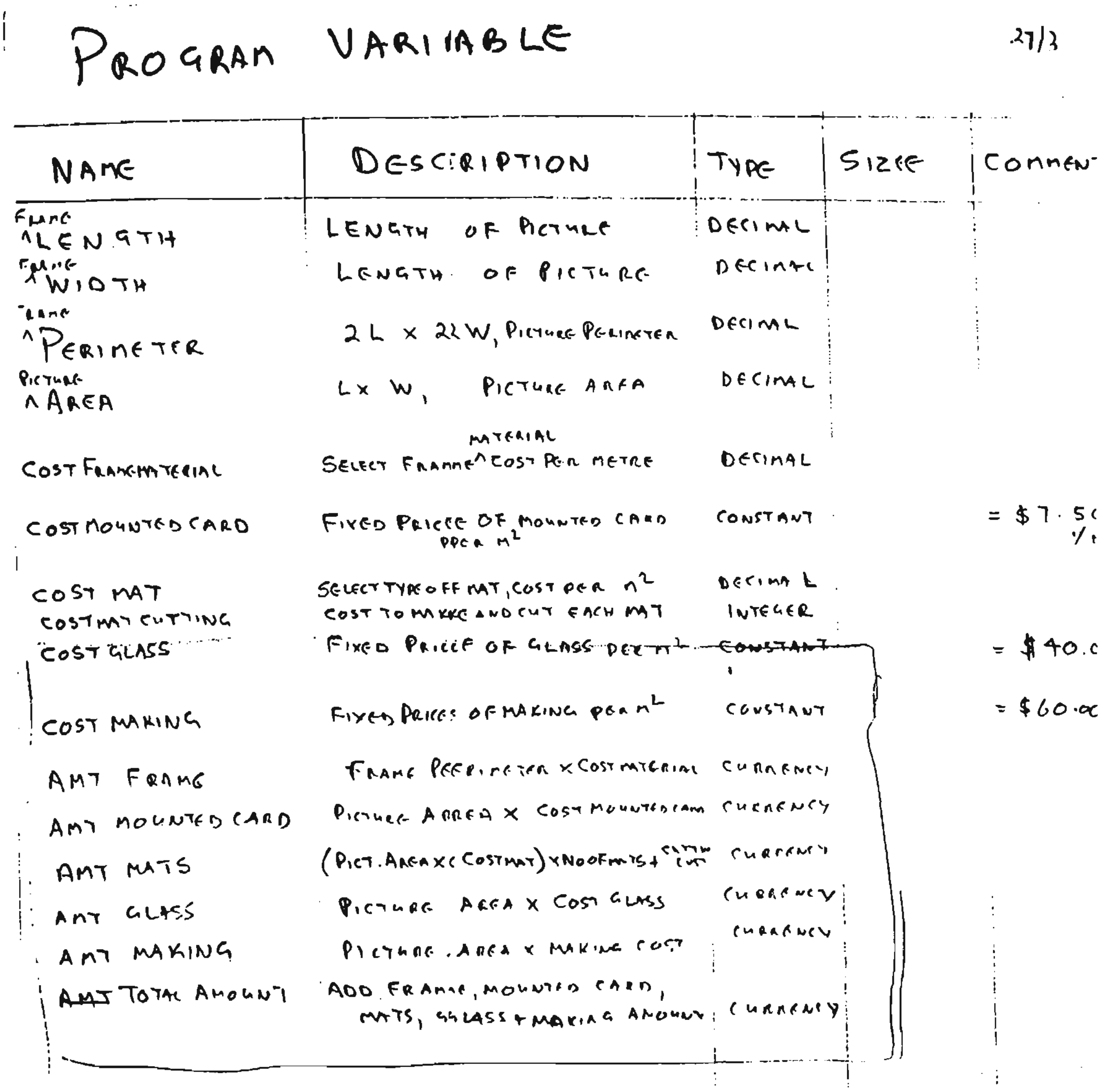
PRO. VAR TABLE

INPUTS

N	D	T	S	C.
L	IN Length of Frame	DECIMAL		
W	IN	DECIMAL		
PER	VAR 2L × 2W	DEC	/SINGLE?	
AREA	VAR. L × W	DEC		
~~COST~~ FRAME COST	IN COST FRAME/m	DEC ~~DEC~~ ~~CURRENCY~~		
CARD COST	CONT.	~~CONSTANT~~ CONT. CONT		= $7.50
GLASS COST	CONT	CONT		
MAKING COST	CONT	CONT		
TOTAL AMOUNT	VAR	~~CURRENCY~~		

AMOUT OF LABEL BOXES F, C, G, M. (VAR) CURRENCY

In his final draft, Peter used the Description section of the program variable table as a method of defining how each variable was calculated. He wrote:

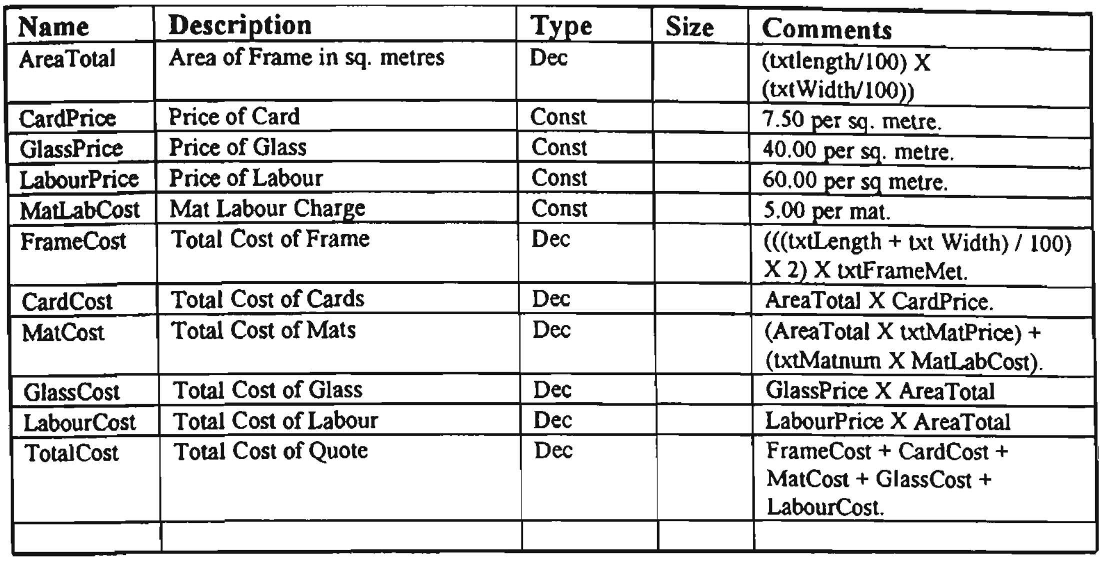


PROGRAM VARIIABLE 27/3

NAME	DESCRIPTION	TYPE	SIZEE	COMMEN
FRAME LENGTH	LENGTH OF PICTURE	DECIMAL		
FRAME WIDTH	LENGTH OF PICTURE	DECIMAL		
FRAME PERIMETER	2L × 22W, PICTURE PERIMETER	DECIMAL		
PICTURE AREA	L × W, PICTURE AREA	DECIMAL		
COST FRAMEMATERIAL	SELECT FRAME MATERIAL COST PER METRE	DECIMAL		
COST MOUNTED CARD	FIXED PRICEE OF MOUNTED CARD PER M²	CONSTANT		= $7.5 /
COST MAT	SELECT TYPE OFF MAT, COST PER M²	DECIMAL		
COST MAT CUTTING	COST TO MAKKE AND CUT EACH MAT	INTEGER		
COST GLASS	FIXED PRICEF OF GLASS ~~PER M²~~	~~CONSTANT~~		= $40.0
COST MAKING	FIXED PRICES OF MAKING PER M²	CONSTANT		= $60.0
AMT FRAME	FRAME PEERIMETER × COST MATERIAL	CURRENCY		
AMT MOUNTED CARD	PICTURE AREA × COST MOUNTED CARD	CURRENCY		
AMT MATS	(PICT. AREA × (COSTMAT) × NOOFMATS + CUTTING CUT	CURRENCY		
AMT GLASS	PICTURE AREA × COST GLASS	CURRENCY		
AMT MAKING	PICTURE . AREA × MAKING COST	CURRENCY		
AMT TOTAL AMOUNT	ADD FRAME, MOUNTED CARD, MATS, GGLASS + MAKING AMOUNT	CURRENCY		

It is interesting to note how some students use a section of the program variable table as a method of describing aspects of the algorithm. Another post-graduate student, Peter P, did not declare any input variables which took values from text boxes and used the Comment section to give a brief but comprehensive description of the calculations for output variables. He wrote:

6. Program Variable Table.



Name	Description	Type	Size	Comments
AreaTotal	Area of Frame in sq. metres	Dec		(txtlength/100) X (txtWidth/100))
CardPrice	Price of Card	Const		7.50 per sq. metre.
GlassPrice	Price of Glass	Const		40.00 per sq. metre.
LabourPrice	Price of Labour	Const		60.00 per sq metre.
MatLabCost	Mat Labour Charge	Const		5.00 per mat.
FrameCost	Total Cost of Frame	Dec		(((txtLength + txt Width) / 100) X 2) X txtFrameMet.
CardCost	Total Cost of Cards	Dec		AreaTotal X CardPrice.
MatCost	Total Cost of Mats	Dec		(AreaTotal X txtMatPrice) + (txtMatnum X MatLabCost).
GlassCost	Total Cost of Glass	Dec		GlassPrice X AreaTotal
LabourCost	Total Cost of Labour	Dec		LabourPrice X AreaTotal
TotalCost	Total Cost of Quote	Dec		FrameCost + CardCost + MatCost + GlassCost + LabourCost.

Object-based languages, such as Visual Basic, enable programmers to use the properties of objects as operands without requiring them to assign them to data variables. The programmer must make a decision whether to create a variable or use a property of an object. Most students appear to create a variable to hold temporary results and to hold an output before it is assigned to the caption property of a label box. The choice of whether to create an input variable or not is confusing for many novice programming students at this early stage and this was recognised by Lecturer M:

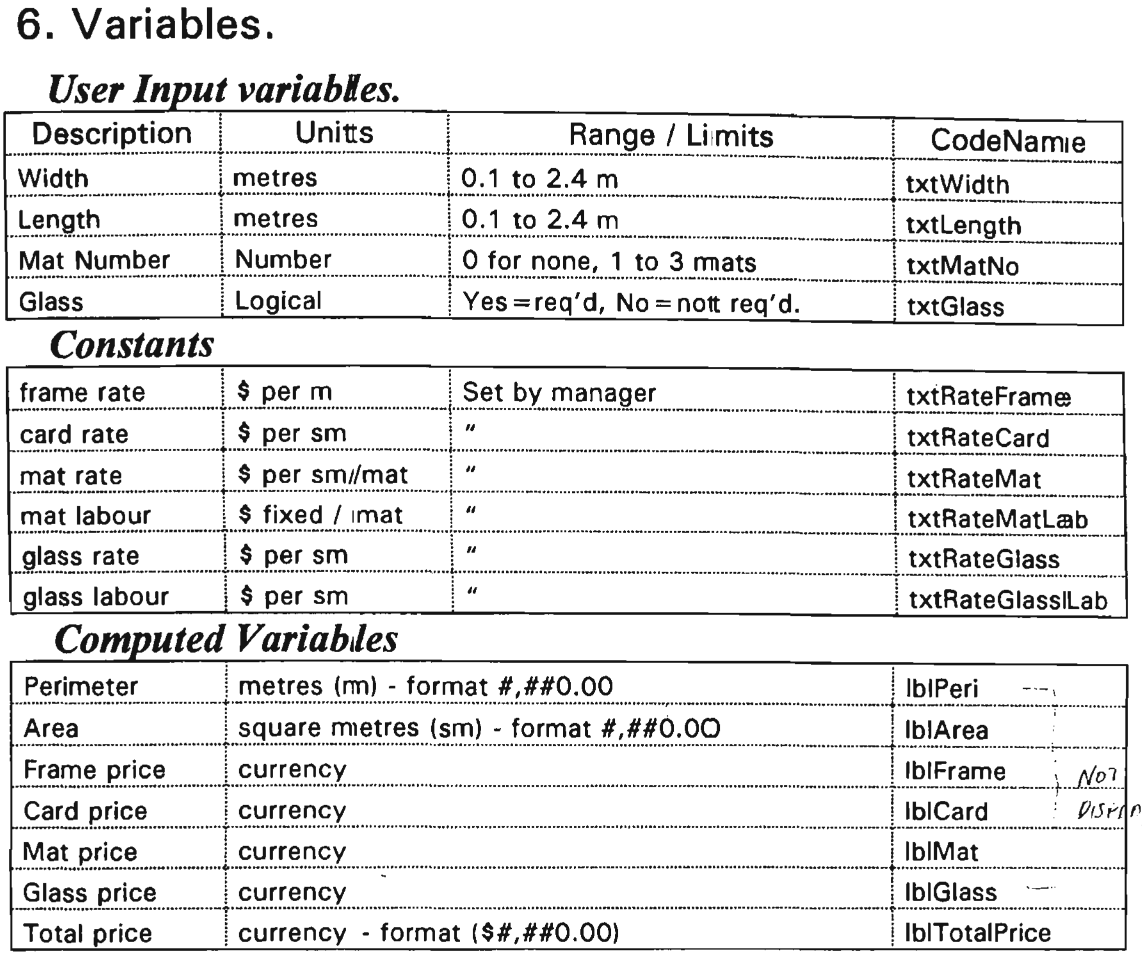
> *I still have got problems with the program variable table. I think although I have done it a bit differently from the book where I try to say now where in the coding, anyway, I have tried to say each value that comes from an input box is allocated to a variable - we haven't done that before. I have got problems because they don't know. I think they should define the variables they are likely to use and then use them in the algorithm. Now some of them don't, some of them still use other words in the algorithm or something or other (Lecturer M).*

Lecturer M suggested that the comment column of a program variable table could be used to show the assignment from an object property to a data variable (the ← symbol) and from an object property to a variable (the → symbol). The program variable table from an earlier exercise using this suggestion would be:

Name	Description	Type	Size	Comments
Hours	Number of whole hours worked	Integer		← txtHours less than 65
Minutes	Number of minutes worked in last part of hour	Integer		← txtMinutes less than 60
TimeWorked	Convert time to real number format. Decimal hours calculated from hours and minutes worked	Decimal		→ lblTimeWorked
PayRate	Hourly rate of pay	Decimal		← txtPayRate
GrossPay	Gross pay for an employee	Decimal		→ lblGrossPay

> Lecturer M has a number of other criticisms of the table which he believes may be confusing for students:
>
> *I don't think that it needs all that other stuff with it. I don't know whether it needs a description column for a start, it probably does. It just seems to be a big cumbersome table with columns that they never seem to use. And the other thing I have problems with is putting constants in a variable table. I think they have to declare their constants with their values in there so maybe have, I don't know what to call it, you very rarely put comments on anything other than constants.*
>
> *The other thing at this stage is that we are talking about the program variable table and do you include the local variable or do you only put in the global ones. Each module should really have a program variable table but that is going a little overboard. (Lecturer M)*

One postgraduate student, George divided the program variable table into sections according to function; constants, input variables and output (or computed) variables. Moreover, he developed a separate section to define the formulae for computed variables:



6. Variables.

User Input variables.

