# A Comparison of Programming Notations for a Tertiary Level Introductory Programming Course

**Charmain Barbara Cilliers**

Submitted in fulfilment of the requirements for the degree of
Philosophiae Doctor in the Faculty of Science at the
University of Port Elizabeth

September 2004

**Promoter**: Prof A. P. Calitz
**Co-promoter**: Prof J. H. Greyling

UPE

Department of Computer Science and Information Systems

*Trust in the Lord with all your heart. Never rely on what you think you know.*
*Remember the Lord in everything you do and He will show you the right way.*

*Proverbs 3: 5 – 6*

# Acknowledgements

The completion of this thesis would not have been possible without the invaluable support, assistance and guidance of a number of individuals over the past few years.

Professors André Calitz and Jéan Greyling, my promoters in the Department of Computer Science and Information Systems (CS/IS) at the University of Port Elizabeth (UPE), whose continued guidance and encouragement is much appreciated. Their experience as well as their never-ending interest and enthusiasm in the research were integral to its completion.

David Brown, Justin Thomas and Allan Yeh, exceptional Honours students in the Department of CS/IS at UPE, for developing the B# programming notation and development environment included as the core instrument in the empirical study conducted.

Mr Danie Venter, lecturer in the Department of Mathematical Statistics at UPE, for expert advice and confirmation of the appropriateness of the quantitative statistical analysis techniques applied.

James Smith and Greg Saunders from the Computer Centre at UPE for assistance in the extraction of historical student mark data.

Each and every one of my colleagues in the Department of CS/IS at UPE, especially Dieter Vogts, MC du Plessis and Marinda Taljaard, introductory programming course lecturers for their continued encouragement and acceptance of the conducting of the experiment with their students; as well as Lynette Barnard and Janine Nel for assistance in lecturing load during these past few months.

The 2003 introductory programming students for being such willing and co-operative participants in the investigative study.

The National Research Foundation (NRF) and UPE for financial assistance.

My parents for standing in for mom’s taxi and as supporters of my sons’ in their respective sports team matches these past 2 years.

My husband, Pieter, for support, understanding and encouragement as well as handling family catering arrangements so as to prevent another burnt meal; and to my young sons, Jacques and Pierre, for letting Mom get on with her “thing”.

This material is based upon work supported by the National Research Foundation (NRF) under Grant number 2054293. Any opinion, findings and conclusions or recommendations expressed in this material are those of the author and therefore the NRF does not accept any liability in regard thereto.

# Summary

Increasing pressure from national government to improve throughput at South African tertiary education institutions presents challenges to educators of introductory programming courses. In response, educators must adopt effective methods and strategies that encourage novice programmers to be successful in such courses. An approach that seeks to increase and maintain satisfactory throughput is the modification of the teaching model in these courses by adjusting presentation techniques.

This thesis investigates the effect of integrating an experimental iconic programming notation and associated development environment with existing conventional textual technological support in the teaching model of a tertiary level introductory programming course. The investigation compares the performance achievement of novice programmers using only conventional textual technological support with that of novice programmers using the integrated iconic and conventional textual technological support.

In preparation for the investigation, interpretation of existing knowledge on the behaviour of novice programmers while learning to program results in a novel framework of eight novice programmer requirements for technological support in an introductory programming course. This framework is applied in the examination of existing categories of technological support as well as in the design of new technological support for novice programmers learning to program. It thus provides information for the selection of existing and the design of new introductory programming technological support.

The findings of the investigation suggest strong evidence that performance achievement of novice programmers in a tertiary level introductory programming course improves significantly with the inclusion of iconic technological support in the teaching model. The benefits are particularly evident in the portion of the novice programmer population who have been identified as being at risk of being successful in the course. Novice programmers identified as being at risk perform substantially better when using iconic technological support concurrently with conventional textual technological support than their equals who use only the latter form. Considerably more at risk novice programmers using the integrated form of technological support are in fact successful in the introductory programming course when compared with their counterparts who use conventional textual technological support only.

The contributions of this thesis address deficiencies existing in current documented research. These contributions are primarily apparent in a number of distinct areas, namely:

- formalisation of a novel framework of novice programmer requirements for technological support in an introductory programming course;
- application of the framework as a formal evaluation technique;
- application of the framework in the design of a visual iconic programming notation and development environment;
- enhancement of existing empirical evidence and experimental research methodology typically applied to studies in programming; as well as
- a proposal for a modified introductory programming course teaching model.

The thesis has effectively applied substantial existing research on the cognitive model of the novice programmer as well as that on experimental technological support. The increase of throughput to a recommended rate of 75% in the tertiary level introductory programming course at the University of Port Elizabeth is attributed solely to the incorporation of iconic technological support in the teaching model of the course.

**Keywords**: Novice Programmer, Introductory Programming, Iconic Programming, Visual Programming, Programming Notation, Technological Support

# Table of Contents

# List of Figures

# List of Tables

# List of Definitions

# Chapter 1

# Research Context

## 1.1 Introduction

Transformations in the South African political and educational scenario over the past few years have resulted in increasing pressure from national government to improve student throughput rates at tertiary institutions (Department of Education 2001), the University of Port Elizabeth (UPE) being one example of such an institution. At UPE, specifically, the minimum pass rate for a course in an academic Department in the Faculty of Science as well as other faculties is recommended to be 75% (UPE 2002; Wesson 2002). The problem of sustaining recommended satisfactory throughput rates in tertiary level courses is compounded by the fact that currently increasingly more under-prepared students are entering South African tertiary education institutions (Warren 2001; Monare 2004).

The higher incidence of under-prepared students in South African tertiary education institutions has a particular significance for introductory programming courses which rely heavily on the use of technological tools as components of the teaching model[1]. As a direct consequence of the resulting diversity of the introductory programming course student population, an important factor that cannot be ignored is the general lack of exposure and access to technology, specifically computers, in some groups of the student population (Loxton 1997). While some introductory programming

[1] Includes learning resources, learning activities and learning supports.

students have had limited or no prior exposure to computers, others have Computer Studies as a secondary education subject credit.

Recent research in the Department of Computer Science and Information Systems (CS/IS) at UPE shows that certain student groups originate from environments where there is limited or no exposure to electrical equipment such as video recorders or microwave ovens (Streicher 2003). The higher incidence of technologically under-prepared students in introductory programming courses consequently impacts on the group profile of the students and overall throughput rate of these courses.

Maintaining satisfactory group and individual performance rates in introductory programming courses is not constrained to South African tertiary education institutions. The sustaining of acceptable levels of performance remains an issue that is constantly being addressed by tertiary education institutions worldwide (Lister & Leaney 2003). Acknowledged as being of great importance in efforts to elevate the throughput rate in an introductory programming course at tertiary level are effective methods and strategies that assist novice programmers to overcome difficulties associated with computer programming (Carbone *et al.* 2001).

Typical difficulties (Section 1.3.2) experienced by novice programmers in introductory programming courses include deficiencies in problem-solving strategies (AC Nielsen Research Services 2000), misconceptions related to programming language constructs (Studer *et al.* 1995; Proulx *et al.* 1996; Deek 1999; McCracken *et al.* 2001), the use of traditional textual programming notations and their associated environments (Satratzemi *et al.* 2001) and individual form of motivation (Jenkins 2001a; Chamillard *et al.* 2002; Jenkins 2002). Attempts to address these difficulties have resulted in increased demands on lecturing and computing resources. As a consequence of the resulting situation as well as high failure and attrition rates among students, there exists an urgent need for sound academic methods to raise the successful completion percentage of candidates of already over-subscribed introductory programming courses without reducing the quality of the course (UCAS 2000; Boyle *et al.* 2002).

The approaches to this problem involve either the identification of potentially successful introductory programming candidates or the modification of the introductory programming course teaching model (Wilson & Braun 1985; Austin 1987). The former approach involves the discovery of factors that could predict success in CS/IS and thereby provide a mechanism for the selection of introductory programming course candidates from applicants most likely to succeed. This type of approach has prompted numerous international research projects over the past two decades (Fowler & Glorfield 1981; Butcher & Muth 1985; Koubek *et al.* 1985; Sauter 1986; Carbone *et al.* 2001). As a result of the findings, related research concerned specifically with the satisfactory progress of students in introductory computer programming courses has been ongoing for many years internationally (Calloni & Bagert 1997; Astrachan 1998; Byrne & Lyons 2001; Carter & Jenkins 2001; Wilson & Shrock 2001; Chamillard *et al.* 2002) and nationally (Mostert 2002; Misthry *et al.* 2003; Naudé & Hörne 2003).

Specifically at UPE, the selection of first year CS/IS students has been an ongoing research project since 1982 (Calitz 1984; Calitz *et al.* 1992; Calitz 1997). The main emphasis of these earlier research projects was on the identification of matriculation results and the use of psychometric variables on predicting success in CS/IS (Calitz 1997). Since the completion of Calitz's research, changes in the South African political and educational scenario necessitated the need for further research. Subsequent research conducted by Greyling (2000) formed an integral part of the work done by the UPE Admissions and Placement Team (Foxcroft 1997), whose primary task was to implement and monitor a series of computerised tertiary selection and placement assessments. The recommendations led to the implementation of a selection and placement model in the Department of CS/IS at UPE in 2001.

The second type of approach that seeks to increase the throughput rate of introductory programming students involves the modification of course presentation techniques to support high risk students, being those students who have been identified as being of low-ability in an introductory programming course. This type of approach is the primary factor that prompted the current study. Consequently, the goal of the current investigation is to establish and assess the effects of an iconic programming notation in the role of a development environment for an introductory programming course.

This chapter provides an outline of the international and national context that prompted the current investigation (Section 1.2). It highlights UPE's context and prior research with respect to the first of the strategies identified as being an approach to the problem of increasing the throughput rate of introductory programming students. An overview on the issues that contribute to the overall relevance of the current research follows (Section 1.3).

The first issue identified is the comprehensive study and limited success observed in the implementation of an acknowledged selection and placement model for novice programmers in a tertiary level introductory programming course at UPE (Section 1.3.1). The second issue involves the identification of factors that contribute to the acknowledgement that the task of programming is a difficult skill for novice programmers (Section 1.3.2).

Based on the issues raised, a number of research questions are identified for investigation (Section 1.4) with the core focus of the investigation being the determination and measurement of the impact of a locally developed iconic programming notation and development environment, B#[2] (Brown 2001a, b; Thomas 2002a, b; Yeh 2003a, b), on novice programmers in an introductory programming course at tertiary level.

## 1.2 Background and Prior Research

Since the early 1980's both international and national higher education institutions have experienced a phenomenal increase in their enrolment for CS/IS programmes (Konvalina *et al.* 1983; Butcher *et al.* 1985; UCAS 2000). A similar pattern (Figure 1.1) is evident in the UPE first year CS/IS enrolment for the introductory programming course (UPE 2003a). At UPE, the relationship of the enrolment figures in the introductory programming course in the previous two decades shows an increase from an average of 188 students in the 1980's to an average of 278 students in the 1990's, an increase of 48%.

[2] Name selected by initial developer (Brown 2001a). The name has no relationship with any existing programming notation or development environment. No connotation with any existing programming notation or development environment is intended.

Students have been motivated and encouraged to enrol for CS/IS courses with the inception of personal computers and the introduction of end-user software like word processing and spreadsheet packages (Downes 2002; Jenkins 2002). High-paying computer-related careers and a world wide shortage of Information Technology (IT) professionals at the time (Downes 2002) provided further motivation[3]. The explosion of the World Wide Web (WWW) and the Internet during the 1990's was also a stimulus for increased interest in CS/IS, relating in a further growth in enrolments at departments of CS/IS (Denning 1996). These factors, together with changing requirements in degree programmes and implementation of placement strategies at UPE resulted in an average of 352 students enrolling in the introductory programming course at UPE in the current decade up to the end of 2002 (UPE 2003a), being an increase of 27% on the average of the previous decade.

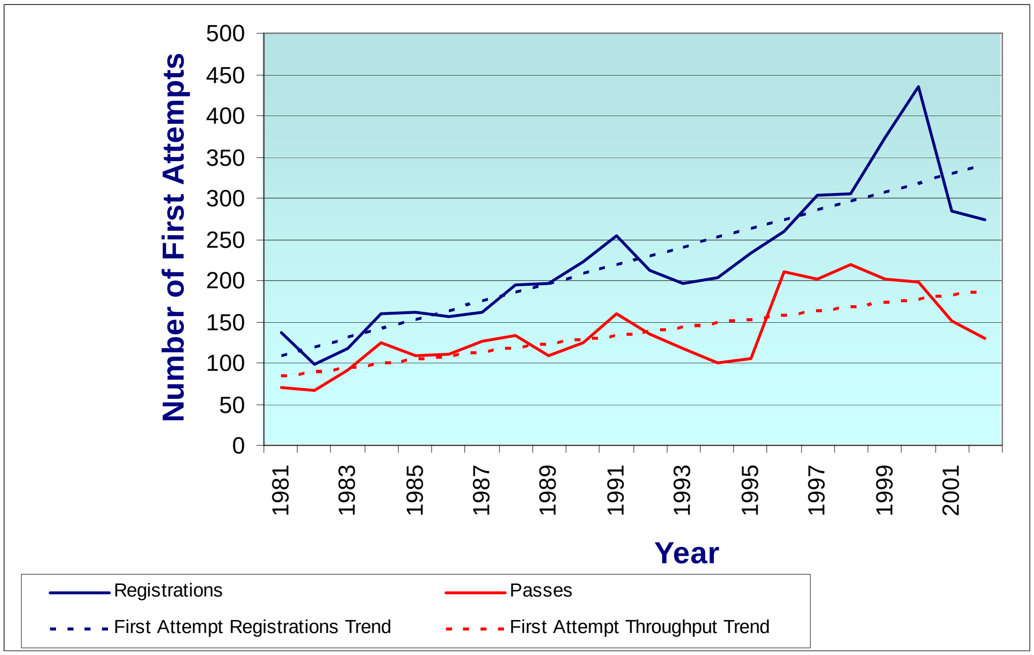


*Figure 1.1: Registrations and successful completion tendency*

Despite the continuous development of new technologies, a large percentage of introductory programming students over the past few decades have consistently found the challenge of computer programming more difficult than initially anticipated (Konvalina *et al.* 1983; Calitz 1997; Lister 2000; Carbone *et al.* 2001; Boyle *et al.* 2002; Thomas *et al.* 2002; Garner 2003). Figure 1.1 illustrates the successful

[3] The shortage of IT professionals is currently not as critical as in the recent past.

throughput trend in relation to that of number of initial first attempt registrations in an introductory programming course at UPE. It is evident that the distance between the number of initial registrations and the number of successful students in the introductory programming course at UPE has been growing steadily over the past two decades. Specifically, the pass rate of first year CS/IS enrolments for the introductory programming course steadily decreased from an average of 68% in the 1980's, to an average of 61% in the 1990's and an average of 49% in the current decade up to the end of 2002 (UPE 2003a).

In attempts to address the constantly declining pass rate of first year CS/IS students in the UPE introductory programming course over the past two decades, validated selection and placement strategies were implemented since the 1980's (Calitz 1984; Calitz *et al.* 1992; Calitz 1997; Greyling 2000; Greyling *et al.* 2002; Greyling & Calitz 2003). Even with the implementation of these strategies, numerous students who indicate a potential to be successful are in fact unsuccessful in the introductory programming course. This is evident from the fact that only 59% of the first year students predicted to be successful in the introductory programming course at UPE in 2002 were in fact successful on their first attempt (UPE 2003a). An enhanced strategy for improving the learning process of introductory programming students that has been confirmed in practice (CC 2001) is thus encouraged.

The goal of this investigation is to persist with research aimed at increasing the throughput rate of introductory programming students at UPE, particularly with respect to the modification of the introductory programming course teaching model. In particular, this investigation will determine the impact of a specialised technological tool used in an introductory programming course teaching model. The subjects of the study are novice programmers who have been identified as being potentially successful in an introductory programming course at tertiary level (Greyling 2000; Greyling *et al.* 2002; Greyling *et al.* 2003). Specifically, it will be investigated whether novice programmers at a tertiary level can benefit individually and as a group from exposure to an iconic programming notation in a development environment for an introductory programming course.

The current investigation follows on similar quantitative international research on the use of iconic programming notations that has been conducted during the mid to late 1990's (Calloni & Bagert 1994; Calloni *et al.* 1997)[4]. National research related to the study is restricted to the identification of potentially successful students (Mostert 2002; Nash 2003; Naudé *et al.* 2003), and specifically to the selection and placement of CS/IS students at UPE that has been conducted since the early 1980's (Calitz 1984; Calitz *et al.* 1992; Calitz 1997; Greyling 2000; Greyling *et al.* 2002; Greyling *et al.* 2003).

Since the completion of Greyling's initial research in 2000, research findings (Greyling *et al.* 2002; Greyling *et al.* 2003) on the implementation of a selection and placement model at tertiary level have prompted further research in the area of introductory programming courses at tertiary level. The specific issues that prompted the current research form the focus of the following section.

## 1.3 Relevance of the Investigation

Approaches to the problem of increasing the throughput in introductory programming courses involve either the identification of potentially successful introductory programming candidates or the modification of the introductory programming course teaching model (Wilson *et al.* 1985; Austin 1987). The former approach has been exhaustively researched and implemented over the past two decades in the Department of CS/IS at UPE[5].

Consequently, the main factor (Section 1.3.1) prompting the current investigation is that a limited improvement in the throughput of first year introductory programming students has been observed (Greyling *et al.* 2002; Greyling *et al.* 2003) at UPE upon the implementation of an approved selection and placement model since 2001 (Greyling 2000).

[4] No further documentary evidence of follow-up studies since 1997 has been located. In fact, it has been discovered that the author of the research, Ben A. Calloni, has diversified and is currently employed at a different institution to that where the empirical studies were originally conducted (Calloni 2002).

[5] (Calitz 1984; Calitz *et al.* 1992; Calitz 1997; Greyling 2000; Greyling *et al.* 2002; Greyling *et al.* 2003)

The other factor (Section 1.3.2) that has a bearing on the research is the observation that programming is especially difficult for novice programmers (Whitley 1997; Warren 2000; Lischner 2001). When programmers implement a solution, two areas of operation are prevalent, namely the problem and program domains (Mattson undated-a). The problem domain is that area where the problem is stated and defined. The program domain is that area containing the programmed solution. Novice programmers experience that a large amount of effort is required in converting a mental interpretation of the problem domain into a corresponding program solution representation in the program domain (Smith *et al.* 2000).

Novice programmers are furthermore faced with mastering the challenges of superficial and in-depth learning in the program domain (Mayer 1981; Perkins & Salomon 1988; Carter *et al.* 2001; Jenkins 2002). Superficial learning skills in programming include that of memorising the syntactical issues of a particular programming notation. In-depth learning skills necessitate the comprehension of programming constructs required for the future composition of novel program solutions. Novice programmers enrolled for introductory programming courses have to contend with deficiencies in their problem-solving strategies, misconceptions about programming notation constructs, the use of conventional textual programming notation development environments traditionally utilised for the teaching and learning of programming as well as individual type of motivation (Section 1.3.2).

The aforementioned factors have prompted exploration at UPE of the second approach to increase throughput, namely the modification of the teaching model for an introductory programming course. One approach to the modification of the teaching model for an introductory programming course is the variation of presentation techniques of learning resources by means of technological support. One such type of technological support is a class of programming notations known as visual programming notations, of which iconic programming notations and their associated development environments, the type of tool used in the current investigation, is a category. The selection of the type of technological tool used in this study is prompted by the features of the identified category (Chapter 3) as well as evidence of positive results of similar quantitative international research using

BACCII©[6], an iconic programming notation, which occurred in the mid to late 1990's (Calloni *et al.* 1994, 1997).

The following two sections in turn emphasise each of the two abovementioned issues that stress the relevance of the current investigation. Thereafter, a section consolidates, reviews and highlights the identified significant aspects.

### *1.3.1 Limited Improvement in Throughput upon Implementation of Selection and Placement Model*

The implementation of a selection and placement model at UPE resulted in a limited improvement in the performance of introductory programming students. Greyling *et al.*'s (2002; 2003) observations after the 2001 implementation of a computerised selection and placement model for CS/IS students at UPE recorded an increase in group performance of introductory programming students. Results showed an increase of 17% from that of the previous year in the pass rate of introductory programming students.

An individual average performance achievement increase of 8% – 19% on the predicted marks for students was also observed for those students streamed into an alternative slower paced introductory programming course. Despite these positive observations, the pass rate for the introductory programming course at UPE remained below the recommended throughput rate of 75% (UPE 2003a).

The burden is on the educators of tertiary level courses with relatively low pass rates, specifically introductory programming courses, to provide instruments that will improve the situation without the lowering of standards. Further research in the area of group and individual performance with respect to the teaching model applied in an introductory programming course has thus been provoked. The research must take into account the specific issues that novice programmers have to contend with when learning to program. These issues are elaborated on in the following section.

[6] Pronounced ba-chee.

### *1.3.2 Difficulty of Programming Task for Novice Programmers*

Any academically competent student is thought to be able to program due to the universality of computing in education and the advent of user-friendly interfaces, (Byrne *et al.* 2001). However, tertiary level students who are proficient in other tertiary level courses may fail to achieve success in programming (Bonar & Soloway 1983; Proulx 2000; Byrne *et al.* 2001; McCracken *et al.* 2001). Tertiary educators who hope to teach programming effectively need to understand precisely what makes learning to program difficult for so many students (Jenkins 2002) and adjust their teaching techniques to support these students when required (Byrne *et al.* 2001; Thomas *et al.* 2002).

Computer programming is not an easy task (Bonar *et al.* 1983; Warren 2000) and has been defined as a difficult form of problem-solving (Whitley 1997). By their nature, problem-solving intensive courses require the ability to take a vague problem description that is stated in the problem domain and construct a well designed solution (Lister *et al.* 2003) in the program domain. Problem-solving intensive courses consequently require a considerable amount of effort in several skills. In the case of programming, these skills include the simultaneous mastering of various programming language constructs at both the superficial and in-depth levels of learning (Studer *et al.* 1995; Proulx *et al.* 1996; Carter *et al.* 2001; Jenkins 2002). The programming notation constructs are specifically syntax, semantics, structure and style. The mastering thereof is required for the solving of problems using a particular programming notation within a given programming development environment.

There is a need for an introductory programming teaching model that incorporates tools to simplify programming language syntax, semantics, structure and style. Any tool used in this teaching model should also serve to minimise the errors produced by novice programmers as a result of erroneous interpretation of programming language constructs.

Programming represents a formal task of precision (Koelma *et al.* 1992; Lischner 2001; Pane *et al.* 2001). Consequently, novice programmers in introductory programming courses can only achieve success by applying a very high level of precision to their tasks (Jenkins 2002). This level of precision may represent a much higher level than that required of most other tertiary level courses (Jenkins 2002). Teaching model support in achieving and maintaining this level of precision with respect to the application of **problem-solving strategies**, **comprehension of programming constructs** and **use of conventional development environments** can impact on the **motivation** of the novice programmer, and ultimately on performance achievement. Each of these aspects is overviewed in the following four subsections.

### Deficiencies in Problem-Solving Strategies

In most efforts to teach programming, the focus tends to be on the cultivation of programming skills (Perkins *et al.* 1988; Lister 2000) and algorithmic problem-solving (Thomas *et al.* 2002). This approach is the one currently in practice in the Department of CS/IS at UPE, specifically using a semiotic teaching sequence, requiring that syntactical knowledge (superficial learning level) of a specific programming construct be mastered prior to the teaching of the semantics and pragmatics of the same programming construct (in-depth learning level) (Kaasbøll 1998). The result of this approach is often the over simplification of the programming process with too little emphasis on the problem-solving steps of analysis, design and testing (CC 2001). Novice programmers often then experience a restricted sense of discipline in the programming process (CC 2001), and find it difficult to adapt to different kinds of problems and problem-solving contexts (AC Nielsen Research Services 2000).

Supplemental to the teaching approach, educators should strive to improve the novice programmers’ performance by teaching them strategies for putting pieces of program code together, thereby reinforcing in-depth learning (Spohrer & Soloway 1986). This process supports the problem-solving process by encouraging the identification and mapping of known solutions to novel problems. Further, introductory programming course educators should promote a primary educational objective while teaching an introductory programming course, namely that novice programmers should learn to perform structural decomposition in their approach to solving problems (Cockburn & Churcher 1997).

During problem-solving, the difference in the way programmers comprehend a problem and the way the solution must be expressed in a specific programming notation makes it difficult for novices to learn how to program. A large part of any programming task is to transform a mental plan for solving a problem into the program solution representation for the particular programming notation being used. Ideally, the effort involved in the conversion of the plans to the program solution representation should be minimal (Pane *et al.* 2001), but in reality for novice programmers it is vast (Smith *et al.* 2000).

Any small cognitive transformation required, especially by a novice programmer, detracts effort from the intended task, namely implementing a solution to a given problem. Many novice programmers program experimentally by randomly including programming constructs to determine whether the solution will work (Buck & Stucki 2001). They thus tackle their assignments head-on with ad-hoc development strategies despite training and guidance to the contrary (Cockburn *et al.* 1997).

There are two ways in which to assist a novice programmer reduce the effort involved in the conversion of programming plans to implemented solutions, namely move the novice programmer nearer to the program domain or move the program domain nearer to the novice programmer (Smith *et al.* 2000). Traditional lectures in introductory programming courses attempt the former method by providing formal instruction of the program domain. The latter method could be realised by addressing the **programming notation** together with its associated **development environment**. These topics form the core of the discussions of the following two subsections.

Comprehension of Programming Notation Constructs

Introductory programming course educators agree that the main purpose of an introductory programming course is to teach novice programmers to program while using a specific programming notation as an implementation tool (Lockard 1986; Kushan 1994; Studer *et al.* 1995; Proulx *et al.* 1996; Dingle & Zander 2001; McCracken *et al.* 2001; Jenkins 2002). It is thus not the intention of such educators to teach a particular programming notation. Novice programmers should merely be taught the mechanics of the programming notation so that they will be able to create and manipulate the data types and control flow mechanisms, amongst other programming concepts, in order to solve the program domain requirements of their problem domain solution (Cockburn *et al.* 1997).

Novice programmers can thus be expected to practice the use of the programming notation by undertaking practical assignments. However, when learning to program, novice programmers are faced with misconceptions about the syntax, semantics and pragmatics of programming notation constructs (Studer *et al.* 1995; Proulx *et al.* 1996; Deek 1999; McCracken *et al.* 2001; Satratzemi *et al.* 2001).

A large share of the literature on computer science education has been dedicated to debates on the most appropriate programming notation and associated development environment as well as the programming paradigm to use in order to teach novice programmers (McIver 2000; Pane *et al.* 2001; Warren 2001; De Raadt *et al.* 2002; Howell 2003). Under discussion is the trade-off between choosing a programming notation for its educational suitability versus the extent and acceptance of its use in industry. Potential employers of IT graduates have a clear expectation of expertise following an introductory programming course, hence the demand from industry to use a programming notation and development environment with a significant market share (Dingle *et al.* 2001).

When making a choice of programming notation for use by novice programmers, consideration must be given to how easily the novices will learn the chosen notation, the existence of any notation features that might interfere with the understanding of the fundamental programming concepts, as well as any notation features that ease the transformation of the novice programmer to one who is competent (Dingle *et al.* 2001). Further, it is maintained that the programming notation choice for an introductory programming course can influence learning in subsequent computer programming courses (Applin 2001).

Teaching a particular programming notation should not replace teaching the core concepts of programming (Lockard 1986; Kushan 1994; Studer *et al.* 1995; Proulx *et al.* 1996; Dingle *et al.* 2001; Jenkins 2002). Many current programming notations were not designed for teaching (Ziegler & Crews 1999; McIver 2001; Jenkins 2002).

The predominantly conventional commercial textual programming notations do not accurately reflect the cognitive strategies used by novice programmers (Bonar *et al.* 1983). They require the programmer to make large transformations from the intended tasks to implementation in the textual program solution representation (Pane *et al.* 2001). The goal of an appropriate programming notation should be to apply cognitive directness, thereby minimising the mental transformations that a programmer must make and so ensure that the conceptual distance between the novice's mental plan of a solution and one compatible with the computer is as small as possible.

In a recent study of Pane *et al*. (2001) investigating the notation and structure of non-programmers' solutions to programming problems, many places were identified as exhibiting unnecessary large gaps which are imposed by the features and requirements of current programming notations. This study concluded that textual programming notations do not support the goal of applying cognitive directness. The effect is that when novice programmers are confused, they attempt to transfer their knowledge of natural language notation to the programming task at hand (Bonar *et al.* 1983; Pane *et al.* 2001).

In natural language solutions, there exists a large amount of imprecision and under-specification, which is contradictory with the formality evident in the task of programming (Koelma *et al.* 1992; Lischner 2001; Pane *et al.* 2001). It is therefore important to find ways to help novices make their specifications more complete so that they may become effective programmers (Pane *et al.* 2001). Correct solutions in natural language tend to appear in a different style to those required by conventional textual programming notations. The conclusion is that programming notations that are usually designed for use by professional programmers are thus clearly not suitable for novice programmers in that they pose an extra burden on the novices (Ziegler *et al.* 1999; Satratzemi *et al.* 2001; Warren 2001; Jenkins 2002).

Programming may consequently seem more difficult than it actually is because it requires solutions to be expressed in ways that are not familiar or natural for novice programmers, who are more comfortable with concrete than abstract thinking (Pane *et al.* 2001). There is consequently a need for novice programmers to learn how to design, develop, verify and debug a program when given certain programming notations and development environments as technological support in a learning environment (Satratzemi *et al.* 2001).

Concerns exist as to the cost/benefit ratios when using technological support in learning environments, specifically regarding the maintenance of the balance between learning about the learning support software and learning about the course content by means of the supporting software (Rader *et al.* 1998). It has been observed, therefore, that implementation issues in traditional textual programming notation development environments can distract novice programmers so that they do not comprehend the programming abstractions required for the correct implementation of program solutions (Reek 1995; Lidtke & Zhou 1998; Ziegler *et al.* 1999; Proulx 2000; Warren 2000, 2001).

Use of a Traditional Programming Development Environment

The most common technological learning environment used by novice programmers in introductory programming courses is a commercial textual programming notation and its associated development environment (De Raadt *et al.* 2002; Reid 2002). Even though these conventional textual programming environments exhibit the characteristic of concurrently displaying many programming constructs on the screen, they tend to under-determine the novice programmer by providing no constraints on the textual symbols that can be entered (LaLiberte 1994). The result is a large amount of effort being required to accurately transform the mental model of the desired program solution with that of the supported programming notation (Wright & Cockburn 2000). The novice programmer is forced to provide precisely correct syntax before receiving any response to the solution plan and implementation thereof (Crews & Ziegler 1998). This situation results in an overload at the superficial level of learning, indirectly impacting on the effectiveness of the in-depth learning.

Further, the lack of sufficient visual feedback in the use of such technological support makes the comprehension of programming notation semantics more difficult for a novice programmer (Satratzemi *et al.* 2001). This situation results in limited support at the in-depth level of learning.

Research has also indicated that traditional programming development environments do not meet their goals as technological support for novice programmers and are not suited to the types of problems experienced by novice programmers. These challenges are specifically deficiencies in problem-solving strategies (AC Nielsen Research Services 2000) and misconceptions related to programming notation constructs, both having been discussed in detail earlier in this section (Studer *et al.* 1995; Proulx *et al.* 1996; Deek 1999; McCracken *et al.* 2001).

Features of conventional textual programming development environments include complex hierarchical menu structures and intricate user interfaces. These properties are often experienced by novice programmers as distractions from the task of programming (Reek 1995; Lidtke *et al.* 1998; Ziegler *et al.* 1999; Proulx 2000; Warren 2000, 2001). Consequently, the difficulties experienced by novice programmers when using conventional programming development environments can impact on the level of motivation, which is the issue under discussion in the following subsection.

Form of Motivation

In addition to the previously discussed points of deficiencies in problem-solving strategies, comprehension of programming constructs and use of conventional development environments, the individual performance of novice programmers in introductory programming courses can also be attributed to personal motivation. Consequently, a major challenge for educators of introductory programming courses is finding ways to motivate CS/IS students to excel and to help them enjoy the course.

As the student enrolment in an introductory programming course becomes more diverse, so too does students' motivation for taking the course. Anecdotal evidence from several international tertiary institutions is that students are becoming more tactical, are extrinsically motivated and, consequently will only engage in those learning activities which they see as contributing to an eventual highly paid job (Jenkins 2001a).

Increasing literature exists on innovative techniques that would enhance introductory programming courses and improve student motivation. Much of this literature discusses innovative ideas including the use of visual props (Astrachan 1998), theatre and singing (Siegel 1999). Even though there is evidence from novice programmer feedback that they enjoy the innovative modes of delivery of the introductory programming course material, it has not been shown that these methods of teaching have any real impact on the learning process required by novice programmers when learning to program (Jenkins 2002).

Students should be motivated to expect to succeed in their studies (Jenkins 2001a), and they should value the eventual outcome, examples being individual success in the course with respect to performance level, final programme qualification or the possession of a qualification that guarantees employability. If students are not able to appreciate these expectations of success and outcome value, they will become discouraged and consequently will not learn. Thus, educators need to address the issue of motivation in order to promote a better, more effective learning environment.

The motivation problem appears to be exacerbated in courses containing both computer science majors and non-majors (Chamillard *et al.* 2002). The introductory programming course in the Department of CS/IS at UPE is such a combined course. In the academic year (2003) during which the investigation is administered, the combined introductory programming course recorded an initial intake of 39% (*n* = 153) CS/IS majors and 61% (*n* = 239) non-majors (UPE 2003a).

Knowledge of factors influencing performance such as the source of motivation, could be useful in supporting students who enter combined introductory programming courses with an alleged weakness with respect to successful performance (Byrne *et al.* 2001). Researched information in this area could be useful in amending teaching techniques of introductory programming course educators so that they reflect the disparities in performance and motivation present in any particular group of students. Any methods used to provide the motivation necessary to increase performance and foster enjoyment in an introductory programming course must be selected and implemented with much thought so that novice programmers at the start of their formal programming education still learn the foundational programming constructs required (Chamillard *et al.* 2002).

### *1.3.3 Concluding Remarks on Relevant Factors*

The approach of teaching core programming concepts instead of a particular programming notation and associated development environment has been discussed in the previous section to address novice programmer deficiencies with respect to problem-solving strategies, misconceptions related to programming notation constructs and the use of traditional development environments. This approach has been implemented so widely over the past 2 decades[7] that it can be assumed that it has been evaluated as successful. However, from a recent multi-national multi-institutional report, it is evident that many novice programmers do not know how to program at the successful conclusion of their introductory programming courses (McCracken *et al.* 2001). This observation supports a conclusion that the educational approach described may no longer be as successful as previously established (Jenkins 2001a).

A recent report on the programming notations and development environments used by 37 Australian universities in introductory programming courses concluded that there is clearly a lack of technological support that is designed specifically for novice programmers, is freely available, easy to use, does not obscure the details of the programming process, and in which educators can be confident in teaching (De Raadt *et al.* 2002).

In related work that compares textual and graphical notations in programming notations, it has been shown that graphical notations are not necessarily an ultimate solution (Green *et al.* 1991; Moher *et al.* 1993). This observation is made despite claims of graphical notations providing an outline of the program structure, supplying more information and being easier to read. It follows that the crucial issues about any programming notation and in the case of this research, specifically a graphical programming notation, are to what degree does the programming notation make

[7] (Lockard 1986; Kushan 1994; Studer *et al.* 1995; Proulx *et al.* 1996; Dingle *et al.* 2001; Jenkins 2001a; McCracken *et al.* 2001; Jenkins 2002)

certain types of information available to the novice programmer, and does the novice programmer have sufficient experience with the type of programming notation being presented (Ramalingam & Wiedenbeck 1997). Unfortunately there is currently both insufficient research and empirical evidence, both internationally and nationally, to justify the use of non-conventional programming technological support, specifically non-textual programming notations like iconic programming notations and their associated development environments in the education and training of novice programmers (Whitley 1997).

It can be concluded from the discussions in Sections 1.3.1 and 1.3.2 that the following problems contribute to the relevance of the present research in introductory programming courses for novice programmers at tertiary level:

- continued unsatisfactory pass rates, despite the implementation of remedial strategies of selection and placement of students;
- deficiencies in novice programmer problem-solving strategies;
- amount of conversion required by novice programmers between plan and implementation of solution;
- high level of precision required in the syntax, semantics, structure and style of textual programming notations used in learning environments;
- lack of visual feedback in introductory programming development environments used for teaching and learning; and
- increased enrolment for CS/IS introductory programming courses and the impact of the diverse student population on the motivation of students to be successful in these courses.

From the same discussion, it is furthermore apparent that in order to address the above mentioned problems, there is a requirement for appropriate technological support in terms of a programming notation that:

- enhances understanding of introductory programming for a recognised teaching model; and

- is specifically designed for novice programmers and motivates and encourages the novice to be successful in using it.

The identified problems and solution summarised above serve to provide a focus for the current investigation, the details of which are elaborated on in the following section.

## 1.4 Focus of the Investigation

The current investigation focuses on an implementation of the second type of approach that has been proposed as a solution to increase the successful throughput of students in introductory programming courses at tertiary level. This approach is namely the modification of the teaching model (Wilson *et al.* 1985; Austin 1987). This investigation is a natural extension of the comprehensive studies already conducted at UPE with respect to the first of the approaches, namely the identification, selection and placement of potentially successful students (Calitz 1984; Calitz *et al.* 1992; Calitz 1997; Greyling 2000; Greyling *et al.* 2002; Greyling *et al.* 2003). It is also an extension of related international studies in programming with respect to the second type of approach (Calloni *et al.* 1994, 1997; Ramalingam *et al.* 1997; Crews 2001).