Description	Units	Range / Limits	CodeName
Width	metres	0.1 to 2.4 m	txtWidth
Length	metres	0.1 to 2.4 m	txtLength
Mat Number	Number	0 for none, 1 to 3 mats	txtMatNo
Glass	Logical	Yes=req'd, No=not req'd.	txtGlass

Constants

frame rate	$ per m	Set by manager	txtRateFrame
card rate	$ per sm	"	txtRateCard
mat rate	$ per sm/mat	"	txtRateMat
mat labour	$ fixed / mat	"	txtRateMatLab
glass rate	$ per sm	"	txtRateGlass
glass labour	$ per sm	"	txtRateGlassLab

Computed Variables

Perimeter	metres (m) - format #,##0.00	lblPeri
Area	square metres (sm) - format #,##0.00	lblArea
Frame price	currency	lblFrame NOT
Card price	currency	lblCard DISPLD
Mat price	currency	lblMat
Glass price	currency	lblGlass
Total price	currency - format ($#,##0.00)	lblTotalPrice

In larger systems, the program variable table, if completed as the algorithm is developed, could assist a programmer with the tracking of data variables. In smaller event driven systems, particularly those where only local variables are required in methods, students appear to find little benefit from completing the table. Moreover, event-driven languages have three types of variables, local, form and global, and this adds to the confusion on what or what not to include in the table. The modifications suggested by Lecturer M, may help to clarify in the minds of students the difference between an object property and a data variable. Sections based around the function of the variable, such as input or output may further enhance the table. It is clear, however, that the table becomes cumbersome when multiple forms are employed.

6.8 Algorithm

6.8.1 Pseudocode

Traditionally, pseudocode has been used as a tool in procedural languages to outline the logic of a detailed algorithm. In procedural languages where control rests with the computer at all times there is a clear logical flow from the start to the finish of a program. This is not the case in event-driven programming where the control of the flow is determined by the actions of the user. Compared to a procedural language, event-driven systems divide a main-line controlling algorithm into segments and place these against events in objects. The use of pseudocode, therefore, as a tool in the development of event-driven systems is seen by some experts as unnecessary:

> *I doubt with something like Visual Basic the relative importance of pseudocode too. I think pseudocode is really on about designing algorithms more than a program - I am making the distinction that a program might consist of algorithms but it doesn't necessarily have to have a lot of heavy algorithms, it might or it might not. Pseudocode is really good to work out the logic of an algorithm but I am not sure that pseudocode that shows a whole lot of print statements is actually all that necessary any more (Lecturer T).*

Nevertheless, Lecturer T advocates some step in the design process that can help a student to develop some detail to the logic in each method:

> *I think you can make the distinction between pseudocode and structured English, a number of people regard them as the same. Structured English really was English; pseudocode was more like the first version of a programming language. I mean, you might say in structured English format this now, or get the following information from the user where as in pseudocode you would say get this get that. I think the Structured English approach might be a better way of handling that side of it given that really most of the examples we are looking at are not really that algorithmic. I mean a sort is an algorithm but adding up two figures who cares.*

Moreover, other experts appear to advocate a similar stage in the design process:

> *Yes, I have no worries. I think you have got to get them to write something down, even if it is only a list of what they are going to do (Lecturer M).*

> *I draw a structure chart. I draw a function hierarchy chart, I teach in pseudocode. Because you still have to produce the functionality so you need to describe the functionality, the right way of describing functionality is Structured English and structure charts. Now if that functionality is obtained by moving from one spot to another with an event it still doesn't change the functionality. It would not effect the function hierarchy it would not effect the structure chart the Structured English. If I want the price of cake I need to obtain the information needed for the cake and I need to produce an answer. That functionality description is the same.(Lecturer R).*

> *I have never been one for getting people to do things for the sake of just doing them, it is not going to help you in the design process then lets not do it. ... I asked them all to do pseudocode so that you could think of your problem and go and put it in the programming language but don't both handing in the pseudocode to me, chuck it in the dustbin because no one keeps that (Lecturer D).*

It is clearly evident that the majority of students used some form of generic pseudocode prior to coding. Moreover, most appear to see this stage in the design process as beneficial:

> *Yes, I write it down on paper. A system of both pseudocode and Visual Basic.*
> *I could not sit in front of the computer and just code.(Dianne).*

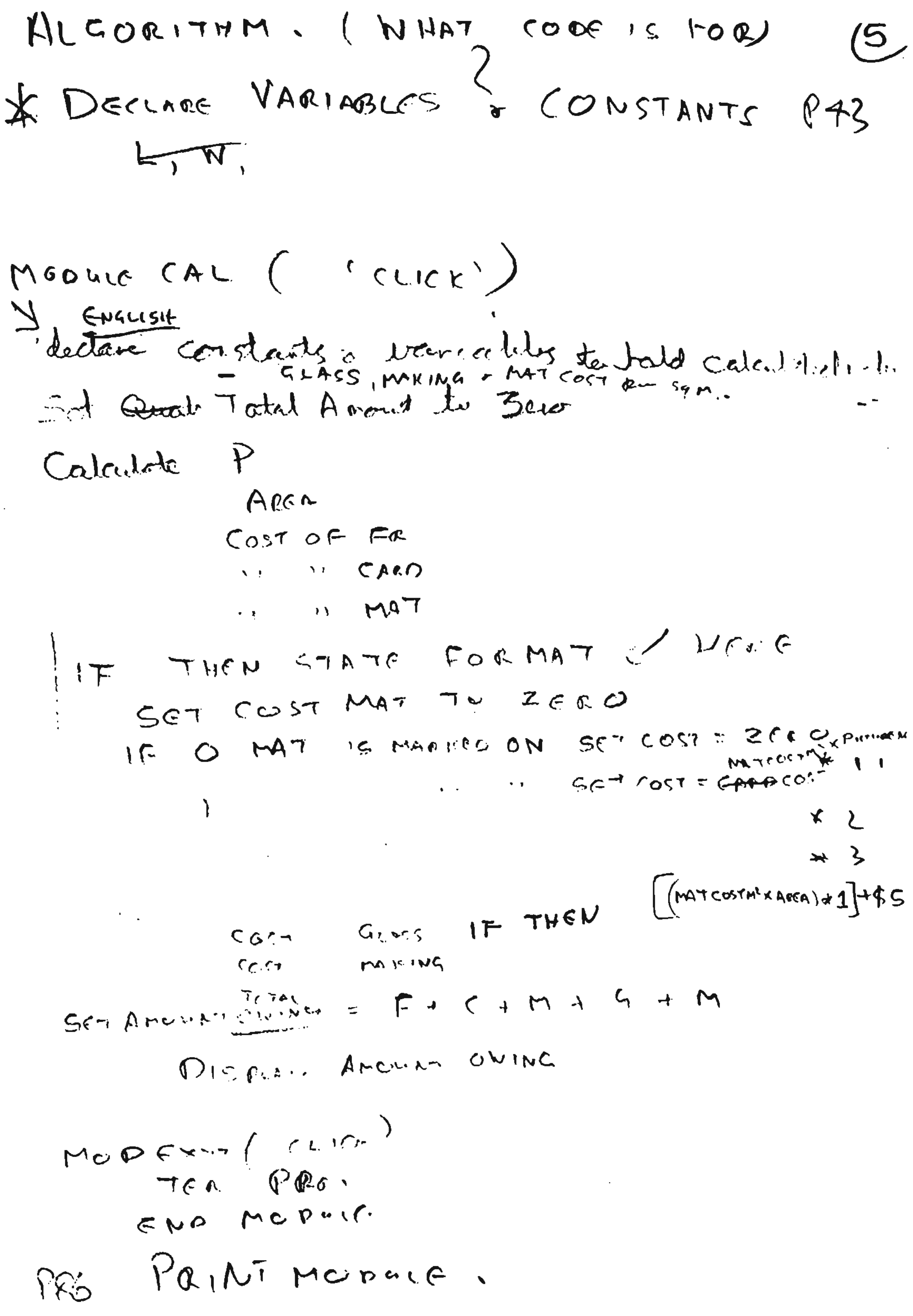
> *It depends on what sort of mood I am in. Usually I like to do it into as much detail as I can, it makes it easier when you go to the program code. Definitely. It is plain*

English and you get your ideas down on how you think it should be. It makes things a lot easier.(Billy).

Yes it is beneficial. To some degree working out whether it is a complete process of creating your algorithm or whether you jot down some of the difficult or more complex features of the program down onto a piece of paper and then putting them into the program first. It is fairly difficult to work out everything that you put into a program. So for example if you have taxation with all the tax brackets then write down on a piece of paper how you would manually go about calculating the tax.(Clinton).

Yes it is beneficial, but I have trouble with it. If I code up strange to the code I lose track completely but with my algorithm there I know which track to follow (Chigden).

In the early stages of the subject Peter used a very crude form of pseudocode to help develop the method for the cmdCalcuate command button in the first assignment:



ALGORITHM. (WHAT CODE IS FOR) 5

* DECLARE VARIABLES & CONSTANTS P43

L, W,

MODULE CAL ('CLICK')

ENGLISH

'declare constants & variables to hold calculation [illegible]

— GLASS, MAKING & MAT COST per sq m..

Set ~~Quat~~ Total Amount to Zero

Calculate P

AREA

COST OF FR

" " CARD

" " MAT

IF THEN STATE FORMAT ✓ NONE

SET COST MAT TO ZERO

IF O MAT IS MARKED ON SET COST = ZERO x PRIMARY

MATCOSTM * 1 1

" " SET COST = ~~CARD~~ COST

* 2

* 3

[(MATCOSTM² x AREA) * 1] + $5

COST GLASS IF THEN

COST MAKING

SET AMOUNT TOTAL OWING = F + C + M + G + M

DISPLAY AMOUNT OWING

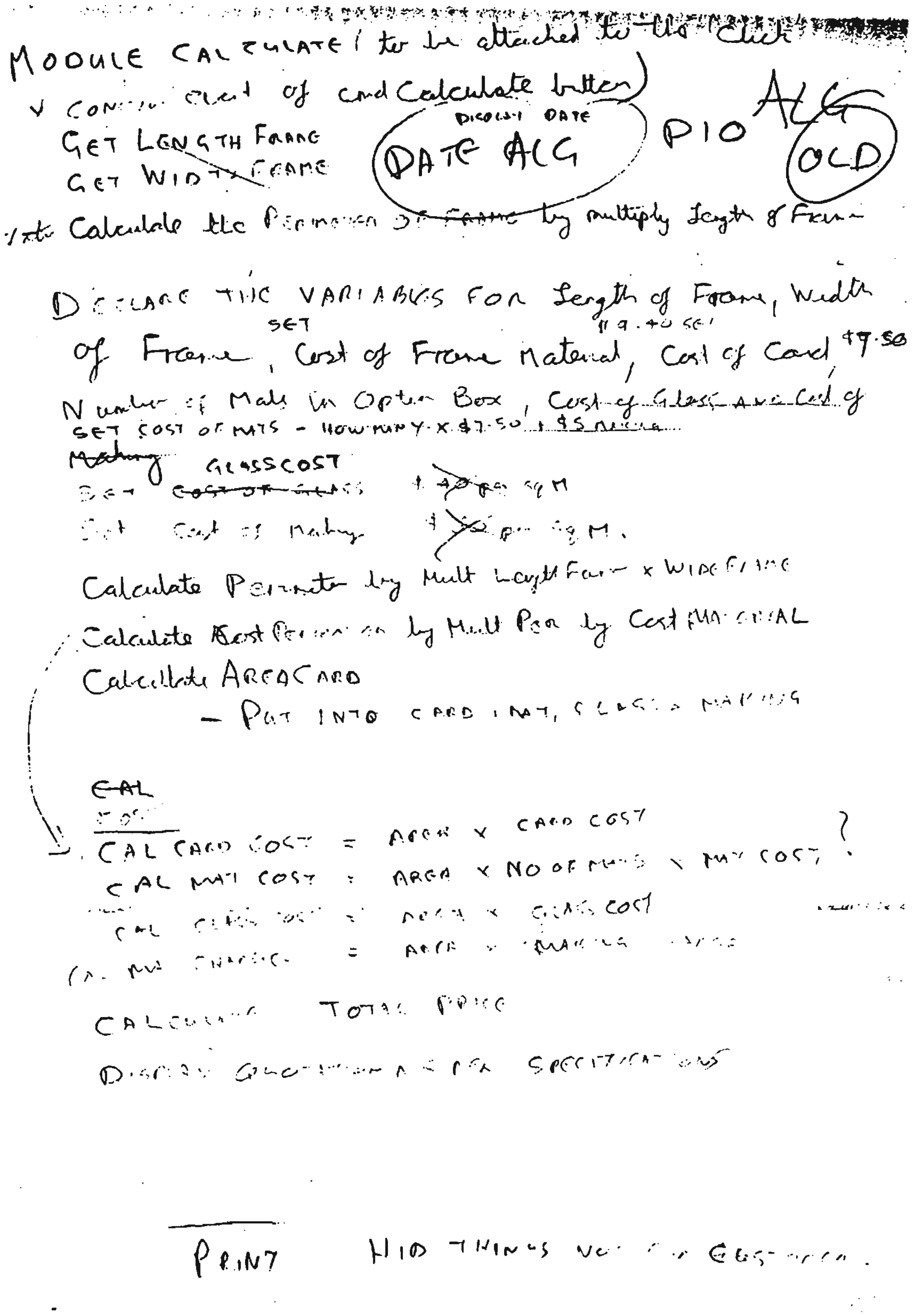
MODEXIT (CLICK)

TERM PROG

END MODULE

PRG PRINT MODULE.