During the 1990's, quantitative international research in the use of an iconic programming notation in an introductory programming course at tertiary level was conducted (Calloni *et al.* 1994, 1997). The study by Calloni *et al*. indicated a significant improvement in the results obtained by novice programmers in an introductory programming course using BACCII$^{©}$, an iconic programming notation and development environment. It seems apt to perform a similar study on South African CS/IS students and thereby determine the suitability of an iconic programming notation as an alternative form of technological support for novice programmers in an introductory programming course at tertiary level within the South African context.

Furthermore, as a result of the continued increase in enrolment figures in CS/IS introductory programming courses, the continually growing diversity in first year introductory programming student populations (as discussed in Section 1.3) is of relevance when the issue of a technological support in the learning environment is considered. A cognitive model that provides insight into the process followed by novice programmers while learning to program (Mattson undated-a), can be used to compare different programming notations and their associated development environments more effectively.

Preferably, the programming notation used to educate and train novice programmers in an introductory programming course should be chosen for teaching and learning suitability and not because it is accepted and endorsed by industry (Jenkins 2002). The courses in which these notations are used should be designed to be flexible so as to allow different students to learn in different ways. This is supported by the results of a study that examined the correlations between preferred learning style of novice programmers and performance on both the exam and practical aspects of an introductory programming course (Thomas *et al.* 2002). The study observed that novice programmers with specific preferred learning characteristics[8] may be disadvantaged by certain traditional methods of teaching. These findings add to the present research's focus on the preferred teaching and learning programming notation in a development environment by also focussing on the issue of the use of a commercial textual programming notation and associated development environment as the accepted and prescribed technological support.

The focus of the investigation is highlighted in terms of the study's primary objectives (Section 1.4.1). One of the main objectives is the conducting of a comparative study using different categories of technological support as learning environment instruments for subjects in an introductory programming course (Section 1.4.2). The scope (Section 1.4.3) and feasibility (Section 1.4.4) of the research are presented, together with the specific research questions that are to be addressed by the current investigation (Section 1.4.5).

[8] Namely active (learning by trying), sensing (concrete learning), and visual (learning with diagrams) (Thomas *et al*. 2002; Soloman & Felder undated).

### *1.4.1 Goal and Objectives*

This thesis investigates and addresses the challenges of existing misconceptions about the syntax, semantics and pragmatics of programming notation constructs that are evident in novice programmers when using programming notations within the context of an introductory programming course teaching model. The thesis specifically reports on a quantitative and qualitative behavioural study of novice programmers in an introductory programming course. The behavioural study determines whether novice programmer performance achievement with respect to the comprehension and composition of program solutions is independent of the programming notation used as technological support in the learning environment for an introductory programming course at tertiary level.

The investigation encompasses a comprehensive study of the cognitive model of novice programmers while learning to program. The acknowledgement of such a cognitive model can assist in the design of programming notations and development environments that more closely match the way programmers, specifically novice programmers, think. Consequently, a framework of novice programmer requirements is proposed as the result of a comprehensive literature study of the cognitive model of such programmers.

The framework of novice programmer requirements is applied as the criteria against which programming notations related to that forming part of the current study are measured in support of suitability for novice programmers. Various distinct categories of experimental technological support for novice programmers are identified from existing research and evaluated according to these criteria. Furthermore, the framework of novice programmer requirements has a direct influence on the design and implementation of the locally developed visual iconic programming notation and development environment, B#. The role of B# in the current study is that of the treatment technological support in a between-groups empirical analysis.

The empirical analysis component of this investigation is conducted by qualitatively and quantitatively comparing two different programming notations and their associated development environments used as technological support in an introductory programming course learning environment in the Department of CS/IS at UPE during 2003. The present research is primarily concerned with the comparison of performance achievements observed for each of the identified technological supports. This research is partly in response to the observation of a continued unsatisfactory trend in the pass rate of introductory programming courses at tertiary level in South Africa (Naudé *et al.* 2003; UPE 2003a).

A further issue that impacts on introductory programming course throughput is the influence of a student's motivation when using specific technological support in the learning environment of an introductory programming course. If a student's preferred learning style matches that presented by a particular programming notation, the level of personal motivation might be more positive, thereby resulting in increased performance achievement and ultimately increased throughput (Thomas *et al.* 2002).

Although focussing on a visual iconic programming notation as the preferred technological learning environment in a CS/IS introductory programming course, this thesis also reports on whether a visual iconic programming notation is beneficial as a technological learning environment that is supplemental to that of a conventional textual programming notation. The analysis of the study is done in terms of maximising throughput as well as individual average performance achievement.

### *1.4.2 Technological Support and Subjects of Comparative Study*

The subjects for the research are first year introductory programming students in the Department of CS/IS at UPE. In the introductory programming course at UPE, students are traditionally taught the basic concepts of programming in a procedural, or imperative, fashion, using a conventional textual programming notation (PASCAL) and associated commercial development environment, Delphi™ Enterprise, as the technological support (CS&IS 2003).

The programming notations considered in the investigation are the aforementioned textual programming notation and development environment (discussed further in

Section 5.4.1) and, B#, a visual iconic programming notation developed in the Department of CS/IS at UPE (discussed further in Section 5.4.2).

### *1.4.3 Scope*

The hierarchical placement of the current investigation is shown in Figure 1.2 by means of an orange colour scheme. The focus of the investigation is on the impact of using B# as technological support in the teaching model of an introductory programming course in an attempt to increase the throughput rate.

The other components in the diagram illustrated in Figure 1.2 are research related to the approach of the modification of the teaching model. The components appearing in green and purple shading indicate alternative techniques for addressing the same problem addressed by the focus of the current investigation. Some of these alternative techniques (shown by means of a green colour scheme), amongst others, are the themes of related recent and current research at UPE but fall outside the scope of the current investigation (Christians 2003; Gamieldien 2003; Van Tonder 2003; De Jager 2004; Henning 2004; Leppan 2004; Mamtani 2004; Naudé 2004; Vogts 2004; Yeh 2004).

The theme indicated by yellow shading is that of the first approach identified as being an approach that increases the throughput rate in introductory programming courses. This approach that identifies potentially successful students in an introductory programming course has in the past been comprehensively researched in the Department of CS/IS at UPE (Calitz 1984; Calitz *et al.* 1992; Calitz 1997; Greyling 2000; Greyling *et al.* 2002; Greyling *et al.* 2003). The selection model based on the findings of this research has been applied at UPE since 2001 and consequently has a bearing on the profile of the subjects selected as participants in the current comparative study.

Figure 1.2 emphasises that one way in which to modify the teaching model for an introductory programming course is by means of the inclusion of technological support within the teaching model. Many types of experimental technological support in terms of programming notations and development environments have been used to support students of introductory programming courses internationally[9] and nationally (Warren 2000, 2001, 2003), yet the evidence is that not one has thus far gained widespread acceptance for various reasons elaborated on in this thesis (Chapter 3).

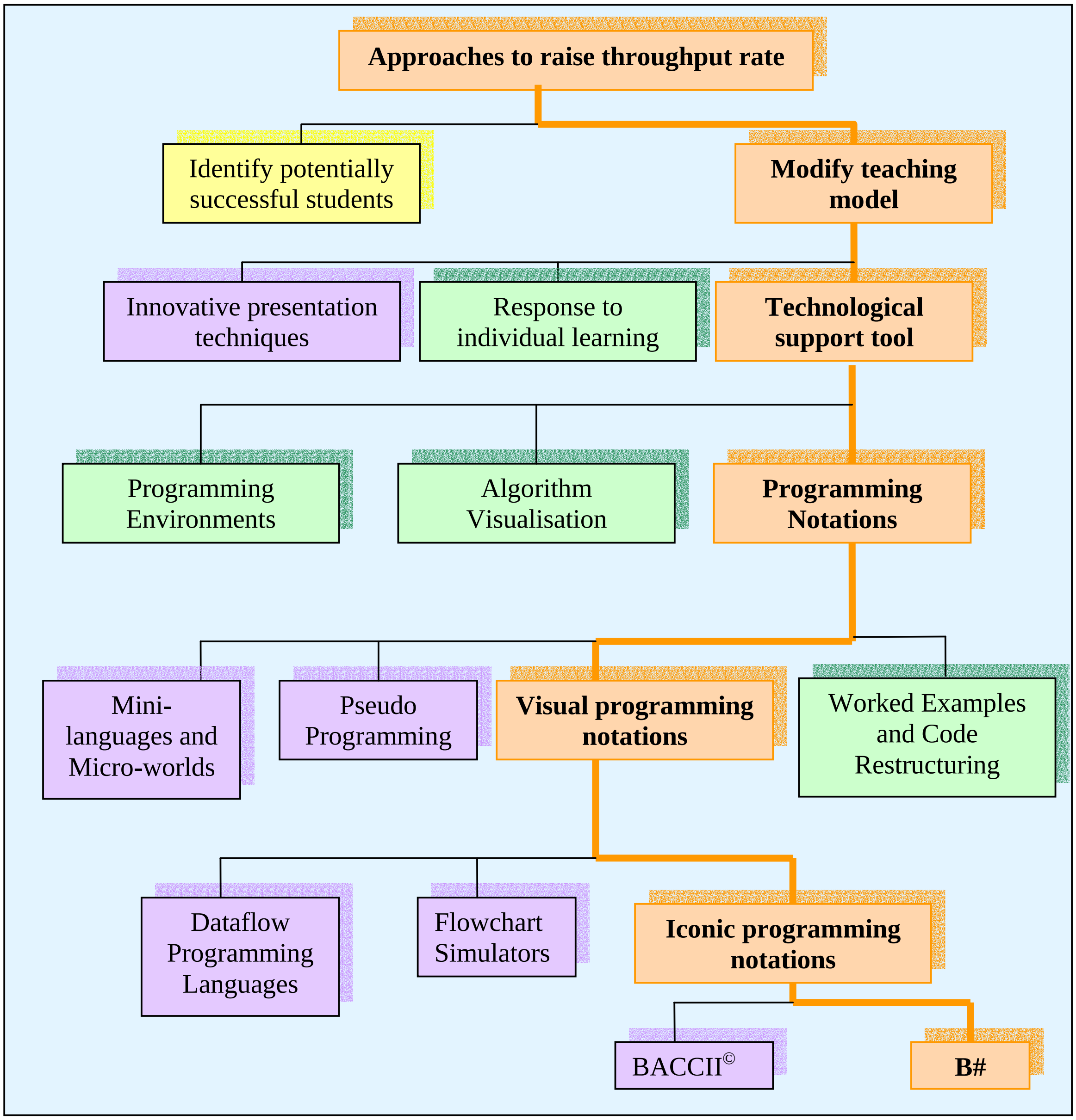

Approaches to raise throughput rate
Identify potentially successful students
Modify teaching model
Innovative presentation techniques
Response to individual learning
Technological support tool
Programming Environments
Algorithm Visualisation
Programming Notations
Mini-languages and Micro-worlds
Pseudo Programming
Visual programming notations
Worked Examples and Code Restructuring
Dataflow Programming Languages
Flowchart Simulators
Iconic programming notations
BACCII©
B#

*Figure 1.2: Scope of the investigation*

[9] (Bonar & Liffick 1990; Lyons *et al.* 1993; Calloni *et al.* 1994; Calloni & Bagert 1995; Studer *et al.* 1995; Liffick & Aiken 1996; Calloni *et al.* 1997; Cockburn *et al.* 1997; Crews *et al.* 1998; Blackwell & Green 1999a; Good 1999; Cooper *et al.* 2000; Garner 2000; Stajano 2000; Blackwell 2001; Dagiano *et al.* 2001; Materson & Meyer 2001; M$^{c}$Iver 2001; Navarro-Prieto & Cañas 2001; Baas 2002; Chamillard *et al.* 2002; De Raadt *et al.* 2002; Fergusson 2002; Gibbs 2002; McIver 2002; Quinn 2002; Burrell 2003; Donaldson 2003; Carlisle *et al.* 2004; Hickey 2004; Mahmoud *et al.* 2004; Zelle undated)

### *1.4.4 Feasibility*

The validity of B# with respect to goodness of fit to the envisaged study was initially determined by means of a pilot study that was conducted on students in the introductory programming course in the Department of CS/IS at UPE during 2002. This pilot study used B# version 1 (Brown 2001a, b).

The results of the pilot study suggested evidence of improved academic performance achievement for students who used B# as technological support (Cilliers & Vogts 2002; Cilliers *et al.* 2003). The findings of the pilot study also suggested programming notation and development environment enhancements that were incorporated into version 2 of B#, which was in development concurrent with the conducting of the pilot study.

The enhanced version of B#, version 2 (Thomas 2002a, b), is used as the experimental programming notation and development environment in the comparative study reported on in this thesis.

### *1.4.5 Research Questions*

The investigation attempts to answer the specific research questions posed in Table 1.1. Various methods are used to determine the answers to the research questions posed. The different methods used are indicated in Table 1.1. Where appropriate, a literature review forms the basis for the research.

An acknowledged experimental design used in studies in programming is adopted, modified and applied in the design of the experiment, collection of data and analysis of the results. The chapter(s) that address each of the identified research questions is(are) also listed in Table 1.1. The significance and relationship of the chapters to the overall investigation is discussed in the following section.

| Research Question | Research method | Chapter |
|---|---|---|
| 1. What is the mental model of novice programmers when learning to program? | *Literature review* | 2 |
| 2. What are the criteria for selecting an appropriate programming notation and associated development environment for novice programmers? | *Analysis of literature review* | 2 |
| 3. What categories of programming notations and associated development environments are used in introductory programming courses at tertiary level? | *Literature review* | 3 |
| 4. What categories of programming notations and associated development environments satisfy the selection criteria for an appropriate programming notation and development environment for novice programmers? | *Critical analysis of literature review* | 3 |
| 5. What is the contribution of previous research at UPE to the current investigation? | *Literature review* | 4 |
| 6. What technological support is developed for use in the current investigation? | *Design and implementation of experimental programming notation and development environment (B#)* | 5 |
| 7. Why were the specified tools chosen as instruments in the current investigation? | *Critical analysis of literature review* | 5 |
| 8. What process is followed in the empirical analysis relevant to the current investigation? | *Literature review*<br>*Experimental design* | 6 |
| 9. How well do novice programmers perform depending on the technological learning environment exposed to? | *Empirical evaluation*<br>*Deliberation on findings* | 7 and 8 |
| 10. What is the impact of a visual iconic programming notation on novice programmer performance and motivation in an introductory programming course at tertiary level? | *Comparative study of textual and iconic programming notations and associated development environments*<br>*Deliberation on findings* | 7 and 8 |
| 11. How should a visual iconic programming notation be used in an introductory programming course at tertiary level? | *Evaluation* | 9 |

*Table 1.1: Research questions and methods used to answer each*

## 1.5 Structure of Thesis

The thesis will specifically address the level of the impact of B#, a visual iconic programming notation (Brown 2001a, b; Thomas 2002a, b; Yeh 2003a, b), on novice programmers in a tertiary level introductory programming course. The organisation of the thesis (based on that documented in Mouton 2001) is illustrated in Figure 1.3 and elaborated on in the following paragraphs.

Chapters 2 and 3 provide an overview of the main underlying theoretical components of the thesis. Chapter 2 defines a framework for novice programmer requirements which is evident in the behaviour of novice programmers when learning to program in both the problem and program domains. The discussion of the latter domain is detailed in terms of the superficial and in-depth levels of learning. Based on the identified requirements of a novice programmer, Chapter 2 concludes with a list of criteria against which programming notations and associated environments are measured in Chapter 3.

An overview of both conventional textual and alternative programming notations and their associated development environments for teaching introductory programming is presented in Chapter 3. This overview incorporates a categorisation and evaluation of technological support most commonly used as the educational and experimental learning environments in introductory programming courses at tertiary level. These programming notations and development environments are identified as being related to those involved in the comparative study on which this thesis focuses.

Chapter 4 summarises related work that has been used to date by the Department of CS/IS at UPE in the selection and placement of introductory programming students. The subjects of the current investigation have been exposed to this procedure of pre-selection prior to being participants in the empirical study. Chapter 5 describes the design as well as the implementation of the B# programming notation and development environment. The discussion highlights support for the framework of novice programmer requirements derived in Chapter 2.

The methodology that is applied in the empirical component of the investigation is presented in Chapter 6, with the results of the empirical study being offered in Chapter 7. A detailed discussion of the composite findings on the literature and empirical investigation follows in Chapter 8. A concluding discussion in Chapter 9 focuses on the evaluation of the initial objectives of the investigation, the contribution of this thesis to existing research, limitations that were identified during the investigation, as well as recommendations for future and further research.

All appendices integral to the investigation reported on in this thesis appear in a separate publication due to the volume thereof.

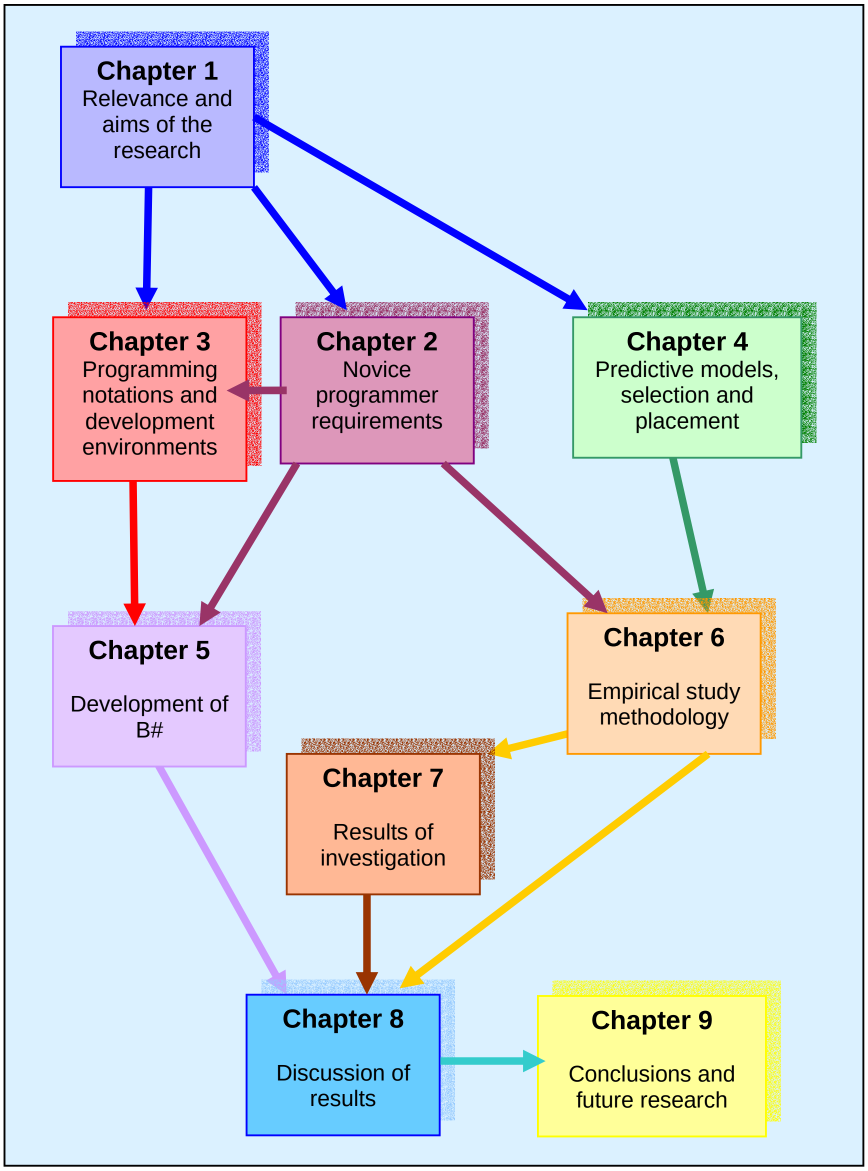

Chapter 1
Relevance and aims of the research
Chapter 3
Programming notations and development environments
Chapter 2
Novice programmer requirements
Chapter 4
Predictive models, selection and placement
Chapter 5
Development of B#
Chapter 6
Empirical study methodology
Chapter 7
Results of investigation
Chapter 8
Discussion of results
Chapter 9
Conclusions and future research

*Figure 1.3: Thesis Outline*

# Chapter 2

# Cognitive Model of the Novice Programmer

## 2.1 Introduction

In Chapter 1, it was emphasised that the current investigation focuses on the use of a programming notation and its associated development environment as an appropriate technological tool in the teaching model of an introductory programming course. In order to validate the choice of a suitable programming notation and development environment, there is a requirement for the comprehension of the manner in which novice programmers learn to program. The focus of this chapter, therefore, is to determine and describe the way that novice programmers function when solving a problem using programming techniques while learning to program.

In the implementation of a solution to a problem, programmers typically operate in two areas, namely the problem and program domains (Mattson undated-a). The problem domain is the statement of the problem and the program domain the solution itself in the form of a representation in a particular programming notation. The successful act of programming is thus the accurate transformation of an appropriate mental representation of the problem domain to a corresponding solution in the program domain (Cockburn *et al.* 1997; Pane *et al.* 2001; Mattson undated-a). The large amount of conversion required by novice programmers between the mental representation of the problem domain and the implementation of a solution in the program domain is highlighted in Chapter 1 (Section 1.3.2). This is a deficiency that contributes to the lack of success for novice programmers in introductory programming courses.

In the program domain, novice programmers function on two further sub-levels when learning to implement solutions, namely the superficial and in-depth levels (Mayer 1981; Perkins *et al.* 1988; Carter *et al.* 2001; Jenkins 2002). During superficial learning, novice programmers typically apply rote-learning techniques in attempts to master programming concepts. In contrast, the exercise of in-depth learning of programming concepts requires a novice programmer to comprehend the effect of the programming concepts within the context of a program solution, irrespective of whether the solution is provided or requiring composition. The in-depth level of learning is typically that level of learning that is primarily required as an outcome of an introductory programming course.

Prior to in-depth knowledge for any particular programming construct being achieved, the novice programmer masters knowledge of that construct at a superficial level. Introductory programming courses enforce the practice that superficial and in-depth knowledge of multiple programming concepts is mastered concurrently. The superficial knowledge for a specific programming construct precedes that of the in-depth knowledge for the same construct, but possibly occurs at the same time as the in-depth knowledge of some other programming construct. Any hindrance at the superficial learning level thus impacts on the progress of the in-depth learning of the novice programmer.

One deficiency identified at the superficial learning level is the high level of precision required in the syntax of textual programming notations typically used as learning environments in introductory programming courses (Section 1.3.2). The previous chapter highlights the following additional deficiencies that also contribute to the lack of success for novice programmers in introductory programming courses, these deficiencies being at the in-depth learning level:

- deficiencies in novice programmer problem-solving strategies;
- a high level of accuracy required in the semantics, structure and style of textual programming notations used as learning environments in introductory programming courses; and

- the need for appropriate technological support in terms of a programming notation that is specifically designed for novice programmers and motivates as well as encourages the novice to be successful in using it.

Recognising the concerns highlighted as deficiencies in the process of learning to program, this chapter describes the manner in which novice programmers, specifically students at tertiary level, experience the process of learning to program within the problem and program domains, and in the latter domain, at both the superficial and in-depth learning levels. A novice programmer's level of success achieved while learning to program impacts on the type and level of individual motivation present in introductory programming courses, and ultimately on performance achievement measurements. An overview of the types of motivation observed in novice programmers when learning to program appears in Section 2.3.

The current chapter concludes with a set of measurement criteria deduced from the described cognitive model of the novice programmer. The list of criteria serves as the requirements of a novice programmer for a programming notation and development environment that forms the technological support in an introductory programming course teaching model (Table 2.1 in Section 2.4). The criteria are further recommended as the instrument against which various categories of technological educational programming notations and development environments are measured in Chapter 3. The list of criteria also forms a share of the basis of the discussion on the development of the investigative instrument in the current study (Chapter 5). The discussion of the methodology appropriate to the current investigation (Chapter 6) includes the criteria as being the dimensions against which performance achievement of novice programmers in the form of first year introductory programming students is quantitatively and qualitatively measured in the current empirical study.

## 2.2 Novice Programmers and Learning to Program

Programming is defined as a process of transforming a mental plan of the problem domain that is in familiar terms into one that is more compatible with the computer, namely the program domain (Pane *et al.* 2001; Mattson undated-a). The problem domain is the definition of the problem and is generally represented in an abstract form. The program domain is the representation corresponding to the solution of the presented problem in some programming notation. Novice programmers are those computer users who typically have little or no experience in the program domain (Mayer 1981; Sutherland 1995).

The theory of learning to program emphasises the interaction between the novice programmer's conceptual model of the problem domain together with the programming notation and associated development environment in the program domain, where a conceptual model is defined as the understanding of the scenario described (Shih & Alessi 1993). It is generally accepted that programmers construct complex hierarchical conceptual models of the problem domain that represent similar structures in the program domain (Toombs 1987). The construction of conceptual models of the problem domain is categorised as intrinsic cognitive load (Garner 2001).

The intrinsic cognitive load evident in novice programmers is high (in terms of large amount of effort required), and is, unfortunately, not directly subject to modification by the use of technological support in the teaching model of an introductory programming course. In contrast, extraneous cognitive load is subject to modification in the teaching model of an introductory programming course. Extraneous cognitive load comprises of the interaction with the programming notations and associated development environments forming the instructional format used in the teaching and learning process in the program domain. Extraneous cognitive load should thus be lowered in order to minimise the total cognitive load of a novice programmer learning to program (Garner 2001).

The extraneous cognitive load can be reduced for a novice programmer by a programming notation and development environment being sensitive to the way in which a novice programmer functions in the program domain, defined previously. A novice programmer functions on two levels of learning when implementing solutions in the program domain, namely at the superficial and in-depth levels of learning (Mayer 1981; Perkins *et al.* 1988; Carter *et al.* 2001; Jenkins 2002). Superficial learning is characterised by novice programmers memorising programming concepts for automatic future use. The other level of learning, in-depth learning, requires comprehension of the effects of programming concepts so that they can be applied in the future implementation of solutions. In-depth learning is often the source of confusion for novice programmers (Shneiderman 1983).

An overview of the problem domain of novice programmers focussing on the format of the conceptual model constructed by a novice programmer in the problem domain is provided (Section 2.2.1). The limitations evident in the conversion by novice programmers of the conceptual model in the problem domain to a solution in the program domain are identified. A detailed discussion of the program domain of novice programmers (Section 2.2.2) focuses on novice programmer experiences of the superficial and in-depth levels of learning.

### *2.2.1 Problem Domain*

The problem domain conceptual model for any problem typically consists of dataflow and task knowledge (Ramalingam *et al.* 1997). Dataflow knowledge is concerned with the transformations which data items undergo, consequently being fundamental to the goals of the tasks of a corresponding program domain representation (Shih *et al.* 1993; Wiedenbeck *et al.* 1999). Even though all programmers experience intrinsic cognitive overload, it remains particularly difficult for novice programmers since they have not yet developed strategies to reduce and effectively manage the intrinsic cognitive load. Consequently, novice programmers may not detect losses of information from working memory, thereby neglecting small but important portions of the problem domain (Spohrer *et al.* 1986). An appropriate and precise conceptual model representing the novice programmer’s conceptual understanding of a specified problem is therefore necessary before an accurate conversion to the program domain can be made.

Performance in the problem domain can also be affected by the way in which a problem is presented and phrased (Whitley 1997). Novice programmers frequently do not translate a given problem into a conceptual model that is independent of the original external representation, namely the presented problem. The external representation of a problem can therefore restrict the internal conceptual model representation. Furthermore, novice programmers will not necessarily use efficient conceptual models to represent problems (Petre & Blackwell 1999). The type of conceptual model typically used by novice programmers in programming exercises is indicative of the level of confidence experienced by the novice when learning to program.

The types of conceptual models have been observed to prominently feature the use of diagrams as notation and structure in the interpretation of programming problems by non-programmers (Pane *et al.* 2001). The visual imagery typically appeared early during the problem-solving process. In related studies of the conceptual models used for learning control constructs in programming problems, it was also evident that figures featured prominently during the initial analysis of a problem (Green 1997; Ginat 2001). Visual conceptual models as opposed to verbal conceptual models were the preferred medium by subjects in another related study (Shih *et al.* 1993).

The results of the aforementioned studies and others (Dillon *et al.* 1994; Astrachan 1998) support early evidence (Mayer *et al.* 1986) that success in learning to program is related to the problem representation skill of diagramming. It can thus be concluded that the conceptual models of programmers within the problem domain tend to feature visual imagery as opposed to textual imagery. The acknowledgement of this factor is applicable in the selection of an appropriate technological support in the learning environment of an introductory programming course (Chapter 5).

Appropriate training and support in the learning environment of introductory programming courses can assist novice programmers in the construction of effective conceptual models which provide appropriate representations of the states and relationships of data elements within the problem domain (Ben-Ari 1998, 2001; Dagdilelis *et al.* 2002). Using these conceptual models, the misconceptions of novice programmers are reduced and the novices thus perform tasks more like expert programmers, especially in the conversion process from the problem domain to that of the program domain (Shih *et al.* 1993).

In a review of studies of novice programmers using textual programming notations, one of the most prominent problems was identified as the limitation on the closeness of the mapping of the problem domain to the program domain and the use of unfamiliar terminology (Pane & Myers 2000). The finding was also evident in an earlier related study conducted by Perkins *et al.* (1988) where it was found that the formal context of the program domain might have hindered the transfer of problem-solving skills (problem domain) to be practiced in programming a solution to a problem (program domain).

The observations just presented support the fact that novice programmers are perhaps able to determine what needs to be done when solving a problem, but often are confused only about how to convert and convey that requirement in a specific programming notation in the program domain (Bonar *et al.* 1983). Attempts to compensate for this limitation include the situation where introductory programming educators design learning activities that explicitly encourage and support the correct and appropriate conversion from problem domain to program domain (Shih *et al.* 1993).

In order to perform the conversion from problem domain to program domain successfully, a novice programmer must concurrently master the skills of syntax, semantics, structure and style of the specific programming notation being used for accurate implementation in the program domain (Studer *et al.* 1995; Proulx *et al.* 1996; Carter *et al.* 2001; Jenkins 2002). The majority of these skills, which form a substantial part of the discussion in the following section, are furthermore required to be mastered at a high level of accuracy (Lischner 2001; Pane *et al.* 2001; Jenkins 2002).

### *2.2.2 Program Domain*

Learning to program using a specific programming notation is one of the first and most fundamental ways to learn about the functionality of computers, thus computer programming has been the major focus of most computer introductory courses (Urban-Lurain & Weinshank 2000). Program solution generation in a programming notation has consequently become one of the primary skills desired as an outcome of an introductory programming course. Therefore, the specific practice of program solution generation is recommended in order to cultivate programmers skilled in the writing of computer program solutions (Shih *et al.* 1993).

The writing of computer program solutions can be viewed as a type of symbolic encoding (Toombs 1987) and is a formal task requiring a high level of accuracy (Lischner 2001; Pane *et al.* 2001; Jenkins 2002). Programming is learnt by a novice programmer in two ways, namely at the superficial level and the in-depth level (Mayer 1981; Perkins *et al.* 1988; Carter *et al.* 2001; Jenkins 2002). The superficial level which typically precedes the in-depth level, incorporates techniques of memorisation in the representation of problems in the program domain. In contrast, the in-depth level requires the comprehension of programming constructs used in the program domain, with a view to the composition of novel solutions to programming problems.

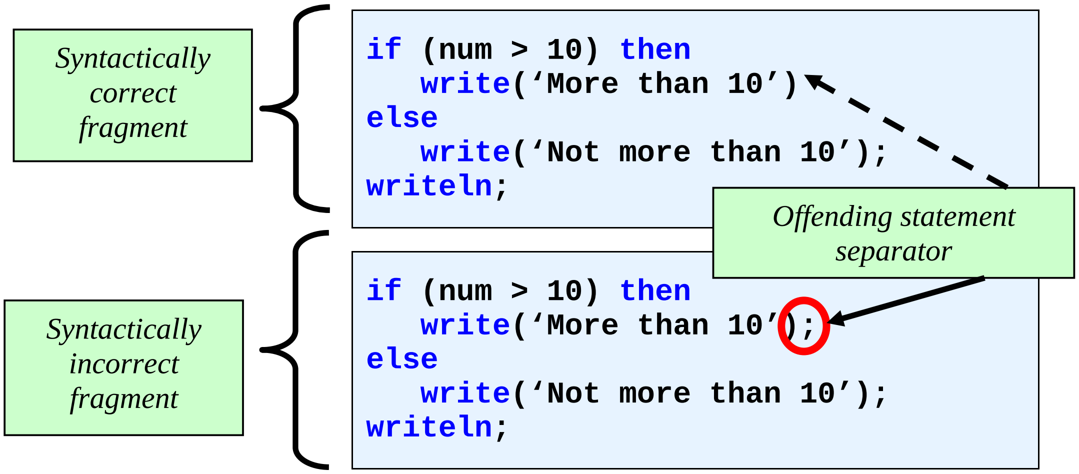
One of the objectives of an introductory programming course is to produce programmers who will be able to create original solutions to problems (Mayer 1981; Shih *et al.* 1993). Consequently, novice programmers are expected not to merely be proficient in a specific programming notation at the superficial level, but to exhibit expertise at the in-depth level of the program domain (Lockard 1986; Kushan 1994; Studer *et al.* 1995; Proulx *et al.* 1996; Dingle *et al.* 2001; McCracken *et al.* 2001; Jenkins 2002). This section thus focuses on the characteristics of the superficial and in-depth levels of learning, these being the components of the program domain.

Superficial Level of Learning

The superficial level of learning in the program domain is characterised by attention to finer detail and typically occurs at a low level of abstraction (Ramalingam *et al.* 1997; Wiedenbeck *et al.* 1999). Mastering of the superficial level is evident in the accurate and correct application of the lexical and syntactical rules of a particular programming notation. Both the lexical and syntactical rules of a particular programming notation can be accurately memorised with minimum effort by novice programmers for application in solutions to programming problems.

In order to master the lexical and syntactical rules of a programming notation, the novice programmer must be sufficiently knowledgeable in the use of the individual programming notation operations and their associated grammatical restrictions to produce program solutions that are of the correct format for the programming notation being used. This involves the correct combination of program statements using the available programming notation operations in the correct sequence (Howe 2003). The rigidity of the requirements for evidence of successful superficial learning in the program domain does, however, possess the potential to hinder the overall progress of learning to program for novice programmers, especially during the composition of novel solutions to problems.

Conventional textual programming notations traditionally used in introductory programming courses require absolute accuracy in the representation of the format of a program solution in the program domain. The precision of the programming notation's lexical and syntactical constraints must be in place before any evaluation of the actual program tasks corresponding to the conceptual model of the problem domain can take place. The ultimate task of having a functioning program at the in-depth level is thus delayed until such time as the program satisfies all of the lexical and syntactical constraints of the programming notation being used at the superficial level. The following discussion illustrates the potential for a delay in the process of learning to program with respect to the level of accuracy required in the syntax of a textual programming notation.


Syntactically correct fragment
Offending statement separator
Syntactically incorrect fragment

```
if (num > 10) then
   write('More than 10')
else
   write('Not more than 10');
writeln;
```

```
if (num > 10) then
   write('More than 10');
else
   write('Not more than 10');
writeln;
```

*Figure 2.1: Textual programming notation fragments: Syntax*

Figure 2.1 illustrates two textual programming notation program solution fragments. They are a correct syntax (top) and incorrect syntax (bottom) for a conditional statement in the commercial textual programming development environment Delphi™ Enterprise using the PASCAL textual programming notation. Even though the fragments are visually very similar, the subtle incorrect placement of the statement separator, the semicolon (`;`), in the bottom fragment results in a syntactical error being detected by the textual programming notation compiler. In fact, the compiler highlights the line containing the **`else`** statement and displays a message similar to **`';' not allowed before 'ELSE'`**. There is, in fact, no **`;`** before the **`else`** statement. The offending **`;`** occurs in the previous line which is not directly highlighted as containing an error. The novice programmer is thus alerted to the incorrect line of the program solution.

The potential thus exists that a novice programmer will not speedily detect the offending semi-colon, especially within a larger and more complex program solution fragment, thereby delaying the progress of constructing a functioning solution representation in the program domain.

The potential consequences of the aforementioned scenario are aggravated by the fact that supporting texts for introductory programming courses typically focus on the instruction of the syntax of textual programming notation constructs (Hennefeld & Burchard 1998; Williams & Walmsley 1999; Kerman 2002). The novice programmer is encouraged to view the result of the programming process as being the textual program solution itself. Attention to detail at the superficial level is perceived by novice programmers as the focus of the problem-solving process (Soloway 1986; Hilburn 1993; Wiedenbeck *et al.* 1999).

Novice programmers may consequently quickly acquire an understanding of the lexical and syntactical constraints of a programming notation, yet this narrow focus of exclusive memorisation can result in a novice programmer missing the broad function of the solution representation in the program domain (McIver 2000). The effect is that the program logic content that is learned at the superficial level can rarely be effectively transferred because of a lack of conceptual understanding of the underlying tasks of the program solution (Jonassen 2000).

Using the superficial learning approach exclusively, novice programmers are consequently not able to relate the programming constructs in a program domain representation to an understanding of how the desired results were produced (Mayer 1981; Liffick *et al.* 1996). A comprehension of the higher level effects of programming constructs within a solution representation in the program domain empowers a novice programmer to construct novel solutions to similar problems (Spohrer *et al.* 1986; Ginat 2001; Mattson undated-a). Such comprehension and transfer is evidence of in-depth learning in the program domain, which is the focus of the next subsection.

### In-depth Level of Learning

In the process of learning to program, the novice programmers' comprehension of the effects and implementation of individual programming constructs is the abstract understanding that complements the superficial level of learning in the program domain. The successful transference and application of the comprehension of programming constructs to new but similar problems is evidence of achievement in the in-depth learning level of the program domain (Mayer 1981; Shih *et al.* 1993; Wiedenbeck *et al.* 1999).

An effect of a low level of conceptual understanding in novice programmers is that they neglect to successfully transfer existing programming knowledge and consequently approach each problem as if it was unique (Urban-Lurain *et al.* 2000). Novice programmers thus require strategies to enable them to view a current problem in terms of previously encountered problems, so that known solution strategies can be transferred to the current problem (Soloway 1986; Perkins *et al.* 1988; Jonassen 2000). The application of this in-depth learning in the program domain is a reflective process, in contrast to the reflexive process of the application of superficial learning, and is evidence of successful problem-solving in the program domain.

During the development of in-depth learning, novice programmers are required to comprehend existing solutions in a particular programming notation. The comprehension process typically occurs in two successive phases (Navarro-Prieto & Cañas 1999). During the first phase the novice programmer develops an understanding of the control flow of the solution in terms of the structure of the solution and outcomes of individual programming constructs making up the solution. The second phase is a determination of the transformation procedures and tasks applicable to data elements of the problem, known as dataflow comprehension. The latter phase is classified as comprehension at a higher level of abstraction and is thus experienced as being the more difficult of the two phases.

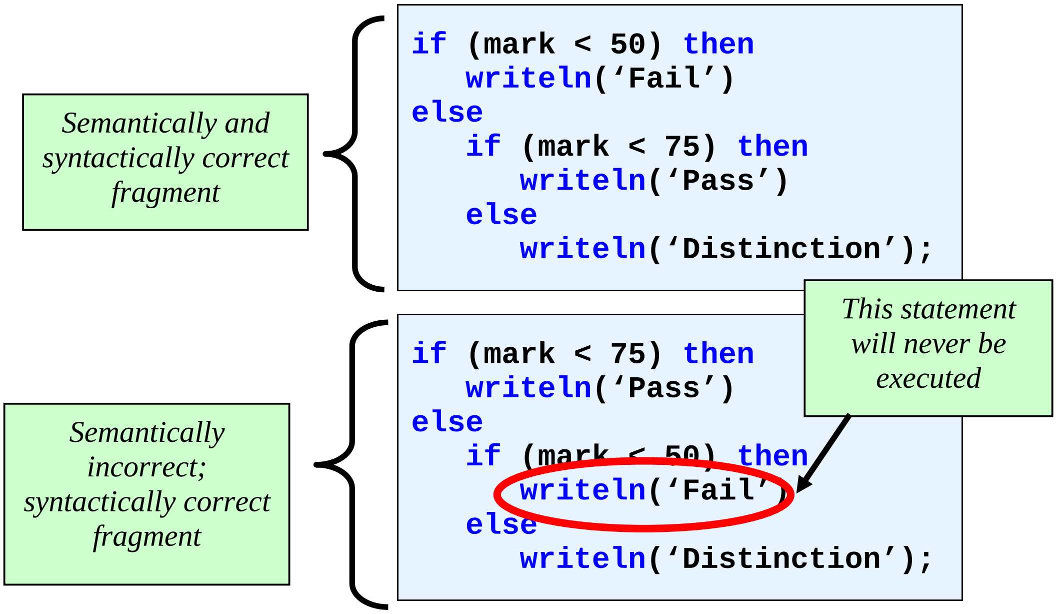
Each of the control and dataflow comprehension phases is discussed in turn. If the necessary support in the learning environment that encourages successful in-depth learning is lacking, novice programmers might develop a conceptual understanding of the control and dataflow of solutions which is ineffective and flawed (Ben-Ari 2001), ultimately resulting in misconceptions. This trait has been identified as one of the greatest obstacles novice programmers face in learning to program (Shih *et al.* 1993; George 2000). For this reason, the subsection concludes with a description of the requirements for educational support of in-depth learning in the program domain.

### *Control Flow Comprehension*

Comprehension of the control flow of a program solution results in a knowledge structure representation of how the program functions (Navarro-Prieto *et al.* 1999) and is usually as a result of reading an existing programming notation solution with evidence of pre-existing superficial learning (Ramalingam *et al.* 1997). Typical examples of control flow concepts are looping and branching programming constructs, as well as the ordering of individual programming constructs in relation to one another, also referred to as program structure.

Novice programmers who master the comprehension of the control flow of existing solutions in a particular programming notation in the program domain are able to provide evidence of understanding the outcome of each individual program statement in the solution (Mayer 1981). They have thus successfully learnt both the syntactical and semantical constraints of the programming notation, as well as appropriate program structure (Shih *et al.* 1993; Deek 1999; Howe 2003).

Figure 2.2 and Figure 2.3 illustrate effects of applied misconceptions related to control flow comprehension. Figure 2.2 shows two sample PASCAL textual programming notation program solution fragments, each being syntactically correct. The program solution fragments are attempts to solve the problem to display an appropriate message depending upon the value of the variable **`mark`**. The upper program solution fragment illustrates the use of PASCAL textual programming notation programming constructs in a correct semantic interpretation. The lower program solution fragment will not work correctly should the variable **`mark`** take on a value less than **`50`**. Due to an erroneous interpretation of the semantics of the PASCAL textual programming notation conditional statement in the lower program solution fragment a logical error is introduced into the program solution.


Semantically and syntactically correct fragment
Semantically incorrect; syntactically correct fragment
This statement will never be executed

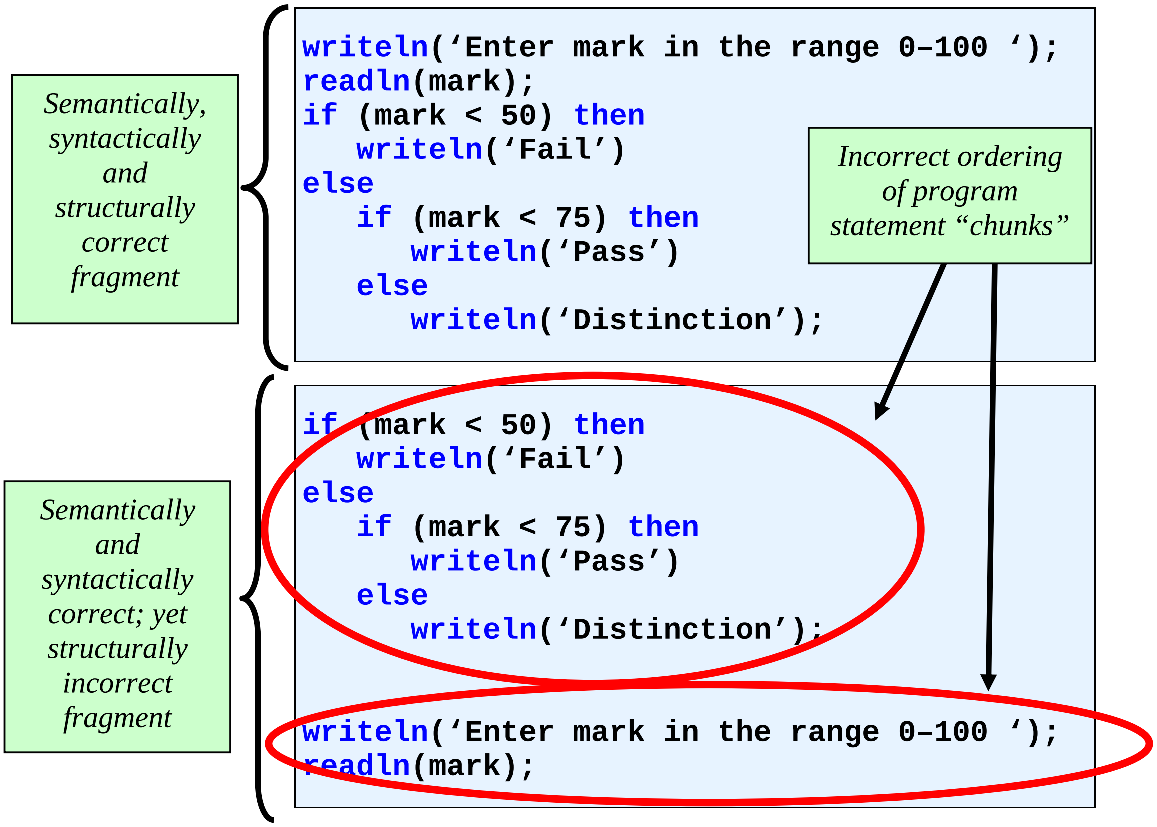
```
if (mark < 50) then
   writeln('Fail')
else
   if (mark < 75) then
      writeln('Pass')
   else
      writeln('Distinction');
```

```
if (mark < 75) then
   writeln('Pass')
else
   if (mark < 50) then
      writeln('Fail')
   else
      writeln('Distinction');
```

*Figure 2.2: Textual programming notation fragments: Semantics*

Similarly, in order to master the structure of a programming notation, the novice programmer is required to comprehend the restrictions placed on the order of programming notation statements in relation to one another. Students often do not see a textual program solution as a sequence of steps that must be executed one at a time (Crews *et al.* 1998). They tend to view program solutions as a collection of statements that execute when necessary, and thus do not pay attention to the correct and exact placement of statements within a program.

Figure 2.3 illustrates the effect of structure comprehension by means of two sample PASCAL textual programming notation program solution fragments, each of which exhibits syntactically correct textual programming notation statements. The program solution fragments are attempts to solve the same problem described for Figure 2.2 but also require that a value for the variable **mark** be entered.


Semantically, syntactically and structurally correct fragment
Incorrect ordering of program statement "chunks"
Semantically and syntactically correct; yet structurally incorrect fragment

```
writeln('Enter mark in the range 0-100 ');
readln(mark);
if (mark < 50) then
   writeln('Fail')
else
   if (mark < 75) then
      writeln('Pass')
   else
      writeln('Distinction');
```

```
if (mark < 50) then
   writeln('Fail')
else
   if (mark < 75) then
      writeln('Pass')
   else
      writeln('Distinction');

writeln('Enter mark in the range 0-100 ');
readln(mark);
```

*Figure 2.3: Textual programming notation fragments: Structure*

The upper program solution fragment exhibits the correct PASCAL programming notation programming structure, namely that the value for the variable **`mark`** be obtained prior to any computation on the variable. The lower program solution fragment also requires that a value for the variable **`mark`** be obtained, but the ordering of the program statements is incorrect. The lower program solution fragment will perform computation on the variable **`mark`** but the result will be unpredictable since the obtained value for the variable **`mark`** will have no bearing on the computation.

Again, a logical error has been introduced due to erroneous programming notation structure interpretation.

In dealing with the types of misconceptions previously illustrated, the novice programmer must first successfully comprehend how a program solution in a particular programming notation functions. Thereafter comprehension of the solution at a higher level of abstraction can take place, namely at the level of comprehending what the program solution does. This comprehension takes place during the dataflow comprehension phase, which is the focus of the following subsection.

*Dataflow Comprehension*

Dataflow comprehension of an existing program solution consists of a top level understanding of the main tasks of the program and the knowledge that is required to understand exactly what the program does (Navarro-Prieto *et al.* 1999). Research has established that the better comprehenders of program solutions are distinguished from the poorer by their higher level of dataflow comprehension (Fix *et al.* 1993).

Novice programmers who have mastered the difficult cognitive skill of dataflow comprehension are able to illustrate their knowledge by means of the recognition of solution patterns (also termed beacons) when converting problem domain conceptual models to solutions in the program domain (Perkins *et al.* 1988; Whitley 1997; Jenkins 2002; Mattson undated-a). The mastery of dataflow comprehension thus supports mental simulations of program solutions used to propose and test assumptions about possible solutions to a problem in the program domain (Shih *et al.* 1993; Mattson undated-a).

A further effect of mastering dataflow comprehension skills is the use of programming construct “chunking”, where a single dataflow item may, in effect, be a collection of related sequential programming constructs that perform a single top level task and that have been individually previously comprehended during the control flow phase (Curtis 1981; Soloway *et al.* 1982; Sutherland 1995; McIver 2000; Urban-Lurain *et al.* 2000). Examples of chunked dataflow items are the average, sum and maximum subrountines.

Chunking encourages a higher level of abstraction and comprehension of solution schemes in procedural form that is programming notation independent (Liffick *et al.* 1996). The comprehension of the higher schematic level characterised by the chunking of program solutions is known as germane cognitive load (Garner 2001). An increase in the use of germane cognitive load by a novice programmer results in a reduced demand on intrinsic cognitive resources. An advantage of chunking is therefore that fewer finer programming details are required to be recalled when comprehending and transferring knowledge of a program solution and thus the total cognitive load on the novice programmer is reduced.

Research literature discusses methods that encourage successful in-depth learning through the control and dataflow comprehension phases (Curtis 1981; Mayer 1981; Astrachan 1998; Applin 2001). The requirements for effective in-depth learning are summarised in the following subsection.

### *Requirements for In-depth Learning Support in a Learning Environment*

Successful in-depth learning within the program domain is an example of meaningful learning. Meaningful learning in the program domain is the process by which the novice programmer connects new concepts, or old concepts in a new context, with programming knowledge that already exists in memory (Mayer 1981). This process of gaining programming knowledge is a series of refinements and reorganisations of the understanding to make it fit the existing knowledge structure (Applin 2001). The general framework of the process of meaningful learning is illustrated in Figure 2.4 and is known as assimilation theory, being the presentation of a model prior to learning (Curtis 1981; Mayer 1981).

The purpose of the presented model is to enhance learning because it provides a meaningful context prior to the actual learning process (Mayer 1981; Astrachan 1998). Related to the current investigation is research that discusses the psychology of how novice programmers learn to program. In this research that has been widely cited over the past two decades[10], Mayer (1981) describes that the human cognitive

[10] Refer to (see Curtis 1981; Mayer *et al.* 1986; Deek 1999; George 2000; Pane *et al.* 2000; Applin 2001; McCracken *et al.* 2001 amongst others) amongst others.

system consists of short-term and long-term memory, where the former is a temporary and limited capacity store for holding and manipulating information, and the latter a permanent, organised and unlimited store of existing knowledge. It is against this background that the process of assimilating programming knowledge by a novice programmer is described.

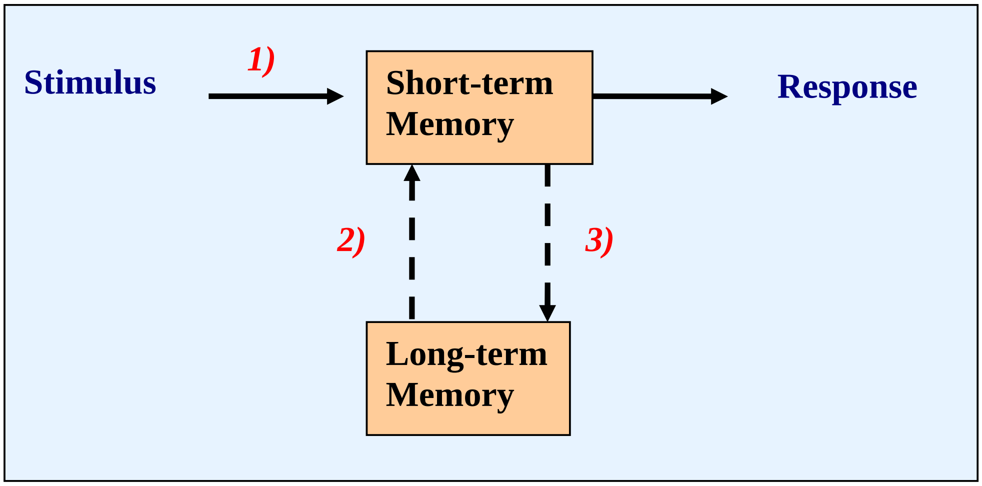


*Figure 2.4: Meaningful learning process*

New programming knowledge is processed sequentially as follows (Figure 2.4 adapted from (Mayer 1981)):

1) On reception of the programming knowledge, it is required that the novice programmer pay sufficient attention to the information so that it reaches short-term memory.
2) Appropriate prerequisite concepts (declarative knowledge) in long-term memory are identified for use in incorporating the new information (Shih *et al.* 1993; West & Ross 2002). These prerequisite concepts are ideas that are used as anchors for the new information and have typically been derived from previous meaningful learning exercises involving the comprehension of existing programming solutions (Mattson undated-a).
3) The prerequisite knowledge is used so that the new programming concept can be connected with it and stored in long-term memory for future use. This is typically the transformation of declarative to procedural knowledge (Shih *et al.* 1993).

Meaningful learning cannot occur if any of the three steps are not processed properly. Each new piece of programming knowledge will need to be memorised as an independent concept to be committed to memory (Mayer 1981). The risk of this is that the new piece of programming information may not be learnt in the manner expected, if at all (Lischner 2001).

Novice programmers need to be supported in the learning environment so that they become skilled in identifying problem prototypes during the assimilation of programming knowledge at the in-depth learning level in the program domain (Dagdilelis *et al.* 2002). The prototype identification skill is necessary for novice programmers to produce solutions to problems by packaging operator sequences into integrated methods (Shih *et al.* 1993). This strategy is identified in the previous subsection as the "chunking" process. Consequently, novice programmers need guidance in developing the skills necessary to better organise information into meaningful groupings.

Meaningful learning of programming concepts is complemented with the use of images (Mayer 1981). Images assist novice programmers to conceptualise the issues relevant to the conversion from problem to program domain in a more useful and consistent way (Kaasbøll 1998). Early research has established that when images are embedded within a technical text, novice programmers tend to perform best on recalling these familiar images and tend to recognise the information adjacent to the image in the text (Mayer 1981). Cockburn *et al*. (1997) support this finding and maintain that novice programmer awareness of program structure can be enhanced through mechanisms that include graphical visualisations as images of program solution content.

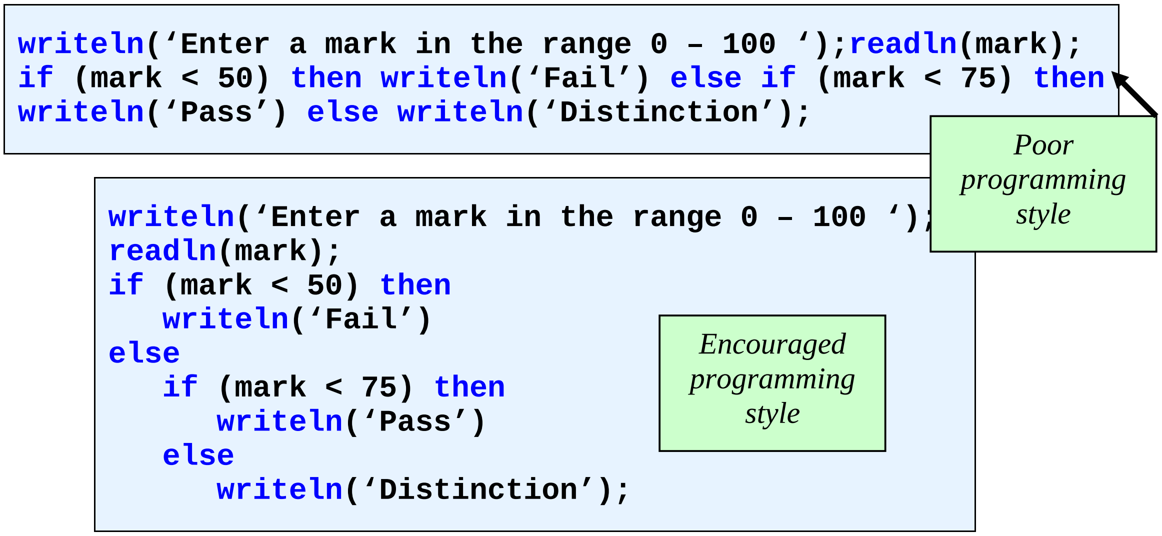
There is also argument that students in technological courses, which include introductory programming courses, are considered to be visual learners (Felder 1993; Fowler *et al.* 2000; Cardellini 2002; Thomas *et al.* 2002). Visual information is defined to include pictures, diagrams, charts, plots and/or animations (Felder 2002). If this is the case, and since it has been found that some novice programmers do indeed learn new material visually rather then verbally, comprehending and developing graphical solutions in the program domain may assist novice programmers to more easily assimilate programming knowledge (Chamillard *et al.* 2002). The use of other visual techniques that aid the in-depth comprehension of program solutions in the program domain is also recommended, one of these techniques being that of programming style.

Mastering the skill of the style of a programming notation, also referred to as secondary notation requires that a novice programmer make use of layout techniques in order to visually aid the comprehension of textual program solution representation (Green & Petre 1993; Whitley 1997; Blackwell & Green 2000). These techniques include the indentation or staggering of program notation statements to indicate several levels of control flow nesting (Kernighan & Plauger 1974) and use of white space within a program.

The program solutions written for and by novice programmers must be readable and simple since they are meant to be read and understood by novice programmers (Kernighan *et al.* 1974). It is therefore essential in the teaching of introductory programming courses to make the purpose of a program solution unmistakable (Kernighan *et al.* 1974). Secondary notation is thus encouraged to make this purpose obvious in the programming notation programming construct detail which will assist in the comprehension of the purpose of the program as a whole. This argument is supported by early research that established that programmers recall more accurately those program solutions that provide evidence of good programming style (Soloway *et al.* 1982).

The lack of correct programming style found in program solutions in the program domain permits novice programmers to build their own sometimes quite flawed model of the control and dataflow of a program solution (Applin 2001). Initial novice programmer programming examples and assignments are often in this badly structured form, an example of which is illustrated in Figure 2.5.

Figure 2.5 illustrates two versions of the correct textual programming notation fragment described for Figure 2.3. The upper fragment in Figure 2.5 does not use secondary notation techniques whereas the lower fragment does. The lack of the appropriate use of secondary notation in the upper fragment makes the visual comprehension of the simple fragment more complex than that of the lower fragment.



```
writeln(‘Enter a mark in the range 0 – 100 ‘);readln(mark);
if (mark < 50) then writeln(‘Fail’) else if (mark < 75) then
writeln(‘Pass’) else writeln(‘Distinction’);
```

Poor programming style

```
writeln(‘Enter a mark in the range 0 – 100 ‘);
readln(mark);
if (mark < 50) then
   writeln(‘Fail’)
else
   if (mark < 75) then
      writeln(‘Pass’)
   else
      writeln(‘Distinction’);
```

Encouraged programming style

*Figure 2.5: Textual programming notation fragments: Style*

A further advantage of good programming style as an aid to novice programmers is that fewer errors will be introduced by them when making changes to a program solution in the program domain. Despite this, programming style is rarely taught in introductory programming courses, either directly or by implication.

The traditional introductory programming education model requires novice programmers to first learn how to perform structural decomposition in their approach to solving problems (Lockard 1986; Kushan 1994; Studer *et al.* 1995; Proulx *et al.* 1996; Dingle *et al.* 2001; McCracken *et al.* 2001; Jenkins 2002). Thereafter, they would be taught the mechanics of the programming notation at the superficial learning level so that they would be able to create and manipulate the data types and control flow mechanisms, amongst other programming concepts, in order to solve the program domain requirements of their problem solution (Cockburn *et al.* 1997). Many novice programmers find the material unexciting because it remains a difficult task for educators to find interesting problems for the novice programmers to solve with the limited set of skills available (Reek 1995; Thomas *et al.* 2002).

Despite this situation, novice programmers should be motivated to expect to succeed in their studies (Jenkins 2001a; Jenkins undated). If they are not motivated to succeed, they will become discouraged and consequently will not learn (Jenkins 2001a). The types and levels of motivation evident in novice programmers enrolled in introductory programming courses is the focus of the following section.

## 2.3 Novice Programmer Motivation

Individual performance of novice programmers in introductory programming courses can also be attributed to personal motivation. A major challenge for educators of introductory programming courses is to find ways to motivate CS/IS students to perform satisfactorily as individuals as well as a group, and to help them enjoy the course.

The traditional teaching method in introductory programming courses is by means of deduction, namely where students are introduced to fundamentals at the superficial level of learning and then proceed with the applications at the in-depth level of learning using conventional textual programming notations and development environments (Felder 2002). This learning environment is often responsible for discouraging many students who have the initial intention and the ability to be successful in programming fields to switch to non-programming fields due to cognitive overload.

It has further been suggested that students whose learning styles are compatible with the teaching style of a course educator tend to retain information longer, apply it more effectively and have more positive post-course attitudes toward the course than do their counterparts who experience learning/teaching style mismatches (Felder 1993). The expectations of introductory programming students with respect to learning resources thus impacts on the type and level of motivation.

The previous argument is supported in part by a recent South African study that examined the effects of a changing society and technology on the way that learners interact with information in an educational environment (Miller 2003). The study revealed that learners today require material in visual format, find or create their own learning content, need fast access to learning material and require learning material with a long-term career value. It is apparent that these learners were motivated by the technology used in information transfer, are active learners, externally motivated and regard learning as a social activity.

Novice programmers should be motivated to expect to succeed in their studies, and they should value the eventual outcome, for example individual success in the course with respect to performance level, final programme qualification or the possession of a qualification that guarantees employability (Jenkins 2001a; Salcedo 2003; Jenkins undated). If novice programmers are not able to appreciate these expectations of success and outcome value, they will become discouraged and consequently will not learn (Jenkins 2001a). Furthermore the motivation dilemma has been observed to be exacerbated in courses containing a diverse population of both computer science majors and non-majors (Chamillard *et al.* 2002). The introductory programming course in the Department of CS/IS at UPE, the context in which the current study takes place, is such a combined course and currently has an intake of 39% (*n* = 144) CS/IS majors and 61% (*n* = 225) non-majors (UPE 2003a).

Motivation amongst novice programmers has been classified as being intrinsic or extrinsic (Jenkins 2002). Novice programmers who are intrinsically motivated have been found to be genuinely interested in the introductory programming course. Extrinsically motivated novice programmers are motivated by the fact that the introductory programming course is a step towards a lucrative career (Wilson *et al.* 1985; Boyle *et al.* 2002; Jenkins 2002; Jenkins undated).

Novice programmers who struggle in the introductory programming course have been more often observed to have a primarily extrinsic motivation (Jenkins undated). One possible way in which to encourage positive motivation amongst novice programmers in introductory programming courses is to provide technological support in terms of a programming notation and development environment that motivates as well as encourages the novice programmers to be successful in using it to compose accurate program solutions.

## 2.4 Conclusion

Success in the transference of programming skills requires a novice programmer to be skilled in the comprehension of existing program solutions at the superficial learning level and especially the in-depth learning level. From the simple textual programming notation program solution fragments illustrated in Figures 2.1 – 2.3 and Figure 2.5, it is obvious that the level of precision required at both levels by novice programmers in order to produce correctly functioning solutions in a conventional textual programming notation is high. Technological support in an introductory programming learning environment should aim to reduce the volume of finer details that a novice programmer has to recall. The impact of this support is a reduction in total cognitive load in the novice programmer while learning to program, especially at the extraneous level.

The aforementioned observation is supported by the fact that novice programmers should begin programming at a level where concepts are what really matters rather than on a lower level where programming notation technicalities become the main issue. Superficial level implementation issues can distract the novice programmer so that the abstractions at the in-depth learning level are not fully comprehended.

Novice programmers should be encouraged in the long-term assimilation of programming knowledge with the use of concrete models that enhance the process of learning to program within technological support in the learning environment in order to ensure that the process of in-depth learning is promoted. Due to novice programmers having been identified as visual learners, the concrete models within the technological support in the learning environment should take the form of imagery.

As a consequence of the discussion in this chapter, 8 requirements are identified as the program domain requirements for technological support in an introductory programming course learning environment. The identified requirements (*R1 – R8*) appear in Table 2.1.

The framework in Table 2.1 emphasises the need for a programming notation and development environment as technological support in the learning environment of an introductory programming course to

- minimise the restrictions placed on novice programmers by a programming notation and associated development environment (*R4*);
- minimise the mundane program solution implementation details at the superficial level of learning (*R1*);
- develop comprehension of the use of programming constructs at the in-depth level of learning (*R2*, *R5*, *R6*, *R7*, *R8*); and
- encourage a positive attitude while learning to program (*R3*).

| **Requirements for Novice Programmer Technological Support** | **Section** |
|---|---|
| ***R1***: Elimination of finer implementation details typically found at the superficial learning level of the program domain | 2.2.1<br>2.2.2 |
| ***R2***: Increased level of program solution comprehension at the in-depth learning level of the program domain | 2.2.2 |
| ***R3***: Increase in level of motivation when using the programming notation | 2.3 |
| ***R4***: Designed specifically for use by novice programmers | 2.2.1 |
| ***R5***: Provision of visual techniques to aid comprehension process at the in-depth learning level of the program domain | 2.2.1 |
| ***R6***: Support for reduced mapping between the problem and program domains | 2.2.1 |
| ***R7***: Increased focus on problem-solving | 2.2.2 |
| ***R8***: Increase in novice programmer performance achievement measured in terms of higher level of accuracy in program solutions in program domain | 2.2.2 |

*Table 2.1: Framework of novice programmer requirements for technological support in the learning environment of an introductory programming course*

The framework of requirements in Table 2.1 is used as the criteria measures in the assessment of different categories of existing introductory programming notations and development environments in Chapter 3. The listed requirements further serve as a partial foundation for the development of the experimental technological support instrument used in the current investigation (Chapter 5) and the description of the methodology applied to the current empirical study (Chapter 6).

# Chapter 3

# Technological Support for Novice Programmers in the Program Domain

## 3.1 Introduction

Novice programmers, when learning to program, are simultaneously faced with the challenges of learning programming concepts and applying them successfully in the program domain. The issues related to the behaviour of novice programmers in their experience of and approach to these challenges are emphasised in Chapters 1 and 2.

Technological support components in the program domain are a programming notation and a development environment (Blackwell *et al.* 2000). The usability of the technological support provided in the program domain is consequently dependent upon both the notation and the development environment (Blackwell *et al.* 2000; Dagiano *et al.* 2001; M$^{c}$Iver 2001). Having to concurrently learn and master the different but related spheres in the program domain often results in cognitive overload for the novice programmer (Garner 2001; DuHadway *et al.* 2002; Garner 2002). One way in which to ease the cognitive load is by making the appropriate selection of programming notation, thereby easing the extraneous cognitive load.

A consideration in the selection of a programming notation for use by novice programmers in the program domain is how easily the novices will learn the chosen notation. Other factors that are just as important are the existence of any notation features that might interfere with the understanding of the elementary programming concepts, as well as any that ease the transformation of the novice programmer to one who is competent (Dingle *et al.* 2001).

The following deficiencies in the textual programming notations and their associated development environments traditionally used as technological support in the program domain for novice programmers in introductory programming courses have been identified in Chapter 1 (Section 1.3.2):

- high level of precision required in the syntax, semantics, structure and style of textual programming notations used in learning environments;
- lack of appropriate technological support in terms of a programming notation and associated environment that enhances understanding of introductory programming for a recognised educational model;
- lack of appropriate technological support in terms of a programming notation that is specifically designed for novice programmers and motivates and encourages the novice to be successful in using it;
- lack of visual feedback in introductory programming notations and associated environments that are used for teaching and learning; and
- amount of conversion required by novice programmers between plan and implementation of solution.

Since all program domains comprise of a programming notation and development environment, this chapter provides an overview of the general features of a program domain for novice programmers. The discussion (Section 3.2) highlights the requirements for a programming notation and associated development environment aimed specifically at novice programmers in the learning environment of introductory programming courses. The discussion focuses on the mapping of the aforementioned requirements in terms of the measurement criteria (*R1* – *R8*) derived in Chapter 2 and duplicated in Table 3.1.

Developments in modern programming development environments and notations that support the teaching and learning of programming by novice programmers address the way in which programming can be taught effectively. These environments and notations form the technological supports which are required to teach and learn programming (Deek 1999). It is further accepted that all of the programming notations and associated development environments presented to novice programmers in introductory programming courses should encourage good program solution design principles (Proulx *et al.* 1996).

| **Requirements for Novice Programmer Technological Support** |
|---|
| ***R1***: Elimination of finer implementation details typically found at the superficial learning level of the program domain |
| ***R2***: Increased level of program solution comprehension at the in-depth learning level of the program domain |
| ***R3***: Increase in level of motivation when using the programming notation |
| ***R4***: Designed specifically for use by novice programmers |
| ***R5***: Provision of visual techniques to aid comprehension process at the in-depth learning level of the program domain |
| ***R6***: Support for reduced mapping between the problem and program domains |
| ***R7***: Increased focus on problem-solving |
| ***R8***: Increase in novice programmer performance achievement measured in terms of higher level of accuracy in program solutions in program domain |

*Table 3.1: Framework of novice programmer requirements for technological support in the learning environment of an introductory programming course*

The programming notation most often used in introductory programming courses is a conventional textual programming notation (Crews *et al.* 1998; De Raadt *et al.* 2002; Reid 2002). Consequently, a discussion of the use of a textual programming notation and associated commercial development environment in an introductory programming course appears in Section 3.3.

A contentious issue surrounding the selection of technological support in a novice programmer program domain is whether to use a textual or a visual programming notation (Pane *et al.* 2000). The use of graphical symbols is considered to be valuable in the instruction of novice programmers (Blackwell 2001). Further, support for teaching and learning in terms of specialised programming notations has become a popular international[11] and national[12] research area over the past decade.

This chapter therefore provides an overview of experimental educational programming notations and associated development environments used in the learning environments of international and national introductory programming courses (Section 3.4). The category of visual programming notations, especially iconic programming notations with characteristics that have influenced the development of B# (Chapter 5), is the primary focus of this section. The success of each of the programming notations and associated development environments presented in this chapter is individually measured in terms of the criteria (*R1 – R8*) listed in Table 3.1.

## 3.2 Program Domain for Novice Programmers

As previously stated, technological support in the program domain consists of both a programming notation and programming development environment (Blackwell *et al.* 2000). The success of technological support in the learning environment of introductory programming courses is thus dependent upon the choice of each of these components of the program domain.

This section focuses on the general requirements for a novice programmer in terms of technological support for each of the programming notation and programming development environment. The measurement criteria applied in the discussion are those derived from Chapter 2 and listed in Table 3.1.

[11] (Bonar *et al.* 1990; Lyons *et al.* 1993; Calloni *et al.* 1994, 1995; Studer *et al.* 1995; Liffick *et al.* 1996; Calloni *et al.* 1997; Cockburn *et al.* 1997; Crews *et al.* 1998; Blackwell *et al.* 1999a; Good 1999; Cooper *et al.* 2000; Garner 2000; Stajano 2000; Blackwell 2001; Dagiano *et al.* 2001; Materson *et al.* 2001; Navarro-Prieto *et al.* 2001; Baas 2002; Chamillard *et al.* 2002; De Raadt *et al.* 2002; Fergusson 2002; Gibbs 2002; McIver 2002; Quinn 2002; Burrell 2003; Donaldson 2003; Carlisle *et al.* 2004; Hickey 2004; Mahmoud *et al.* 2004; Zelle undated)

[12] (Warren 2000; Brown 2001a, b; Warren 2001; Cilliers *et al.* 2002; Thomas 2002a, b; Christians 2003; Cilliers *et al.* 2003; Gamieldien 2003; Warren 2003; Yeh 2003a, b; De Jager 2004; Henning 2004; Mamtani 2004; Naudé 2004; Vogts 2004)

### *3.2.1 Programming Notation*

The choice of programming notation is often justified in terms of the way that it supports mental representations of program solutions (Petre *et al.* 1999). The impact of this is that the level of performance of problem-solving depends upon whether the structure of a problem is matched by the composition of a programming notation used to implement the solution to the problem in the program domain (Whitley 1997). A close match between the structure of a problem and the composition of the program solution in terms of the given programming notation is an illustration of satisfactory support for the novice programmer requirement to maintain a reduced mapping between the problem and program domains (requirement *R6* in Table 3.1).

The observation of the impact of the closeness of the mapping between the problem structure and program solution in a particular programming notation has led to a match-mismatch hypothesis (Gilmore & Green 1984). This theory states that every programming notation emphasises some kinds of information, while hiding others. The match-mismatch hypothesis implies that the benefits of a programming notation are relative to the particular program solution being developed (Blackwell 1996; Whitley 1997; Ko 2003b). In other words, extracting information about a program solution in a program domain is correspondingly easy when the information matches the programming notation and hard when there is a mismatch.

The choice of programming notation used in the development of program solutions by novice programmers can influence the level of success of the program solutions (Kutar *et al.* 2000). The reason for this is that programming notations differ in how well they support the extraction of various kinds of information (Ramalingam *et al.* 1997).

The match-mismatch hypothesis further suggests that in terms of comprehension of program solutions in the program domain, there is unlikely to be any universally superior programming notation. Any particular programming notation may aid comprehension of certain types of information by highlighting that information in some way in the program solution (Ramalingam *et al.* 1997; Wiedenbeck *et al.* 1999).

The visibility and parsability (role expressiveness) of these types of meaningful programming notation structures in the program solution depends on the programming notation itself as well as on the level of expertise of the programmer (Green 1989).

The feature of role expressiveness is an illustration of support for the following novice programmer requirements:

- increased level of program solution comprehension at the in-depth learning level of the program domain (requirement *R2* in Table 3.1); and
- provision of visual techniques to aid comprehension process at the in-depth learning level of the program domain (requirement *R5* in Table 3.1).

Any programming notation provides or fails to provide perceptual prompts which highlight important characteristics of a program solution (Green 1989). This perceptual form of program solution composition, which is often superfluous, can be effective in the process of successful program solution development. However, the ability to make use of this kind of prompt in a program solution depends on the programmer's experience with programming concepts and the actual programming notation itself (Davies 1990).

Research[13] in the area of programming notation has been in the form of attempts to improve the programming process by minimising the complexity of programming notation syntax and increasing the design activities. The minimalism approach to programming notation syntax is an attempt to address the novice programmer requirement to eliminate finer implementation details typically found at the superficial learning level of the program domain (requirement *R1* in Table 3.1). Increasing the design activities in the program domain addresses the novice programmer requirement to increase the focus on problem-solving (requirement *R7* in Table 3.1).

Flexibility in a programming notation is a characteristic noted as being a desirable program domain feature (Dingle *et al.* 2001). Flexibility, however, may favour the experienced programmer, but may hinder, confuse and distract the novice programmer. This feature is in contradiction with the novice programmer requirement to eliminate

[13] (Calloni *et al.* 1994, 1995, 1997; Crews *et al.* 1998; Garner 2000; Crews 2001; Crews & Butterfield 2002; Garner 2002; Crews 2003; Garner 2003)

finer implementation details typically found at the superficial learning level of the program domain (requirement *R1* in Table 3.1). A novice programmer requires a secure comprehension of the underlying structure of the programming notation before attempting to comprehend any type of variation (Dingle *et al.* 2001).