This was later expanded to include more detail:



MODULE CALCULATE (to be attached to the Click
✓ CON[illegible] event of card Calculate button)
GET LENGTH FRAME
GET WIDTH FRAME
DISPLAY DATE
DATE ALG
P10 ALG
OLD
1st Calculate the Perimeter ~~of Frame~~ by multiply Length of Frame

DECLARE THE VARIABLES FOR Length of Frame, Width
SET
of Frame, Cost of Frame Material, Cost of Card, $9.40 SET $7.50
Number of Mats in Option Box, Cost of Glass and Cost of
SET COST OF MATS - HOW MANY X $7.50 + $5 [illegible]
~~Making~~ GLASSCOST
SET ~~COST OF GLASS~~ $ ~~40~~ per sq M
Set Cost of Making $ ~~50~~ per sq M.

Calculate Perimeter by Mult Length Frame x WIDE FRAME
Calculate ~~C~~ost Perimeter by Mult Per by Cost MATERIAL
Calculate AREACARD
— PUT INTO CARD MAT, GLASS + MAKING

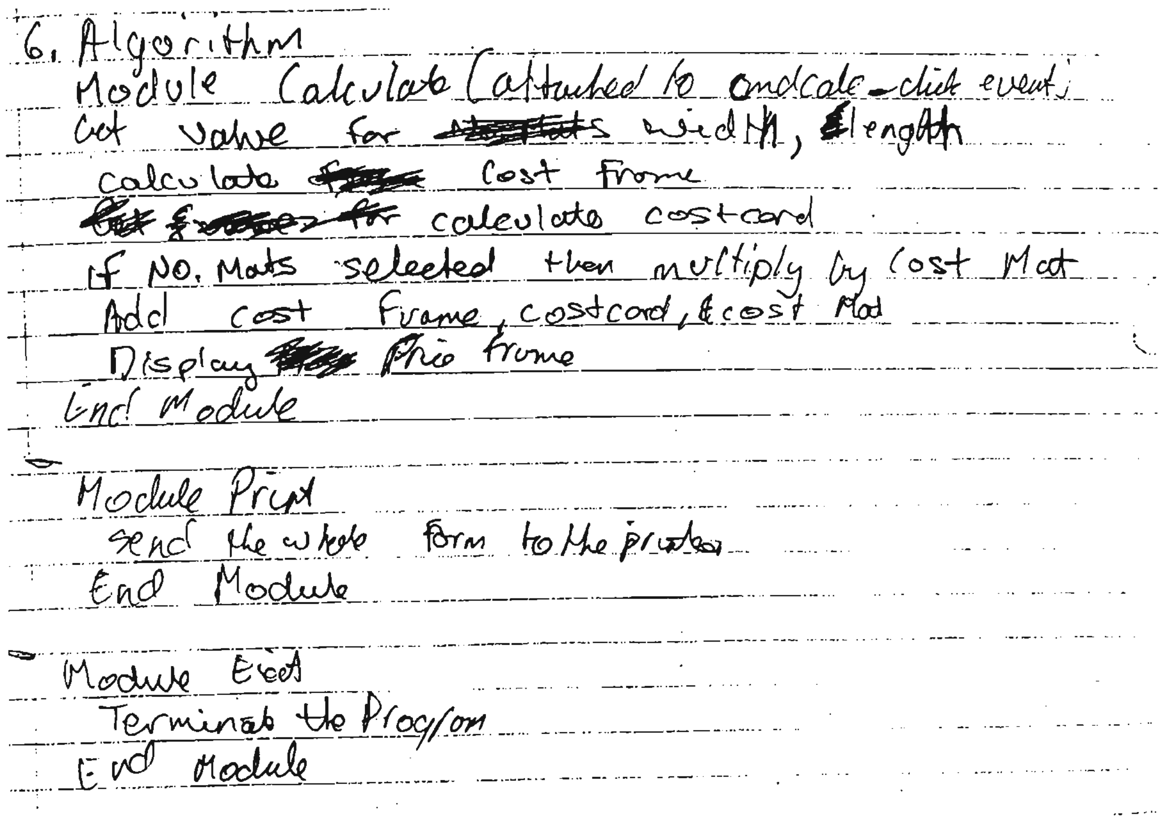
~~CAL~~
[illegible]
CAL CARD COST = AREA X CARD COST ?
CAL MAT COST = AREA X NO OF MATS X MAT COST
CAL GLASS COST = AREA X GLASS COST
CAL MA CHARGE = AREA X MAKING [illegible]

CALCULATE TOTAL PRICE

DISPLAY QUOTATION AFTER SPECIFICATIONS

PRINT HID THINGS [illegible] ESTIMATION.

In contrast, another postgraduate student Gary used a form of Structured English in the early stages to establish the logic of the main algorithm of the first assignment:

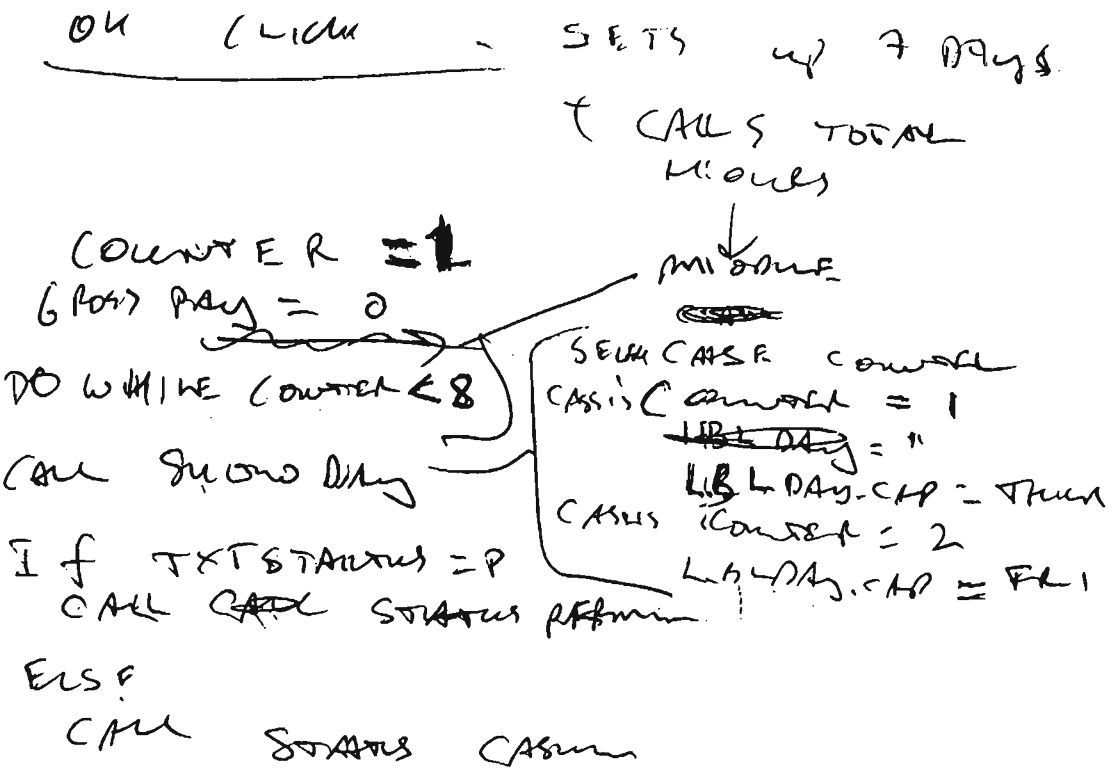


```
6. Algorithm
 Module Calculate (attached to ondcalc_click event;
 Get value for ~~[illegible]~~ width, length
  calculate ~~[illegible]~~ cost frame
  ~~Get [illegible] for~~ calculate costcard
  If No. Mats selected then multiply by cost Mat
  Add cost frame, costcard, & cost Mat
  Display ~~[illegible]~~ Price frame
 End Module

 Module Print
  send the whole form to the printer
 End Module

 Module Exit
  Terminate the Program
 End Module
```

Other comments reveal that some students appear to use pseudocode selectively and tend to concentrate on the complex calculation component of an algorithm. This is understandable given that event-driven programming languages such as Visual Basic enables code to be fragmented, often along functional grounds, into general and event procedures particularly in large systems. It is more likely, however, that the early assignments which often require students to place a large piece of code against an event in one object, necessitate a greater degree of attention than other methods.

> *I would only write down the calculation part - probably in words. I always comment in the Visual Basic code so I know what I am doing but I don't do any pseudocode other than the calculation. Only for calculation before writing the code was beneficial.(Tom).*
>
> *I use the pseudocode in the hard core calculation. I am an accountant, I define the module and then start writing the logic or the mathematical calculation. Once I am happy and I know it is working I then move on ... coming from an accounting background I write out the calculation required to get that. I say to myself, "Well how would I do this with a calculator?", so that is the logical formula, so how do I put that into the code. Do I use a loop, a case, if- then.(Fred).*

Fred's pseudocode for the first assignment is a good example of how a student experiments with the logic of method code at this stage of the design process:

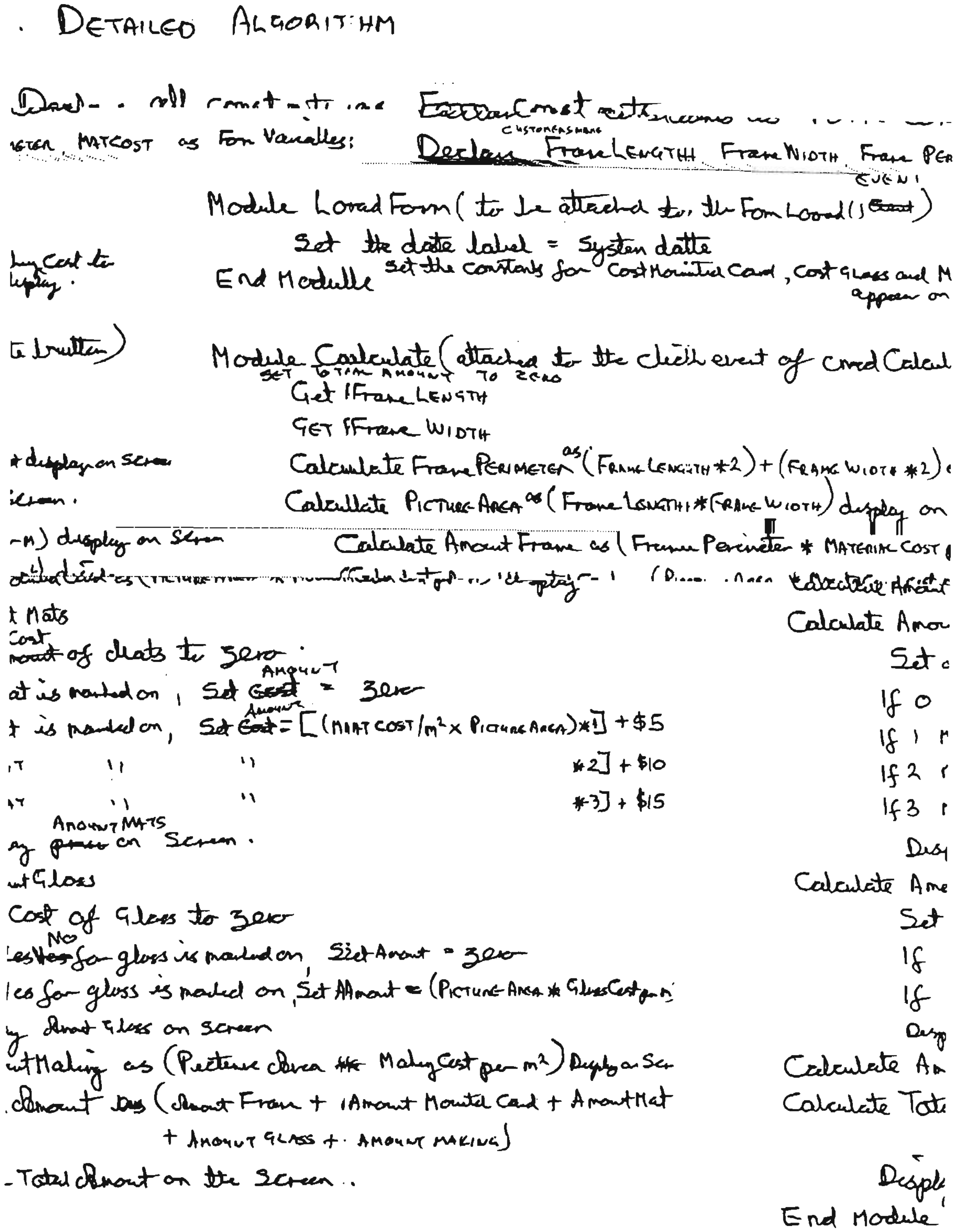


ON CLICK - SETS UP 7 DAYS
& CALLS TOTAL HOURS
MODULE

COUNTER = 1
GROSS PAY = 0
DO WHILE COUNTER < 8
CALL SECOND DAY
IF TXTSTATUS = P
 CALL ~~CASU~~ STATUS PERMAN
ELSE
 CALL STATUS CASUAL

SELECT CASE COUNTER
CASE IS COUNTER = 1
~~LBL DAY~~ = "
LBL DAY.CAP = THUR
CASE IS COUNTER = 2
LBLDAY.CAP = FRI

This concentration on the calculation algorithm is clearly evident in the following pseudocode:



```
. DETAILED ALGORITHM

[illegible] all const[illegible]   ~~External~~Const [illegible]
                                          CUSTOMERSNAME
METER, MATCOST as Form Variables;   Declare FrameLENGTH FrameWIDTH, Frame PER
                                                                        EVENT
                  Module LoadForm ( to be attached to the Form Load() ~~Event~~ )
                        Set the date label = System date
ingCost to        End Module set the constants for CostMountedCard , Cost Glass and M
isplay .                                                                    appear on

te button)        Module Calculate ( attached to the click event of cmdCalcul
                        SET TOTAL AMOUNT TO ZERO
                        Get FrameLENGTH
                        GET FrameWIDTH
+ display on screen     Calculate FramePERIMETER as ( FRAMELENGTH *2 ) + ( FRAMEWIDTH *2 )
screen .                Calculate PICTURE-AREA as ( FrameLENGTH * FRAMEWIDTH ) display on
-m) display on Screen        Calculate Amount Frame as ( Frame Perimeter * MATERIAL COST
[illegible]                                                          ~~Calculate Amount~~
t Mats                                                                Calculate Amou
Cost
~~mount~~ of Mats to zero :                                                    Set
                AMOUNT
at is marked on , Set ~~Cost~~ = zero                                          If 0
                 Amount
t is marked on , Set ~~Cost~~ = [ ( MATCOST/m^2 x PictureArea ) *1 ] + $5      If 1
"         "          "                          *2 ] + $10                     If 2
"         "          "                          *3 ] + $15                     If 3
  AMOUNT MATS
ay ~~price~~ on Screen .                                                       Disp
ntGlass                                                                        Calculate Amo
Cost of Glass to zero                                                          Set
    No
es ~~Yes~~ for glass is marked on, Set Amount = zero                           If
es for glass is marked on, Set Amount = ( PICTURE-AREA * GlassCostperm           If
y Amount Glass on screen                                                       Disp
ntMaking as ( Picture Area ~~*~~ MakingCost per m^2 ) Display on Scr          Calculate Am
Amount ~~as~~ ( Amount Frame + Amount Mounted Card + AmountMat                 Calculate Tota
            + AMOUNT GLASS + AMOUNT MAKING )
-Total Amount on the Screen ..                                                 Displ
                                                                        End Module
```

Although pseudocode or a crude form of Structured English seems to be favoured by students during aspects of algorithm creation, it would appear that in later assignments at least, many students move directly into coding in Visual Basic or believe developing pseudocode will eventually become unnecessary.

> *You draw the screen nice and pretty and then you try do the calculate or click button and then try to do the program, the writing behind it. I probably jot a few notes down like this equals this, where I put everything down, but mainly I try to write the code out without doing the algorithm or pseudocode ... I write down pseudocode when I am stuck.(Elizabeth).*

> *If I knew more about it I would go into code but at the moment I find it beneficial. You are only writing in English. I dare say at the end of the year people will go straight into the code. With experience you don't have to write things down.(Peter).*
>
> *With the assignment I actually did the code first and went home and did the algorithm after the code. I got the printout. The code was not working - I did all I could. I had some things down before hand - the calc tax. But the other one came to me. I do prefer to write it out first and change it and scribble and rub it out and then put it on and keep changing it.... It was pretty big, but I was running out of time and I thought I better get a printout first. I some how found it easier. I had the printout of the Visual Basic code at home. The algorithm is pretty similar. So I found it pretty easy to go back and change a few things (Jaranka).*
>
> *I don't do any pseudocode before coding (Sylvia).*

Lecturer T takes a similar view and, again, advocates Structured English rather than pseudocode as part of the answer:

> *In the extreme example, do you need pseudocode for the Exit button? Definitely not. I suspect that in many cases you would be better off with a one or two line description of what the thing does in a Structured English sort of format more than trying the formal pseudocode because I just don't think they actually do the formal pseudocode before they write the code. I think they probably do that afterwards. Formal pseudocode for an event - I think they do that afterwards, whereas if they had to write down in a couple of sentences what this thing did in a Structured English approach, I suspect they might just do that beforehand.*

In the early weeks of the semester many students requested assistance or had queries concerning their pseudocode solutions. As the semester progressed, it was observed that fewer and fewer students worked from pseudocode. It would appear that there is little difference between the behaviour of students using event-driven languages or procedural languages, that is, there is a temptation to launch directly into coding in the programming language and to by-pass the pseudocode stage. There is a danger, however, that event-driven programming encourages this more so than procedural languages.

6.9 Visual Basic Code

Visual Basic code is based on the original Microsoft BASIC, however there are no line numbers and there are far more instructions. Visual Basic code is based on the procedural paradigm and is placed in event and general procedures. Although programmers are encouraged to declare variables, there is no requirement to do so but instead default to Variant, an all-embracing data type.

A major disadvantage of using Visual Basic as opposed to using a language such as Pascal, was the lack of support materials. There are a huge number of texts for popular procedural languages such as Pascal and COBOL which students can purchase or borrow from libraries. Moreover, in the past students have been able to seek assistance from other students or friends who have a knowledge of procedural languages. In addition, many lecturing staff are not familiar with Visual Basic. In contrast, student texts for Visual Basic were just starting to enter the market while the alternative, technical manuals and how-to-books, were often very expensive and/or difficult to use. Moreover, the text produced by the author concentrated on issues of program design and although it contained a large number of examples, it did not concentrate on Visual Basic syntax. Peter and Dianne summed up the frustration of many students:

> *Not knowing where I was going because I could not follow what I was to do. When I hit a brick wall, I was on the phone. When you are at college you are all together, feedback straight away. But when you come here you work and go home, I felt I was in the room by myself.(Peter)*
>
> *A couple of hours per night late at night in which case the brain is in reverse.(Dianne)*

Irrespective of the language platform used in the subject in the past, it has been found that many students have had problems converting their algorithms into code. Given the shortage of alternative texts and the lack of knowledge from other students and lecturing staff about Visual Basic, it had been expected that many students would experience additional problems developing Visual Basic code. It was surprising to observe, therefore, that this did not turn out the be the case. Lecturer O believes this to be the case:

> *... not a great amount of difficulty. The syntax is very wordy and as such I don't think it caused a lot of problems. I don't think a lot of them thought of it in an OO structured sense where values were built up, the form name the control name of equal value. I don't think they thought of it in the object hierarchical sense but they nonetheless saw those things as good descriptors and most of the code is almost self explanatory. No, I don't think the syntax is a problem*
>
> Question: No more than for another language?
>
> *Probably less so than another language because in some ways there is less bits of it. I mean you are getting around the loop stuff. You do have loops in this but not too many. The only loops we had were rolling through a list. You don't need loops for control.*

Certainly students expressed concern that they could not get access to additional text books on Visual Basic:

> *My Bible. At my level, an idiots guide I don't want, but I want something that will strip some of the jargon so I can understand it and then we can bring the jargon back. ... I never have problem with theory. But it is when it comes to writing the*

> *code, I would like a book to help me write the syntax. I have to look through your examples the book to find things. For example the word lost focus means nothing to me. I tried to look to see what it means and I haven't found it. I think the biggest problem with this book is in finding out where to use it. OK we look at the exercises. Then again I cannot even voice these things because I don't have the knowledge; many people know this anyway.(Gerald)*
>
> *I would really like a dictionary. I am finding it really difficult. It is a drawback for people not to have a resource.(Mary-Beth)*
>
> *I don't find it that easy to follow. When I went back to do the assignment I found it difficult. There is nothing there on how to write the code. Should be something there on how to get there not just on the solution.(Dianne)*

Although a specific Visual Basic text may have provided some assistance to many students who had problems writing the code, it is evident that many students had fundamental problems developing and debugging code and expected a text to solve their problems. Chigden highlighted the difference between a subject like Accounting where there are many fixed procedures compared to Computer Programming:

> *Basically, if I have got a problem I cannot turn to a theory book. OK, this is the problem, go through and fix it up. With Accounting if I am stuck on a particular part then I can go to a book and I can go back and fix it. With programming there is no set rules, the coding itself, if the coding is wrong I don't know where it is wrong. ... I can do up to steps 4 or 5. I understand it but once it comes to the algorithm I don't know whether I am right - if I make a mistake I am not aware of it. It doesn't tell me why I am wrong it just says it is wrong. That is really confusing for me.(Chigden)*

Many others expressed concern that they have trouble with coding:

> *I struggle with the code. A structure for writing code would be good. (Peter)*
>
> *No. I don't know where I start. (Dianne)*
>
> *When you explain things they seem clear but when I apply it I have great problems. I have trouble with the Visual Basic code. (Sylvia)*
>
> *I know what has to be done but the code is difficult. Not easy.(Seeana)*

In small programs, the physical order and the logical order of the code are usually the same. As the size and complexity of a program increases, the logical order departs from the physical order and this is usually the stage when many students have problems writing algorithms and programming code. There is some evidence to suggest that event-driven systems which enables code to be divided into segments matching events, reduces the need for a large amount of program code as is the case in a procedural language, and makes coding easier for students. Observations of students and the nature of their queries would support this view. Like Lecturer

M, observations found a substantial reduction in the number of queries concerning program logic compared to previous semesters when Pascal was used:

> *The problems were generally how do you do it type problems. They were not programming how to just "How would I go about making this do this?", and I would show them there is a particular way you can do it or whatever. I don't think they came with any logic type problems particularly. It would be mostly Visual Basic type problems.*

The interviews with students who had some experience with procedural languages appears to support this view:

> *I know what is expected of the code. If you do the name or the design or how you like you know what you are going to call it, where as if I did this second it is not as easy for me. That is why I like Visual Basic much better because you can see what you are doing where as COBOL and Pascal you hit the code and you only see your end product when the code is correct. This is why this is much more appealing to everyone it is much easier to learn. It is picture wise. ... With COBOL you write the code first. You do this equal this. And when it finally runs and you get all the errors out that is when you can see your screen layout and it is not as good looking as this or not as appealing to anyone learning it. COBOL is really bland. I am more a person. I see things and I can talk about but when it comes to writing I am not as good. I have a better understand at looking. Visual Basic is good to learn straight away. When I first saw it I thought 'This is not programming' I thought grouse. This might be a bit easier that is why I may have understood it a bit more. I have remembered "Oh this is how you do it' It has made me think "Oh that is right in Pascal' Now I understand, it is a bit late, that after I have done it this way I have understood it a bit more.(Elizabeth)*

> *I think it is a lot easier due to the event-response than Pascal.(Billy)*

6.10 Problem Solving Strategies

The literature review and interviews with experts found that most introductory programming subjects using a *procedural* language directly or indirectly advocate a structured, top-down problem solving approach which is usually highly structured and sequential. Moreover, the literature review revealed that this approach may not necessarily match the problem solving strategies used by students in other disciplines or in real life. Constraints on the introductory programming subject from subjects in the second and third years (such as Systems Analysis and Design, other programming subjects, and Database Systems) required students to learn a structured approach to programming hence the need for a design methodology when Visual Basic was introduced. In addition, the design methodology needed to help students understand the concepts of event-driven programming, not to put unnecessary steps in the way.

Moreover, the literature review found that experienced programmers have more *schema* than novice programmers, that is, a greater number of templates or standard algorithms in their long-term memory. When faced with solving a problem, experienced programmers are more likely than novice programmers to identify components of a system's requirements and use schema for parts of the solution. The literature review identified research that advocates an algorithmic approach in the teaching of programming, that is, developing a student's semantic knowledge and building up their schema.

In addition to covering many traditional topics in the introductory programming subject, we gradually introduce students to more and more standard procedures, routines and algorithms as the semester progresses. By the end of the semester students should be able to recall these limited number of standard routines and to apply them to solve similar problems. Although this method of teaching is probably followed in introductory programming subjects in other institutions, it is a *specific* aim of introductory programming subject at Victoria University.

It was therefore interesting to investigate the strategies students use to solve computer programming problems.

6.10.1 Problem Solving Strategies

In the interview, students were asked to outline the way they would normally go about solving a small but unexpected problem in their normal everyday life. As one would expect, there were a range of responses:

> *... organising a school sports. I look at all the things I have to do then do one thing at a time; organise the bus, organise coaches for the teams. I know from experience that if don't have all those things going then you haven't got those things all organised. Always cut it down (Peter).*
>
> *I stew over things for ages. I don't rush into things. If things go wrong I panic and then I would think of something else to do. I think of things I would do in my head as I am going through. For some things I consult someone. If something does not work then we go back (Dianne).*
>
> *I first have to first understand. I have to sit and think and if it is something I can read - it is a problem that is written out, then I have to read it a great many times. I must have an overview. I never rush into solving it and making the mistake. I am very cautious in everything that I do. I always stand back and look at it and then attack it. I test it out in my head before I actually try it and fail at it because I don't want to fail at it.... I like to find my own way and make my own mistakes but I like to know I can have the help of other people. But I do like to work by myself (Gerald).*
>
> *Help. If I am left by myself, I ring someone up and nag them. If I cannot do that, I will just read the problem and try and understand it and break it down and then solve one part at a time and it will go to the second part. If you break it down into*

smaller pieces it is easier to handle than one big problem. It is easier to handle (Elizabeth).

I try to figure out pros and cons of a decision. And I try to weigh that up. Normally if there is more pros I will do it although if I don't want to do it I will.
Before that I would try to do some research - some background to it. And then weigh up the pros and cons and then make a decision. ... If it was a very large problem put up by someone I would probably try and go to them and get their opinion and try to ask other people and see what they would try to do in that position (Jaranka).

Always ask someone who I think may know about the problem and then I try to do it. If no one available then go for it. If I cannot do it then step back and go for it (Georgina).

I first find out what the problem is? I may ask someone. And then try to work out ways to solve the problem. If I have no knowledge of it I ask someone. Read about it (Mark).