The selected novice programmer programming notation should also encourage germane cognitive load (Garner 2001). This is the ability to "chunk" programming constructs. The development of this cognitive skill encourages novice programmers to think and create the necessary schemata for programming knowledge assimilation in long-term memory. In this way, the novice programmer requirement to eliminate finer implementation details typically found at the superficial learning level of the program domain (requirement *R1* in Table 3.1) is enforced. Support for "chunking" also serves to reduce the total cognitive load of the novice programmer.

A distinction is made between programming notations which are fundamentally sensory and those which are fundamentally conventional (Ware 1993). Sensory notations are defined as being well matched to the early stages of neural processing of sensory information and tend to be stable across individuals and cultures. Conventional programming notations are defined as being influenced by the society in which they are used. Their success regarding use thus depends on the particular cultural environment of an individual. An example of a sensory programming notation is one that makes use of visual representations within its programming constructs, whereas a conventional programming notation is typically of a textual nature.

The programming notation is not the sole component of the program domain that influences the level of success in program solution development. The other component of the program domain that has an influence is that of the development environment (Blackwell *et al.* 2000).

### *3.2.2 Programming Development Environment*

An integrated development environment (IDE) has the ability to be a support for novice programmers learning to program and yet most have been designed for professional programmers (Ziegler *et al.* 1999; Satratzemi *et al.* 2001; Warren 2001; Jenkins 2002;

Garner 2003). One consequence is that these environments tend to have a very steep learning curve, thereby contributing to the extraneous cognitive load (Garner 2001) of the novice programmer, and increasing their already high total cognitive load (Garner 2003). Consequently, these types of IDEs do not support the novice programmer requirement that a development environment be designed specifically for use by novice programmers (requirement *R4* in Table 3.1).

IDEs as program domain support in introductory programming courses have also exhibited a steady increase in level of difficulty (Proulx 2000; Warren 2001). One of the results of this is that modern texts (Hennefeld *et al.* 1998; Williams *et al.* 1999; Kerman 2002) that support the introductory programming learning process have suffered a loss in algorithmic complexity, and consequently in-depth level of learning in the program domain, when compared with older learning resources (McGettrick & Smith 1983). This situation is due to the fact that considerable more of the volume of the learning resources is dedicated to the mastering of the interface of the IDEs. A further consequence is that specifically with regards to introductory programming courses in tertiary education institutions, program domain support for teaching (Section 3.4) has become a popular research area in the past decade[14].

Further, the IDE of any introductory programming learning environment should support three primary learning activities in the comprehension and composition of program solutions, namely the writing, reading and watching of program solutions (Wright *et al.* 2000). These activities offer natural scaffolding that promotes learning and can encourage transfer effects that ease the shift from elementary to advanced problem-solving and programming concepts. A novice programmer's program domain should provide an interactive platform so that when a novice programmer writes a program solution, the program solution's behaviour is a dynamic image (the watchable form) of the program solution (the readable form) that the novice programmer has expressed (the written form). An IDE exhibiting these features supports the following novice programmer requirements:

[14] (Calloni *et al.* 1994; Hansen *et al.* 1994; Calloni *et al.* 1997; Crews *et al.* 1998; Kaasbøll 1998; Christensen *et al.* 2000; Garner 2000; Stajano 2000; Warren 2000; Crews 2001; Warren 2001; Baas 2002; Gibbs 2002; M$^c$Iver 2002; Wright & Cockburn 2002; Burrell 2003; Donaldson 2003; Garner 2003; Ko 2003b; Carlisle *et al.* 2004; Hickey 2004; Mahmoud *et al.* 2004; Zelle undated)

- elimination of finer implementation details typically found at the superficial learning level of the program domain (requirement *R1* in Table 3.1); and
- provision of visual techniques to aid comprehension process at the in-depth learning level of the program domain (requirement *R5* in Table 3.1).

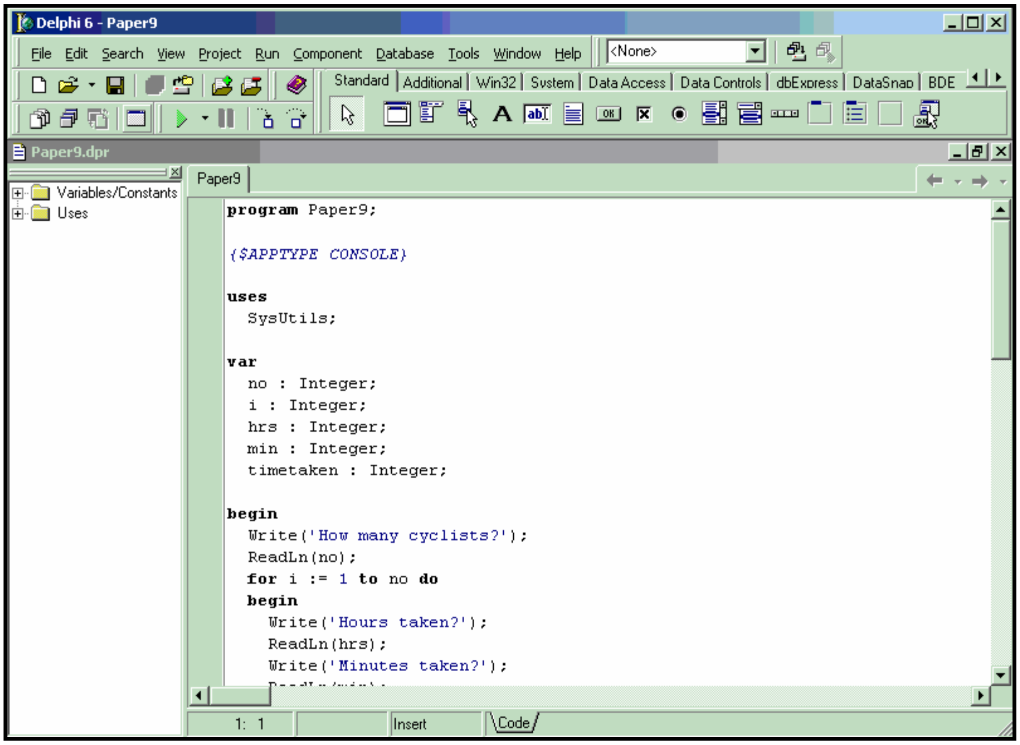
The technological support most often used in the learning environments of introductory programming courses is that of a conventional IDE and a textual programming notation (Crews *et al.* 1998; De Raadt *et al.* 2002; Reid 2002). The focus of the next section is on the use of this type of technological support in the program domain of a novice programmer.

## 3.3 Conventional Interactive Development Environments and Textual Programming Notations

Traditional programming environments typically support textual programming notations. Examples of these types of environments are the Delphi™ Enterprise and Visual Studio commercial development environments that respectively support the textual programming notations PASCAL and C. These types of programming development environments consist of tools aimed at program solution construction, compilation, testing and debugging, with any number of other options included. Examples of these options are pre-coded function libraries, tracers, debuggers and graphical user interface tools. This is the kind of integrated development environment provided to most novice programmers in introductory programming courses at tertiary education institutions (Crews *et al.* 1998; De Raadt *et al.* 2002; Reid 2002).

The aforementioned type of programming development environment and notation (namely textual) is the category of prescribed programming notation and development environment for novice programmers in the introductory programming course at UPE (CS&IS 2003). An example of the interface provided by UPE's conventional commercial programming development environment, namely Delphi™ Enterprise is illustrated in Figure 3.1 (Borland 2003).

Conventional programming development environments like the one illustrated in Figure 3.1 have the density advantage of text, implying that many programming notational primitives can be displayed on the screen at the same time (LaLiberte 1994). These types of environments thus have the ability to concurrently display a multitude of programming constructs.


Delphi 6 - Paper9
File Edit Search View Project Run Component Database Tools Window Help
<None>
Standard Additional Win32 System Data Access Data Controls dbExpress DataSnap BDE
Paper9.dpr
Variables/Constants
Uses
Paper9
1: 1
Insert
Code

```
program Paper9;

{$APPTYPE CONSOLE}

uses
  SysUtils;

var
  no : Integer;
  i : Integer;
  hrs : Integer;
  min : Integer;
  timetaken : Integer;

begin
  Write('How many cyclists?');
  ReadLn(no);
  for i := 1 to no do
  begin
    Write('Hours taken?');
    ReadLn(hrs);
    Write('Minutes taken?');
    ReadLn(min);
```

*Figure 3.1: Borland© Delphi™ Enterprise version 6 Programming Environment*

Despite this advantage, conventional IDEs tend to under-determine the novice programmer by providing no constraints on the textual notational symbols that can be entered. The result is a larger effort in the conversion of the mental model of the desired program solution to one in the required programming notation (Wright *et al.* 2000). The novice programmer is forced to provide precisely correct textual notational syntax before receiving any response to the solution plan and implementation thereof (Crews *et al.* 1998). The implication of the aforementioned observations is that the following novice programmer requirements are not supported by conventional programming development environments like Delphi™ Enterprise:

- elimination of finer implementation details typically found at the superficial learning level of the program domain (requirement *R1* in Table 3.1); and

- support for reduced mapping between the problem and program domains (requirement *R6* in Table 3.1).

Research has also indicated that conventional programming development environments do not meet their goals as support tools for novice programmers and are not suited to the types of problems experienced by novice programmers (Deek 1999). This observation provides evidence to conclude that another of the novice programmer requirements listed in Table 3.1 is not supported, namely that conventional programming development environments are not designed specifically for use by novice programmers (requirement *R4*).

One way in which modern IDEs are considered inappropriate as supporting tools for learning programming is with respect to functional weaknesses (Deek 1999). Functional weaknesses are characterised as the

- lack of the facilities necessary to aid a novice programmer in the formulation of a problem, as well as the planning and design of an appropriate program solution; and
- overemphasis on the programming notation being used as a result of the separation of the composition of a program solution from that of developing the program solution.

The functional weaknesses identified above illustrate that the novice programmer requirements of *R1* (elimination of finer implementation details) and *R7* (increased focus on problem-solving) listed in Table 3.1 are not supported by conventional commercial IDEs promoting the use of textual programming notations. The result is that the novice programmer experiences that a larger effort is required to master the superficial level of learning in the program domain, with little technological support to encourage the mastering of in-depth learning.

Further, another manner in which modern IDEs provide inadequate support for novice programmers is the typical way in which errors within a program solution are highlighted. This observation is supported by the fact that system-generated messages are often typically correct but difficult for a novice programmer to understand (Dagdilelis *et al.* 2002).

The aforementioned observation is further evidence of a lack of support for the following novice programmer requirements:

- designed specifically for use by novice programmers (requirement *R4* in Table 3.1); and
- increased level of program solution comprehension at the in-depth learning level of the program domain (requirement *R2* in Table 3.1).

In response to the resulting situation of using conventional textual programming notations and development environments, the area of alternative programming notations and development environments for novice programmers has become a popular research focus in both the international and national sectors. The next section therefore provides an overview of the educational programming development environments and programming notations used as experimental technological support in the learning environment of an introductory programming course.

## 3.4 Educational Programming Development Environments and Programming Notations

Separating the programming concepts from programming notation implementation details has been proposed as an effective way of teaching and learning introductory programming concepts (Lidtke *et al.* 1998). The motivation behind this recommendation is that novice programmers will not experience the usual confusion of trying to decide if the errors identified are due to syntax, semantics, failure to understand the basic computing concepts or the existence of faulty logic in the program solution. The learnability of a programming notation can also be significantly improved by integrating into the programming development environment learning supports that allow novice programmers to be educated about the syntax, semantics and applications of the notation (Dagiano *et al.* 2001).

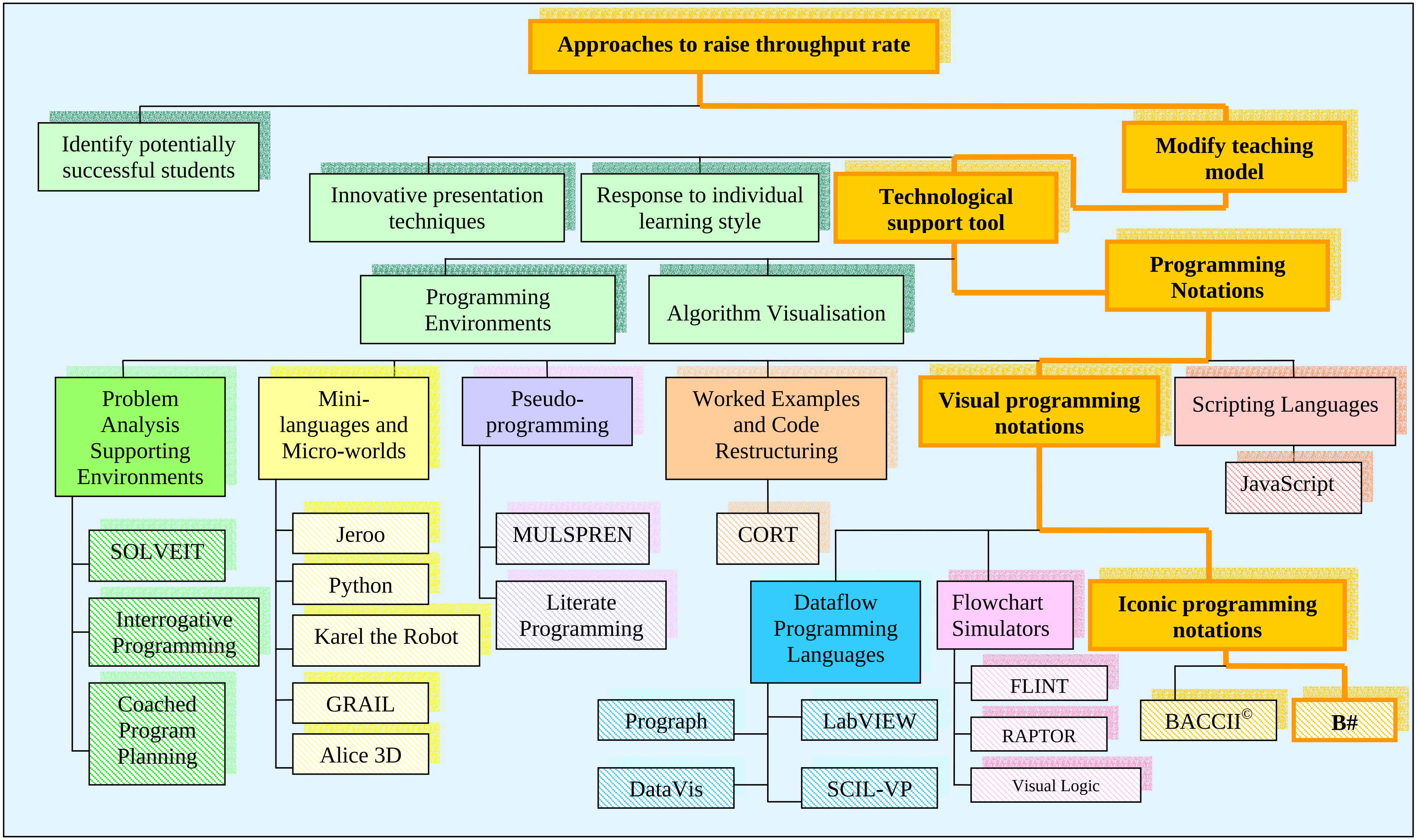

Approaches to raise throughput rate
Identify potentially successful students
Modify teaching model
Innovative presentation techniques
Response to individual learning style
Technological support tool
Programming Notations
Programming Environments
Algorithm Visualisation
Problem Analysis Supporting Environments
Mini-languages and Micro-worlds
Pseudo-programming
Worked Examples and Code Restructuring
Visual programming notations
Scripting Languages
JavaScript
SOLVEIT
Interrogative Programming
Coached Program Planning
Jeroo
Python
Karel the Robot
GRAIL
Alice 3D
MULSPREN
Literate Programming
CORT
Dataflow Programming Languages
Flowchart Simulators
Iconic programming notations
Prograph
LabVIEW
DataVis
SCIL-VP
FLINT
RAPTOR
Visual Logic
BACCII©
B#

*Figure 3.2: Related work in Educational Programming Development Environments and Programming Notations*

A number of different types of approaches for programming development environments and notations that support teaching and learning in introductory programming courses have been proposed as being alternative to traditional commercial programming development environments and textual programming notations. The taxonomy of these approaches in relation to the current investigation is illustrated in Figure 3.2. The path shown in orange is the focus of the current research in the context of the taxonomy of educational programming notations and development environments. The lowest level nodes that are shaded by means of a texture are specific examples of each type of approach.

The categories of programming notations and development environments related to that of the current investigation (Figure 3.2) are namely **problem analysis supporting development environments**[15], **mini-languages and micro-worlds**[16], **pseudo-programming**[17], development environments that support **worked examples and code restructuring**[18], **scripting languages**[19] as well as **visual programming notations and development environments**, incorporating, amongst other categories[20], flowchart simulators[21] and iconic programming notations[22]. An overview of each of these categories is presented in this section. The review of each of these categories includes a measurement of the support provided in terms of the framework of novice programmer requirements derived in Chapter 2 and listed in Table 3.1.

### *3.4.1 Problem Analysis Supporting Development Environments*

A problem analysis supporting development environment is one that integrates a problem-solving methodology with the program solution development tasks (Kimmel *et al.* 1999). This type of technological support provides mechanisms for the formulation of the problem, the planning and design of the program solution, as well as the monitoring

[15] (Deek 1999; Kimmel *et al.* 1999; Lane 2002; Quinn 2002; Lane 2003; Lane & VanLehn 2003; Lane 2004; Lane & VanLehn 2004a)

[16] (Wright *et al.* 2000; Crews 2001; Burrell 2003; Ko 2003b)

[17] (Cockburn *et al.* 1997; Lidtke *et al.* 1998; Crews 2001; Wright *et al.* 2002)

[18] (Garner 2000)

[19] (Stajano 2000; Warren 2000, 2001; Baas 2002; Gibbs 2002; Donaldson 2003; Warren 2003; Hickey 2004; Mahmoud *et al.* 2004; Zelle undated)

[20] (Lyons *et al.* 1993; Hansen *et al.* 1994; LaLiberte 1994; Blackwell & Green 1999b; Navarro-Prieto *et al.* 1999)

[21] (Crews *et al.* 1998; Crews 2001; Carlisle *et al.* 2004)

[22] (Calloni *et al.* 1994, 1997; Calloni 1998)

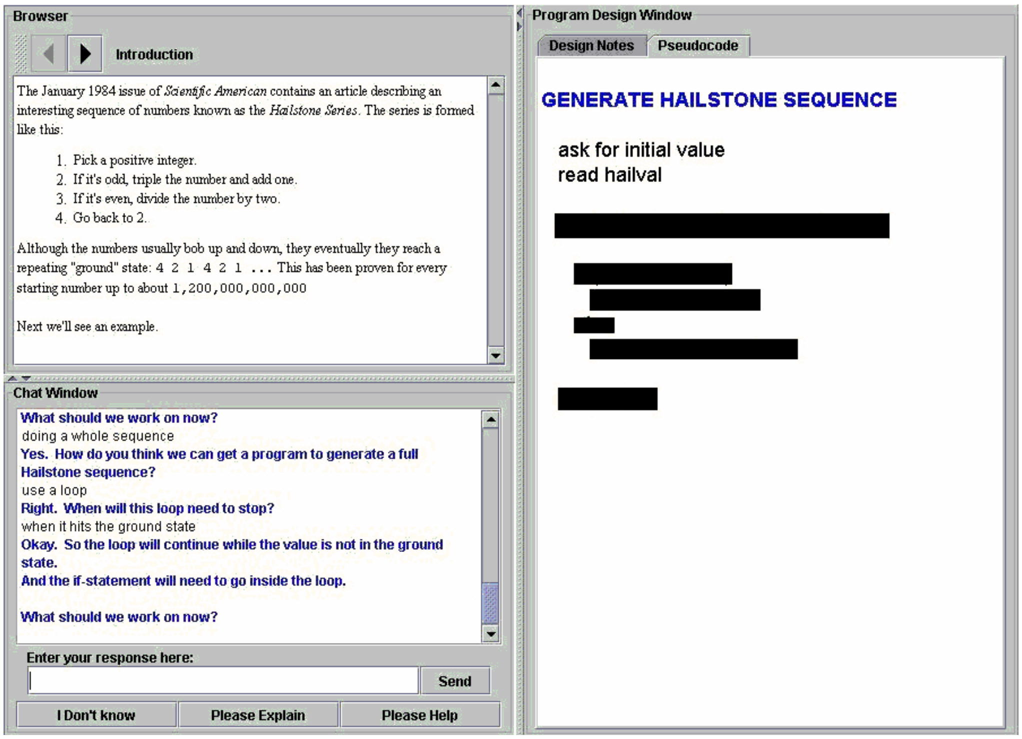
and evaluation of the program solution's progress. Further, the environment aims to encourage novice programmers to comprehend the problem and to consider possible program solutions prior to the implementation thereof. Examples of problem analysis supporting development environments are the **S**pecification **O**riented **L**anguage in **V**isual **E**nvironment for **I**nstruction **T**ranslation (SOLVEIT) (Deek 1999) and enquiring programming development environments (Lane 2002; Quinn 2002; Lane 2003; Lane *et al.* 2003; Lane 2004; Lane *et al.* 2004a).

SOLVEIT is a prototype learning development environment that adapts and enhances the general problem-solving process to the area of programming (Kimmel *et al.* 1999; Garner 2003). This particular environment, which was specifically designed to be used by novice programmers while solving programming problems, has been developed to support the problem-solving and program development process (Deek 1999). The main goals of the techniques used in this tool are to assist and encourage the novice programmer's development of problem-solving and related cognitive skills, encourage novice programmers' positive awareness, manner and motivation towards the learning of problem-solving and programming, and increase the transfer and retention of these skills.

In the evaluation of this tool, the assumption was that novice programmers using SOLVEIT would exhibit improved performance on problem-solving and program development tasks when compared with the performance of novice programmers not using the tool (Deek 1999; Deek & McHugh 2002). The experimental group's scores provided evidence of statistically significant improvements both with respect to problem-solving and program development skills (Deek *et al.* 2002), with significantly positive results for qualitative research issues related to the novice programmers' awareness, manner and motivation during the program development process (Deek 1999).

The second type of problem analysis supporting development environment, enquiring programming development environments are characterised by two types of environments, namely interrogative programming (Quinn 2002) and coached program planning (Lane 2002, 2003; Lane *et al.* 2003; Lane 2004; Lane *et al.* 2004a).

**Interrogative programming** is a method used to determine the novice programmer's program solution intent by means of related closed-ended questions (Quinn 2002). A current prototype of this type of novice programmer learning environment requires that the novice programmer solve all problems in a depth-first manner.



The January 1984 issue of *Scientific American* contains an article describing an interesting sequence of numbers known as the *Hailstone Series*. The series is formed like this:

1. Pick a positive integer.
2. If it's odd, triple the number and add one.
3. If it's even, divide the number by two.
4. Go back to 2.

Although the numbers usually bob up and down, they eventually they reach a repeating "ground" state: 4 2 1 4 2 1 ... This has been proven for every starting number up to about 1,200,000,000,000

Next we'll see an example.

**What should we work on now?**
doing a whole sequence
**Yes. How do you think we can get a program to generate a full Hailstone sequence?**
use a loop
**Right. When will this loop need to stop?**
when it hits the ground state
**Okay. So the loop will continue while the value is not in the ground state.**
**And the if-statement will need to go inside the loop.**

**What should we work on now?**

```
GENERATE HAILSTONE SEQUENCE

  ask for initial value
  read hailval

  [redacted]

    [redacted]

  [redacted]
```

*Figure 3.3: ProPl coached program planning development environment (Lane & VanLehn 2004b)*

A similar environment used in the teaching of programming principles to novice programmers is a dialogue-based style of tutoring, known as **coached program planning** (Lane 2002, 2003; Lane *et al.* 2003; Lane 2004; Lane *et al.* 2004a). The goal of this type of environment is to assist a novice programmer with the comprehension of a problem and the construction of the first step of the program solution design in the form of a natural-language style pseudo-code. A current prototype for the environment **Pro**gram **Pl**anner (ProPl[23]) provides a dialogue window that facilitates natural language communication between a novice programmer and the coached program planning development environment (Figure 3.3). Draggable tiles containing pseudo-code text representing steps in the solution appear in the pseudo-code editor. It was observed that novice programmers exposed to coached program planning adopted more desirable programming behaviours, specifically regarding that of secondary notation at the in-depth level of learning in the program domain (Lane *et al.* 2003).

[23] Pronounced pro-PELL.

| Requirements for Novice Programmer Technological Support | Supported |
|---|---|
| ***R1***: Elimination of finer implementation details typically found at the superficial learning level of the program domain | |
| ***R2***: Increased level of program solution comprehension at the in-depth learning level of the program domain | ✓ |
| ***R3***: Increase in level of motivation when using the programming notation | ✓ |
| ***R4***: Designed specifically for use by novice programmers | ✓ |
| ***R5***: Provision of visual techniques to aid comprehension process at the in-depth learning level of the program domain | |
| ***R6***: Support for reduced mapping between the problem and program domains | ✓ |
| ***R7***: Increased focus on problem-solving | ✓ |
| ***R8***: Increase in novice programmer performance achievement measured in terms of higher level of accuracy in program solutions in program domain | ✓ |

*Table 3.2: Support for Novice Programmer Requirements by Problem Analysis Supporting Development Environments*

Table 3.2 illustrates the support provided by problem analysis supporting development environments for the framework of novice programmer requirements derived in Chapter 2. Inconclusive evidence in the available literature on these kinds of development environments is the reason for a lack of confirmation in respect of whether the novice programmer requirements *R1* and *R5* are indeed supported or not (Table 3.2).

### *3.4.2 Mini-languages and Micro-worlds*

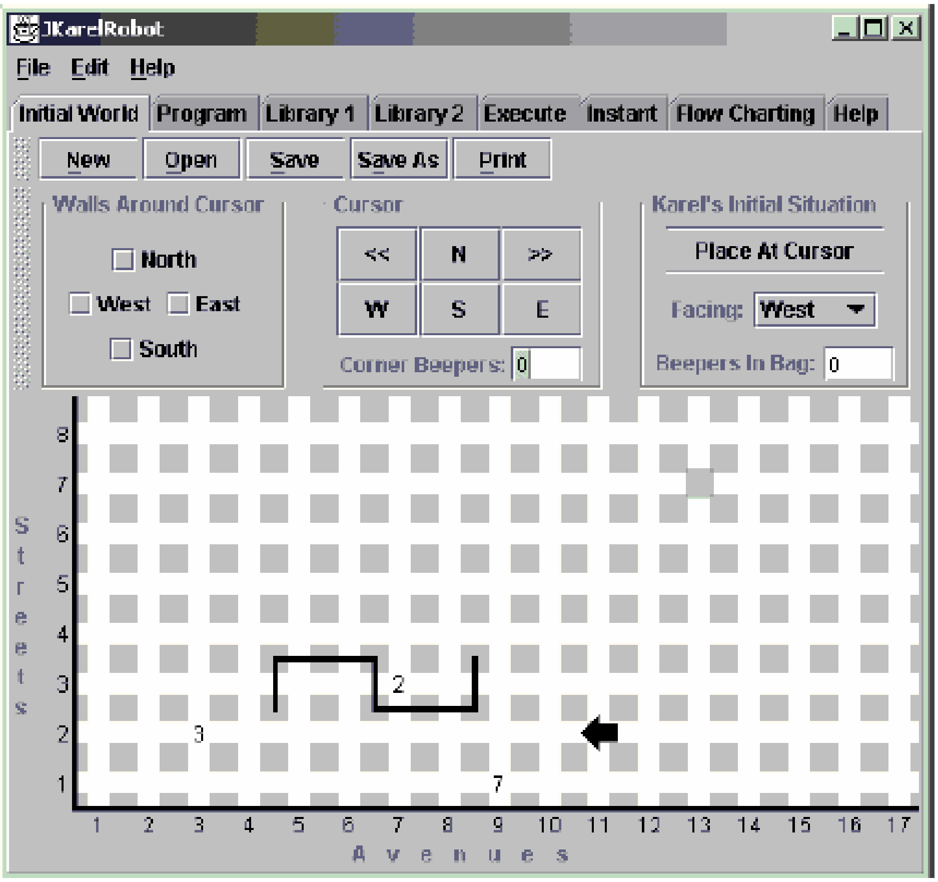
A **mini-language** is a programming development environment containing a small set of programming commands (Crews 2001). This kind of environment is designed to be more instinctive to novice programmers, thus allowing them to unreservedly explore the problem-solving context without being hampered by mundane notational syntax construction. One of the advantages of using a mini-language to teach novice programmers how to program is that interesting and significant program solutions can be implemented by the novice programmer very quickly (Shannon 2003). A recent example of a mini-languages is GRAIL (McIver & Conway 1999; McIver 2001).

The GRAIL programming development environment was designed to be simple, predictable and familiar to the novice programmer (McIver *et al.* 1999; McIver 2001). The programming notation of the control flow mini-language is small and makes use of familiar operations, based primarily on the novice programmers' prior mathematical experience. GRAIL is intended to assist novice programmers in acquiring introductory programming concepts with minimum syntax knowledge and is meant for teaching programming in the short term. Preliminary findings of a study using GRAIL suggest that there is an increase in the level of motivation amongst novice programmers learning to program.

A **micro-world** is a programming environment that simulates a real or imaginary world that has been selected, simplified and perfected to allow concepts and relationships between concepts to be easily observed and discovered by the novice programmer (Burrell 2003). This type of programming development environment attempts to remove irrelevant detail and areas of little direct interest thus intentionally being diminished so as to focus attention on the more important higher-level programming concepts and skills. Examples of micro-worlds are Karel the Robot (Burrell 2003), Jeroo (Sanders & Dorn 2003a, b) and Alice (Cooper *et al.* 2000; Ko 2003b).

A number of micro-worlds, for example Karel the Robot (Burrell 2003) and Jeroo (Sanders *et al.* 2003a, b), provide a dual view of textual program solution representations implemented by the novice programmer as well as visual synthesis in response to the programming instructions occurring in the program solution code (Buck *et al.* 2001). These visual features aid in the comprehension at the in-depth level of learning in the program domain. Figure 3.4 is an illustration of the interface provided by the programming development environment of Karel the Robot.

Jeroo is an integrated programming development environment and micro-world that is designed specifically for novice programmers (Sanders *et al.* 2003a). Jeroo has been successfully used for the first 30% of an introductory programming course at Northwest Missouri State University. The transition from Jeroo to the prescribed Java textual programming notation was perceived as being seamless. This can be attributed to one of the specified design goals of Jeroo, namely that the programming notation of Jeroo be close to that of Java and C$^{++}$. A further observation of novice programmers is that their average confidence level increased significantly after using Jeroo.



□ North

□ West □ East

□ South

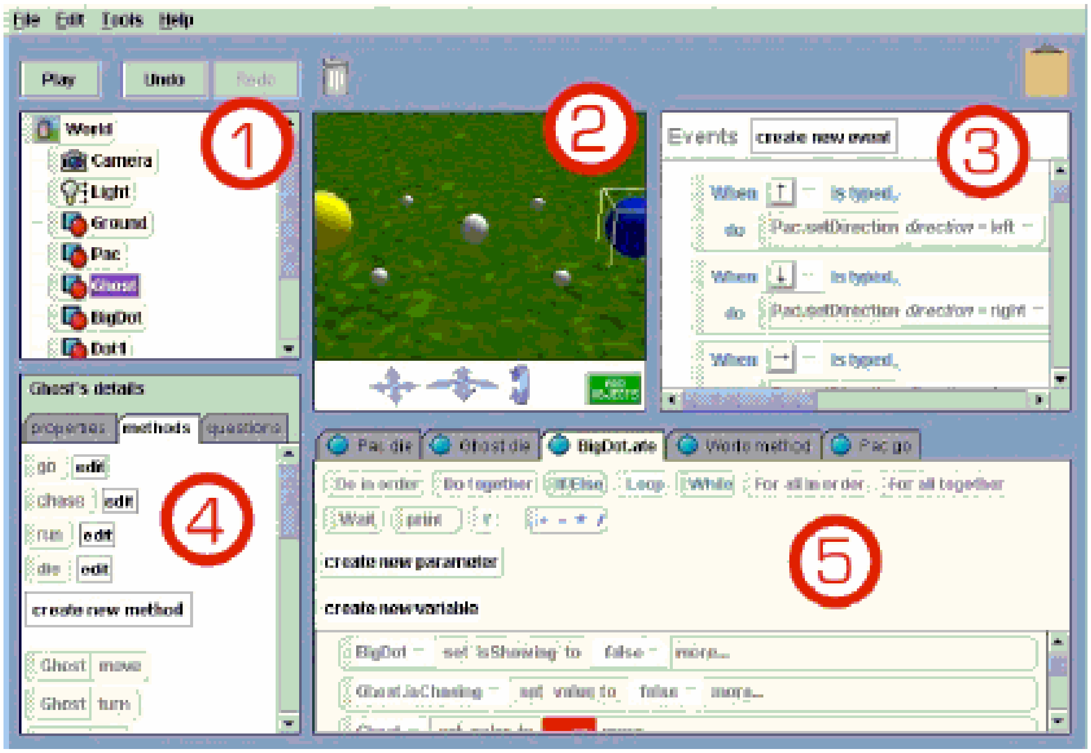
Corner Beepers: 0

Facing: West

Beepers In Bag: 0

*Figure 3.4: Karel's world*

A further example of a micro-world, Alice (Cooper *et al.* 2000; Ko 2003b), described as a three-dimensional (3-D) interactive graphics programming development environment (Cooper *et al.* 2000), is a scripting and prototyping environment that prevents all syntax and data type errors by providing a drag-and-drop controlled editing environment (Ko 2003a). The interface provided by the programming development environment of Alice is illustrated in Figure 3.5. The interface shows (1) an object list, (2) a 3-D world view, (3) an event list, (4) details about the selected object, and (5) the methods being edited (Ko 2003a).



```
When ↑ is typed,
do   Pac.setDirection direction = left
When ↓ is typed,
do   Pac.setDirection direction = right
When → is typed,

BigDot set isShowing to false more...
Ghost.isChasing set value to false more...
```

*Figure 3.5: Alice micro-world*

In a recent study using Alice to teach introductory programming concepts instead of Java, at-risk students' average grade was raised from 1.3 to 2.8 on a 4.0 scale (Cooper *et al.* 2003). The same study observed that the attrition rate of at-risk students was decreased from 90% to 10%. It has been observed, however, that in order to use Alice effectively, novice programmers are required to have an understanding of the co-ordinate system and the spatial relationship of 3-D objects to one another. Further, the error messages produced by Alice are at times obscure.

Table 3.3 illustrates the support provided by mini-languages and micro-worlds as technological support in the program domain for novice programmers, according to the requirements derived in Chapter 2. A review of the available literature provides insufficient evidence to deduce whether these kinds of programming development environments support novice programmer requirements *R6* and *R7* or not (Table 3.3).

| Requirements for Novice Programmer Technological Support | Supported |
|---|---|
| ***R1***: Elimination of finer implementation details typically found at the superficial learning level of the program domain | ✓ |
| ***R2***: Increased level of program solution comprehension at the in-depth learning level of the program domain | ✓ |
| ***R3***: Increase in level of motivation when using the programming notation | ✓ |
| ***R4***: Designed specifically for use by novice programmers | ✓ |
| ***R5***: Provision of visual techniques to aid comprehension process at the in-depth learning level of the program domain | ✓ |
| ***R6***: Support for reduced mapping between the problem and program domains | |
| ***R7***: Increased focus on problem-solving | |
| ***R8***: Increase in novice programmer performance achievement measured in terms of higher level of accuracy in program solutions in program domain | ✓ |

*Table 3.3: Support for Novice Programmer Requirements by Mini-languages and Micro-worlds*

### *3.4.3 Pseudo-programming*

Pseudo-programming is a procedural programming notation and development environment that supports less rigid syntactical rules than conventional textual programming notations (Crews 2001). The use of such a programming notation to describe data structure and algorithms in an introductory programming course permits the concentration on issues relevant to abstraction and not implementation (Lidtke *et al.* 1998). An example of a pseudo-programming notation and development environment is the **MU**ltiple **L**anguage **S**imulation **PR**ogramming **EN**vironment (MULSPREN).

MULSPREN provides multiple representations of a program solution, with one representation being an English-like language and the other a conventional textual programming notation similar to Java (Wright *et al.* 2002). Novice programmers are able to work with either representation of the program solution. Modifications in the one representation are replicated immediately in the other.

The aim of MULSPREN is that multiple representations of a program solution together with program solution visualisation support will assist in the transfer of programming concepts between the familiar domain, namely the English-like programming notation, and the unfamiliar domain of conventional textual programming notations (Wright *et al.* 2002). Consequently, MULSPREN is designed to minimise the mapping between a novice programmer's mental model of a program solution and the implementation thereof in the program domain. MULSPREN also animates both programming notation representations concurrently, thereby reducing the mapping between the novice programmer's mental model of the behaviour of a program solution and actual behaviour by inspection, thus further supporting the transfer of programming concepts.

A sub-category of pseudo-programming is that of literate programming. Literate programming permits programmers to design, document and construct program solutions in the order that best supports the programmers' mental models (Cockburn *et al.* 1997). In this way the structure of the program solution is not restricted by the requirements of the programming notation's compiler or interpreter. Consequently, the major aim of the technique of literate programming is to ensure that program solutions are more comprehensible to programmers.

The technique of literate programming has been designed in such a way that novice programmers will benefit (Cockburn *et al.* 1997). Specific benefits for novice programmers include that the correspondence between program solution representation and documentation will be encouraged and promoted. Further, due to the minimal syntactic requirements of the programming notation, the possibility of syntactic errors in the specification of the program solution will be minimised by the development environment providing syntactic correctness within its interface. The novice programmer is provided with facilities to expand and contract "chunks" of a program solution and they are thus able to control the amount of programming abstraction and detail displayed.