When you come to the problem, I would look for alternatives and go with the alternative. I would go and see someone else I know. Researching as well. I would look to see that I could get to do the job (Sylvia).

Define the problem - by the evidence available to you - noises in the engine, What are the symptoms? Does it go? Does it turn over or not? Does it drive on the road or not? Next, almost a process of elimination (Fred).

Think about the steps need to solve it and then go through it Seek help if need it (Seena).

If it a general problem I would go to my parents or friends to get information (Chigden).

The height of craziness. To try to get to a solution in the quickest way is to obviously ask someone who knows a little bit more than you but that doesn't seem to be the approach that students take here (Mary-Beth).

Although some students are less adventurous than others, the responses indicate that most students would seek information on a problem from someone who may have more experience. Moreover, most students, particularly the post-graduate students, appear to place more emphasis on the initial analysing of a problem before attempting a solution.

6.10.2 Strategies Used in Solving a Programming Problem

Students were then asked to identify differences between these strategies and the strategy used in the introductory programming subject. A large number of those students interviewed saw very little difference although many recognised the distinctive steps involved in programming and the penalties that are paid if these are not followed:

> *I don't think it is that much different. It is only more theory. You do define the problem and you do try to test the data - what would happen. It is similar to what we do (Elizabeth).*
>
> *It is very similar but not as rigorously applied. I do all the steps (Clinton).*
>
> *There is not all that much difference. You have your problem and then you have to try to work out the problem then in between there will be a whole lot of problems which you have to solve (Mark).*
>
> *Sort of the same, but with real world have more alternatives to chose (Seena).*
>
> *Similar in the sense and come to you or my friends and ask around in a Visual Basic sense. Theory wise there is not a lot of options I can go to, like in a book because it is not there. Basically you have to be one on one to solve my problem (Chigden).*
>
> *No I don't think so. You are required to have a certain amount of knowledge in order to determine the problem. If you cannot define what a problem is then you cannot go any further in terms of those 6 other steps. And in order to define the problem need to be able to be competent about your knowledge about saying what the problem is. You might have one solution but 15 different problems come out of it. And how people solve it will be 15 different ways (Mary-Beth).*

Some students, particularly the post-graduate students, reveal the difficultly they have adapting their real-life problem solving strategies, to the structured approach adopted in the subject:

> *I find they are very different because the problems I solve are to do with people and their emotions and all of the other things that you cannot really quantify. With a computer I am finding that it is so logical, it is so cut out that if you don't do that, the other things simply won't happen. Like my printer it wouldn't work and later when it worked it printed out 5 things that I had gone tap tap tap and those 5 taps were all lined up. and I couldn't over-ride that they had to do all of that. But that applies to all of this computer work. There are certain steps and you have got to follow them. It is very different to the way I think. ... I can pick up something and I cannot even understand the English in it. That was my problem with maths I could not make any sense of it. But by reading it several times, I will then understand it (Gerald).*
>
> *There is a systematic step to every problem that we have been given. There is a systematic step we have to take to solve it. In everyday life everything is different and I don't think you can really apply something set like that to a problem that you come across everyday (Billy).*
>
> *This is very structured. You have said if you don't do certain things it won't work. That is what I have found. If something does go wrong then you have only got a set way you can do things. So soon or later if you follow the right steps you get the achieve the right answer (Dianne).*
>
> *You have more of a procedure - steps you can follow those. Whereas other problems are not as structured (Joanne).*

Clearly, a large number of students view the program design methodology used in the introductory programming subject as more structured with less alternatives available than would be the case in everyday problems. A large number of students feel that unless they can master the steps in the process, they will not be able to develop a computer solution. Moreover, although many lecturers use examples to introduce new topic areas, it is interesting to note the importance students place on these examples to assist with their understanding. In a paradigm where many students do not have a clear mental picture of how an event-driven system operates, the use of examples appears to play an important role in the teaching of programming concepts.

6.11 Summary

The analysis so far has revealed the strengths and weaknesses of each component of the program design methodology. How effective, however, was this design methodology for the teaching of event-driven systems? As one would expect, there were a range of views about the methodology but it is clearly evident from observations and from interviews with lecturers who taught the introductory programming subject and with students, that the vast majority of students use tools of some form or anther to assist with the design of event-driven systems. Moreover, this would finding matches the views of the experts who were interviewed. Lecturer M believed that the methodology had a number of benefits:

> *Well I thought it was a great idea we had to come sort of way they could work through it logically and, just by giving them a defining diagram and a structure chart really wasn't enough to get them into it. I think you have to give them steps so that they can build it up and I think the 5 - 7 steps get them to look at it from different aspects from the point of view from what the screen looks like, from the point of view from what the responses are going to be, the inputs and outputs and their program variables. And I think for them to look at all that before they start gives them a better idea or a wider range of aspects of what the program is going to do or should do or whatever. Well, whether it actually does that or not I don't know. I think it does - I think we finished up with some very good programs towards the end of semester*

Lecturer O recognised the problems trying to lay down a structured approach to event-driven programming and suggested some changes:

> *It is a very difficult task to come up with a structured approach to something so tempting unstructured as visual programming and I think the event response thing which I think is the core to the whole thing is the key. I think what is missing from that design methodology is some way of coping with multiple forms in a more reliable fashion, I don't think that is formally built into it. ... You are trying to represent a dynamic thing, event driven things necessarily have built into them a time scale which you cannot represent nicely on a piece of paper. There is a precedence that you cannot represent nicely on a piece of paper. I would in such a system, build into the structure of the little event response list the precedence that happens within the program so that I have on the on load event... on got focus would have before on change before on lost focus, you would insist that things happened in that sequence.*

Lecturer T supported Lecturer O's view that the design methodology broke down when multiple form systems were required:

> *After doing it , I think there are a couple of holes in it but it is basically good... I suppose the major hole I have seen in the past is that it doesn't work for multiple forms.*

It is therefore interesting to compare the views of students with lecturers. A number of students felt the methodology used in the introductory programming subject helped then in the design process:

> *The steps are a big help. I know what is coming. I follow them closely (Chigden).*
>
> *This semester I have been doing the 7 steps first then the programming code and that find that, related to the problem statement, and I find that it is a lot easier to go through the 7 steps work out everything and then go and do the program. It breaks down the problem, I guess you could say "Well", and gives you a good understanding of the problem (Billy).*
>
> *I love it. That is my nature to have these all broken into very clear boxes. You should see the way I live. It is just me. To have those things to develop into that. When you get us to do a little thing in the tut one day, I was going back to the problem statement - you were saying don't hack the code straight off - I never will because that is how I think and for a new thing like that that is how I must think. I must see it from that overview that problem statement. So breaking into modules is Gerald! ... I like it makes more sense (Gerald).*
>
> *Yes I do. I try to do as much as I can (Jaranka).*
>
> *I like it. It is easier for me rather than go straight into the coding (Georgina).*
>
> *When I get a picture, I will start going through the 7 steps which I definitely follow (Fred).*

The analysis revealed that many of these students moved back and forth between the various steps before coding their systems. Other students left steps out or changed the order:

> *Yes except for one. The defining diagram and the screen layout is OK. The one where you have to put the variables in I don't understand (Elizabeth).*
>
> *I would do the program variable later and the event response list later as well. It makes more sense to me. I think, this is what it should do, so I will try and do it and then I go back and write an event-response list from that. I would probably do the code and go back and do the others. Only because it would make more sense because I have got the code then I will know to define the program variable table or I might define part of the program variable table, define a few things and add to it (Dianne).*
>
> *I do the defining diagram and the screen layout. I don't do the program variable table, the test data and the event response list. I know what I want so I don't need to do the event response list (Tom).*
>
> *Sometimes I do the screen layout straight after the problem statement so I have a better knowledge of the inputs and outputs depends on the problem. Then I do the*

> *event response list. I usually test at the end. I then do the Program Variable Table (Joanne).*
>
> *The program variable table is very difficult. When I think I have it down pat, when I go to use it again it is difficult. ... I usually jump into the algorithm, into the programming because I think I would be better off that way. What I have found, that I went to do the 7 steps but it appears like a waste of time unless I get the program to go. I need help with the programming. I make it work and move my way back (Sylvia).*
>
> *I usually get the problem and go through the problem to work out what you want the program to do. And then I do my screen layout based around that, you can put everything in there and get everything out you want. And then I do the screen layout on the computer and try and type in the code then (Mark).*
>
> *I prefer to have the variable table after the event response and then the algorithm and then the test data at the end. It is jump like you are trying to structure the program once you get the event response and then it is the test data - it seems out of place for me. Once you have worked through the algorithm and the program variables I think it is just logical to do the test data (Billy).*

Observations reveal that the vast majority of students adopted, in varying degrees, the first four steps ie. the problem statement, the defining diagram, the screen layout and the event-response list, and a limited form of pseudocode or Structured English. There is clear evidence to suggest that the majority of students were confused by the program variable table or found it unnecessary, and that most students adopted an ad hoc approach to testing. Lecturer M states:

> *I still have got problems with it. I think although I have done it a bit differently in the book where I try to say now where in the coding any way I have tried to say each value that comes from an input box is allocated to a variable - we haven't done that before. I have got problems because they don't know..., I think they should define the variables they are likely to use and then use them in the algorithm. Now some of them don't. Some of them still use other words in the algorithm or something or other.*

Lecturer T expresses concern about the program variable table as well as pseudocode:

> *I think that up to the event response has definite possibilities, there is a little bit of change suggested but the pseudocode I have doubts about because what I was saying before I think what we tend to be doing tends not to be highly algorithmic for the parts that are fine but to make them do pseudocode for the whole lot I am not sure. I don't know about the programming variables to some point they have got to do variables I am not sure whether that's a good place.*

It is possible that a major benefit of this methodology is its flexibility, that is, it enables students to select, re-order or modify individual steps that better suit their particular problem solving strategies, although it was never intended to be so. A concern, however, is that the deletion of too many steps could lead to an ad hoc approach to programming. Mary-Beth states:

> *I think they do the defining diagram because I think in the long run it actually helps them to get to where they are going. It is irrelevant that it is the last stage but maybe the first step is defining the problem, the second step is drawing a visual of the screen and the next step is input/output. Maybe that is something that*

> *people need to know as well. Just because it says 1 through 7 it doesn't mean that is the sequence.*

Although some modifications may be necessary the analysis of the data suggests that the program design methodology has been successful in teaching a structured approach to programming event-driven systems.

In summary, the analysis of the methodology indicates that, although some students find the steps advantageous in the development process, many students are heavily reliant upon the screen layout and are confused about the meaning of events and where to place the code. Moreover, many students often fail to establish a clear mental picture that depicts the operation of event driven systems. Conversion of pseudocode to code syntax and testing routines, which are common problems experienced by students using procedural languages, continues to plaque inexperienced students using event driven languages.

Chapter 7 Conclusion

One of the unique characteristics of the computing discipline is rapid and on-going change. When tertiary institutions first established Information Systems courses to cater for the growing demand of business for graduates, they too locked themselves into this endless cycle of change. Now, as computers are becoming increasingly powerful and permeate more of the daily operations of businesses, tertiary institutions are being forced to re-evaluate the content of their courses.

7.1 Findings

7.1.1 Language Trends in Introductory Programming Subjects

Programming, always a fundamental building block in Information Systems curriculum, is one area which has not escaped pressure to change. Programming environments have changed significantly in recent times with the advancements in technology and software. The literature reveals that the traditional procedural programming paradigm, so long the backbone of introductory programming subjects, is increasingly being replaced with languages from other paradigms. The literature also reveals that, with the widespread use of application languages, it is possible to categorise these languages according to their use, rather than by the paradigm. The first category is languages used by professional programmers such as C or C++, the second category is power-user languages such as Delphi or Visual Basic, and the final category is user languages such as macros in spreadsheets. This study reveals that programming subjects in Information Systems courses appear to be moving away from procedural languages and towards power-user languages opening up a clear distinction from Computer Science courses in the area of the teaching of programming. It is possible to represent this trend diagrammatically (see Figure 7. 1) with the shaded areas in the diagram depicting the domain of each respective discipline area.

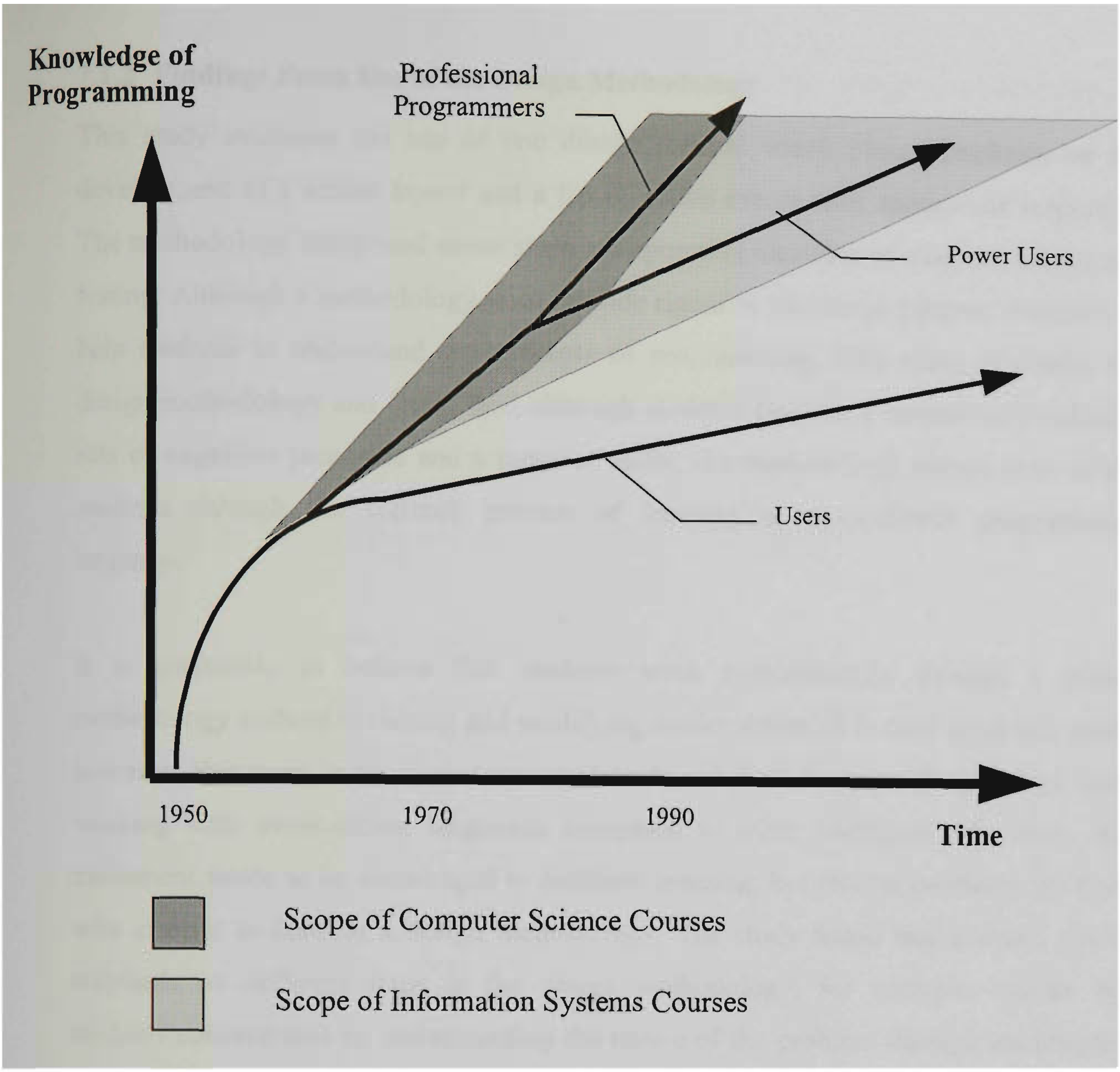


Figure 7. 1: Scope of Information Systems Courses in Teaching Programming

It is clear that introductory programming subjects aim to teach a range of programming concepts rather than just the syntax of a language. Fundamental to these subjects is the notion that students should plan before acting. Structured programming is the most popular method used for procedural languages and has a number of conceptual modelling tools such as structure charts and hierarchy charts. There is, however, no recognised methodology for the event-driven paradigm and it is clear from the study that lecturers who are using event-driven languages are still trying to determine an appropriate methodology to enhance learning programming concepts and in presenting material.

7.1.2 Findings From Use of the Design Methodology

This study evaluates the use of one design method which places emphasis on the development of a screen layout and a list of active events with appropriate responses. The methodology comprised seven steps incorporating elements of analysis, design and testing. Although a methodology must provide rigour in the design process, it must also help students to understand the concepts of programming. This study evaluated the design methodology and found that, although students came to a subject with differing sets of cognitive processes and a range of skills, the methodology helped most of the students through the difficult process of learning an event-driven programming language.

It is unrealistic to believe that students work systematically through a design methodology without revisiting and modifying earlier stages. It is clear from this study, however, that there is far more movement back and forth between design steps when working with event-driven languages compared to other platforms. Moreover, this movement needs to be encouraged to facilitate learning, but creates problems for those who attempt to develop a design methodology. The study found that students placed emphasis on different steps in the design methodology; for example, mature age students concentrated on understanding the nature of the problem through the program statement, while younger undergraduates appeared to launch straight into screen design.

The literature suggests that human memory is essentially associative and must be presented with concrete retrieval cues to make contact with relevant knowledge. Although most students come to Information Systems courses having worked in a Windows environment, many have great difficulty in the early stages understanding the concepts of event-driven systems. The study showed that many students have very little trouble developing and modifying a screen layout but have great difficulty conceptualising the operation of the system to assist them with development beyond this stage. A major cause of this problem is the inability of most students to develop a corresponding mental representation of event-driven systems, which is exacerbated by the fact that there is no suitable conceptual model that can adequately describe the operation of an event-driven system. Although event-driven languages have unique characteristics, they incorporate aspects of the object-oriented and procedural paradigm and the study found that lecturers, often unknowingly, find themselves describing an event-driven system in one of these other paradigms creating even more confusion for students. Moreover, the study found that lecturers do not have a range of real-world examples and metaphors that can assist them to describe event-driven systems and teach programming in this paradigm. This may be partially due to the limited amount of support for the language at this early stage but the study found lecturers often face difficulty developing an appropriate mind-set for event-driven languages and leaving behind their procedural habits.

7.2 Recommendations for Further Research

7.2.1 Revised program design methodology

In recent years a number of object-oriented analysis and design methodologies have adopted a Use Case approach to analyse system data structures and behaviours. An alternative to the first two steps in the design methodology could be to develop a use statement. The Use Statement may comprise a number of cases depending upon the number of functions performed by a system eg. a Use statement for a system which can add details to a database, delete data or modify details would comprise at least three cases. Each case in the statement would have a name, detail the purpose and show the typical course of the system. Where an alternative course may occur such as for validation of an input, an alternative course of the system would be outlined. As an example, suppose we need a program that would allow the user to input the hours and minutes an employee worked during the week together with the rate of pay, so that the program could then calculate the decimal value of the hours worked and produce a pay slip for that employee. We could write the Use Statement as:

Name:	Produce Pay Slip
Purpose:	Describes the process of producing a pay slip for an employee

Typical Course of System:

1. The user enters the employee name.
2. The user enters the hours worked, the minutes worked and the pay rate in any order.
3. On entry of hours worked, the minutes worked or the pay rate the time worked and gross pay are calculated and displayed.

Another advantage of a use statement is that it caters for multiple screen systems.

The event response table in the current methodology could be altered to identify the object and the type of event for *active* events. Using the example above an event response table could include heading and entries as follows:

Object	Object Type	Type of Event	Response
txtHours	Text box	Lost Focus	Calculate and display time worked and gross pay
txtMinutes	Text box	Lost Focus	Calculate and display time worked and gross pay
txtPayRate	Text box	Lost Focus	Calculate and display time worked and gross pay
cmdExit	Command button	Click	The program is ended

Finally, the testing routines could be modified to include test cases and the test data. This may help students to move methodically through the testing process.

7.2.2 Recommendations for further research

Traditionally, programming has been taught in the small with the belief that many of the concepts could be extrapolated to the large. A major criticism of the program design methodology used in this study is that it is limited to teaching small systems. There is a need for further research which will enable a design methodology to be used for teaching and for the development of large systems. It is, frankly unacceptable that a paradigm so popular as the event-driven paradigm, comes with no design methodology and therefore has the potential to undo so much of the long term benefits that come from properly designed systems.

Another area that also would seem to warrant further investigation is the impact on the content and order of proceeding subjects in an Information Systems curriculum when an introductory programming subject adopts an event-driven language as its platform. What changes would need to be made to subjects such as systems analysis and design? Is it important to include new subjects such as user interface design in the curriculum and, if so, how early in the course should it be taught? How much importance should be place on the teaching of other languages such as procedural languages in the Information Systems curriculum?

It would be unrealistic to expect that one of the first program design methodologies developed to teach novices how to program using an event-driven language would be entirely successful. Nevertheless, this study makes a significant contribution in attempting to develop a structured approach to the teaching of these types of systems and provides a valuable insight into some of the problems faced by lecturers who adopt event-driven languages as part of their courses.

Bibliography

Aaronson, D. (1976), 'Performance Theories for Sentence Coding: Some Qualitative Observations', *Journal of Experimental Psychology: Human Perception and Performance*, 2, 42-45.

Abelson, H., Sussman, G. & Sussman, J. (1985), *Structure and Interpretation of Computer Programs,* 2nd Edition, MIT Press, New York.

Adams, Joel C. (1996), 'Object-Oriented Design: A Five-Phase Introduction to Object-Oriented Programming in CS1-2', *in* Klee, Karl J., (ed), *Twenty-Seventh SIGCSE Technical Symposium on Computer Science Education*, Philadelphia, Pennsylvania, ACM Press, 78 - 82.

Alexander, Patricia & Judy, Judith (1988), 'The Interaction of Domain-Specific and Strategic Knowledge in Academic Performance', *Review of Educational Research,* 58(4), 375-404.

Ang, Ang Yang & Lo, Bruce Wai Ning (1991), 'Changing Emphasis on Information Systems Curriculum: An Australian Perspective', *Second Annual Australian Conference on Information systems and Databases Special Interest Group*, University of New South Wales.

Armstrong, A. (1989), 'The Development of Self-Regulation Skills Through the Modelling and Structuring of Computer Programming', *Educational Technology Research & Development,* 37(2), 69-76.

Attwood, M. E. & Ramsay, H. R. (1978), *Cognitive Structures in the Comprehension and Memory of Computer Programs: An investigation of Computer Debugging*, U.S. Army Research Institute for the Behavioural and Social Sciences, Report Number TR-78A21, Alexandria, VA.

Bayman, Piraye & Mayer, Richard (1988), 'Using Conceptual Models to Teach BASIC Computer Programming', *Journal of Educational Psychology,* 80(3), 291 - 298.

Beck, Kent (1994), 'Patterns and Software Development', *Dr Dobbs Journal,* February, 1994, .

Bell, J. (1987), *Doing Your Research Project*, Open University Press, Stratford, England.

Bellin, D. (1992), 'A Seminar Course in Object Oriented Programming', *SIGCSE Bulletin,* 24(1), 134 - 137.

Benbasat, I., Goldstein, D. & Mead, M. (1987), 'The Case Research Strategy in Studies of Information Systems', *MIS Quarterly,* 11(3 (September, 1987)), 369-386.

Biddle, Robert & Tempero, Ewan (1996), 'Explaining Inheritance: A Code Reusability Perspecitve', *in* Klee, Karl J., (ed), *Twenty-Seventh SIGCSE Technical Symposium on Computer Science Education*, Philadelphia, Pennsylvania, ACM Press, 217 - 221.

Bogdan, R. & Biklen, S. (1992), *Qualitative Research for Education: An Introduction to Theory and Methods*, Allyn and Bacon, U.S.A.

Bogdan, R. & Taylor, S (1975), *Introduction to Qualitative Research Methods*, Wiley, New York.

Boland, R. J. (1979), 'Control, Causality and Information Systems Requirements', *Accounting, Organizations and Society,* 4, 259 - 272.

Bonnici, Joseph & Warkentin, Merrill E. (1995), 'Revisted: Fabbri and Mann's Criticism of the DPMA Model Curriculum', *Journal of Computer Information Systems,* 35(3), 96 - 98.

Booch, Grady (1991), *Object Oriented Design with Applications*, Benjamin/Cummings Publishing, Redwood City, California.

Bostrom, Robert, Olfman, Lorne & Sein, Maung (1988), 'End-User Computing: A Research Framework for Investigating the Training/Learning Process', *in* Carey, Jane M. (Ed.) *Human Factors in Management Information Systems*, Norwood, New Jersey, Ablex Publishing, 221 - 250.

Boulay, B. & O'Shea, T. (1981), 'The black box inside the glass box: presenting computing concepts to novices', *International Journal of Man-Machine Studies,* 14(1), 237-249.

Braithwaite, Richard (1996), 'Design Tools for Event-driven Software Systems', *Australasian Conference on Information Systems*, Hobart, Tasmania, Australia, University of Tasmania, 69 - 78.

Brooke, J. & Duncan, K. (1980), 'Experimental studies of flowchart use at different stages of program debugging.', *Ergonomics,* 23(11), 1057-1091.

Brooks, R. (1977), 'Towards a theory of the cognitive processes in computer programming', *International Journal of Man-Machine Studies,* 9, 737-751.

Brooks, R. (1983), 'Towards a theory of comprehension of computer programs', *International Journal of Man-Machine Studies,* 18, 543-554.

Burnett, Margaret M & McIntyre, David W. (1995), 'Visual Programming', *IEEE Computer,* March, 1995, 14 - 16.

Burrell, G. & Morgan, G. (1979), *Sociological Paradigms and Organization Analysis,* Heinemann Books, London.

Carroll, John, Thomas, John & Malhotra, Ashok (1979), 'Clinical-experimental Analysis of Design Problem Solving', *Design Studies,* 1(2), 84 - 92.

Carroll, John, Thomas, John & Malhotra, Ashok (1985), 'Presentation and Representation in Design Problem-solving', *British Journal of Psychology,* 71, 143 - 153.

Carroll, J., Thomas, J., Miller, L. & Friedman, H. (1980), 'Aspects of Solution Strucutre in Design Problem-solving'', *American Journal of Psychology,* 93(2), 269-284.

Cattell, R., (Ed.) (1987), *Intelligence: Its structure, growth and action,* North Holland : Elsevier Science Publishers.

Chang, Shi-Kuo (1990), *Principles of Visual Programming Systems,* Prentice-Hall Inc., Englewood, Cliffs, New Jersey.

Coad, Peter & Yourdon, Edward (1990), *Object Oriented Analysis,* Yourdon Press, Englewood Cliffs, New Jersey.

Cooke, R. (1990), 'Full Circle', *Byte,* August, 1990, 211-214.

Cornell, Gary (1993), *The Visual Basic 3 for Windows Handbook*, Osbourne McGraw-Hill, California, USA.

Craig, Annemieke (1996), *Encouraging Female Students in Business Computing*, M. Business Thesis, Victoria University of Technology, Melbourne, Australia.

Cross, James H. II & Philips, Thomas M. (1996), 'Successfully Integrating Traditional and Object-Oriented Approaches with ADA 95', *in* Klee, Karl J., (ed), *Twenty-Seventh SIGCSE Technical Symposium on Computer Science Education*, Philadelphia, Pennsylvania, ACM, 19 - 23.

Curtis, Bill, (Ed.) (1985), *Human Factors in Software Development*, IEEE Computer Society Press, Washington.

Decker, Rick & Hirshfield, Stuart (1994), 'The Top 10 Reasons Why Object-Oriented Programming Can't Be Taught in CS 1', in Joyce, Daniel, (ed), *Twenty-Fifth SIGCSE Technical Symposium on Computer Science Education*, Phoenix, Arizona, ACM, 51 - 55.