Table 3.4 illustrates the support provided by pseudo-programming as technological support in the program domain for novice programmers, according to the requirements derived in Chapter 2. A review of the available literature provides insufficient evidence to deduce whether this approach as a development environment supports novice programmer requirements *R3*, *R7* and *R8* or not (Table 3.4).

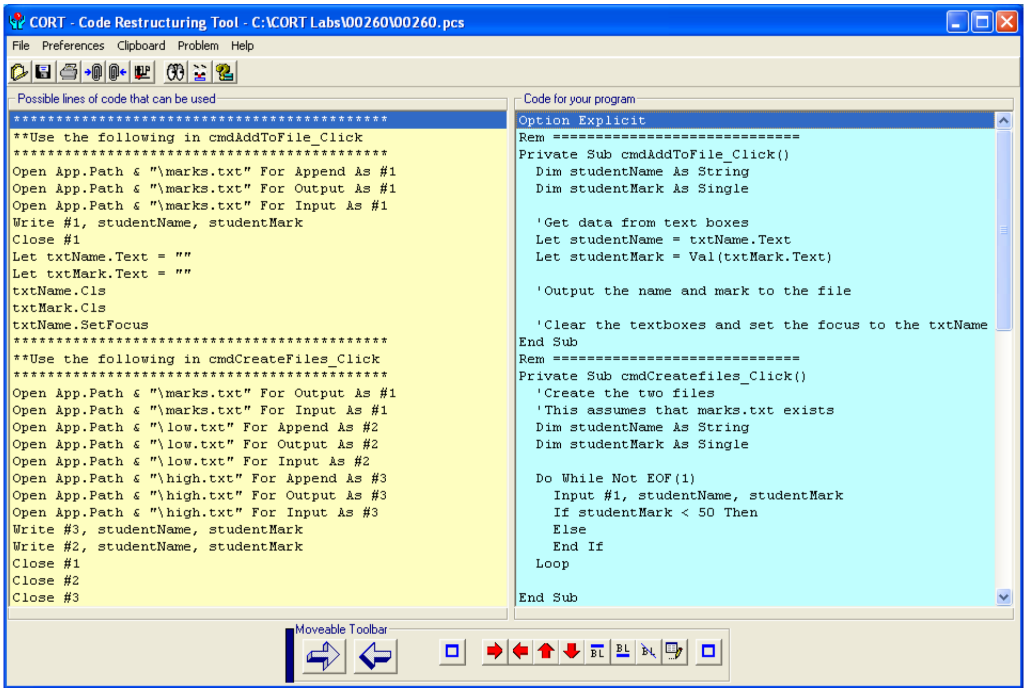
| Requirements for Novice Programmer Technological Support | Supported |
|---|---|
| ***R1***: Elimination of finer implementation details typically found at the superficial learning level of the program domain | ✓ |
| ***R2***: Increased level of program solution comprehension at the in-depth learning level of the program domain | ✓ |
| ***R3***: Increase in level of motivation when using the programming notation | |
| ***R4***: Designed specifically for use by novice programmers | ✓ |
| ***R5***: Provision of visual techniques to aid comprehension process at the in-depth learning level of the program domain | ✓ |
| ***R6***: Support for reduced mapping between the problem and program domains | ✓ |
| ***R7***: Increased focus on problem-solving | |
| ***R8***: Increase in novice programmer performance achievement measured in terms of higher level of accuracy in program solutions in program domain | |

*Table 3.4: Support for Novice Programmer Requirements by Pseudo-programming*

### *3.4.4 Worked Examples and Code Restructuring*

A further technique used in the teaching and learning of programming is the use of worked examples (Garner 2000). This technique involves the exposure to novice programmers of correct program solutions in the form of a conventional textual programming notation. Novice programmers are required to familiarise themselves, with educator interference where necessary, with the algorithms within the worked examples. The aim of the technique is to provide novice programmers with the skills to make modifications to and restructure the code of other similar program solutions. This reading technique of learning programming thus aims to emphasise the reading, comprehension and modifications of non-trivial, well-designed program solutions.

Technological support that implements this category of teaching and learning technique is the **CO**de **R**estructuring **T**ool (CORT) (Garner 2000). CORT permits the novice programmer to interactively complete partial program solutions using a set of provided program solution instructions. The benefit of this method is that the novice programmer is prevented from introducing syntax errors into the program solution and will thus always be exposed to syntactically correct programming instructions and program solutions. An example of the interface supported by CORT appears in Figure 3.6.


CORT - Code Restructuring Tool - C:\CORT Labs\00260\00260.pcs
File
Preferences
Clipboard
Problem
Help
Possible lines of code that can be used
Code for your program
Moveable Toolbar

```
********************************************
**Use the following in cmdAddToFile_Click
********************************************
Open App.Path & "\marks.txt" For Append As #1
Open App.Path & "\marks.txt" For Output As #1
Open App.Path & "\marks.txt" For Input As #1
Write #1, studentName, studentMark
Close #1
Let txtName.Text = ""
Let txtMark.Text = ""
txtName.Cls
txtMark.Cls
txtName.SetFocus
********************************************
**Use the following in cmdCreateFiles_Click
********************************************
Open App.Path & "\marks.txt" For Output As #1
Open App.Path & "\marks.txt" For Input As #1
Open App.Path & "\low.txt" For Append As #2
Open App.Path & "\low.txt" For Output As #2
Open App.Path & "\low.txt" For Input As #2
Open App.Path & "\high.txt" For Append As #3
Open App.Path & "\high.txt" For Output As #3
Open App.Path & "\high.txt" For Input As #3
Write #3, studentName, studentMark
Write #2, studentName, studentMark
Close #1
Close #2
Close #3
```

```
Option Explicit
Rem =============================
Private Sub cmdAddToFile_Click()
  Dim studentName As String
  Dim studentMark As Single

  'Get data from text boxes
  Let studentName = txtName.Text
  Let studentMark = Val(txtMark.Text)

  'Output the name and mark to the file

  'Clear the textboxes and set the focus to the txtName
End Sub
Rem =============================
Private Sub cmdCreatefiles_Click()
  'Create the two files
  'This assumes that marks.txt exists
  Dim studentName As String
  Dim studentMark As Single

  Do While Not EOF(1)
    Input #1, studentName, studentMark
    If studentMark < 50 Then
    Else
    End If
  Loop

End Sub
```

*Figure 3.6: Example of CORT program solution*

The main aim of CORT is to reduce the extraneous cognitive load of the novice programmer by providing a scaffolding learning environment in an introductory programming course (Garner 2000, 2002; Garner undated). In order for a novice programmer to select the correct program solution instructions from the provided list for inclusion in the presented program solution, the novice programmer is required to successfully comprehend the adjacent partial program solution.

Experimental results provide evidence that the method is less time-consuming than conventional methods where novice programmers are required to practice the implementation of program solutions to problems in a conventional textual programming notation (Garner 2000). Further, novice programmers using CORT made fewer errors in subsequently solving similar problems than those novice programmers exposed to a conventional practice-based method. There was, however, no significant difference between the two groups when solving novel problems.

It was however noted that the presentation of worked examples in a specific programming notation to novice programmers is insufficient as the novice programmers may not abstract the correct in-depth knowledge from them (Garner 2000).

| Requirements for Novice Programmer Technological Support | Supported |
|---|---|
| ***R1***: Elimination of finer implementation details typically found at the superficial learning level of the program domain | ✓ |
| ***R2***: Increased level of program solution comprehension at the in-depth learning level of the program domain | |
| ***R3***: Increase in level of motivation when using the programming notation | |
| ***R4***: Designed specifically for use by novice programmers | ✓ |
| ***R5***: Provision of visual techniques to aid comprehension process at the in-depth learning level of the program domain | |
| ***R6***: Support for reduced mapping between the problem and program domains | |
| ***R7***: Increased focus on problem-solving | |
| ***R8***: Increase in novice programmer performance achievement measured in terms of higher level of accuracy in program solutions in program domain | ✓ |

*Table 3.5: Support for Novice Programmer Requirements by Worked Examples and Code Restructuring*

Table 3.5 illustrates the support provided by worked examples and code restructuring as technological support in the program domain for novice programmers, according to the requirements derived in Chapter 2. A review of the available literature only provides evidence to deduce that this approach as a development environment supports novice programmer requirements *R1*, *R4* and *R8* (Table 3.5).

### *3.4.5 Scripting Languages*

The use of scripting languages is another alternative programming notation proposed for use in an introductory programming course[24]. Examples of scripting languages that have

[24] (Stajano 2000; Warren 2000, 2001; Baas 2002; Gibbs 2002; Donaldson 2003; Warren 2003; Hickey 2004; Mahmoud *et al.* 2004; Zelle undated)

been documented as being used in introductory programming courses are JavaScript (Warren 2000; Mahmoud *et al.* 2004), VBScript (Gibbs 2002), Ruby (Baas 2002) and Python (Stajano 2000; Donaldson 2003; Zelle undated).

Python (also portrayed as a mini-language by Shannon (2003)) is described as having a simple syntax and uncomplicated program structure that encourages the use of secondary notation. This feature enhances support for the in-depth level of learning in the program domain. Anecdotal evidence from instructors using Python as a programming development environment for novice programmers is that there has been a positive increase in the level of motivation of the novice programmers and a decrease in the novice programmers' level of frustration.

The use of scripting languages like Python eliminates the rigidity of data typing usually required by conventional textual programming notations. Further, all examination of the syntax is performed incrementally at the time of program solution execution.

The aforementioned feature of an interpreted development environment provides an environment that is appropriate for novice programmers to use (Stajano 2000). The novice programmer can thus concentrate on algorithmic problem-solving issues instead of implementation details (Stajano 2000; Warren 2000, 2001; Baas 2002; Donaldson 2003; Hickey 2004; Zelle undated). The result is that the comprehension of program solutions at a higher-level is encouraged (Gibbs 2002).

Furthermore, anecdotal evidence suggests that novice programmers are more motivated to be successful when using a scripting language (Mahmoud *et al.* 2004). Scripting languages, however, still require the novice programmer to learn a form of programming notation and basic programming techniques (Quinn 2002).

Table 3.6 illustrates the support provided by scripting languages as technological support in the program domain for novice programmers, according to the requirements derived in Chapter 2. A review of the available literature on the use of scripting languages in introductory programming courses provides evidence to deduce that this approach as a development environment supports novice programmer requirements *R1*, *R2*, *R3*, *R4* and *R7* (Table 3.6).

| Requirements for Novice Programmer Technological Support | Supported |
|---|---|
| ***R1***: Elimination of finer implementation details typically found at the superficial learning level of the program domain | ✓ |
| ***R2***: Increased level of program solution comprehension at the in-depth learning level of the program domain | ✓ |
| ***R3***: Increase in level of motivation when using the programming notation | ✓ |
| ***R4***: Designed specifically for use by novice programmers | ✓ |
| ***R5***: Provision of visual techniques to aid comprehension process at the in-depth learning level of the program domain | |
| ***R6***: Support for reduced mapping between the problem and program domains | |
| ***R7***: Increased focus on problem-solving | ✓ |
| ***R8***: Increase in novice programmer performance achievement measured in terms of higher level of accuracy in program solutions in program domain | |

*Table 3.6: Support for Novice Programmer Requirements by Scripting Languages*

### 3.4.6 Visual Programming Notations and Development Environments

Visual programming notations are collectively defined as sensory programming notations in contrast with conventional textual programming notations (Ware 1993). A visual programming notation facilitates programming by means of the interactive manipulation of visual expressions such as graphics or icons within some specific graphical programming notation in order to construct a program solution (LaLiberte 1994). This kind of programming notation facilitates the solving of a problem in the problem domain, as well as the representation of the solution in the program domain (Green & Blackwell 1996; Blackwell *et al.* 1999a). The use of sophisticated interface technology assists with the representation of visual program solutions in the program domain (Koelma *et al.* 1992; Lord 1994; Chang 1995; Blackwell 1996; Freeman *et al.* 1996).

Visual representations of program solutions are considered valuable because they are more intuitive when requiring comprehension by novice programmers (Navarro-Prieto *et al.* 1999; Crews 2001; Quinn 2002; Cranor & Apte undated). These types of program solution representations are also considered to contain more information than corresponding textual programming notation representations (Shu 1988). An example of this type of implied semantic information is that relationships between programming concepts are more explicitly represented without the need for labelling and more easily recognised when appearing in a graphical programming notation than in one that is textual based (Larkin & Simon 1987; Burnett *et al.* 1995; Schiffer & Fršhlich 1995).

A visual programming notation typically provides for the implementation of a restricted set of primitive programming concepts to focus the attention of the novice programmer (Meyer & Masterson 2000). The graphical notation of a visual program solution, however, should not be a sequence of textual programming notation instructions with arrows indicating control flow. The programming notation should rather use its graphical characteristics to support the comprehension of the program solution.

Visual programming notations are described as providing usability advantages for novice programmers because they are easier to learn than textual programming notations (Koelma *et al.* 1992; Lord 1994; Blackwell *et al.* 1999a). This advantage is observed despite the fact that visual programming notations sometimes specify the behaviour of a program solution to an identical level of detail as that specified by an equivalent textual notation program solution (Blackwell *et al.* 2001).

The images generally used in visual programming notations assist the novice programmer to transfer programming knowledge thereby making it simpler for the prediction of the behaviours of program solutions (Blackwell *et al.* 1999a). Consequently, novice programmers are provided with a framework in which to understand the effects of programming instructions and are thus encouraged to construct appropriate mental models in the program domain (Koelma *et al.* 1992; Chang 1995; Blackwell *et al.* 1999a; Navarro-Prieto *et al.* 1999).

Visual programming notations are classified according to the type and degree of visual expression used, with icon-based programming notations being one of these categories (LaLiberte 1994). Visual expressions may also be used in programming development environments as graphical interfaces for conventional textual programming notations. Visual programming notations have also been defined as multi-dimensional programming notations, incorporating specifically the dimension of time within a program solution structure (Chang *et al.* 1999). This section's discussion will however be restricted to the characterisation of a visual programming notation as programming with the use of graphics or icons within a specific paradigm (dataflow or control flow) using a particular type of visual presentation (diagrams or icons) (Burnett & Baker 1994).

This section therefore restricts the discussion to a subset of the categories of visual programming notations and development environments, this subset being directly relevant to the experimental programming notation and development environment used in the current investigation. The specific categories of visual programming notations and development environments focussed on are namely **dataflow programming notations**, **flowchart simulators** and **iconic programming notations**. The section concludes with a discussion on the level of support provided by the visual programming notations discussed for the novice programmer requirements derived in Chapter 2.

Dataflow Programming Notations

Dataflow visual programming notations are one of the most popular paradigms for visual programming notations (Ghittori *et al.* 1998). They are distinguished from other types of visual programming notations in that they consist of visual icons depicting functionality. The connections between the icons depict input to and output from each function (LaLiberte 1994). A programming instruction in a dataflow visual program solution executes as soon as all the inputs required are available, at which time the outputs are produced. In this way the execution order is controlled by the flow of data between connected programming constructs (Jamal & Ronpage 1996).

Examples of dataflow visual programming notations that are usable by novice programmers (Schmucker 1996) are LabVIEW (Hansen *et al.* 1994; Whitley 1997), Prograph (Hansen *et al.* 1994; Meyer *et al.* 2000) and SCIL-VP (Koelma *et al.* 1992). A general scientific dataflow visual programming notation designed for use by non-programmers, specifically scientists, is **Data Vis**ualization (DataVis) (Hils 1999). Both LabVIEW and Prograph are commercially available dataflow visual programming notations (Blackwell *et al.* 2001).

The dataflow visual programming notation Prograph is classified as a visual object-oriented dataflow notation (Hansen *et al.* 1994; De Roure *et al.* 1998) that is a visual form equivalent to that of the LISP textual programming notation (Meyer *et al.* 2000). All programming constructs are created and manipulated through direct manipulation of the icons on the screen. An example of a Prograph program solution that calculates the factorial of a user entered value is shown in Figure 3.7. An evaluation of the programming constructs implemented by Prograph determined that the programming constructs are confusing, which could lead to the misinterpretation of a program solution, especially by novice programmers (Meyer *et al.* 2000).

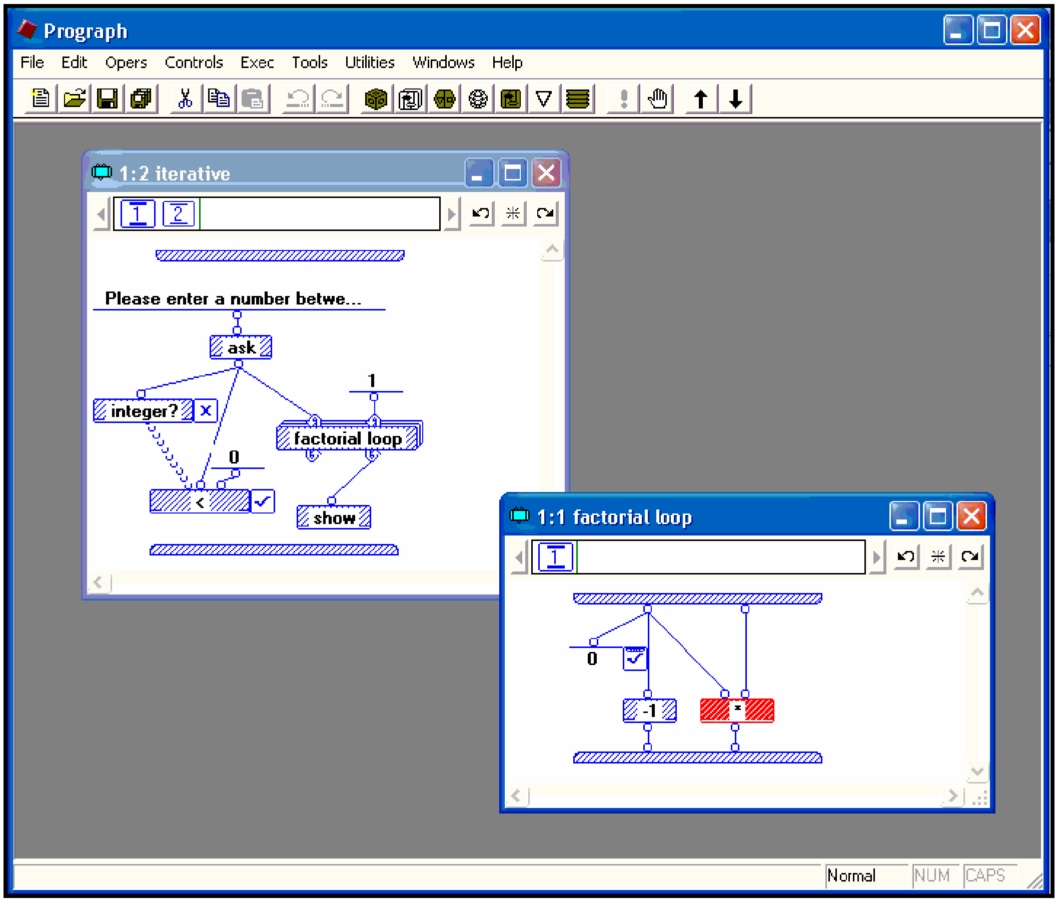

Prograph
File Edit Opers Controls Exec Tools Utilities Windows Help
1:2 iterative
Please enter a number betwe...
ask
integer?
1
factorial loop
0
<
show
1:1 factorial loop
0
-1
*
Normal
NUM
CAPS

*Figure 3.7: Prograph for Windows version 1.2 dataflow programming notation interface (Pictorius 1998)*

A further example of a dataflow programming notation, LabVIEW was written specifically for end-user programming, where the end-users are scientists and engineers (Whitley 1997). In a documented appraisal on this programming environment, the strongest effect observed was the difficulty of the visual programming notation. It was found that novice programmers were distinguished from their expert counterparts in the lack of ability to read and use the secondary notation evident in LabVIEW (Whitley 1997; Whitley & Blackwell 2001). Further, it was observed that programming with LabVIEW requires much the same level of precision as that required by conventional textual programming notations (Lavonen *et al.* 2003).

SCIL-VP (Koelma *et al.* 1992) is a multi-layer dataflow programming language environment that provides support for both the novice and experienced programmer. A feature of SCIL-VP is that it combines the dataflow programming notation with the textual programming notation C so that the more experienced programmer is provided with support to address implementations that would not normally be possible using only the dataflow programming notation.

DataVis (Hils 1999) is a dataflow visual programming notation that has been designed to be used by scientists for the visualisation of scientific data. DataVis makes use of a “plumbing” metaphor in the implementation of a program solution and has been described as being easy for non-programmers to learn. DataVis provides a library of predefined visual programs that can be associated with icons representing data values. It is thus not necessary for a scientist using DataVis to program since the only tasks that require completion are the association of data icons with predefined program icons.

Figure 3.8 illustrates an example of a DataVis **`CASE`** statement (adapted from Hils 1999). Inputs to the **`CASE`** statement originate from the **`Selector`**, **`Number1`** and **`Number2`** data icons. The outcome of the computation is processed in any of the various frames of the **`CASE`** statement and is output to the **`Result`** data icon.

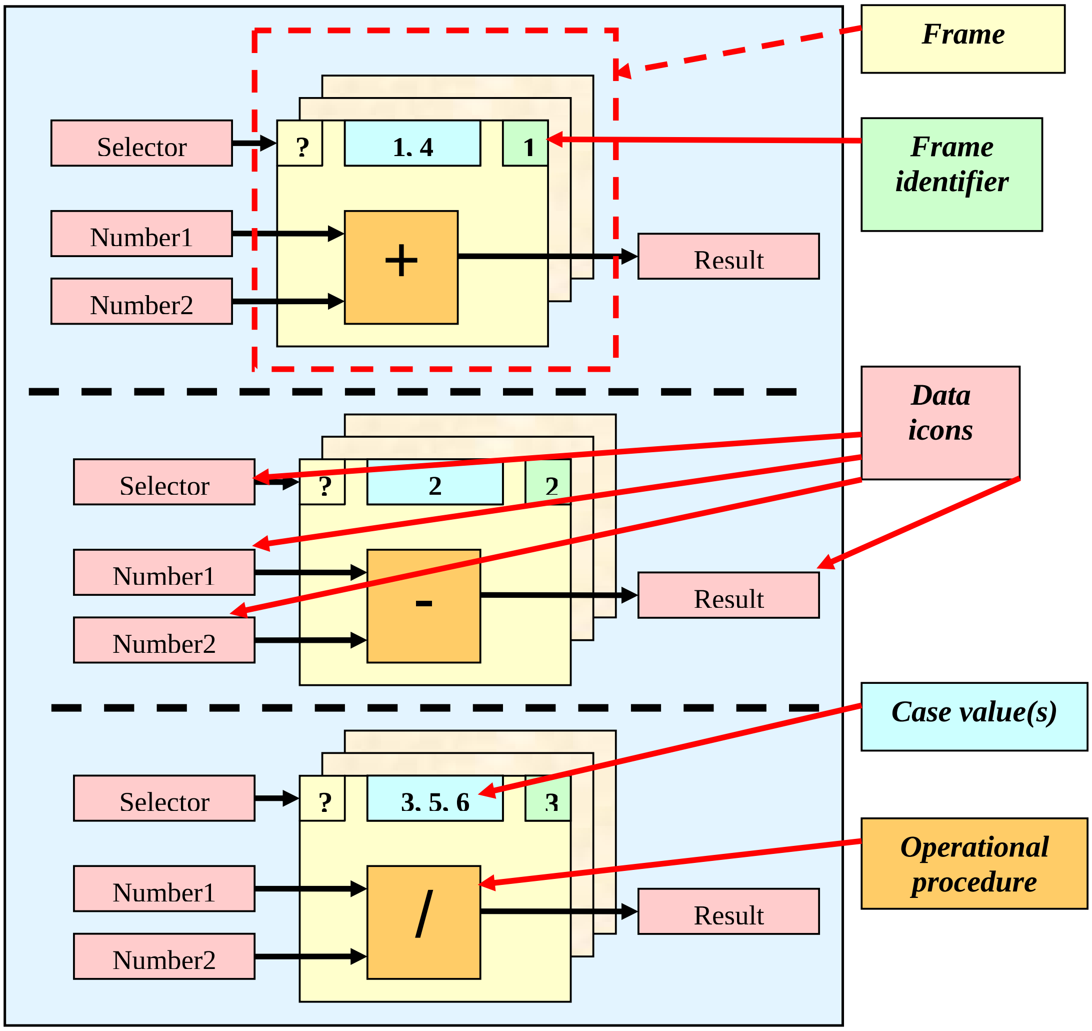


*Figure 3.8: Example of a DataVis CASE statement*

Each frame in the **CASE** statement (the particular one in Figure 3.8 comprises of three) is numbered at the top right hand side for ease of identification. The value in the **Selector** data icon connected to the **?** icon at the top left of each frame determines which of the frames is executed. Specifically, if the value of the **Selector** data icon is **1** or **4**, the **Result** data icon would contain the sum of the values in data icons **Number1** and **Number2** (output from process in frame **1**). If the value of the **Selector** data icon is **2**, the **Result** data icon would contain the difference of the values in data icons **Number1** and **Number2** (output from process in frame **2**). If the value of the **Selector** data icon is **3**, **5** or **6**, the **Result** data icon would contain the quotient of the values in data icons **Number1** and **Number2** (output from process in frame **3**).

A type of visual programming notation that supports the control flow programming paradigm is that of flowchart simulators. This type of visual programming notation and programming development environment is the focus of the next section.

Flowchart Simulators

An early example of a visual programming notation is the technique of flowcharts (Waddel & Cross 1988; Meyer *et al.* 2000). A flowchart is a graphical notation (Meyer *et al.* 2000) of icons that illustrates the flow of control through a process (Crews 2001). Flowcharts thus assist in the illustration of algorithms (Cranor *et al.* undated). An example of an elementary flowchart that illustrates the algorithm to determine and display the area of a circle appears in Figure 3.9.

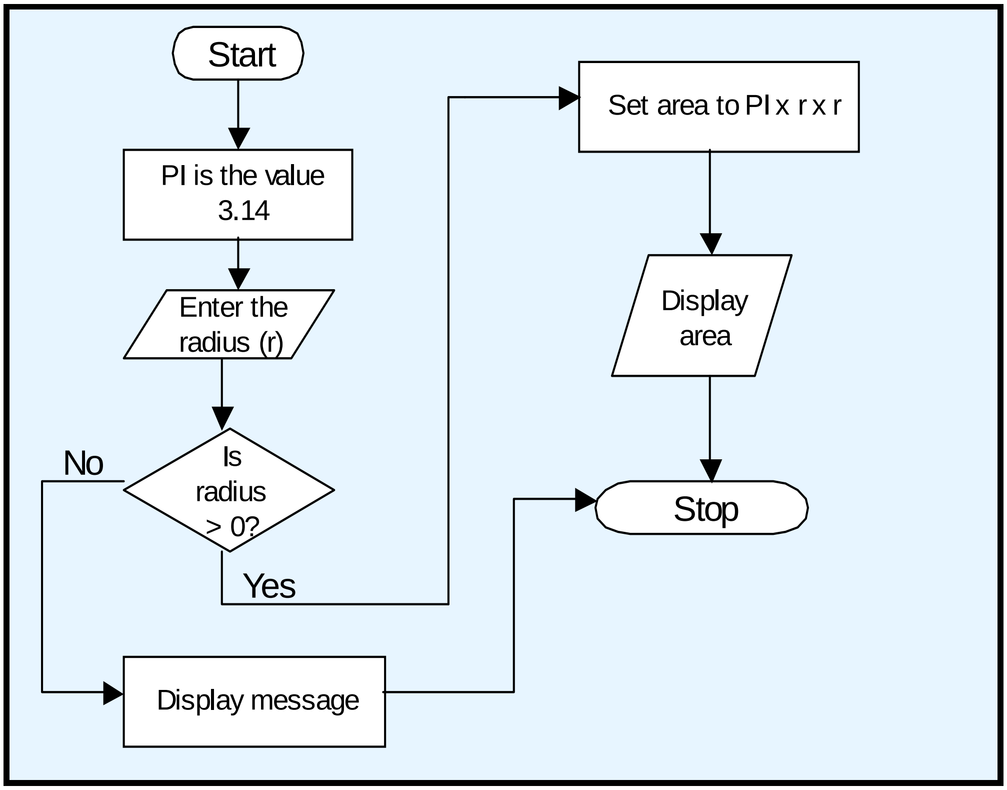


*Figure 3.9: Example of a flowchart that calculates and displays the area of a circle*

A flowchart thus facilitates a conceptual representation of the sequence of significant programming constructs such as input, output, assignments, conditions and looping structures in relation to one another. Consequently, flowcharts support higher level learning activities in the program domain (Crews 2001). Flowcharts assist novice programmers in following a program's decision structure (Curtis 1981).

It has been observed that flowcharts, when compared with structured textual programming notation program solution, are useful conceptual tools for assisting novice programmers to better understand abstract problems specifically with respect to solution accuracy and time to completion when generating program solutions (Crews *et al.* 2002). It was also observed that significant benefits existed with respect to novice programmer confidence and incidence of errors in program solutions. Examples of technological tools that support the flowchart technique are flowchart simulators, namely the **S**tructured **F**low **C**hart Editor (SFC) (Watts 2003), **FL**owchart **INT**erpreter (FLINT) (Crews *et al.* 1998) and the **R**apid **A**lgorithmic **P**rototyping **T**ool for **O**rdered **R**easoning (RAPTOR) (Carlisle *et al.* 2004).

A novice programmer develops a flowchart representation of a program solution using SFC (Watts 2003). An equivalent textual programming notation representation (in C or PASCAL-like programming notation) of the program solution is concurrently displayed. In order to complete the program solution, the novice programmer is required to copy and paste the textual programming notation representation into a text editor and make modifications in the latter environment.

FLINT is a visual instructional programming development environment that makes use of minimal programming notation top-down design flowcharts as visual representations of program solutions (Ziegler *et al.* 1999). FLINT facilitates the removal of attention from the syntactic details of a programming notation by providing novice programmers with an iconic interface for developing flowcharts (Crews *et al.* 1998; Garner 2003). Programmers create flowcharts by dragging-and-dropping the flowcharting icons onto a design panel. Timely and helpful feedback is presented to support the learning process by means of automated feedback of the graphical program solution entered. An example of a program solution in the FLINT visual programming development environment appears in Figure 3.10.

The FLINT development environment was specifically designed to provide programmer interaction with solution design activities without the need to interact with a traditional textual programming notation. FLINT is thus designed for novice programmers and is intended to be easy to use (Crews 2001). The environment is an attempt to address the problems associated with programming notation syntax, problem-solving and support for program solution execution in a unified manner (Crews *et al.* 1998).

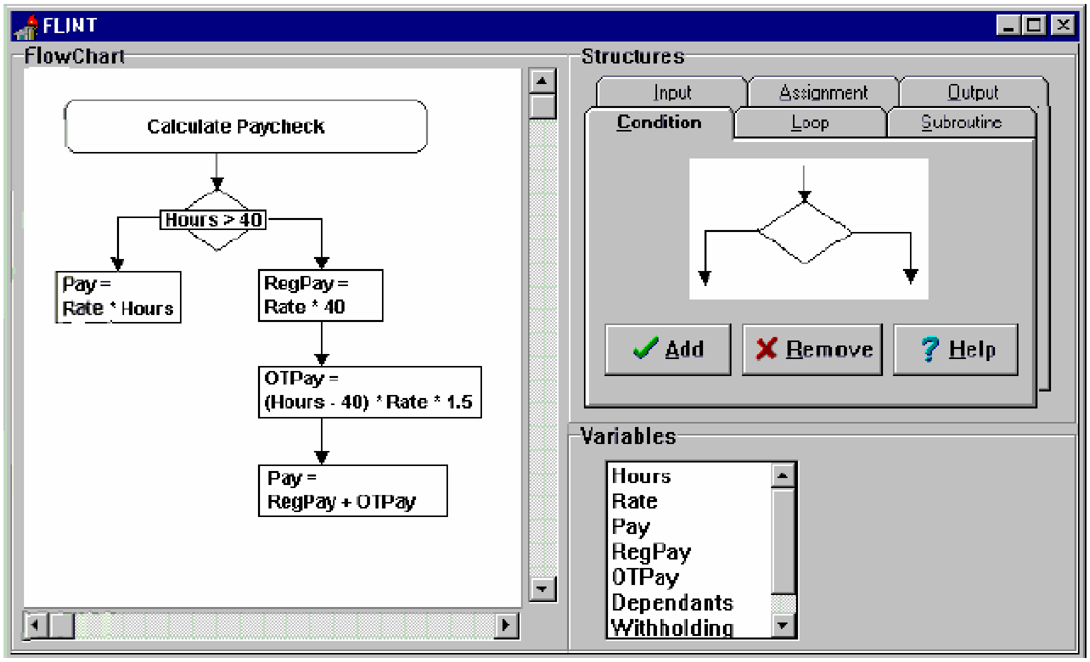


*Figure 3.10: FLINT flowchart simulator*

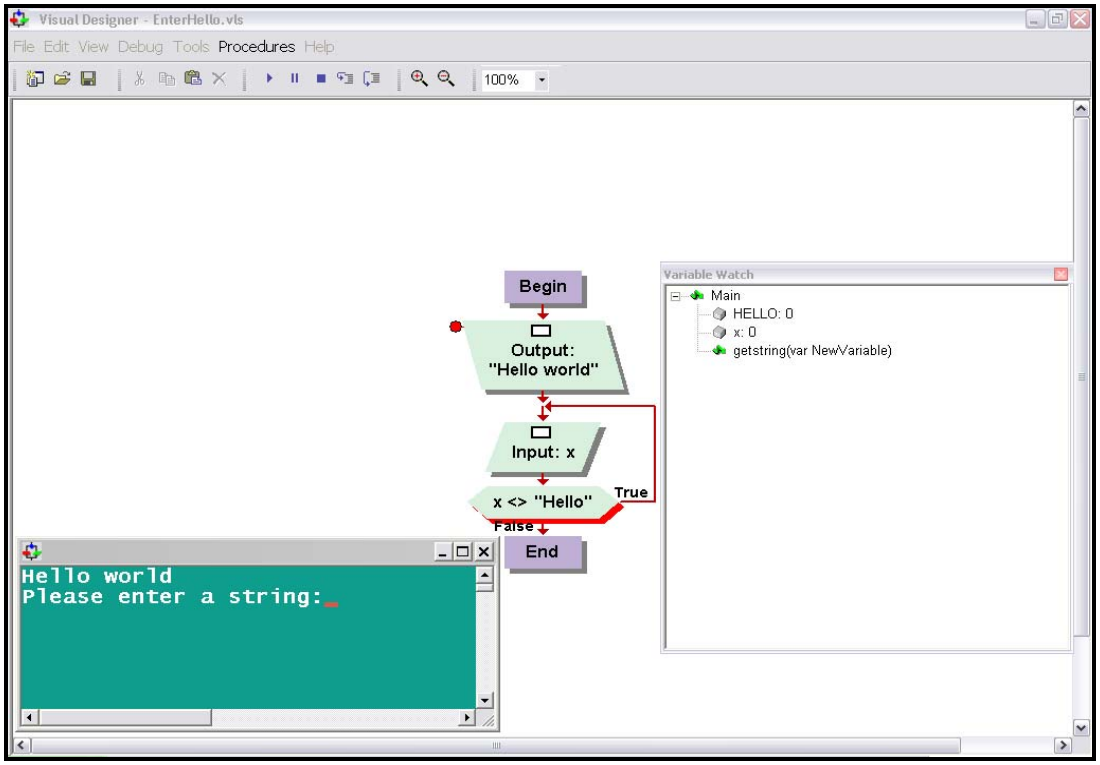
The main strength of FLINT is in the role of a support tool for novice programmers within a laboratory learning environment (Ziegler *et al.* 1999). An evaluation of the use of FLINT with novice programmers suggests that instructional benefits exist for novice programmers who use the interactive flowcharts provided by FLINT to emphasise logic and design early in an introductory programming course (Crews 2001; Crews *et al.* 2002). There is, however, an argument that FLINT does not have the capability to support a complete range of programming constructs (Garner 2003).

A general argument that is also applicable to FLINT is that flowcharts that are too large to be displayed wholly on a screen offer poor visibility and lead to discontinuities of focus of attention. This is apparent as the novice programmer scrolls the view window to a different part of the flowchart, thereby becoming distracted from the problem-solving process (Blackwell *et al.* 1999b). Flowcharts themselves, however, have been questioned as being an effective means of aiding program solution comprehension (Waddel *et al.*

1988; Warren 2003). They are prone to allow too many structural deviations (Meyer *et al.* 2000) and could thus become just as unstructured as equivalent textual programming notation program solutions (Curtis 1981).

The initial argument against the use of flowcharts had little to do with the flowchart design technique itself and was more concerned with the tediousness involved with maintaining modifications to them manually (Curtis 1981). FLINT, as a flowchart simulator tool, is an attempt to address this issue.

The most recent successor to FLINT, Visual Logic (Crews 2003), continues the support for the development of structured flowcharts to represent program solutions. Testing and debugging of program solutions is supported through a built-in flowchart interpreter, using a graphical trace to step through the flowchart one programming construct at a time. The system supports the exporting of flowcharts to a number of traditional textual-based programming notations. An example of the interface presented by Visual Logic appears in Figure 3.11.



HELLO: 0
x: 0
getstring(var NewVariable)

```
Hello world
Please enter a string:_
```

*Figure 3.11: Visual Logic*

As with FLINT and its successor, Visual Logic, RAPTOR permits novice programmers to compose and execute program solutions within the same development environment (Carlisle *et al.* 2004). Although both environments support a similar visual programming notation, the difference between the two development environments is that RAPTOR allows novice programmers to compose program solutions incrementally whereas FLINT and Visual Logic enforce composition on program solutions in a top-down fashion. RAPTOR ensures that it is not possible to compose a syntactically incorrect flowchart. In an experiment comparing the performance achievement of novice programmers using RAPTOR with others using a conventional textual programming notation, it was observed that the former group of novice programmers outperformed the latter group[25].