DeClue, Tim (1996), 'Object-Orientation and the Principles of Learning Theory: A New Look at Problems and Benefits', *in* Klee, Karl J., (ed), *Twenty-Seventh SIGCSE Technical Symposium on Computer Science Education*, Philadelphia, Pennsylvania, ACM Press, 232 - 236.

Department of Employment, Education and Training. Economic and Policy Analysis Division (1990), *Education and Training Needs of Computing Professionals and Para-professionals in Australia*, Information Paper No. 1, Volume 1: Data and Findings, Australian Government Publishing Service, Canberra, Australia.

Department of Employment, Education and Training, Department of Industry, Technology and Commerce & Information Industries Education and Training Foundation (1992), *Report of the Discipline Review of Computing Studies and Information Sciences Education*, Volume 2, Australian Government Publishing Service, Canberra, Australia.

DeGrace, P. & Hulet Stahl, L. (1990), *Wicked problems, righteous solutions: a catalogue of modern software engineering paradigms*, Prentice-Hall, New Jersey.

Denenberg, S. (1990), 'A course for the nonmajor: Teaching Collaborative Problem Solving via the Classical Computer Science Space-Time Tradeoff', *in* McDougall, A. & Dowling, C. (Eds.), *Computers in Education*, Amsterdam, North-Holland.

Dijkastra, Edward (1968), 'Go To Statement Considered Harmful', *Communications of the ACM,* 11(3), 147 - 148.

Dumas, Joseph & Parsons, Paige (1995), 'Discovering the Way Programmers Think About New Programming Environments', *Communications of the ACM,* 38(6 (June)), 45 - 56.

Duntemann, Jeff (1993), 'The (Shower) Curtain Falls', *Dr Dobb's Journal,*(April), .

Eberts, R (1987), 'Human Computer Interaction', *in* Hancock, P. (Ed.) *Human Factors Psychology Advances in Psychology Series*, New York, New Holland.

Evans, M.D. (1996), 'A New Emphasis & Pedagogy for a CS1 Course', *SIGCSE Bulletin,* 28(3), 12 - 16.

Fabbri, Tony & Mann, Ronald A. (1993), 'A Critical Analysis of the ACM and DPMA Curriculum Models', *The Journal of Computer Information Systems,* 34(1), 77 - 80.

Fienup, Mark (1996), 'Rethinking the CS-2 Course with an Object-Oriented Focus', *SIGCSE,* 28(3), 23 - 25.

Foreman, K. (1990), 'Cognitive Characteristics and Initial Acquisition of Computer Programming', *School of Education Review,* 2(Spring), 55-61.

Galliers, R. (1991), 'Choosing Appropriate Information Systems Research Approaches: A Revised Taxonomy', *in* Nissen, H., Klein, H. & Hirschheim, R. (Eds.), *Information Systems Research: Contemporary Approaches & Emergent Traditions,* Amsterdam, North-Holland, 327-346.

Galliers, R. (1992), *Information Systems Research: Issues, Methods and Practical Guidelines,* Blackwell Scientific Publications, London.

Gersting, Judith (1994), 'A Software Engineering Frosting on Traditional CS-1 Course', *Proceedings of the 25th SIGCSE Technical Symposium,* ACM, 233 - 237.

Giddens, Anthony (1987), *Social Theory and Modern Sociology,* Standford University Press, Stanford, California.

Gilmore, D. & Green, T. (1984), 'Comprehension and recall of miniature programs.', *International Journal of Man-Machine Studies,* 21, 31-48.

Glinert, E. (1990), *Visual Programming Environments: Applications and Issues,* IEEE Computer Society Press, Washington.

Glinert, P (1990), 'Nontextual Programming Environments', *in* Chang, Shi-Koo (Ed.) *Principles of Visual Programming Systems,* New Jersey, Prentice Hall, 144 - 230.

Goodson, Ivor F & Mangan, J. Marshall (1991), 'An Alternative Paradigm for Educational Research', *in* Goodson, Ivor F & Mangan, J. Marshall (Eds.), *History Context and Qualitative Methods in the Study of Education,* , RUCCUS Occasional Papers Vol 1, University of Western Ontario, Canada.

Gorgone, John T., Couger, J. Daniel, Davis, Gordon, et al. (1994), 'Information Systems '95 Curriculum Model - A collaborative Effort', *Data Base,* 25(4), 5 - 8.

Goschnick, S. (1993), 'Object-Oriented Language Set to be the Inventive Code-Cutter's Best Ally', *The Age,* 12th October, 1993.

Goslar, Martin D. & Deans, P. Candace (1993), 'A Pilot Study of Information Systems Curriculum in Foreign Educational Institutions', *The Journal of Computer Information Systems,* 34(1), 8 - 17.

Greeno, J. G. (1973), 'The Structure of Memory and the Process of Problem Solving', *in* Solo, R. (Ed.) *Contemporary Issues in Cognitive Psychology,* Washington, Winston.

Gugerty, Leo & Olson, Gary M. (1986), 'Comprehension Differences in Debugging by Skilled and Novice Programmers', *in* Soloway, Elliott & Sitharama, Iyengar (Eds.), *Empirical Studies of Programmers,* Norwood, New Jersey, Ablex Publishing Corporation, 13 - 27.

Güvenir, H. Altay (1995), 'An Object-Oriented Tutoring System for Teaching Sets', *SIGCSE Bulletin,* 27(3), 39 - 46.

Guzdial, Mark (1995), 'Centralized Mindset: A Student Problem with object-Oriented Programming', *in* White, Curt, (ed), *Twenty-Sixth SIGCSE Technical Symposium on Computer Science Education*, Nashville, Tennessee, ACM Press, 182 - 185.

Hanif, Mohammed (1996), *The Need for a Representation System for Visual Basic Program*, Victoria University of Technology, Melbourne, Australia.

Harmon, Paul (1995), 'Object-Oriented AI: A Commercial Perspective', *Communications of the ACM,* 38(11 (November)), 80 - 86.

Johnson-Laird, P.N. (1989), *The Computer and the Mind - An introduction to Cognitive Science.*, Fontana Press, London.

Kant, Elaine (1985), 'Understanding ans Automating Algorithm Design', *Transactions on Software Engineering,* 11, 1361 - 1374.

Kant, E. & Newell, A. (1984), 'Problem Solving Techniques for the Design of Algorithms', *Information Processing & Management,* 20(1-2), 97-118.

Kaplan, B. & Duchon, D. (1988), 'Combining Qualitative and Quantitative Methods in Information Systems Research: A Case Study', *MIS Quarterly,* 12(4 (December)), 369-386.

AGPS (1985), *Quality of Education in Australia*, , Canberra, Australia.

Keen, P.G.W. (1987), 'MIS Research: Current Status, Trends and Needs', *in* Buckingham, R.A, Hirschheim, R.A., Land, F.F. & Tully, C.J. (Eds.), *Information Systems Education: recommendations and implementation*, Cambridge, Cambridge University Press, 1 - 13.

Kennedy, Stuart (1995), 'Object Occult', *Information Age*, May, 1995, 16 -19.

Kirk, J. & Miller, M. (1990), *Reliability and Validity in Qualitative Research*, Sage Publications, Newbury Park, California.

Klein, H., Nissen, H. & Hirschheim, R. (1991), 'A Pluralist Perspective of the Information Systems Research Arena', *in* Nissen, H., Klein, H. & Hirschheim, R. (Eds.), *Information Systems Research: Contemporary Approaches & Emergent Traditions*, Amsterdam, North-Holland, 1-20.

Kölling, Michael, Koch, Bett & Rosenberg, John (1995), 'Requirements for a First Year Object-Oriented Teaching Language', *in* White, Curt, (ed), *Twenty-Sixth SIGCSE Technical Symposium on Computer Science Education*, Nashville, Tenessee, ACM, 173 - 177.

Kölling, Michael & Rosenberg, John (1996), 'An Object-Oriented Program Development Environment for the First Programming Course', *in* Klee, Karl J., (ed), *Twenty-Seventh SIGCSE Technical Symposium on Computer Science Education*, Philadelphia, Pennsylvania, ACM Press, 83 - 87.

Koubek, R., Salvendy, G. & Dunsmore, H. (1989), 'Cognitive issues in the process of software development: review and reappraisal', *International Journal of Man-Machine Studies,* 30, 171-191.

Kraft, P. (1977), *Programmers and Managers - The Routinization of Computer Programming in the United States*, Springer Verlag, New York.

Kurland, D., Clement, C. & Mawby, R. (1986), 'A Study of the Development of Programming Ability and Thinking Skills in High School Students', *Journal of Educational Computing Research,* 2(4), 429-458.

Letovsky, S. (1986), 'Cognitive Processes in Program Comprehension', *in* Soloway, E. & Ivengar, S. (Eds.), *Empirical Studies of Programmers*, New Jersey, Ablex Publishing, 58-79.

Levy, Suzzane Pawlan (1995), 'Computer Language Usage in CS1: Survey Results', *SIGCSE Bulletin,* 27(3), 21 - 26.

Lincoln, Y.S. & Guba, E.G. (1985), *Naturalistic Inquiry*, Sage Publications, Beverley Hills, California.

Linn, Marcia C. (1985), 'The Cognitive Consequences of Programming Instruction in Classrooms.', *Educational Researcher,* 14(5), 14-16.

Linn, Marcia C. & Songer, Nancy Butler (1991), 'Cognitive and Conceptual Change in Adolescence', *American Journal of Education,* 99(4), 379 - 343.

Lisack, Susan (1996), 'Making the Plunge: Switching from COBOL to a New Language', *Tenth Annual Midwest Conference on Information Systems*, Loyola University, Chicago, 1 - 6.

Lisack, Susan (1996), 'The Move to a GUI Programming Environment: Selecting an Appropriate Language for the Programming Curriculum', *Tenth Annual Midwest Conference on Information Systems*, Loyola University, Chicargo.

Littman, David C., Pinto, Jeannine, Letovsky, Stanley & Soloway, Elliot (1986), 'Mental Models and Software Maintenance', *in* Soloway, Elliot & Iyengar, Sitharama (Eds.), *Empirical Studies of Programmers*, Norwood, New Jersey, Ablex Publishing Corporation.

Longenecker, Herbert E. Jnr, Feinstein, David L., Gorgone, John T. & Davis, Gordon G. J (1995), *Information Systems '95: Model Curriculum and Guidelines for Undergraduate Degree Programs in Information Systems - Draft Report*, ACM, DPMA, AIS, U.S.A.

Longenecker, Herbert E. Jnr, Feinstein, David L., Couger, J. Daniel, Davis, Gordon G. & Gorgone, John T. (1994), 'Information Systems '95: A Summary of the Collabarative IS Curriculum Specification of the Joint DPMA, ACM, AIS Task Force', *Journal of Information Systems Education,* Winter, 174 - 186.

Lyytinen, K. (1987), 'Different Perspectives on Information Systems: Problems and Solutions', *ACM Computing Surveys,* 19(1), 5 - 46.

Malhotra, A., Thomas, J., Carroll, J. & Miller, L. (1980), 'Cognitive processes in design', *International Journal of Man-Machine Studies,* 12, 119-140.

Marion, Bill (Moderator), Chi, Paul Ping-Chung, Smith, Suzanne, Moses, Louise & Stoecklin, Sara (1996), 'Experiences in Teaching Object-Oriented Design and Programming with C++ in the Computer Science Curriculum', *in* Klee, Karl J., (ed), *Twenty-Seventh SIGCSE Technical Symposium on Computer Science Education*, Philadelphia, Pennsylvania, ACM Press, 405 - 406.

Martin, James & McClure, Carma (1984), *Diagramming Techniques for Analysts and Programmers*, Prentice Hall Inc., New Jersey.

Martin, James & Odell, James (1992), *Object-Oriented Analysis & Design*, Prentice Hall, Englewood Cliffs, New Jersey.

Matsumura, Kazuo & Tayama, Shuichi (1990), 'Visual Man-Machine Interface for Program Design and Production', *in* Ichikawa, Tadao, Jungert, Erland & Korfhage, Robert R. (Eds.), *Visual Languages and Applications*, New York, Plenum Press, 99 - 119.

McNeill, P. (1990), *Research Methods.*

McNutt, L. J. & Lo, B. W. N. (1991), 'The Cognitive Dimensions of Computer Programming: Implications for Teaching Programming in Information Systems Courses', *Second Annual Australian Conference on Information Systems and Databases Special Interest Group*, University of New South Wales, 403-414.

Miles, Matthew (1983), 'Qualitative Data as an Attractive Nuisance: The Problem of Analysis', *in* Van Maanen, John (Ed.) *Qualitative Methodology*, Newbury Park, California, Sage Publications, 117 - 134.

Mintzberg, H. (1983), 'An Emerging Strategy of "Direct" Research', *in* Van Maanen, J. (Ed.) *Qualitative Methodology*, Newbury Park, Sage Publications, 24, 105 -116.

Myers, M. D. (1994), 'Quality in Qualitative Research in Information Systems', Shanks, G. and Arnott, D., (ed), *Proceedings of 5th Australasian Conference on Information Systems*, Monash University, Department of Information Systems, Monash University, 763 - 766.

Navrat, Pavol (1994), 'Hierarchies of Programming Concepts: Abstraction, Generality, and Beyond', *ACM SIGCSE Bulletin,* 26(3 (September, 1994)), 17 - 22.

Neal, Gregory & Lorents, Alden (1995), 'Introduction to Programming with Visual Basic', *in* Little, Joyce Currie, (ed), *ISECON'95 IS Education: Meeting the Challenge of a Global MarketPlace*, Charlotte, North Carolina, DPMA, 54 - 61.

North, Ken (1994), 'Database Development and Visual Basic 3.0', *Dr Dobbs Journal,* February, 1994, .

Oliver, Ron (1991), 'The Development of Semantic Knowledge in Introductory Programming Teaching', *Australian Computers in Education Conference*, Brisbane, Australia, CEGQ, 264-272.

Orlikowski, W.J. & Bardoudi, J.J. (1991), 'Studying Information Technology in Organizations: Research Approaches and Assumptions', *Information Systems Research,* 2(1), 1 - 28.

Osbourne, M. & Johnson, J. (1993), 'An Only undergraduate Course in Object-Oriented Technology', *SIGCSE Bulletin,* 25(1), 101 - 106.

Palumbo, D. & Reed, W. M. (1991), 'The Effect of BASIC Programming Language Instruction on High School Students' Problem Solving Ability and Computer Anxiety', *Journal of Research on Computing in Education,* 23(3 (Spring)), 343 - 372.

Pancake, Cherri M. (1995), 'The Promise and the Cost of Object Technology: A Five-Year Forecast', *Communications of the ACM,* 38(10), 33 - 49.

Pea, Roy. D. & Kurland, D. Midian (1983), *On the Cognitive Prerequisites of Learning Computer Programming (Technical Report No. 18)*, Bank Street College of Education, New York.

Pennington, N. (1982), *Cognitive Components of Expertise in Computer Programming: A Review of the Literature*, Technical Report No 46, University of Michigan Centre for Cognitive Science, Michigan.

Petre, Marian (1995), 'Why Looking Isn't Always Seeing', *Communications of the ACM*, 38(6), 33 - 44.

Ratcliff, B. & Siddiqi, J (1985), 'An Empirical Investigation inot Problem Decompostion Strategies used in Program Design', *International Journal of Man-Machine Studies*, 22(1), 77-90.

Reid, Robert (1994), *The Reid Report*, U.S.A.

Reitman, W. (1965), *Cognition and Thought*, Wiley, New York.

Resnick, Mark (1992), *Beyond the Centralized Mindset: Explorations in Massively-Parallel Microworlds*, Unpublished Thesis, Massachusetts Institute of Technology.

Rich, Charles & Waters, Richard C. (1988), 'The Programmer's Apprentice: A Research Overview', *Computer*, 21(11), 10 - 25.

Richards, Roy Martin & Sanford, Clive C. (1992), 'An Evolutionary Change in the Information Systems Curriculum at the University of North Texas', *Computers & Education*, 19(3), 219 - 228.

Robertson, L. A. (1993), *Simple Program Design*, Second Edition Edition, Thomas Nelson, Melbourne, Australia.

Rosen, M. (1991), 'Coming to Terms with the Field: Understanding and Doing Organizational Ethnography', *Journal of Management Studies.*

Rowan, J (1973), *The Social Individual*, Davis-Poynter, London.

Rumbaugh, James, Balha, Michael, Premerlani, William, Eddy, Frederick & Lorensen, William (1991), *Object-Oriented Modeling and Design*, Prentice Hall, Englewood Cliffs, New Jersey.

Schurmann, Joachim (1994), 'A Program Architecture for Visual Basic Development', *Dr Dobb's Journal*, 19(9 (Fall)), 32- 41.

Seddon, P. (1991), 'Information Systems: Towards a Definition for the 1990's', *Second Annual Australian Conference on Information systems and Databases Special Interest Group*, University of New South Wales, 415-427.

Shackleton, Peter & McConville, Doug (1997), *Program Design Through Visual Basic*, 3rd Edition, Data Publishing, Melbourne, Australia.

Shaler, Sally & Mellor, Stephen J. (1992), *Object Lifecycles: Modelling the World in States*, Yourdon Press/Prentice Hall, Englewood Cliffs, New Jersey.

Shneiderman, Ben (1982), 'Control Flow and Data Structure Documentation: Two Experiments', *Communications of the ACM*, 25(1), 55 - 63.

Shneiderman, B (1983), 'Direct Manipulation: A Step Beyond Programming Languages', *IEEE Computer,* 16(8), 57 - 69.

Shneiderman, B. (1986), 'Empirical Studies of Programmers: The Territory, Paths and Destinations', *in* Soloway, E. & Iyengar, S. (Eds.), *Empirical Studies of Programmers,* New Jersey, Ablex Publishing, 1-12.

Shneiderman, B. & Mayer, R. (1979), 'Syntactic/Semantic Interactions in Programmer Behaviour: A Model and Experimental Results', *International Journal of Computer and Information Sciences,* 8(3), 219-238.

Sime, M.E., Green, T.R.G. & Guest, D.J. (1973), 'Psychological Evaluation of Two Conditional Structures Used in Computer Languages', *Journal of Man-Machine Studies,* 5, 123 - 143.

Sloan, K.D. & Linn, M.C. (1988), 'Instructional Conditions in Pascal Programming Classes', *in* Mayer, R.E. (Ed.) *Teachning and Learning Computer Programming: Multiple Research Perspectives,* New Jersey, Lawrence Erbaum Associates, 207 - 235.

Smith, Dianne & Kelly, Paul (1994), 'A Visual Approach to Computer Programming', *Women in Computing*, Melbourne, Victoria, Victoria University, Melbourne.

Soloway, E., Bonar, J. & Ehrlich, K. (1983), 'Cognitive Strategies and Looping Constructs: An Empirical Study', *Communications of the ACM,* 26(11), 853-860.

Soloway, E. & Elliott, K. (1984), 'Empirical Studies of Programming Knowledge', *IEEE Transaction on Software Engineering,* SE-10(5), 595-609.

Soloway, E. & Iyengar, S. (1986), *Empirical Studies of Programmers*, Ablex Publishing, Norwood, New Jersey.

Sudbury, Neil (ed) (1995), 'Design, not objects, decides useability', *Computer Week*, 4th August, 1995, 18.

Tatnall, A (1993), *A Curriculum History of Business Computing in Victorian Tertiary Institutions from 1960 - 1985*, Master of Arts Thesis, Deakin University, Victoria. Australia.

Tatnall, Arthur & Davey, Bill (1995), 'A Conceptual Development in an Object Environment', *World Conference on Computer Education*, Burmingham, England.

Tesch, Debbie B., Tesch, Robert C. & Albin, Marvin (1996), *The Role of Visual Basic in the CIS Curriculum*, Working Paper, Morehead State University.

Tesch, R. (1990), *Qualitative Research: Analysis Types and Software Tools*, Falmer Press, Great Britain.

Trauth, E. & O'Connor, B. (1991), 'A Study of the Interaction Between Information, Technology and Society: An Illustration of Combined Qualitative Research Methods', *in* Nissen, H., Klein, H. & Hirschheim, R. (Eds.), *Information Systems Research: Contemporary Approaches & Emergent Traditions*, Amsterdam, North-Holland, 131 - 144.

Tucker, Allen (ed) (1991), 'Computing Curricula 1991:A Summary of the ACM/IEEE - CS Joint Curriculum Task Force Report', *Communications of the ACM,* 34(6), 69 - 84.

Turkle, Sherry & Papert, Seymour (1992), 'Epistemological Pluralism: Styles and Voices Within the Computer Culture', *in* Martin, C. Dianne & Murchie-Beyma, Eric (Eds.), *In Search of Gender Free Paradigms for Computer Science Education*, Oregon, Internation Society for Technology in Education, 113 - 139.

Udell, Jon (1994), 'Componentware', *Byte*, May, 1994, 46 - 62.

Van Kirk, D. (1995), 'Visual Basic: a Diamond in the Rough', *Computer Week*, 4th August, 1995.

Van Merrienboer, J. & Paas, F. (1990), 'Automation and Schema Acquisition in Learning Elementary Computer Programming: Implications for the Design of Practice', *Computers in Human Behaviour*, 6(3), 273-289.

Van Papstein, P. & Frese, M. (1988), 'Transferring Skills from Training to the Actual Work Situation: The Role of Task Application knowledge, Action Styles and Job Decision Latitude', Soloway, E., (ed), *CHI'88 Human Factors in Computing Systems*, Washington, New York, Addison-Wesley, 55-60.

Vessy, Iris & Weber, Ron (1986), 'Structured Tools and Conditional Logic: An Empirical Investigation', *Communications of the ACM*, 29(1), 48 - 57.

Victoria University of Technology (1993), *Code of Conduct for Research*, Office of Research, Melbourne, Australia.

Vitalari, N. & Dickson, G. (1983), 'Problem Solving for Effective System Analysis: An Experimental Exploration', *Communications of the ACM*, 26(11), 948-956.

Wallingford, Eugene (1996), 'Toward a First Course Based on Object-Oriented Patterns', *in* Klee, Karl J., (ed), *Twenty-Seventh SIGCSE Technical Symposium on Computer Science Education*, Philadelphia, Pennsylvania, ACM Press, 27 - 31.

Walsham, G. (1993), *Interpreting Information Systems in Organizations*, John Wiley and Sons, Chichester.

Waters, C. (1993), 'Where do we GOTO now?', *Informatics*, April, 1993, 47.

Wick, Michael (1995), 'On Using C++ and Object-Orientation in CS1: The Message is Still More Important than the Medium', in White, Curt, (ed), *Twnety-Sixth SIGCSE Technical Symposium on Computer Science Education*, Nashville, Tennessee, ACM, 322 - 326.

Wiedenbeck, Susan (1991), 'The Initial Stage of Program Comprehension', *International Journal of Man-Machine Studies*, 35, 517 - 540.

Wilkes, M. (1991), 'Software and the Programmer', *Communications of the ACM*, 34(5), 23-24.

Winograd, Terry (1995), 'From Programming Environments to Environments for Designing', *Communications of the ACM*, 38(6 (June)), 65 - 74.

Wolcott, Harry F (1990), *Writing Up Qualitative Research*, Sage Publications, Newbury Park, California.

Yin, R. (1989), *Case Study Research: Design and Methods.*, SAGE Publications, Inc., Newbury Park, CA.

Yourdon, Edward (1994), *Object-Oriented Systems Design*, Prentice Hall, Englewood Cliffs, New Jersey.

Yourdon, E. & Constantine, L. (1979), *Structured Design: Fundamental of a Discipline of Computer Program and Systems Design*, Prentice Hall, Englewood Cliffs, N.J.

Zikmund, W. (1991), *Business Research Methods*, The Dryden Press, Orlando, Florida, U.S.A.

Appendix 1 Student Interview Sample Program

A company requires a program to calculate the cost of sending a parcel by mail. Details of the parcel are to be inputted (Name, Address, the Weight of the Parcel and whether the parcel is to be registered or not (ie. insured during delivery). The charge for sending the parcel depends on the weight of the parcel and whether it is registered or not. The following table represents the cost of sending an unregistered parcel based upon the weight:

Weight of Parcel	Cost
less than 100 grams	$0.45
100 grams to 1 kilogram	$5.30
above 1 kilogram	$7.50

If a parcel is to be registered the cost of sending the parcel is double the unregistered cost as given in the table above. As an example, an unregistered 500 gram parcel would cost $5.30 to deliver, a 500 gram registered parcel would cost $10.60 to send.

The user should be able to print a label from the system which can be stuck onto the parcel showing the cost and registered status.

Please solve the problem showing any documentation, screens and coding you would use. Try to describe what you are doing as you go.

How would you test your solution and at what stages would you test?

Appendix 2 Background of Interviewed Students

Pseudonym	Age	Sex	Family Nationality	Course	Enrolment Status	Years in Course	Occupation	Previous Qualifications	Previous Programming Experience
Billy	20	M	Turkish	BBUS (Computing)	Full-time	3	Student	None	Pascal (repeating subject from last semester)
Chigden	19	F	Turkish	BBUS (Computing)	Full-time	2	Student	None	None
Clinton	19	M	Australian	BBUS (Computing)	Full-time	2	Student	None	None
Dianne	31	F	Italian	Graduate Diploma of Business Computing	Part-time	1	Housewife	B.Arts	None
Fred	43	M	Australian	BBUS (Accounting)	Part-time	3	Accounting Systems Consultant	Advanced Certificate of Management (TAFE)	DataFlex Assembler
Georgina	24	F	Kenyan	BBUS (Computing)	Full-time	2	Student	None	Dbase III+
Gerald	26	M	English	Graduate Diploma of Business Computing	Part-time	1	Classical Dance Instructor	B. English Literature	None

Jaranka	19	F	Yugoslav	BBUS (Computing)	Full-time	2	Student	None	None
Joanne	19	F	Australian	BBUS (Computing)	Full-time	2	Student	None	Pascal (repeating subject from last semester)
Mark	20	M	Australian	BBUS (Computing)	Full-time	1	Student	Advanced Certificate of Accounting (TAFE) Associated Diploma of Accounting (TAFE)	None
Mary-Beth	31	F	Australian	BArts/Business Computing	Full-time	1	Student/Nurse	Diploma of Nursing	None
Peter	39	M	Australian	Graduate Diploma of Business Computing	Part-time	1	Physical Education Teacher	B. Education	None
Seana	19	F	Irish	BBUS (Computing)	Full-time	2	Student	None	None
Sylvia	25	F	Australian	Graduate Diploma of Business Computing	Part-time	1	Insurance Claims Officer	BBUS (Accounting)	None
Tom	20	M	Chinese	BBUS (Computing)	Full-time	2	Student	None	Pascal and DBaseIII+ in VCE

Appendix 3 Student Consent Form

Victoria University of Technology

Dear Student,

I am currently interviewing Bachelor of Business and Graduate Diploma of Business Computing students about their attitudes towards, and experience of, Visual Basic. I am undertaking these interviews as part of a study to determine the appropriateness of Visual Basic as a first year programming language. Information gained will be used to improve the teaching of the language and the development of skills in the introductory programming subject.

If you agree to be interviewed, your interview will be treated with strict confidentiality. Any reference to your interview in the write up of the study will have any identifying data removed. Any reference to you, and any individuals to whom you refer, will be given pseudonyms.

I would like to stress that your participation, or non participation, in these interviews will in no way affect the results of any of your subjects in the Bachelor of Business or Graduate Diploma of Business Computing.

Thank you for your assistance.

Peter Shackleton

I, ...

of ....

agree to be interviewed for the research on:

'Event-Driven Programming for Novice Programmers'

being conducted by Peter Shackleton at Victoria University of Technology.

Signed: .... Date: ....