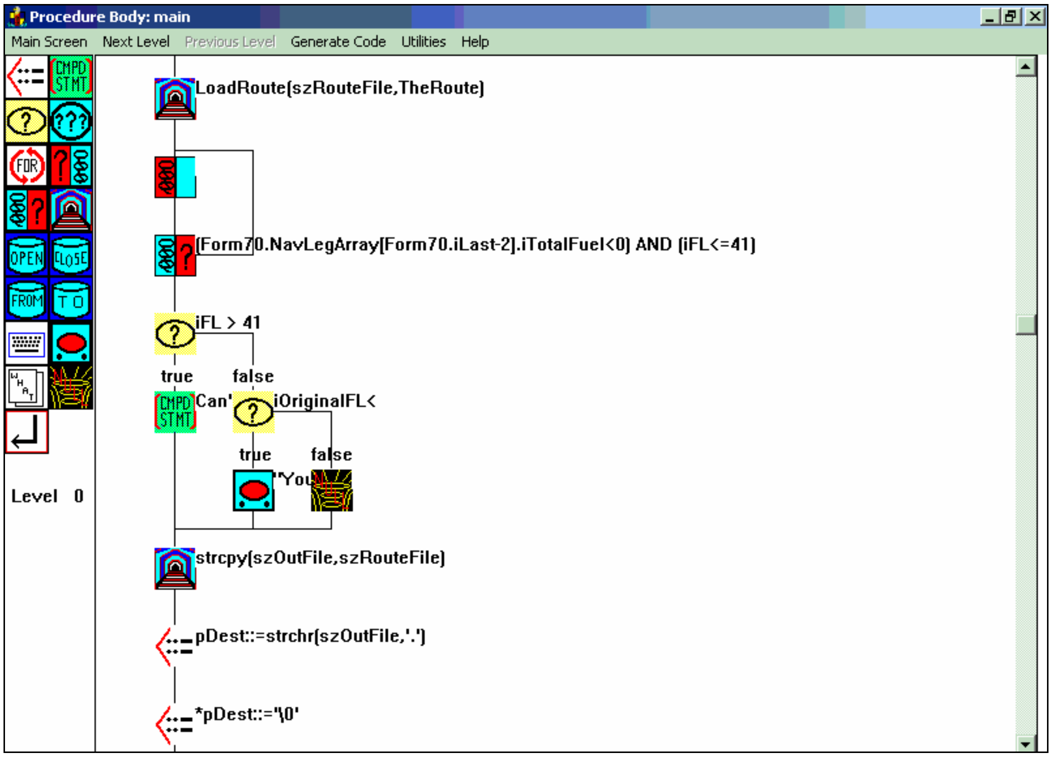
### Iconic Programming Notations

Iconic programming notations are visual notations where each visual sentence is a spatial arrangement of icons (Chang *et al.* undated). The notation is based upon a vocabulary of icons where each icon has a specific meaning. Further semantics are conveyed via the spatial arrangement of the icons.

Iconic programming notations attempt to simplify the programming process by reducing the level of precision and manual typing usually required in equivalent textual programming notations to successfully design and implement program solutions in the program domain (Calloni 1998). This type of notation is potentially more intuitive and comprehensible to programmers than conventional textual programming notations because computing concepts are presented in a graphical fashion (Tanimoto & Glinert 1986). Examples of iconic programming notations used by novice programmers in an introductory programming course are the **B**en **A**. **C**alloni **C**oding **I**conic **I**nterface (BACCII©) (Calloni *et al.* 1994, 1995, 1997; Calloni 1998) and the **SI**mple **VI**sual **L**anguage (SIVIL) (Materson *et al.* 2001). Two other examples of iconic programming notations that are not discussed further in this section, Youngster (Studer *et al.* 1995) and **E**mpirica **C**ontrol (EC) (Lavonen *et al.* 2003), have also been successfully used in the teaching of programming.

[25] The detailed findings of the experiment are revisited in Chapter 8 of this thesis.

Programming constructs in BACCII© are represented by means of graphical images. The creation of program solutions is completed by a series of point, click, drag-and-drop actions by the novice programmer (Calloni 1998). The novice programmer is assisted by intuitive screens to construct a syntactically correct flowchart representing the solution to a given problem. A textual programming notation program solution corresponding to the syntactically correct flowchart may be generated upon completion of the construction of the said flowchart (Calloni *et al.* 1995). Figure 3.12 depicts a program solution in the BACCII++© version 1.50 iconic programming notation.



```
LoadRoute(szRouteFile,TheRoute)
(Form70.NavLegArray[Form70.iLast-2].iTotalFuel<0) AND (iFL<=41)
iFL > 41
Can'
iOriginalFL<
"You
strcpy(szOutFile,szRouteFile)
pDest::=strchr(szOutFile,'.')
*pDest::='\0'
```

*Figure 3.12: BACCII++© version 1.50 Iconic Programming Environment (Calloni 1998)*

Empirical testing has shown a significant improvement in the results obtained by novice programmers in an introductory programming course using the BACCII© iconic programming notation (Calloni *et al.* 1994, 1995, 1997). A conclusion made by the investigators was that the novice programmers benefited from being exposed to the syntactically correct textual programming notation program solutions generated by BACCII©. This conclusion was due to the fact that the group of novice programmers using only BACCII© outperformed the group using only the PASCAL textual programming notation with respect to PASCAL textual programming notation specific examinations[26].

A negative aspect of BACCII© is the requirement for novice programmers to simultaneously function in two independent programming development environments when converting an iconic program solution to an equivalent textual programming notation program solution. Novice programmers are required to generate equivalent syntactically correct textual programming notation program solutions from a previously implemented iconic program solution from within BACCII©. The textual programming notation program solution must then be compiled, tested and debugged in an independent conventional textual programming development environment (Calloni *et al.* 1994, 1997; Crews 2001). The result of this feature is that all error messages and debugging requirements are communicated to the novice programmer using the syntax of the conventional textual programming notation rather than the familiar iconic program solution representation (Crews 2001).

Another iconic programming notation, SIVIL, makes use of labelled icons to teach introductory programming concepts (Materson *et al.* 2001). A visual program solution in SIVIL consists of boxes and arrows to support the control flow paradigm. SIVIL has been proposed as an educational support tool to supplement a traditional textual programming notation.

Based on the preceding discussions on visual programming notations, the following section elaborates on the support provided by such a programming notation in response to the novice programmer requirements derived in Chapter 2.

### Visual Programming Notation Support for Novice Programmer Requirements

In a comparison of visual programming notations, it was observed that some forms of visual program solutions are not easy to read (Hansen *et al.* 1994). The problem is attributed to a lack of standardisation on computational icons and their associated meanings. It was also shown that the use of well selected icons can aid in the

[26] The detailed findings are revisited in Chapter 8 of this thesis.

comprehension of a program solution while the use of badly selected icons can lead to confusion (Tanimoto *et al.* 1986; Koelma *et al.* 1992; Chang *et al.* 1999). Novice programmers using visual programming notations have, however, illustrated different experiences when using alternative visual programming paradigms.

A study that compares novice programmer comprehension of visual program solutions in one of two paradigms, namely the control flow and dataflow paradigms, suggests that specific properties of visual programming notations result in differences in program solution comprehension (Good & Brna 1999; Good 1999; Oberlander *et al.* 1999). The control flow paradigm appeared to be associated with faster program task performance, with the dataflow paradigm being associated with the abstract functional descriptions of program solutions. The study by Good *et al.* (1999) produced evidence that novice programmers were more inclined towards the control flow programming paradigm.

Regardless of the programming paradigm supported, visual programming notations suffer from shortcomings. One major disadvantage of any visual programming notation is that there exists a limitation on the number of visual programming constructs being displayed concurrently on the screen during the development of a program solution (Koelma *et al.* 1992; LaLiberte 1994; Whitley 1997). This limit is somewhat lower than the restriction existing in a conventional textual programming notation (LaLiberte 1994; Whitley 1997). The impact of this restriction is that visual programming notations could be viewed as impractical if the programming development environment does not allow the viewing of sufficient programming constructs on the screen at the same time (Whitley 1997). Combining the visual programming notation with a textual programming notation in a single program solution has been an attempt to address this limitation (Koelma *et al.* 1992). A further disadvantage of visual programming notations is that they tend to be less obviously related to natural language (Blackwell 1996).

Despite disadvantages, there exists an argument that a programming notation that uses two-dimensional diagrams and icons is often more easily understood by novice programmers than a conventional textual programming notation (Koelma *et al.* 1992; Cranor *et al.* undated). There is also evidence that imagery positively influences the mental representations of programmers when comprehending program solutions (Navarro-Prieto *et al.* 1999, 2001). Based on the positive effects of imagery in program solutions, it has been recommended that by adopting a model-based approach using, for example, a flowchart simulator, to teach introductory programming concepts to novice programmers will enhance their ability to think and reason formally, develop program solutions rigorously and conceptually program better (Roussev 2003).

Visual programming notations, however, are still dependent upon some form of syntax (Schiffer *et al.* 1995). Consequently, translating a large collection of textual programming constructs into a large collection of visual programming constructs merely replaces textual complexity with visual complexity (Freeman *et al.* 1996; Shannon 2003). In related research, Pandey & Burnett (1993) have concluded that composing a program solution in a visual programming notation is at least as easy as composing one in a conventional textual programming notation.

| Requirements for Novice Programmer Technological Support | Supported |
|---|---|
| ***R1***: Elimination of finer implementation details typically found at the superficial learning level of the program domain | ✓ |
| ***R2***: Increased level of program solution comprehension at the in-depth learning level of the program domain | ✓ |
| ***R3***: Increase in level of motivation when using the programming notation | |
| ***R4***: Designed specifically for use by novice programmers | ✓ |
| ***R5***: Provision of visual techniques to aid comprehension process at the in-depth learning level of the program domain | ✓ |
| ***R6***: Support for reduced mapping between the problem and program domains | ✓ |
| ***R7***: Increased focus on problem-solving | ✓ |
| ***R8***: Increase in novice programmer performance achievement measured in terms of higher level of accuracy in program solutions in program domain | ✓ |

*Table 3.7: Support for Novice Programmer Requirements by Visual Programming Notations*

In the context of the preceding discussion on various visual programming notations and their associated programming development environments, Table 3.7 summarises the support provided by visual programming notations as technological support in the program domain for novice programmers, according to the requirements derived in Chapter 2.

A review of the literature available on the use of visual programming notations in introductory programming courses provides evidence to deduce that the visual programming notation category in the role of programming notation and development environment supports all of the novice programmer requirements except for requirement *R3* (Table 3.7).

## 3.5 Conclusion

Programming tools are important, if not crucial to the programming activity. It is clear from a review of the literature that there exists much dissatisfaction with the manner in which conventional textual programming notations and their associated development environments fail to provide the necessary support for novice programmers (Section 3.3).

| **Requirements for Novice Programmer Technological Support** | **Supported** |
|---|---|
| ***R1***: Elimination of finer implementation details typically found at the superficial learning level of the program domain | ✘ |
| ***R2***: Increased level of program solution comprehension at the in-depth learning level of the program domain | ✘ |
| ***R3***: Increase in level of motivation when using the programming notation | |
| ***R4***: Designed specifically for use by novice programmers | ✘ |
| ***R5***: Provision of visual techniques to aid comprehension process at the in-depth learning level of the program domain | |
| ***R6***: Support for reduced mapping between the problem and program domains | ✘ |
| ***R7***: Increased focus on problem-solving | ✘ |
| ***R8***: Increase in novice programmer performance achievement measured in terms of higher level of accuracy in program solutions in program domain | |

*Table 3.8: Support for Novice Programmer Requirements by Conventional Textual Programming Notations*

An extensive review of available literature emphasises that conventional textual programming notations fail to support many of the novice programmer requirements (Table 3.8). At the same time, the literature review is unsuccessful to provide evidence of support for any of the novice programmer requirements in the framework derived in Chapter 2. Despite the evidence of discontent, this type of programming notation and development environment remains the most widely used in the learning environment of introductory programming courses.

The amount of dissatisfaction in the use of conventional textual programming notations and their associated development environments is also evident from the many types of alternative programming notations and development environments that have been explored internationally as technological support in the learning environments for novice programmers. There is, however, no evidence of a single type of alternative experimental programming notation and associated development environment being widely accepted for use.

Despite much active research in the area of novice programmer programming notations and development environments over the past decade, it is clear that there remains a requirement for a programming notation to fully support the mental model of the novice programmer, the requirements for which are derived in Chapter 2 (Table 2.1 duplicated as Table 3.1). A novice programmer programming notation should match the task of the novice programmer. This task primarily involves the comprehension and composition of small program solutions. In aiding the novice programmer in this task, the necessary skills for the programming of more complex program solutions are developed.

The programming notation and development environment should strive to be simple by providing support for only the most basic programming concepts and avoid the tendency of complexity found in so many modern textual programming notations and associated development environments. Such a programming notation should be sensitive to the current domain knowledge of the novice programmer and make use of this knowledge as a foundation for creating new programming knowledge, specifically encouraging the skill of “chunking” programming constructs. Finer and mundane implementation details should be separated from the programming concepts and concealed until the novice programmer has developed the necessary skills to manage them with more ease. In so doing, the programming notation should provide a scaffolding learning environment that allows the novice programmer to gradually educate themselves about the syntax, semantics and applications of the programming notation via the supports provided.

The novice programmer programming notation must encourage the secure comprehension of programming concepts. This can be ensured by means of a rigid programming notation that is easy to read as well as providing facilities for the novice programmer to create a program solution, watch the result of the program solution being executed and read the programming statements that caused the production of the results, all within the same integrated development environment. Studies have shown that flowcharting techniques, being programming notation independent, support higher level learning activities in the program domain and provide instructional benefits by assisting in the comprehension of the decision structure of program solutions.

There is also evidence that novice programmers are suited to a control flow sensory notation, of which a visual programming notation is an example. This observation is strengthened by the fact that an extensive literature review of alternative educational programming notations revealed that the category of visual programming notations is most responsive to the novice programmer requirements derived in Chapter 2 (Table 3.7).

The design and implementation of B# has been influenced by the discussion contained in this chapter. B# is an experimental visual iconic programming notation for novice programmers that is integrated with that of a textual programming notation. The details regarding the design and implementation of B# for use in the current investigation is discussed in Chapter 5.

In preparation for the discussion on the methodology (Chapter 6) applied during the current investigation using the experimental instrument described in Chapter 5, Chapter 4 summarises related research on the selection and placement of introductory programming students. The focus of Chapter 4 is on the results of research used to date by the Department of CS/IS at UPE, since the subjects of this empirical investigation have been exposed to this procedure of pre-selection.

# Chapter 4

# Predictive Models and Selection

## 4.1 Introduction

Chapter 1 highlights the strategies that improve and maintain satisfactory throughput in introductory programming courses (Figure 1.2). These strategies are not restricted to the approach of the current investigation that modifies the course presentation technique to cater for those candidates who are not successful. The other approach involves the selection of introductory programming course candidates from those applicants most likely to succeed (Wilson *et al.* 1985; Austin 1987). This latter approach forms part of the ongoing research at UPE to investigate strategies that elevate the successful completion percentage of candidates of already over-subscribed introductory programming courses without reducing the quality of the course (UCAS 2000; Boyle *et al.* 2002). This chapter consequently focuses on the latter approach.

Poor achievement in introductory programming courses has in the past been related mainly to a combination of low programming ability and negative attitudes towards computers. International and national research[27] in the area of effective predictive models and the selection of introductory programming course students prompted the implementation and assessment of a computerised selection method in the Department of CS/IS at UPE since 2001 (Greyling *et al.* 2002; Greyling *et al.* 2003). Despite the

[27] (Calitz 1984; Mayer *et al.* 1986; Evans & Simkin 1989; Calitz *et al.* 1992; Calitz 1997; Greyling 2000)

continued research, the situation of poor achievement persists (Greyling *et al.* 2002; UPE 2002; Greyling *et al.* 2003; Naudé *et al.* 2003; UPE 2003a).

There are many reasons for the importance of being able to predict an individual's potential for mastering computer concepts in the tertiary education environment, with specific reasons (Evans *et al.* 1989) being the:

- ability to classify enrolment applicants;
- identification of CS/IS programming majors;
- identification of productive programmers;
- improvement of introductory programming course learning activities;
- determination of the continued validity of documented predictors of computer competency; and the
- exploration of the relationship between programming abilities and other cognitive reasoning processes.

The main aim of selection for tertiary education programmes is typically to identify students who will succeed in a specific academic programme, since no tertiary education institution can afford to spend large sums of money on large numbers of high risk students (Huysamen 1997). Further, allowing inadequately prepared students accessibility to higher education results in a negative impact on the quality of learning and teaching, as well as on the success rate (Harman 1994).

Any selection process must be shown to be valid in terms of its fairness, effectiveness and efficiency (Zaaiman *et al.* 1998). An effective selection mechanism will select a high percentage of successful students and reject as few as possible of the students who could have been successful if they had been selected. Such mechanisms usually are of the form of an aptitude test administered to prospective students. Many of these tests have a mathematical focus, since there is a belief that the concepts which a student has to comprehend in order to master mathematics problems are similar to those for programming, and consequently mathematics aptitude is thus often a pre-requisite for acceptance into an introductory programming course (Byrne *et al.* 2001).

Early studies (Fowler *et al.* 1981; Konvalina *et al.* 1983; Butcher *et al.* 1985) in the prediction of success in introductory programming courses focussed on establishing associations between academic performance and success in programming in order to define instruments to be used in the selection of students. This chapter overviews these documented studies (Section 4.2), at both the international and national levels. The focus is, however, on research conducted subsequent to that reviewed in Greyling's (2000) study.

The chapter includes a discussion on selection techniques, specifically computerised selection techniques, used to classify applicants for introductory programming courses (Section 4.3). This section concentrates on the implementation and assessment of the computerised selection technique applied in the Department of CS/IS at UPE since 2001 (Greyling 2000; Greyling *et al.* 2002; Greyling *et al.* 2003).

The limitation of the research and how it necessitated the current investigation is highlighted. The selection technique applied at UPE is especially relevant to the current study since subjects of the current investigation have been exposed to the strict application of the selection technique prior to the current study, and are thus controlled by the selection process.

## 4.2 Predictors for Success in an Introductory Programming Course

Research on predictors for success in introductory programming courses has predominantly concentrated on programming aptitude and other cognitive skills. Computer aptitude tests, specifically, have been used in industry since the 1960's for the selection of productive programmers. These aptitude tests were used with varied success in predicting achievement in programming courses (Mazlack 1980; Goldstein 1987).

Research was prompted during the 1980s into the identification of variables that could be used as predictors of success in computer programming. As a result of this research, the skills cited for success in learning to program are problem-solving, language ability and mathematical ability. Supplemental characteristics cited are that of gender, learning style, motivation and prior computing experience.

An overview of the documented international research projects that have been conducted on this topic is presented (Section 4.2.1). A discussion on the comparative national research follows (Section 4.2.2). Although reviewing a large volume of documented research projects relevant to predictors of success in computer programming, the focus is on related research that has been conducted since the most recent research undertaken in the Department of CS/IS at UPE (Greyling 2000).

### *4.2.1 International Research*

International research projects that investigated predictive models and the selection of introductory programming students were initiated during the 1960's by studies of the application and assessment of **computer aptitude tests**. Investigations that followed these initial studies concentrated on the identification of variables that were related to programmer performance. One of the first variables identified in the late 1960's as being relevant was that of **mathematical ability**.

Research on the identification of variables that bore a significant relationship to programmer performance continued, with a number of non-academic variables being identified. The variables identified as being relevant to programmer performance include **language ability and problem-solving ability**, **gender**, **learning style**, **personal motivation** as well as **prior computing experience**. Research related to each of these variables is overviewed in the following 6 subsections.

#### <u>Computer Aptitude Tests</u>

The research projects conducted mainly during the 1960s that are related to the use of specific programmer aptitude tests for the selection of prospective programmers are cited in Koubek *et al* (1985). Although these aptitude tests were often used for predicting success in programming courses, they were originally designed for the purpose of selecting and appointing the better trained programmers.

The most accepted of these tests, the IBM **P**rogrammer **A**ptitude **T**est (PAT) was devised from a form originally developed by Hughes and McNamara in 1955. The test assessed reasoning ability (Mazlack 1980), and by 1962 it was the most popular tool used by business and industry for indicating programming skills (Koubek *et al.* 1985). Varied success was however observed in studies assessing the level at which this programmer aptitude battery could predict success in a programming course (Mazlack 1980; Goldstein 1987).

An aptitude test that is often used as the primary recruiting measure for tertiary education students in the United States of America (USA) is the **S**tandard **A**ptitude **T**est (SAT). SAT is an aptitude test that focuses on measuring verbal and mathematical abilities independent of specific courses or high school curricula (Atkinson 2001). Recently a need has been identified for a standardised tertiary education entrance test that is a curriculum based achievement test instead of an aptitude test like SAT, so that an applicant can be assessed in full complexity and not only on grades and test scores. The motivation for this requirement is that achievement tests are perceived as being fairer to students because these types of tests measure accomplishment rather than imprecise concepts of aptitude.

Mathematical Ability

Towards the end of the 1960s, one of the major skills identified as being indicative of success in programming was identified as being that of mathematical ability (Bauer *et al.* 1968). This finding was reinforced by several other studies that followed during the 1970s cited by Stephens *et al.* (1981) as well as studies in the 1980s[28] and more recently[29].

Despite these findings, related research in the United Kingdom found that there was no evidence that a prior mathematical qualification influenced the achievement in an ultimate programming qualification (Boyle *et al.* 2002). The observation by Boyle *et*

[28] (Konvalina *et al.* 1983; Campbell & McCabe 1984; Danial 1985; Koubek *et al.* 1985; Darius 1987; Loftin 1987)

[29] (Byrne *et al.* 2001; Wilson *et al.* 2001; Jenkins 2002)

*al*. is confirmed by a more recent study into the predictors of success for an object-oriented introductory programming course (Ventura 2003).

Language Ability and Problem-solving Ability

A further variable identified as being a contributor to achievement in computer programming is that of language ability.  Documented studies have respectively linked an individual's command of home language with the ability to program (Sauter 1986), and the ability to speak a second language as a predictor of success in computer programming (Evans *et al.* 1989).

Related studies in the identification of predictors for performance achievement in programming courses have also identified the skill of problem-solving as being relevant.  This skill was found to consist of a number of specific competencies, these competencies being the ability to:

- reason abstractly (Mayer *et al.* 1986; Chen & Vecchio 1992; Chmura 1998; Boyle *et al.* 2002);
- follow a sequence of instructions (Mayer *et al.* 1986; Chen *et al.* 1992; Chmura 1998);
- deduce a suitable result from serially presented information (Mayer *et al.* 1986; Chen *et al.* 1992);
- memorise and recall many details (Chmura 1998);
- visualise information (Mayer *et al.* 1986; Chmura 1998); and
- articulate thoughts (Mayer *et al.* 1986; Chmura 1998).

An early study  concluded that the three competencies of articulation of thoughts, the ability to deduce a suitable result from serially presented information and the ability to follow a sequence of instructions were the better predictors of success in computer programming (Mayer *et al.* 1986).

Gender Differentiation

The following explanations for observed gender performance differences are provided from the many international studies[30] reporting on gender differentiation as being important in the performance achievement in programming:

- males, when compared with females, possess a stronger sense of self-efficacy when working with computers (Howell 1993);
- there is a higher probability for males to have self-initiated prior experience with computers (Howell 1993; Rowell *et al.* 2003);
- the existing male domination makes females feel uncomfortable in lecture learning activities (Howell 1993);
- males tend to register in greater proportions for curricula in which lower average percentage marks are recorded (Rowell *et al.* 2003);
- male students' personalities were more closely matched to the area of computer programming than the female students (Haliburton 1998); and
- structured programming techniques conflict with females' naturally creative and communicative personalities (Howell 1993).

Contradictory research has however been cited by Chen *et al.* (1992) and more recently by Ventura (2003). Chen *et al.* determined that either there was either no gender difference in attitudes towards computer systems, or females tend to have relatively adverse attitudes. Ventura confirmed that no gender bias existed when predicting success for an object-oriented introductory programming course.

A related Australian study reports a finding of a higher participation of Asian females in information technology (IT) education than other Western female students (Nielsen *et al.* 1997). This observation seems to indicate that there exists a relationship between a student's cultural environment and the way in which IT education is perceived.

[30] (Howell 1993; Haliburton 1998; McKenna 2000; Carter *et al.* 2001; Quaiser-Pohl & Lehmann 2002; Rowell *et al.* 2003)

### Learning Style

Learning style can be classified according to four adaptive learning modes, namely concrete experience, reflective observation, abstract conceptualisation and active experimentation (Byrne *et al.* 2001). It is suggested that different learning styles are suited to specific learning environments, depending on whether a subject is conceptual or practical oriented. The tasks of learning to program incorporate these categories of environments, depending on whether a novice programmer is trying to solve a problem, apply skills or understand and identify the relationship between concepts. Consequently different learning styles are evident during the general programming process.

An Australian study found that science (including introductory programming) students experiment more actively when learning to program (Shute 1991). Another study determined that the traditional style of lecturing and conventional textual programming notations used in an introductory programming course was probably favouring novice programmers with certain kinds of learning style preference (Thomas *et al.* 2002). The study also determined that novice programmers who could be categorised as having active, sensing or visual learning styles would be disadvantaged by these traditional course presentation methods.

The emergence of learning style as a statistically significant explanatory variable in the success in an introductory programming course was also observed in a study by Evans *et al* (1989). The conclusion of the study was, however, that the task of finding effective predictors of computer proficiency remained incomplete.

In a subsequent study on learning style, no significant conclusion could be drawn (Byrne *et al.* 2001). The investigators however observed a higher proportion of a convergent learning style among students as well as a higher incidence of success amongst students exhibiting this type of learning style within the overall group.

### Personal Motivation

Personal motivation has also been identified as a predictor of success in an introductory programming course in various studies (Goold & Rimmer 2000; Jenkins 2001a; Wilson *et al.* 2001; Chamillard *et al.* 2002; Rountree *et al.* 2002). One of these studies concluded that the strongest single indicator of success in an introductory programming course was a positive attitude towards the introductory programming course (Rountree *et al.* 2002). An earlier study (Wilson *et al.* 2001) determined that of the 12 variables investigated as being predictive of success in an introductory programming course, the two positive predictive variables in order of importance were motivation and mathematical ability.

In attempting to determine the motivation of a novice programmer in an introductory programming course, it is possible to observe the novice's behaviour and from that deduce their probable motivation, but it is never possible to be certain (Jenkins 2001a). Three categories of motivation have been described (Entwisle 1998), the categories being extrinsic, intrinsic and achievement types of motivation.

Extrinsically motivated students possess an aspiration to complete the introductory programming course in order to achieve some expected reward. Such a student will probably not do much for which there is no summative evaluation recognition. The primary motivator for a student falling into this category of motivation is the career and associated rewards that will follow from successful completion of the course (Jenkins 2001a).

Intrinsically motivated students are motivated by a curiosity in the course. Such students could be expected to research the course topics since they tend to perform more on their own initiative. The primary motivator for an intrinsically motivated student is a profound interest in programming itself (Jenkins 2001a). In some cases, students are intrinsically motivated in order to please some third party whose opinion is valued or for fear of failure. This type of motivation has also been referred to as social motivation (Biggs 1999).

Students who are motivated by achievement tend to be in competition with their peers and will implement strategies that will ensure their best performance in the form of the highest recognition, usually in the form of marks. The primary motivator for an achievement motivated student is to perform well for personal satisfaction (Jenkins 2001a).

Various methods have been applied to increase the motivation of novice programmers in an introductory programming course (Astrachan 1998; Chamillard *et al.* 2002). Anecdotal evidence supports the benefits of integrating the documented strategies, for example the use of graphics in an introductory programming course (Chamillard *et al.* 2002).

Prior Computing Experience

Some of the studies on the relationship between prior computing experience and success in an introductory programming course report that a prior programming course is indeed a predictor for success in an introductory programming course (Newman *et al.* 1999; Hagan & Markham 2000; Morrison & Newman 2001; Wilson *et al.* 2001; Boyle *et al.* 2002; Rountree *et al.* 2002). One specific study determined that novice programmers who had taken at least one prior programming course had a successful outcome rate of 70% compared with a rate of 52% for those who had no prior programming course (Newman *et al.* 1999). A more recent study reported similar conclusions yet indicated that experience in a prior programming course was not a guarantee for success in an introductory programming course (Rountree *et al.* 2002).

A study by Morrison *et al*. (2001) determined that the strongest relationship between prior programming experience and success in an introductory programming course was when the prior programming course was offered at tertiary level when compared to junior college and secondary school level. An earlier study also determined that the more programming languages a novice programmer has experience with, the better the performance in a tertiary level introductory programming course (Hagan *et al.* 2000).

The finding that prior computing experience is a predictor for success in an introductory programming course is contradicted by a recent study conducted by Ventura (Ventura 2003). In his study, Ventura determined that prior computing experience is not a predictor for success in an object-oriented introductory programming course.

### *4.2.2 National Research*

National research in the prediction of success, specifically with respect to introductory programming courses, is limited in volume. Reports related to the prediction of success, but not necessarily in the area of introductory programming courses, and also not originating from UPE, include those relevant to gender differentiation (Huysamen & Roozendaal 1999), personal motivation (Glenn 2000), relationship of English matriculation marks with performance achievement of information system majors (Nash 2003), determination of factors that distinguish achievers from at risk students (Eiselen & Geyser 2003) and predictors of first-year student success (Lourens & Smit 2003).

A review of the non-local national research undertaken specifically in the determination of predictors of success in introductory programming courses specifically is presented in this subsection. Only the studies by Huysamen *et al*. (1999) and Glenn (Glenn 2000) are relevant to this discussion. The details of the research on the prediction of success in an introductory programming course undertaken in the Department of CS/IS at UPE prior to Greyling's (2000) implementation of a computerised selection method is then discussed.

#### <u>Non-local Research</u>

A study by Huysamen *et al*. (1999) in the role of gender differential performance at tertiary level concluded that males at a South African tertiary education institution tend to register in greater proportions for curricula in which lower mean percentage marks are recorded. The observations in this study correspond with the findings in a similar but more recent international study conducted by Rowell *et al*. (2003) (Section 4.2.1). Huysamen *et al*. determined that females tended to score higher when compared to males and deduced that the choice of curriculum impacted this performance.

In a study related, an observation made by Glenn (2000) is that personal motivation plays a role as a predictor of success in an introductory programming course. The specific observation made is that South African programming students are extrinsically motivated in that many of them expect universities and technikons to be a speedy opportunity to a growing number of stimulating and lucrative computer related careers.

A large share of the remaining national research on the prediction of success in an introductory programming course is restricted to studies undertaken in the Department of CS/IS at UPE over the past two decades (Calitz 1984; Calitz *et al.* 1992; Calitz 1997; Greyling 2000; Greyling *et al.* 2002; Greyling *et al.* 2003). The focus of the following section is thus on previous research conducted in the Department of CS/IS at UPE prior to the most recent research on the compilation, validation and implementation of a suitable computerised selection battery. The most recent research (prior to the current investigation) undertaken in the Department of CS/IS at UPE is reviewed in Section 4.3.

Research at UPE

The Department of CS/IS at UPE has been conducting ongoing research into the factors that predict success in introductory programming courses since 1982. Initial research highlighted deficiencies in various computer aptitude test batteries with respect to success in introductory programming courses (Calitz 1984). Consequently, in 1992, research was conducted in the use of matriculation marks and the Swedish point system to predict success in an introductory programming course (Calitz *et al.* 1992).

The independent variables used in the aforementioned research were the Languages (namely English and Afrikaans First and Second Language), Mathematics, Science, Accountancy, Biology, an average mark of the latter three and an average mark of the remaining matriculation subjects. It was observed that 34% of the variance in introductory programming course performance could be attributed to the performance in First Language, Mathematics and the combined average of Science, Accountancy and Biology. The model proposed by this study was implemented at UPE in 1993 (De Kock 1993).

A subsequent study by Calitz (1997) provided evidence that increased the predictive variance to 62%. This study included both biographical and psychometric variables. The biographical variables included in the study were age, gender, race, home language, curriculum registered for and type of registration, namely whether part-time or full-time. The psychometric variables included in the study were obtained from the following psychological tests and questionnaires: the **S**elf-**D**irected **S**earch (SDS) (Holland 1985), the **C**areer **D**ecision **S**cale (CDS) (Osipow 1986), the Career Development Questionnaire (Langley 1992a), the Value Scale (Langley *et al.* 1992), the **S**urvey of **S**tudy **H**abits and **A**ttitudes (SSHA) (Du Toit 1983) and the Life Inventory (Langley 1992b).

Other than the observed increase in predictive variance (62%) , the most significant conclusions of Calitz' (1997) study were the following:

- Even though both the biographical variables of gender and race featured in the predication formulae, it was decided not to make use of these variables for selection purposes.
- The matriculation subjects of English, Mathematics and the combined average of Science, Accountancy and Biology were prominent as variables in the regression formulae. An important skill identified was that of English proficiency. This conclusion differed from the conclusion in the earlier study that First Language, being either English or Afrikaans was identified as a factor for prediction (Calitz *et al.* 1992).
- The inclusion of the *Social* personality type on the SDS psychometric test indicated that social skills, such as communication skills and working with people, have become important qualities of successful CS/IS students.

Subsequent research in the Department of CS/IS at UPE concentrated on the compilation, validation and implementation of an effective and appropriate computerised selection battery for introductory programming students (Greyling 2000; Greyling *et al.* 2002; Greyling *et al.* 2003). The focus of the following section is an overview of the progress of this research.

## 4.3 Computerised Selection in Introductory Programming Courses

In a study conducted by Greyling (2000) in the Department of CS/IS at UPE, the skills measured in the selection battery were mathematical and language ability, spatial and planning ability as well as computer proficiency. A number of different computerised instruments were used in the investigation, namely the ACCUPLACER Computerised Placement Tests (ACC 1997), Computerised Logo System (Hunt 1998), Computerised Mazes System (Roos 1998) and Interactive Learner (Streicher 1998).

Greyling's investigation (2000) concluded that three of the ACCUPLACER Computerised Placement Tests produced the most significant results for an introductory programming course and accounted for 63% of the variance in performance. These tests were namely the reading comprehension, arithmetic and elementary algebra tests.

Analyses conducted on the matriculation subject variables of Mathematics, English Language and the combined average of Science, Accountancy and Biology confirmed the finding of Calitz in an earlier study (1997). Greyling's study (2000) also found that, despite the confirmation that black matriculation results were not good predictors of success in CS/IS introductory programming courses, the matriculation results complemented the results derived from the computerised selection battery. Consequently, research results showed that matriculation results used in conjunction with placement tests continue to play a role in selection and admission strategies at tertiary education institutions.

Obvious advantages were observed from both the administrative and student viewpoints, namely immediate scoring of tests for the former and, for the latter, a more positive attitude than that experienced in traditional pen-and-paper tests. The latter observation is supported by a more recent South African study that examined the effects of a changing society and technology on the way that learners interact with information in an educational environment (Miller 2003). From this more recent study it is evident that learners are motivated by the technology used in information transfer.

Greyling's study (2000) determined that the ability to measure the elapsed time and computer proficiency contributed significantly to the predictive validity of the selection battery for a CS/IS introductory programming course. Greyling concluded further that the combination of the selection battery and matriculation subjects accounted for 62% of the variance in performance in an introductory programming course, a slight decrease (1%) on the variance observed when using the selection battery alone.

Greyling's streaming model was implemented at UPE as from the beginning of 2001 based on the selection results. The model concentrated on the placement of students within alternative introductory programming courses, namely the customary introductory programming course and an extended introductory programming course (Greyling *et al.* 2002; Greyling *et al.* 2003). The extended introductory programming course covers the same material as the customary course but over a longer period of time.

The implication of different learning programmes for different groups of students as categorised by the computerised selection battery is that support mechanisms need to be in place in each of these different programmes. This is in agreement with the strategy adopted by the UPE Placement Task Team (Foxcroft *et al.* 1999).

The greatest risk to the implementation of alternative learning programmes for different groups of students was identified as being the fact that streaming could generate some opposition from students as well as parents. The expected opposition was due to the fact that it would in effect mean for some students that their degree programme would be extended by a further year (Greyling *et al.* 2003). Due to the endeavour on the part of the Department of CS/IS at UPE to communicate the justification of the streaming process, most parents were positive and in fact thankful that such a support mechanism was in place.

The implementation of Greyling's streaming model in 2001 provided an opportunity for further validation of the model. It was observed that the regression formula was successful in predicting the students' final results (Greyling *et al.* 2003). This was apparent in the finding that the actual average mark (69%) differed only slightly from the predicted average mark (66.7%) for a customary introductory programming course.

Furthermore, the pass rate for first time students in the introductory programming course increased to 77% (Greyling *et al.* 2003). A pass rate of 68% was evident in the extended introductory programming course and the actual average mark (59.8%) exceeded the predicted average mark (49.5%) by 10.3%. The results observed in the extended introductory programming course were encouraging as the students enrolled in the extended introductory programming course were predicted as possible failures by Greyling's streaming model.

The following limitations in Greyling's investigation were however identified:

- limited success in implementation (Greyling *et al.* 2002; Greyling *et al.* 2003), especially with respect to the performance rates with the inclusion of repeating students within the student body enrolled for the introductory programming courses; and
- the need for educational support mechanisms in each of the alternative introductory programming courses.

Although the pass rates of the streamed students increased from previously recorded pass rates, the limited success in the implementation of Greyling's streaming model was evident in that the actual pass rate of the extended introductory programming course (68%) was considerably lower when compared with internal and external expectations of 75% (Department of Education 2001; UPE 2002).

Further, the presentation of alternative introductory programming courses by the Department of CS/IS at UPE initiated the requirement for appropriate technological support in the respective learning environments. Adherence to this requirement is in accordance with the strategy adopted by the UPE Placement Task Team (Foxcroft *et al.* 1999).

The need to further improve and maintain satisfactory individual and group performance rates (Greyling *et al.* 2002; UPE 2002; Greyling *et al.* 2003; Naudé *et al.* 2003) amongst novice programmers in an introductory programming course as well as the need for educational support mechanisms in introductory programming courses (Greyling *et al.* 2002) was instrumental in initiating the current investigation into the comparison of programming notations and associated development environments for an introductory programming course at tertiary level.

## 4.4 Conclusion

Limitations on resources, specifically financial, hardware and human, necessitate the implementation of strategies that effectively maximise the throughput rate of introductory programming course students. One category of strategy is the selection of students with the highest potential to succeed in an introductory programming course. The second type of strategy is to modify course presentation techniques. The latter type of strategy is the category into which the current investigation is classified.

Predicting the achievement potential of a novice programmer in an introductory programming course is essential to ensure a satisfactory level in both individual and group performance rates, as well as improve the personal motivation of introductory programming course students. This chapter provided an overview of the international and comparative national research in the selection of candidates for introductory programming courses.

Based on the results of research into the predictors of success in computer programming, the Department of CS/IS at UPE initially implemented a strategy of advisory selection in 1993. Subsequent research in the Department of CS/IS at UPE resulted in the implementation of an effective and efficient computerised selection and placement model at UPE in 2001.

The streaming model implemented at UPE provided the opportunity for students to register and complete degree programmes according to the level of their potential to succeed. Consequently, alternatively paced introductory programming courses were designed and presented and students were placed into the alternative introductory programming course streams based on their performance in a computerised placement test battery. The subjects of the current investigation, namely 2003 customary introductory programming course students in the Department of CS/IS at UPE, were subjected to the rigid implementation of this computerised selection and placement process.

Despite external and internal pressure to increase and maintain satisfactory group performance rates of students in introductory programming courses, CS/IS departments at tertiary educational institutions should strive to maintain the quality of their introductory programming course while adhering to the external and internal expectations. One strategy to ensure the maintenance of the quality of the introductory programming course is with the provision of educational technological support for novice programmers. This is the second type of strategy proposed as a solution to maximise throughput in an introductory programming course.

Chapter 5 discusses the design and implementation of the experimental technological support used in the current investigation. The chapter focuses on the design and implementation of B#, an educational technological support tool that assists in the modification of the introductory programming course presentation techniques.

# Chapter 5

# Design and Implementation of an Iconic Programming Notation and Development Environment

## 5.1 Introduction

Programming tools are essential to the process of learning to program. The need for such technological support that reduces the extraneous cognitive load on a novice programmer and is included in the teaching and learning environment of an introductory programming course is investigated in Chapters 2 and 3. A number of educational programming notations and development environments have been proposed in response to this need.

A survey into the level of support for the framework of novice programmer requirements derived in Chapter 2 (Table 2.1) provided by each of a number of different categories of educational programming notations and development environments is presented in Chapter 3. The literature reviewed provides evidence that there is dispute as to how well current programming tools support the programming task for novice programmers, especially in the areas of error prevention and program comprehension.

The review of the literature reveals that there exists much dissatisfaction in the manner in which widely used conventional textual programming notations and their associated development environments provide support for novice programmers (Section 3.3). Even though the use of imagery in a programming notation and development environment is recommended based on studies of the cognitive model of the novice programmer, the literature review reveals that no single type of alternative experimental programming notation and associated development environment has been widely accepted for use in introductory programming courses. There consequently remains a requirement for a programming notation and associated development environment to sufficiently support the mental model of the novice programmer in an introductory programming course.

One factor prompting the current investigation is that strategies are required to promote the successful completion percentage of candidates of already over-subscribed introductory programming courses without reducing the quality of the course (UCAS 2000; Boyle *et al.* 2002). These strategies involve either the selection of introductory programming course candidates from applicants most likely to succeed or adjusting course presentation techniques to cater for those candidates who are not successful (Wilson *et al.* 1985; Austin 1987). The recent research undertaken, especially at UPE, in response to the former type of strategy is discussed at length in the previous chapter. The latter strategy is the strategy which uses educational technological support, much like that of B#, on whose design and implementation this chapter focuses.

This chapter describes the alternative approaches used as the technological support in the learning environment for an introductory programming course at UPE, which is the context for the current investigation (Section 5.2). For the purposes of completeness, a brief overview (Section 5.2.1) of the conventional textual programming notation and associated development environment, Delphi™ Enterprise, currently used as the prescribed technological support in UPE’s introductory programming course is included.

The focal point of the chapter is, however, on the design and implementation of B#, a locally developed experimental iconic programming notation and educational support tool for novice programmers in an introductory programming course (Section 5.3). The chapter concludes with an assessment of B# in terms of the programming notation and development environment requirements for novice programmers in the learning environment of an introductory programming course (Table 5.4). This evaluation corresponds with those presented in Chapter 3 for the categories of educational technological support reviewed.

## 5.2 Introductory Programming Technological Support at UPE

Technological support forms an integral part of the learning environment for any introductory programming course. Such a successful learning environment for an introductory programming course is described as one that satisfies the following constraints (Brusilovsky *et al.* 1994):

- The initial stages of learning to program are supported by a small, simple subset of a programming notation. As new features of the programming notation are learnt, they can be built onto the existing foundation.
- The visual appearance of a program solution structure makes it possible for the novice programmer to comprehend the semantics of the programming constructs introduced and also serves to shield them from misunderstandings.

Technological support in the learning environment of an introductory programming course in the form of visually based programming notations has become possible due to the rapid development in high speed, high resolution, large random access memory (RAM), low cost computers (Calloni *et al.* 1995). As a consequence of this observation together with the need for a successful learning environment for an introductory programming course, the selection of an experimental programming notation and associated development environment is influenced by a review of existing research (Chapter 3). The experimental technological support identified as being appropriate in the learning environment of an introductory programming course for the purposes of this investigation is classified in the category of visual programming notations (Section 3.4.6).

Visual programming notations and development environments generally minimise the mundane tasks of programming. Forms of this minimalism are ensuring that parentheses match or that lines in a program solution are terminated by a semicolon (Blackwell 1996). Visual programming notations have in the past been described as being potentially more intuitive and comprehensible to novice programmers than conventional textual programming notations because programming constructs are presented in a graphical form (Tanimoto *et al.* 1986).

The current investigation into programming notations and associated development environments in an introductory programming course focuses on the comparison of a traditional commercial textual programming notation and associated development environment with an iconic programming notation and its associated development environment. Delphi™ Enterprise (Borland 2003), a conventional commercial textual programming notation and development environment is one of the instruments of technological support considered in the delivery of the practical experience necessary for novice programmers in an introductory programming course at UPE (Section 5.2.1). The alternative form of technological support is that of B#, a locally developed visual iconic programming notation and development environment (Section 5.2.2).

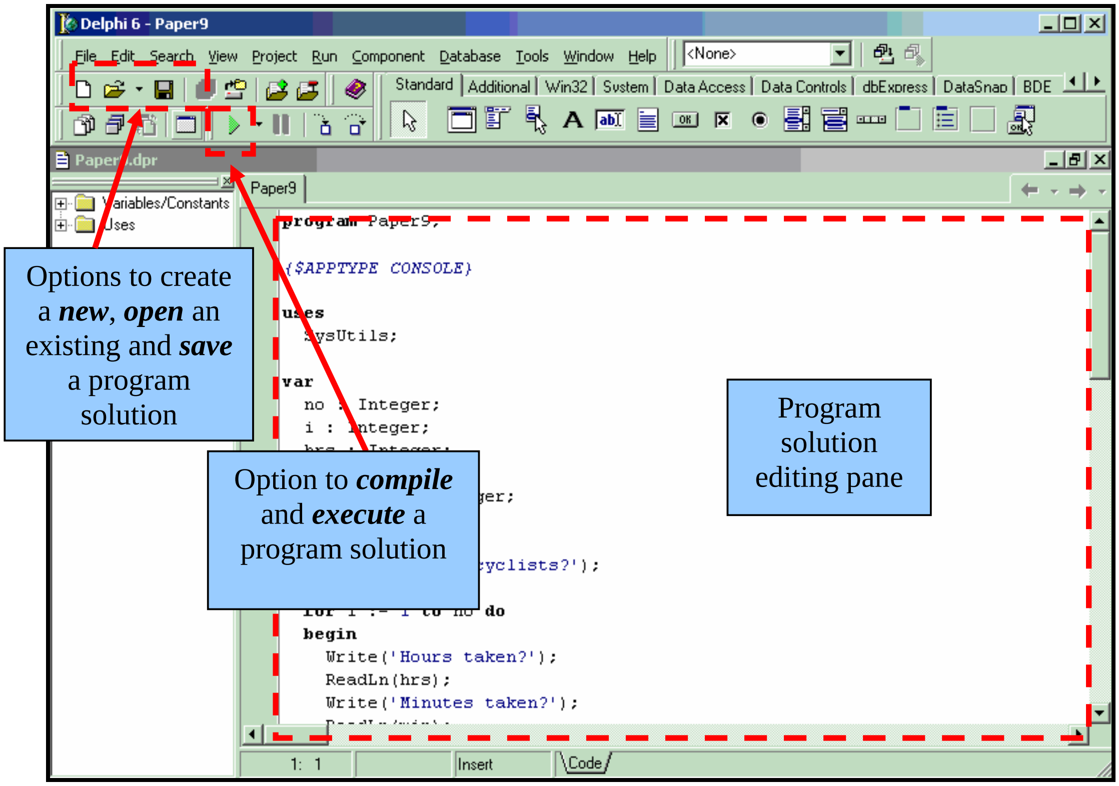
### *5.2.1 Delphi™ Enterprise: A Textual Programming Notation and Development Environment*

Delphi™ Enterprise is the programming notation and development environment currently prescribed for the introductory programming course presented by the Department of CS/IS at UPE (CS&IS 2003). The environment is based on the PASCAL programming notation which was originally developed by Niklaus Wirth specifically as an introductory programming teaching tool (Wirth 1971).

Delphi™ Enterprise is based on a simple textual programming notation with a limited grammar. One of the programming notation's features is that it is easy to learn due to strong data typing and a clear distinction between functions and procedures. An example of the interface provided by Delphi™ Enterprise is illustrated in Figure 5.1. Only those features of the development environment that are emphasised on Figure 5.1 are necessary and used by novice programmers in the introductory programming course at UPE. All other features presented by the programming development environment are superfluous.

Despite the good intentions of the programming notation, the Delphi™ Enterprise programming development environment is an example of an environment to which blame has been partly attributed for the fact that introductory programming courses are perceived as being difficult (Hilburn 1993; Calloni *et al.* 1997; Warren 2000). One reason cited is that commercial textual programming notations and their associated development environments (Section 3.3) like Delphi™ Enterprise tend to have been designed for experienced users who are developing large program solutions (Ziegler *et al.* 1999).

Debugging tools in Delphi™ Enterprise require the use of break points, which are markers placed by the programmer at specific points within a program solution. On execution of the specified sections of the program solution, the tracer is activated. Recognising appropriate programming statements at which to place break points requires the programmer to have a good comprehension of the program solution, the nature of the error present and the possible locations of the error. Furthermore, features such as the watchlist, which permits the values of variables to be displayed as the program solution is executed, are complex to initiate and use, and are often even avoided by more advanced programmers.


Delphi 6 - Paper9
File Edit Search View Project Run Component Database Tools Window Help
<None>
Standard Additional Win32 System Data Access Data Controls dbExpress DataSnap BDE
Paper9.dpr
Variables/Constants
Uses
Paper9
1: 1
Insert
Code
Options to create a ***new***, ***open*** an existing and ***save*** a program solution
Option to ***compile*** and ***execute*** a program solution
Program solution editing pane

```
program Paper9;

{$APPTYPE CONSOLE}

uses
  SysUtils;

var
  no : Integer;
  i : integer;
  hrs : Integer;
            ger;

            yclists?');
  for i := 1 to no do
  begin
    Write('Hours taken?');
    ReadLn(hrs);
    Write('Minutes taken?');
```

*Figure 5.1: Borland© Delphi™ Enterprise version 6 Textual Programming Notation and Development Environment*

In response to the challenge of increasing throughput in introductory programming courses, the Department of CS/IS at UPE identified the need for the development of an experimental iconic programming notation, B# (Brown 2001a; Thomas 2002b; Cilliers *et al.* 2003; Yeh 2003b). B# was deliberately designed to be a short term visual iconic programming notation providing initial technological support in the learning environment of an introductory programming course. A brief overview of B# is presented in the following section with the progress of the design and implementation of this programming notation and development environment being described in Section 5.3.

### *5.2.2 B#: A Visual Iconic Programming Notation and Development Environment*

Based on an intensive literature survey of educational programming notations and development environments (Chapter 3), it was determined that the category of visual programming notations was most responsive to the requirements of a novice programmer (Table 2.1 derived in Chapter 2). Further, it appears as if related international research with respect to the use of an iconic programming notation (BACCII©) in an introductory programming course has been terminated[31] after initial empirical studies (Calloni *et al.* 1994, 1995, 1997). These two observations influenced the choice of category of programming notation on which B# was originally modelled. B# consequently supports a visual programming notation, specifically one that is iconic.

A number of issues need to be considered when designing a visual programming notation and development environment for novice programmers, these issues being summarised below (Chattratichart & Kuljis 2002):

- Different programming notations highlight certain information types while hiding others (Green & Petre 1996).
- In order to reduce the extraneous cognitive load on a novice programmer, there must be a close mapping between the mental representation and the external representation of a program solution (Garner 2001). The closeness of the mapping is determined by a correspondence between the graphical representation and programming task, or a correspondence between programming constructs and the

[31] As noted earlier in Chapter 1, no further documentary evidence of follow-up studies since 1997 has been located (Calloni *et al.* 1994, 1995, 1997). The author of the research, Ben A. Calloni, is currently employed at an institution different to that where the empirical studies were originally conducted (Calloni 2002).

programmer's preferred strategy. An important element in the improvement of programming education is the support for mental model-building with efficient educational support tools (Astrachan 1998).

- Novice programmers are affected by paradigm difference. Research[32] has indicated that novice programmers prefer the control flow paradigm.
- Different representations of the same program solution may require different cognitive effort.
- Graphical representation traversal direction, namely the direction of easiest traversal, for example horizontal or vertical traversal, may affect the cognitive demand on the novice programmers' comprehension of a presented program solution.
- The comprehension of programming constructs can be enhanced by the appropriate design of the visualisation capabilities of the educational programming notation and development environment (Wright *et al.* 2000).

Iconic programming notations traditionally attempt to simplify the programming task by reducing the level of precision and manual typing usually required in conventional textual programming notations (Calloni 1998). The associated development environments also attempt to increase the speed at which problem-solving and implementation of program solutions occur.

Many issues were considered during the design and implementation of the experimental iconic programming notation and development environment, B# that was developed in the Department of CS/IS at UPE over a period of 3 years (Brown 2001a, b; Thomas 2002a, b; Cilliers *et al.* 2003; Yeh 2003a, b; Cilliers *et al.* 2004b). All successive versions of B# were required to be implemented using only the Delphi™ Enterprise programming notation and development environment. The specific design and implementation issues considered are elaborated on in the following section.

[32] (Adelson 1984; Corritore & Wiedenbeck 1991; Good *et al.* 1999; Good 1999; Oberlander, Brna *et al.* 1999; Oberlander, Cox *et al.* 1999; Chattratichart & Kuljis 2000; Chattratichart *et al.* 2002)

## 5.3 Development of B#

In defining the **scope** and **functionality** of the B# visual iconic programming notation and development environment, decisions had to be made concerning the programming structures and generated textual programming notations that were to be supported. Furthermore the task model had to be implemented in such a way that B# could be used as an effective teaching tool which implements a strategy that emphasises the reading of good examples of program solutions (Fincher 1999; Kolling & Rosenberg 2001).

Iconic programming notations are defined as visual programming notations where each visual sentence in a program solution is a spatial arrangement of icons with each icon having a distinct meaning (Chang *et al.* undated). Consequently, various interface issues that were considered in the design of B# included the **choice of icons** to represent the different programming structures, the **program solution representation**, design of different **icon dialogues** as well as the **screen design**.

This section focuses on each of the 6 aforementioned issues considered to be pertinent to the development of B#. The description of each issue is in the format of a report on the design and implementation decisions made during the development of B#. Where appropriate, decisions are supported by citations to related work that influenced the design decision. Deliberation on the design and implementation decisions made appears in Chapter 8.

### *5.3.1 Scope of B#*

The main aim of B# (Brown 2001a, b) is for the provision of a programming notation and development environment for novice programmers that minimises the extraneous cognitive load (Pane *et al.* 2001). Consequently, B# hides low level programming details and encourages the development of problem-solving and programming development skills (McIver 2000; Sanders *et al.* 2003a, b). At the same time novice programmers are exposed to automatically generated and syntactically correct textual programming notation program solutions in an integrated development environment.

The programming notation of B# is required to particularly provide support for a small number of powerful, non-overlapping programming constructs. B#, as a first programming notation, is thus limited to support only the foundation programming constructs of sequence, selection, iteration and variables (Mayer 1981; Shu 1985; Blackwell *et al.* 2000; McIver 2000; Howell 2003; Warren 2003). The effect of this design decision is that many powerful programming constructs are excluded from the B# programming notation in order that the programming notation remain as simple as possible. In this way, it is intended that B# support a programming notation that is easy to learn.

A further exclusion from B# is the representation of primitive arithmetic operations such as addition and subtraction as independent programming constructs. This design decision not only reduces the level of complexity, but also simplifies the functionality of the programming notation.

The intention is not to classify B# as a programming development environment that solely simplifies the generation of textual programming notation program solutions. B# is designed to provide an iconic programming notation in support of a textual programming notation within an integrated development environment that allows novice programmers to easily write, read and watch the effect of program solutions. Furthermore, B# is deliberately designed to be a short term programming notation and development environment to be used in the context of the initial stages of an introductory programming course.

### *5.3.2 Functionality of B#*

Throughout the development of the consecutive versions of B#, the developers of the development environment were constantly aware that novice programmers would be the main users of the programming notation and development environment. A great deal of consideration was consequently given to the design and implementation of the development environment interface as well as the functionality (Cilliers *et al.* 2003, 2004a, b).

The functionality of B# incorporates provision for support in each of the following areas:

- composition of program solutions;
- introductory programming constructs;
- integration with equivalent textual programming notation program solutions; and
- appropriate visual feedback on program solution execution.

This section discusses each of the above mentioned areas of functionality, concluding with a presentation of the general task model for composing a program solution in B#.

Program Solution Composition

Novice programmers using the B# programming notation and development environment use a drag-and-drop technique to compose the program solution to a programming problem in the form of a flowchart of icons, with each icon representing a specific and unique programming construct (Section 5.3.4). The program solution representation is thus in the form of a flowchart, with the flowchart being the program solution to which icons can be appended in any order (Blackwell *et al.* 2001; Mattson undated-b). The flowchart was selected as the visual representation of a program solution in B# since it provided a natural progression from the prescribed methodology used to illustrate problem-solving in the initial stages of the introductory programming course at UPE (Wright *et al.* 2000).

Variables and constants can be declared as and when needed by programming constructs during flowchart creation in B# (Blackwell *et al.* 2000). A novice programmer using B# is thus not restricted to the order in which subtasks of the programming task is done.

It has been observed in a recent study conducted by Wright (2002) that multiple representations of a program solution combined with good program solution visualisation support assists novice programmers in building a good mental model of conventional textual programming constructs during the task of programming. In B# the novice programmer would thus be concurrently visually exposed to the equivalent and correct textual programming notation of program solutions generated by the development environment (Mayer 1981; Blackwell *et al.* 2000).

It has further been observed in various related studies (Mayer 1981; Astrachan 1998; Howell 2003) that when well-defined images are contained in a technical textual notation, novice programmers tend to perform best on recollecting these familiar images and tend to recognise the information in the textual notation associated with the images. The association of generated textual notation program solution extracts with corresponding B# iconic programming notation programming construct images would be required to be supported by B#. B# was, however, required to be robust enough to be used for program solution development without a novice programmer requiring to learn or know the syntax of the textual programming notations supported (Calloni *et al.* 1994).

The textual programming notation program solution generated by B# is required to always exhibit properties of good programming practice and secondary notation[33] to encourage the transfer of textual programming notation knowledge (Cant *et al.* 1995; Hendrix *et al.* 1998; Blackwell *et al.* 2000; Lister 2000; Applin 2001). The requirement for programming knowledge transfer necessitated sensitivity to the equilibrium between shielding novice programmers from all forms of notational syntax and permitting them to learn from their errors.

B# was expected to immediately detect syntactical errors as data is progressively entered by the novice programmer. As a result, the implementation of its own compiler module was essential. Considerable effort was required to be put into the design and development of this module, with a view to make it easily extendible for future programming constructs.

Programming Construct Support

The B# development environment had to ultimately provide for general programming constructs, particularly those required during the initial phase of an introductory programming course at UPE. Time constraints on the development of the first version of the system, however, necessitated a restriction on the number of programming constructs

[33] Extra information present in a program solution representation other than formal programming notation syntax. Examples are indentation, colour highlighting and grouping of related constructs. These techniques' main function are to convey meaning to the human reader, thereby enhancing comprehension of a program solution (Green & Petre 1996).

initially supported (Brown 2001a, b). The programming constructs supported by B# ver. 1.0 appear in the upper blue portion of the list in Table 5.1.

**<u>Programming constructs supported by B# ver. 1.0 (Brown 2001a, b)</u>**

- Constants and variables of type integer, float, char, string and boolean;
- Assignment programming construct;
- Simple conditional programming constructs, restricted to the use of a single branching construct, equivalent to the **IF** programming construct;
- Complex conditional programming constructs, restricted to the use of a multiple branching construct using multiple nested single branching constructs, equivalent to nested **IF** programming constructs;
- Iteration or looping programming constructs, restricted to the use of a counting looping construct, equivalent to the **FOR** programming construct;
- Keyboard input; and
- Screen output.

**<u>Additional programming constructs supported by B# ver. 2.0 (Thomas 2002a, b)</u>**

- Simple conditional programming constructs including the use of a single multiple branching construct, equivalent to the **CASE** programming construct;
- Looping programming constructs including the use of sentinel looping constructs, equivalent to the **WHILE** and **REPEAT** programming constructs;
- Procedures; and
- Functions.

*Table 5.1: Programming constructs supported by B# (Brown 2001a, b; Thomas 2002a, b)*

In its support of the iteration programming construct, B# was required to facilitate both horizontally parallel and temporally dependent forms of iteration. Horizontally parallel iteration is where the outcome of one cycle of an iteration has no affect on the outcome of a subsequent cycle, whereas temporally dependent iteration implies a sequential dependence between consecutive cycles (Mosconi & Porta 2000).

The B# development environment had to be developed and continuously adapted in such a way that future additions of programming constructs could be supported without major difficulties. This requirement was necessary because of the limitations placed on the development of the initial version of B# and the need to use B# in an introductory programming course requiring future additional programming constructs. The result was that specific consideration was required in the design of a generic internal and external data model that would facilitate this requirement. The successful inclusion of the additional programming constructs listed in the lower green section of Table 5.1 into B# ver. 2.0 by an independent developer determined that the generic data model designed in B# ver 1.0 was effective (Thomas 2002a, b).

Textual Programming Notation Support

The initial B# development environment was required to concurrently produce syntactically correct textual programming notation program solutions in either $C^{++}$, PASCAL or JAVA forms during flowchart program solution composition. At any time during program solution composition the programmer could select that the textual program solution representation be in an alternative textual form. It would not, however, be possible to view all three textual programming notation forms concurrently. As the flowchart of icons is built, the corresponding textual programming notation is generated simultaneously and is able to be viewed by the programmer.

During the development of B# ver. 2.0, the addition of the procedure and function programming constructs (Blackwell *et al.* 2000) necessitated the decision to facilitate the generation of three textual programming notations to be revisited. The reason for this is that the textual programming notations of $C^{++}$ and JAVA implement both procedures and functions in a similar fashion, with procedures being a special type of a function. PASCAL, however, implements procedures and functions as distinct programming constructs. With Delphi™ Enterprise being the prescribed textual programming notation and development environment in the introductory programming course used in the current investigation, a decision was made to restrict the generation of equivalent textual program solution representation to a single programming notation, namely PASCAL.

Visual Feedback on Program Solution Execution

Novice programmers often do not comprehend that a program is a sequence of steps that are executed consecutively (Crews *et al.* 1998). Programs tend to be viewed by novice programmers as a collection of programming statements that are executed when necessary. Novice programmers thus neglect to pay attention to the accurate placement of programming constructs within a program solution. Consequently, many misconceptions and logical errors are overlooked due to a lack of sufficient and relevant program solution execution feedback.

In order to encourage the transfer of knowledge from introductory to intermediary programming concepts, B# was designed to support two types of mechanisms that allow novice programmers to manipulate the visualised behaviour of program solutions. One such mechanism that is supported is the feature that allows a novice programmer to view the final outcome, in terms of output produced, of a program solution. The second mechanism supported is the feature that allows the sequential nature of the program solution to become visible as a B# flowchart program solution is executed step-by-step (Wright *et al.* 2000).

The latter mechanism formed the basis for modifications and enhancements to the second version of B# which resulted in B# ver 3.0 in 2003 (Yeh 2003a, b). B# ver. 3.0 facilitates a view on variable values which permits a novice programmer to follow the effects of each program construct on the variables. This form of controlled simulation familiarises the novice programmer with program solution tracing and encourages the development of debugging skills to complement their programming skills.

Programming Task Model

Program solutions created in B# are saved in one of two formats, namely the generated textual programming notation representation which is then available for use in a conventional textual programming development environment, or in a format specific to B#. B# facilitates the opening of existing program solutions saved in only the customised B# format.

The general task model for developing any flowchart program solution in B# can thus be summarised as follows:

1. ***Create a new or open an existing flowchart program solution*.**

   In B# ver. 1.0, this included the selection of one of the three textual programming notations supported, namely C$^{++}$, PASCAL or JAVA. This step is eliminated in B# ver. 2.0 for reasons already elaborated on previously.

2. ***Construct or modify the flowchart representation of the program solution by continuously applying the steps 2.1 – 2.3***.

   2.1. ***Select an icon representing a required programming construct***.

   2.2. ***Attach the icon to the flowchart program solution by dragging it to the correct position***.

   As the icon is attached a dialogue box is opened to permit the specification of properties for the particular programming construct.

   2.3. ***Move icons around on the flowchart program solution by dragging and dropping, or delete an icon from the flowchart program solution***.

   The technique implemented simplifies the modification process by successfully moving or deleting any nested icons as a single unit together with the parent icon (Blackwell *et al.* 2000).

   2.4. Throughout this process (repetitive application of steps 2.1 – 2.3 above) ***the corresponding textual programming notation representation of the flowchart program solution can be viewed***.

   Changes to the flowchart program solution are immediately reflected in the equivalent textual program solution representation.

3. ***Execute and debug the flowchart program solution***.

   The flowchart program solution can be executed at any time, allowing the debugging of a program solution. The tracer facility may be invoked by the novice programmer for assistance in the process. B# displays both the final result of the program solution execution as well as the effect of the execution on the declared variables. The tracing facility progresses in a step-wise fashion and is controlled by the novice programmer.

4. ***Save the flowchart program solution***.

   The B# flowchart program solution can be saved for future access from within the B# development environment. The textual programming notation program solution may also be exported in a standard textual programming notation form for future access from within a conventional textual programming development environment.

### *5.3.3 Design of B# Icons*

As mentioned previously, the flowchart representing a program solution contains programming constructs, metaphorically represented by icons and interconnected with lines. Each icon represents a specific and unique programming construct. The purpose of each icon is therefore to indicate the presence of a particular programming construct within a program solution (Dale 1998; Blackwell *et al.* 2000).

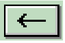
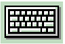
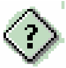
| **Programming Construct** | **Icon** | **Programming Construct** | **Icon** |
|---|---|---|---|
| Assignment |  | Input |  |
| Simple conditional construct (**IF**) |  | Output | |
| Counter iteration loop (**FOR**) | | | |

*Table 5.2: Metaphorical images of programming icons in B# ver. 1.0*

Application of an icon design methodology (Chang *et al.* undated) for the design of B#'s icons resulted in the foundation programming constructs of sequence, selection and iteration to be represented by icons (Mayer 1981; Shu 1985; Blackwell *et al.* 2000; McIver 2000; Howell 2003; Warren 2003). An icon image which attempted to instantly induce the correct meaning of the construct, was associated with each of the identified foundation programming constructs (Blackwell 2001). Table 5.2 shows the icons associated with the programming constructs supported by B# ver. 1.0 (Brown 2001a, b).

The level of intuitiveness attributed to a B# icon is dependent upon how close the mapping between the icon image presented and the programming construct represented. If the mappings are not optimal, novice programmers require more time to comprehend and assimilate the knowledge content of the underlying programming constructs (Ben-Ari 2001). In acknowledgement of the possibility that the B# mappings might not be ideal, an informal survey (Appendix E) of the closeness of mapping between the icon metaphors used in B# ver. 1.0 and their associated programming constructs was conducted.

An evaluation of the icons used in the initial version was conducted during the development of B# ver. 2.0 in 2002 by means of a survey amongst CS/IS major students of a second year computer programming course in the Department of CS/IS at UPE (Thomas 2002b). Participants in the survey were firstly presented with a two-column table (Section E.2), one column containing the icon images and the other the programming construct supported by B#. The participants were required to associate with an image the programming construct which in their mind was immediately induced by the metaphorical image.

| Programming Construct | Icon | Programming Construct | Icon |
|---|---|---|---|
| Assignment | := | Input | ⌨ |
| Simple conditional construct (**IF**) | ◇? | Output | 🖥 |
| Multiple conditional construct (**CASE**) | ?⊟ | Conditional iteration loop (**REPEAT_UNTIL**) | R↻ |
| Counter iteration loop (**FOR**) | F↻ | Operation to return to calling function (**RETURN**) (*appears within a user-defined function*) | ↵ |
| Conditional iteration loop (**WHILE**) | W↻ | Procedure call | P(x) |

*Table 5.3: Metaphorical images of programming icons in B# ver. 2.0*

Upon collecting the survey forms, the administrator of the survey then informed the participants of the correct associations. An hour later the same participants were required to complete the second part of the survey using the form replicated in Section E.3. This questionnaire determined the level of recollection of the programming constructs for each of the B# programming construct icons.

One of the findings of the first section of the survey was that some of the participants did not immediately correctly match the associated programming constructs with some of the icons (Thomas 2002b). The responses of the second part of the survey did however show that participants retained knowledge of the associations between icons and programming constructs, once correctly advised. As a result of this study, some of the icons were redesigned resulting in B# ver. 2.0 supporting the programming constructs and icons shown in Table 5.3.

Further semantics associated with each B# programming construct are conveyed by the spatial arrangement of the icons within the flowchart program solution representation.

### *5.3.4 Program Solution Representation*

Iconic programming notations, like all visual programming notations, still require some kind of syntax to represent the semantics of a program solution (Schiffer *et al.* 1995). One reason for using a flowchart as the development representation notation for B# is that flowcharts involve minimal syntax, where the syntax is visual rather than textual. Another reason is that flowchart program solutions are easier and more intuitive for novice programmers to comprehend than structured textual program solutions (Scanlan 1989; Crews 2001; McKinney 2003).

In the Department of CS/IS at UPE, novice programmers initially practice problem-solving using traditional flowcharting techniques in the introductory programming course. Novice programmers at UPE are thus familiar with the methodology. B#'s program solution representation in the form of a flowchart takes advantage of this prior experience of the novice programmers (Chang *et al.* 1999). Further, the curriculum of the introductory programming course presented by the Department of CS/IS at UPE uses the procedural paradigm, corresponding to the preferred novice programmer paradigm of control flow (Good *et al.* 1999; Good 1999; Oberlander, Cox *et al.* 1999) which is visually supported by the flowcharting methodology implemented by B#.

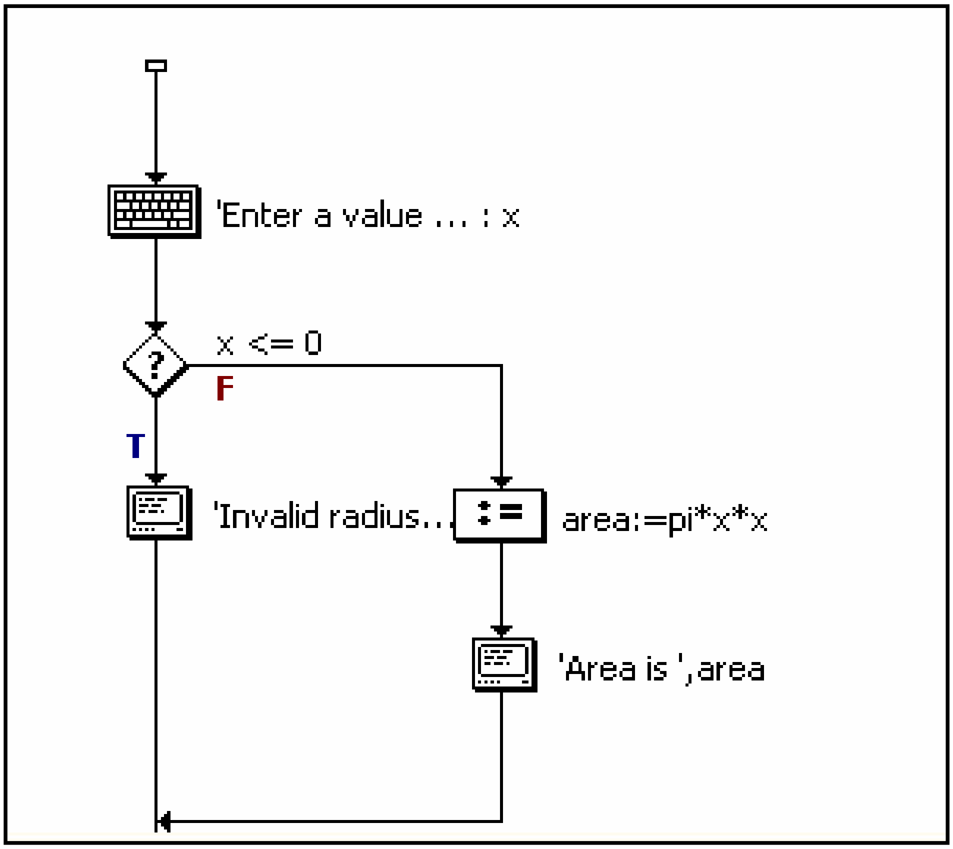


*Figure 5.2: Flowchart program solution in B#*

Figure 5.2 illustrates a typical example of a flowchart program solution composed in B#. The B# flowchart program solution is a top-down single-sequence box-and-line construction typical of visual programming notations (Green & Petre 1996). It consists of icons, each representing a distinct foundation introductory programming construct (Chattratichart *et al.* 2002). The icons are connected by non-crossing lines to indicate the sequential nature of the programming constructs in relation to one another (Lyons *et al.* 1993; Green & Blackwell 1996; Chattratichart *et al.* 2000, 2002; Lyons undated).

Visual secondary notation in a B# flowchart program solution is supported by the vertical and horizontal arrangement of icons in relation to one another (Figure 5.2). Vertical arrangement of icons is primarily an indication of order of sequence, or control flow, from top to bottom. Horizontal arrangement of icons is an indication of mutually exclusive selection.

Manipulation of the icons in the flowchart is the only means by which a novice programmer can specify a program solution using the B# development environment. The flowchart program solution is, however, complimented by an equivalent textual programming notation program solution in order to encourage the transfer of textual notation programming knowledge ultimately required for the introductory programming course at UPE. The positioning of the display of these alternative program solution representations was one issue considered as being important during the design of the B# development environment screen layout.

### *5.3.5 Screen Layout*

The B# screen design process involved developing an interface that could be used effectively by novice programmers. Of special significance was the requirement that a minimal number of windows should be required to be manipulated by a novice programmer at any one time (Green & Blackwell 1996). Figure 5.3 depicts the generic layout of the main screen.

The menu bar contains standard Win32 application options. The tool bar contains specific options from the menu bar that are used regularly by a novice programmer. Examples of such options are saving and executing B# program solutions. The icon palette contains the metaphorical icon images that are associated with the programming constructs supported by B#.

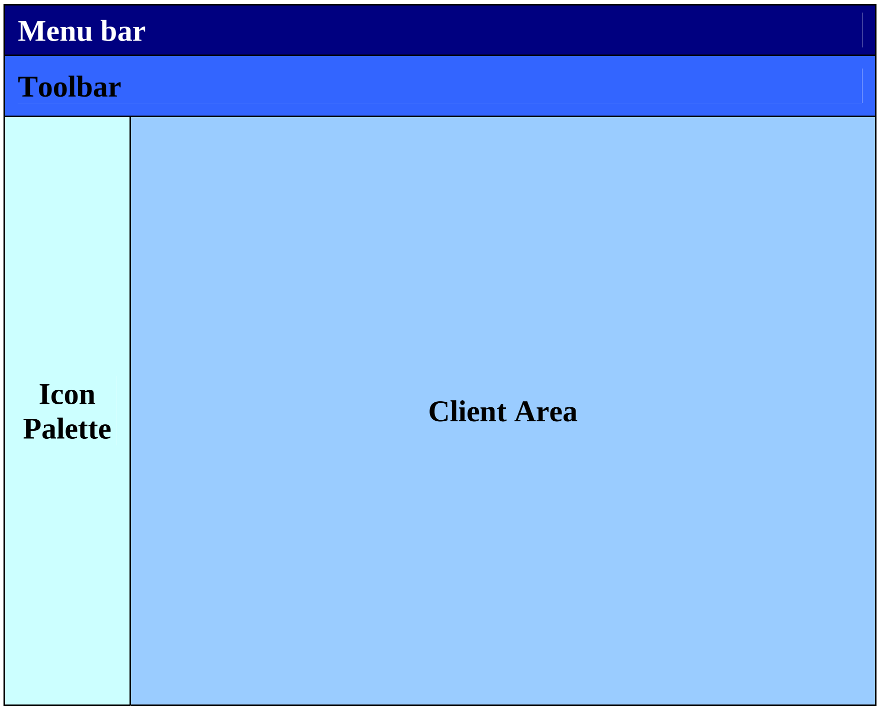


*Figure 5.3: Generic screen layout*

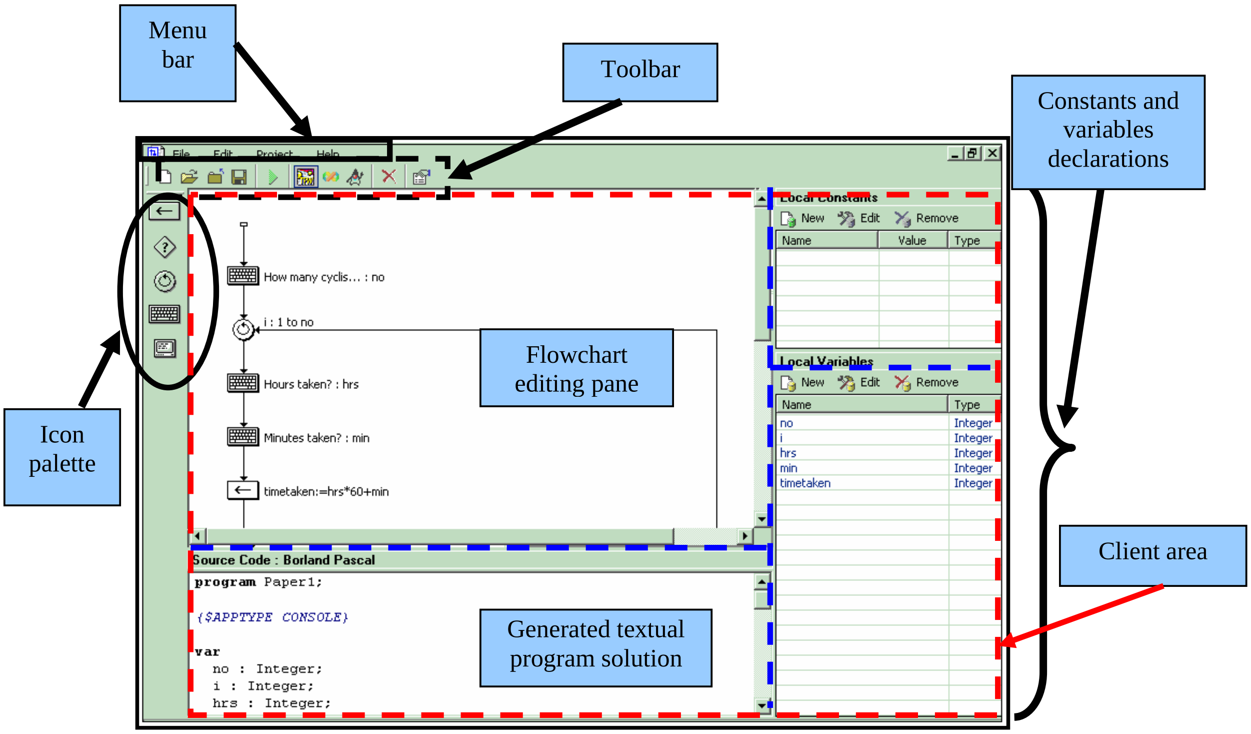
The B# programming development environment makes use of a multiple document interface (MDI) allowing it to contain multiple flowcharts in its client area (Thomas 2002b). Context-sensitive views are implemented to make it easier for the novice programmer to know which choices are available and consequently be shielded from the occurrence of unnecessary errors (Pane *et al.* 2002). The main screen of B# has two states, namely a starting and working state.

In the starting state, the icon palette is not visible. The reason for this is to guide the novice programmer to open a new or existing B# program solution. Once a B# program solution has been created or opened, a state transition to the working state takes place. In the working state at least one flowchart program solution is open in the client area and the icon palette becomes visible.

The evolution of the screen design across the 3 subsequent versions of B# appears in Figures 5.4 – 5.6.

B# ver 1.0

Figure 5.4 shows an example of the screen presented to a novice programmer in the working state while the novice programmer is in the process of developing a program solution in B# ver. 1.0. The client area is divided into four sections: the flowchart editing pane, listing of declared constants and listing of declared variables, as well as the textual programming notation representation corresponding to the flowchart program solution.



| Name | Value | Type |
|---|---|---|

| Name | Type |
|---|---|
| no | Integer |
| i | Integer |
| hrs | Integer |
| min | Integer |
| timetaken | Integer |

Source Code : Borland Pascal

```
program Paper1;

{$APPTYPE CONSOLE}

var
  no : Integer;
  i : Integer;
  hrs : Integer;
```

*Figure 5.4: B# Ver. 1.0 screen layout*

The icon palette facilitates the addition of an icon to the flowchart program solution by the novice programmer simply dragging it from the palette into the correct position in the flowchart. The flowchart is completely interactive, continuously allowing for the addition, moving, deleting and editing of icons. Variables and constants can also be added, edited and deleted on the right hand side of the client area. The textual program solution area at the bottom left hand side of the client area displays the automatically generated equivalent textual program solution representation.

In B# ver. 1.0, the textual program solution would be displayed in the preferred textual programming notation, namely one of $C^{++}$, JAVA or PASCAL. The textual program solution is immediately modified whenever any change is made to either the flowchart program solution or the variables and constants ensuring that both program solution representations are mutually consistent (Cockburn *et al.* 1997; Blackwell *et al.* 2000).

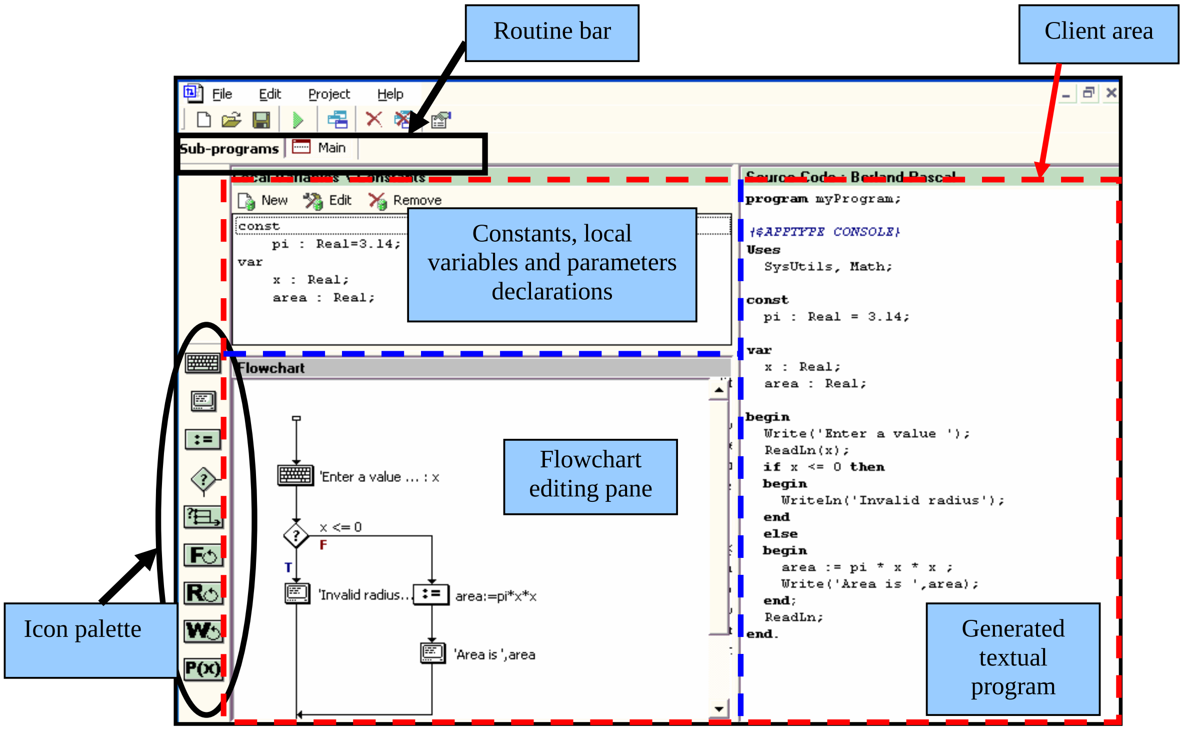
### B# ver 2.0

Informal usability testing of B# ver. 1.0 using the questionnaire in Appendix E revealed that improvements were required in the screen layout (Thomas 2002b). The detailed analysis of the results of this survey appears in Appendix F.

The juxtaposibility property (Blackwell *et al.* 2000) was not sufficiently well supported by B# ver. 1.0 since the textual program solution representation pane appeared below that of the flowchart editing pane, and was also much smaller. This resulted in extra effort by the novice programmer to locate textual program solution extracts corresponding to a particular icon in the flowchart program solution. The consequence of these observations was a reimplementation of the screen layout in B# ver. 2.0. The resulting screen design is illustrated in Figure 5.5.

In the revised screen design (Figure 5.5), the equivalent program solution representations are located horizontally adjacent to one another. In this way, B# visually enhances the correspondence between icon and textual programming notation representations for programming constructs. The generic screen layout was also modified to contain an additional partition, the routine bar. This bar allows the novice programmer to switch between user-defined procedures and/or functions that have already been declared for a particular B# program solution.

The left hand side of the client area in Figure 5.5 presents the B# program solution being composed. Constants, variables and, in the case of procedures and/or functions, parameter declarations, as well as the flowchart of icons are displayed in this section. The union of the various declarations resulted from a requirement to optimise screen space usage. This design decision upholds the natural flow of a routine in an equivalent textual programming notation program solution where the declarations appear at the top of the program solution.



```
const
    pi : Real=3.14;
var
    x : Real;
    area : Real;
```

```
program myProgram;

{$APPTYPE CONSOLE}
Uses
  SysUtils, Math;

const
  pi : Real = 3.14;

var
  x : Real;
  area : Real;

begin
  Write('Enter a value ');
  ReadLn(x);
  if x <= 0 then
  begin
    WriteLn('Invalid radius');
  end
  else
  begin
    area := pi * x * x ;
    Write('Area is ',area);
  end;
  ReadLn;
end.
```

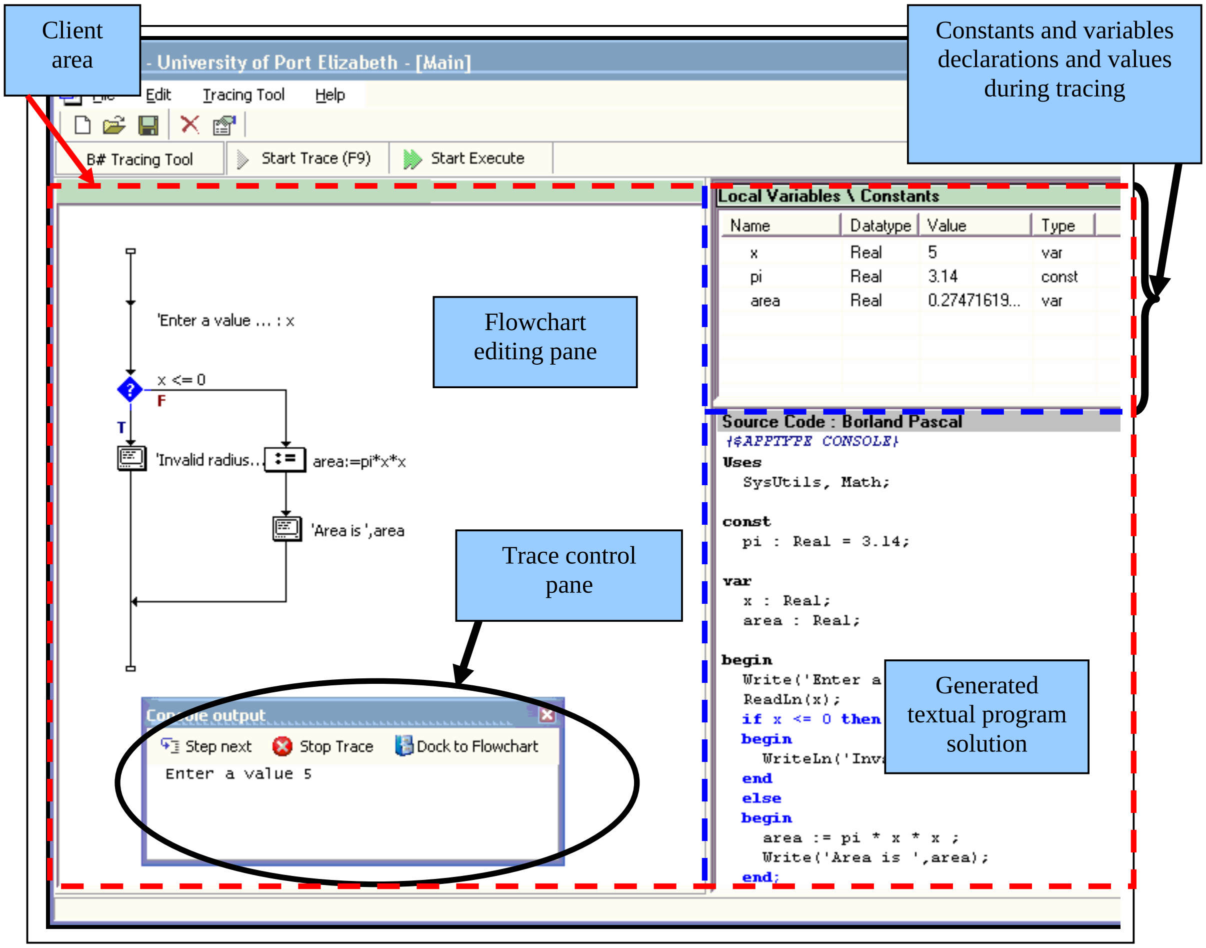
*Figure 5.5: B# ver. 2.0 screen layout*

In support of the B# feature for facilitating user-defined routines, the contents of the flowchart editing pane in Figure 5.5 determines the section of the corresponding textual programming notation program solution displayed. If the flowchart editing pane represents the main section (driver) of the program solution, the entire equivalent textual program solution is displayed. In the case of the flowchart editing pane representing a user-defined function or procedure, only the textual program solution extract relevant to the particular sub-routine is displayed. In all cases, minor effort is required on the part of the novice programmer to search for the textual program solution extract corresponding to the displayed flowchart extract (Blackwell *et al.* 2000).

<u>B# ver 3.0</u>

To further minimise the effort required to search for corresponding textual program solution extract for a flowchart icon, B# ver. 3.0 includes a code-highlighting feature. On a novice programmer selecting an icon in the flowchart program solution, B# highlights the associated generated textual program solution extract thereby visually focussing the attention of the novice programmer to the equivalent program solution extract.

The third version of B# also provides two ways in which a B# program solution can be executed and tested (Yeh 2003b). The first is to execute a program solution through an external compiler with only the final results of the execution, if any, being displayed. This method was also supported by both previous versions of B#. The second method is to execute a program solution within the B# programming environment allowing the novice programmer to control and follow the progress of the execution on both the flowchart and generated textual program solution extract, one programming construct at a time.



```
Enter a value 5
```

Local Variables \ Constants

| Name | Datatype | Value | Type |
|---|---|---|---|
| x | Real | 5 | var |
| pi | Real | 3.14 | const |
| area | Real | 0.27471619... | var |

Source Code : Borland Pascal

```
{$APPTYPE CONSOLE}
Uses
  SysUtils, Math;

const
  pi : Real = 3.14;

var
  x : Real;
  area : Real;

begin
  Write('Enter a
  ReadLn(x);
  if x <= 0 then
  begin
    WriteLn('Inv
  end
  else
  begin
    area := pi * x * x ;
    Write('Area is ',area);
  end;
```

Figure 5.6: B# ver. 3.0 screen layout

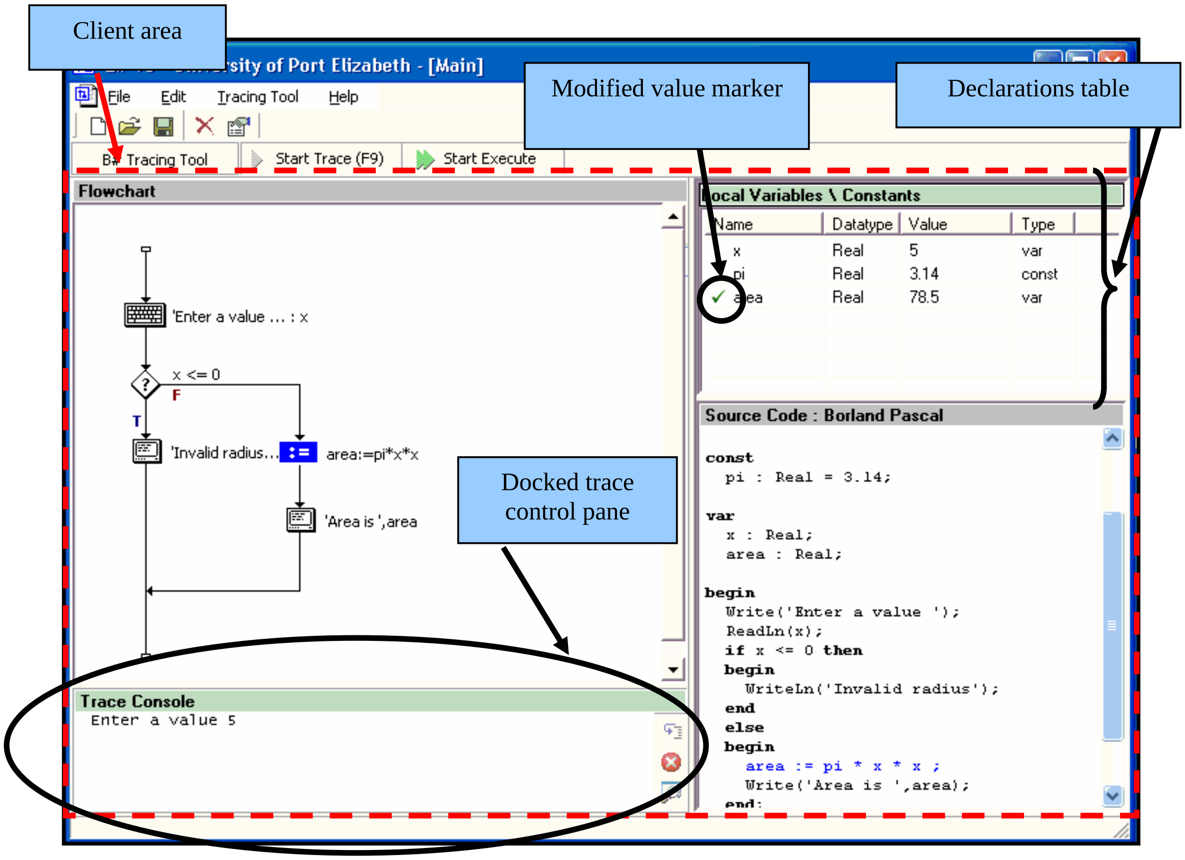
Real-time visual feedback on the progression of the programming logic, the listing of variables and their current values as well as the displaying of any results from the program solution is supported. The need for the behaviour of program solutions to be visualised in B# ver. 3.0 required that both the flowchart and the generated textual program solution extract be animated simultaneously (Yeh 2003a, b). This would serve to reduce the mapping between the novice programmer's mental model of the behaviour of the program solution and the actual behaviour evident by observation (Wright *et al.* 2002), and assist further in supporting retention and transfer of programming knowledge, thereby encouraging in-depth learning. Figure 5.6 illustrates the screen layout implemented in B# ver. 3.0.

The screen design of B# ver. 3.0 retains the juxtaposibility property. The code-highlighting is implemented using the colour blue to animate both the flowchart and textual program solutions simultaneously as the novice programmer manipulates the animation, or trace, of the program solution using the controls in the trace control pane (see blue conditional icon in flowchart editing pane and associated blue text in textual program solution extract in Figure 5.6).

Since graphical versions of program solutions tend to make use of more screen space than their textual counterparts (Blackwell *et al.* 1999b), the layout of the client area was adapted to maximise screen space usage. The variables/constants declarations table pane was repositioned to be above the generated textual program solution representation.

The trace control pane is designed to be visible at all times during the tracing process. B# supports two ways in which the trace control pane can be used. The first is by means of a floating window, allowing the novice programmer to move the trace control pane around freely (illustrated in Figure 5.6). The trace control pane consequently remains on top of the other visual components of B# until the tracing process has been completed.

The second method permits the novice programmer to stabilise the positioning of the trace control pane by docking it to the bottom of the flowchart editing pane. Figure 5.7 illustrates a B# program with the trace control pane docked. The latter method reduces the flowchart display area but ensures that the trace control pane does not obscure any of the other visual components.



| Name | Datatype | Value | Type |
|---|---|---|---|
| x | Real | 5 | var |
| pi | Real | 3.14 | const |
| ✓ area | Real | 78.5 | var |

```
const
  pi : Real = 3.14;

var
  x : Real;
  area : Real;

begin
  Write('Enter a value ');
  ReadLn(x);
  if x <= 0 then
  begin
    WriteLn('Invalid radius');
  end
  else
  begin
    area := pi * x * x ;
    Write('Area is ',area);
  end;
```

```
Enter a value 5
```

Figure 5.7: *B# ver. 3.0 showing docked trace control pane*

Both methods of using the trace control pane provide the novice programmer with the same manipulation operations for tracing a B# program solution. These manipulation operations are namely to

- proceed to the next step/programming construct in the program solution;
- stop the trace; or
- toggle between docking and free floating the trace control pane.

The declarations table (Figure 5.7) is comparable with the watchlist feature of Delphi™ Enterprise but has been designed to be more accessible for novice programmers. During the trace of a B# program solution, all the variables and constants declared for the program solution are listed in the declarations table. At the stage where a variable's value is changed as a result of a programming construct effect, a marker (green tick) appears alongside the variable name to visually focus the attention of the novice programmer to the changed value.

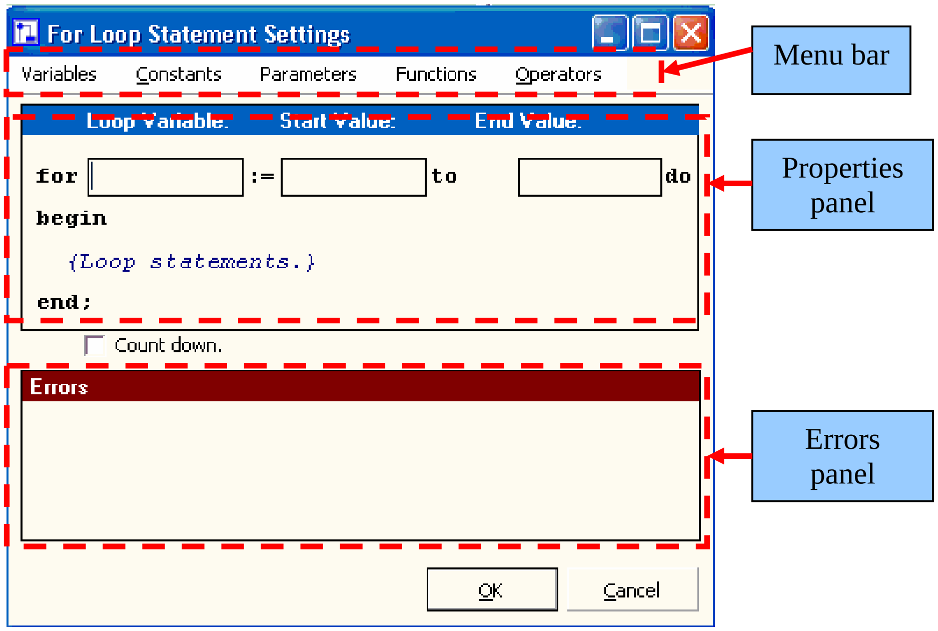
An additional feature of B# ver. 3.0 is one that encourages novice programmers to always initialise variables. This feature is implemented by the automatic initialisation of all variables to randomised values upon declaration. During the debugging of program solutions, novice programmers are consequently faced with unpredictable program solution output until variable initialisation is included in the program solution.

As previously described, all B# program solutions are in the form of a flowchart of icons. On the addition of an icon to the flowchart, a customised icon dialogue guides the novice programmer in the specification of the properties required for the particular programming construct metaphorically represented by the icon.

### *5.3.6 Icon Dialogue Design*

An icon dialogue was designed for each programming construct icon, giving a total of nine icon dialogues being implemented in B# ver. 2.0. An example of an icon dialogue, namely the counting iteration programming construct dialogue (corresponding to the PASCAL textual programming notation **FOR** statement), is shown in Figure 5.8.

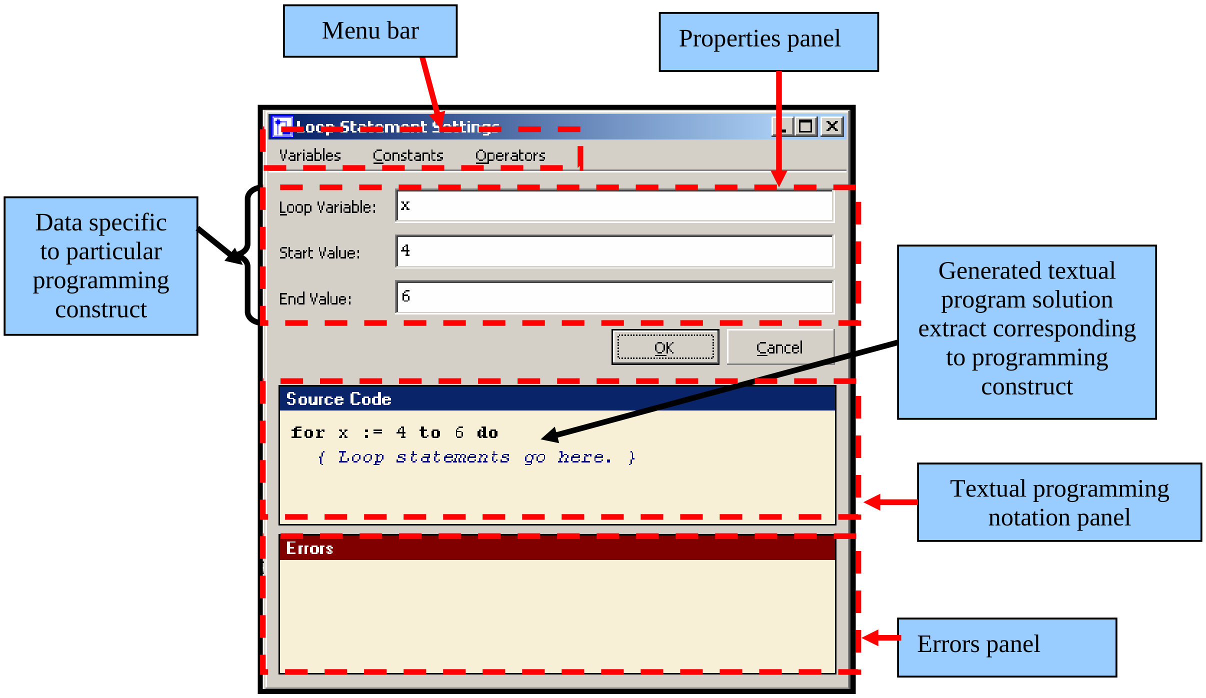
An icon dialogue allows for the associated programming construct's essential data to be viewed and maintained by the novice programmer, in effect eliminating the need for the memorisation of these details by the novice programmer. For example, the icon dialogue illustrated in Figure 5.8 requires the novice programmer to enter a looping variable, starting and ending values. These data items are essential to the counting loop programming construct and considered the properties of the counting loop programming construct. A screen shot of each of the remaining B# icon dialogues appears in Appendix B, together with other selected screenshots from the B# programming development environment.



```
for [          ] := [          ] to [          ] do
begin
   {Loop statements.}
end;
```

□ Count down.

*Figure 5.8: B# ver. 2.0 counting loop programming construct icon dialogue*


Menu bar
Properties panel
Loop Statement Settings
Variables
Constants
Operators
OK
Cancel
Source Code
Errors
Data specific to particular programming construct
Generated textual program solution extract corresponding to programming construct
Textual programming notation panel
Errors panel

Loop Variable: x

Start Value: 4

End Value: 6

```
for x := 4 to 6 do
  { Loop statements go here. }
```

*Figure 5.9: B# ver. 1.0 counting loop programming construct icon dialogue*

Each icon dialogue has a similar design. The icon dialogues implemented in B# ver. 1.0 each consisted of four panels. An example of such an icon dialogue, the counting loop programming construct icon dialogue, is illustrated in Figure 5.9.

The top panel consisted of a menu bar containing shortcuts to previously declared variables and constants, as well as available primitive programming operators. The panel just below the menu bar, the properties panel, allowed the novice programmer to enter/edit information concerning the specific programming construct. The next panel was a textual programming notation panel displaying the generated textual program solution extract corresponding to the combination of the current icon with its essential data item properties.

The lower panel was an errors panel that displayed meaningful error messages related to all errors detected by the compiler with respect to the data entered in the icon dialogue. All entered data was thus required to be error free before the dialogue could be successfully closed (Blackwell *et al.* 2000). In this way B# ensured that the program solution remained free from syntactical errors at all times, and could thus be executed at any time.

Informal usability evaluation of the B# ver. 1.0 icon dialogues design (Thomas 2002b) determined that novice programmers did not pay much attention to the generated textual program solution extract presented in the third panel (Figure 5.9) and consequently might fail to retain sufficient textual programming notation knowledge for transfer to a conventional textual programming notation development environment. This observation resulted in the generic design shown in Figure 5.8 being implemented in B# ver. 2.0, where the novice programmer is encouraged to enter the programming construct's essential data in-line, that is, within the context of a conventional PASCAL textual programming notation statement.

Because of the positioning of the OK button in the icon dialogues in B# ver. 1.0, a novice programmer could very easily avoid looking at the textual source code. B# ver. 2.0's icon dialogue design (Figure 5.8) catered for a repositioned button that encouraged a novice programmer to view both the textual program solution extract and error messages before attempting to close the dialogue. A further implication of this design decision is that the icon dialogues are presented in a less cluttered and simpler form having only three panels, namely the menu bar, the properties panel incorporating the textual program solution extract, and the errors panel.

Much deliberation was required when interpreting issues that arose during the design of the icon dialogues. Noting the requirement that an iconic programming notation for novice programmers should support simple input and output (McIver 2000) as well as provide enhanced error diagnosis, which is a vital facet of learning to program, a good deal of consideration was paid to the compiler error messages presented by B#. The motivation for the consideration is the fact that error messages are the main form of interaction between the novice programmer and the programming development environment (McIver 2000).

A related issue was the amount of assistance the icon dialogue should provide to the novice programmer when entering icon data. Design methodology for iconic programming notations recommends that the development environment should minimise unproductive errors (Cockburn *et al.* 1997; McIver 2000). Unproductive errors are those errors at the superficial level of learning which hinder and discourage the novice programmer, without providing learning opportunities. Examples of this type of error are trivial syntactical errors like a missing semi-colon, or mismatched braces, which do not have the same educational value as, for example, semantic or incorrect programming construct placement errors. Unproductive errors may hinder in-depth learning as they divert novice programmers from the essentials of programming and problem-solving.

During the process of the completion of icon dialogues in B#, one method that was considered was that essential data be automatically filtered by B# and provided to the novice programmer by means of a drop down list. Figure 5.8 is used in the illustration of such a scenario.

On a loop variable being required for the counting loop programming structure in Figure 5.8, B# could provide a filtered drop down list that displays only those variables of the correct data type for the current usage, in this case, for example, only variables of type integer. This would be in agreement with the concept of having minimal syntactical involvement on the part of the novice programmer during the development of a program solution. The novice programmer unfortunately will not benefit from any learning experience using this mode of input. It was, thus, argued by the developers that novice programmers needed to also be introduced to some of the syntactical issues related to programming with a conventional textual programming notation.

Novice programmers had to be sufficiently equipped for a potentially seamless future transfer to a conventional textual programming notation and its associated development environment. Consequently a design decision was that novice programmers would be encouraged to enter data into the text boxes instead of always selecting from filtered drop down lists, even though the selection and drop down list feature would be supported. However, any drop down lists presented to the novice programmer would contain the full set of available and relevant data, with potential distracters included. This intentional feature provides a development environment that allows the natural scaffolding of learning in a novice programmer to develop (Wright *et al.* 2000). Novice programmers are therefore permitted to make mistakes such as specifying a variable that is not of a valid data type.

B#, however, immediately detects such errors and the novice programmer is alerted to a mistake through descriptive error messages in the errors panel. In this way, novice programmers are encouraged to learn from their errors (De Koning *et al.* 2000).

## 5.4 Conclusion

The goal of B# is not to eliminate textual programming notations but rather to complement such a notation in an integrated visual development environment, and thereby enhance the learning experience of the novice programmer. B# supports a small, simple subset of programming constructs. Furthermore, the visual appearance of the program solution structure simplifies the comprehension of the semantics of the supported programming constructs.

The visual iconic programming notation supported by the B# programming development environment lessens the extraneous cognitive load of the novice programmer by reducing the cognitive effort and conceptual mapping required between the problem and program domain representations of a program solution. B# supports this through the visual support of both the flowchart and equivalent textual programming notation program solutions. Further, the B# iconic programming notation and development environment exhibits a strong mapping with the procedural paradigm tasks required by the curriculum of the introductory programming course at UPE.

Because B# satisfies the constraints of a successful visual programming notation (Section 5.2), B# can be classified as such a visual programming notation that is intended for use by novice programmers. The use of a flowchart representation in which to specify a program solution is an educational and significant visual representation. The textual programming notation program solutions generated by B# are examples of well-written textual program solutions. The reading of these program solutions by the novice programmer will encourage the writing of well-written textual program solutions. Further, the presentation of the textual programming notation program solution in the familiar context of a flowchart program solution will encourage the retention of the associated textual programming construct concepts.

B# supports an iconic programming notation that is limited, easy but meaningful enough to visually represent the structure of program solutions and their associated executions. B#'s iconic programming notation highlights the control flow characteristic of a program solution, this paradigm being that preferred by novice programmers. The top-down, left-right traversal direction of solutions created in B# is natural to novice programmers since it mimics an accepted practice of reading natural language text. Consequently the comprehension of the programming constructs is enhanced by the appropriate design of the visual capabilities of the B# programming development environment.

Program solution composition is generally using the mouse in the dragging, dropping and moving of icons representing programming constructs, as well as in the selection of icon dialogue properties. A scaffold learning approach is supported where novice programmers, as they develop and become more familiar with the programming process, can use the keyboard to type in the properties required by the icon dialogues.

Feedback in B# is immediate, automatic, accurate and requires no additional resources. A major strength of the B# programming development environment is the hiding from the novice programmer of mundane syntactical issues usually associated with conventional textual programming notations. B# addresses the problem of how to represent a program solution and its execution in an integrated way showing not only the final results of a program solution, but also providing visual support concurrently indicating the sequence of execution in both the flowchart and equivalent textual programming notation program solutions.

A disadvantage of the visual characteristic of B# is that a flowchart program solution uses more screen space than an equivalent textual programming notation program solution. An implication of this is that a novice programmer might be forced to scroll through the flowchart editing pane during the composition of a flowchart, this action placing extra load on the working memory of the novice programmer. Depending on the extent of the scrolling action required, the extra load on working memory could result in the decay of programming information currently being manipulated in working memory.

| **Requirements for Novice Programmer Technological Support** | **Supported** |
|---|---|
| ***R1***: Elimination of finer implementation details typically found at the superficial learning level of the program domain | ✓ |
| ***R2***: Increased level of program solution comprehension at the in-depth learning level of the program domain | ✓ |
| ***R3***: Increase in level of motivation when using the programming notation | **To be determined by the current study** |
| ***R4***: Designed specifically for use by novice programmers | ✓ |
| ***R5***: Provision of visual techniques to aid comprehension process at the in-depth learning level of the program domain | ✓ |
| ***R6***: Support for reduced mapping between the problem and program domains | ✓ |
| ***R7***: Increased focus on problem-solving | ✓ |
| ***R8***: Increase in novice programmer performance achievement measured in terms of higher level of accuracy in program solutions in program domain | **To be determined by the current study** |

*Table 5.4: Support for Novice Programmer Requirements by B#*

The novelty of B# lies in the fact that the programming development environment presents multiple representations of the same program solution in an integrated environment in an attempt to support and enhance the learning experience of the novice programmer.

In line with the analysis of programming notations and development environments presented in Chapter 3, Table 5.4 summarises the support provided by B# for the framework of novice programmer requirements derived in Chapter 2 (Table 2.1). Support for the novice programmer requirements *R3* and *R8* (Table 5.4) remain undetermined.

Chapter 6 discusses the methodology used during the process of the current investigation that determines the value to novice programmers with respect to these two requirements (*R3* and *R8* in Table 5.4) in an introductory programming course using B# as the supporting programming development environment.

# Chapter 6

# Investigative Research Methodology

## 6.1 Introduction

The Department of CS/IS at UPE has in the past contributed useful and successful research in response to the challenge of increasing throughput in introductory programming courses. Despite these innovative methods (Chapter 4) implemented at UPE since the 1980's to select and place students into alternative streams of introductory programming courses according to their measured potential, poor achievement in both individual and group performances in the introductory programming course, although improved, continues to persist.

The persistence of unsatisfactory individual and group performance rates in the introductory programming courses was consequently one of the motivating factors for the current investigation. The methodology for this investigation into the manner in which a particular programming notation and development environment supports the requirements of a novice programmer (Chapter 2) is described in this chapter.

The most recent strategy of selection and placement implemented in the Department of CS/IS at UPE has been active since 2001 (Chapter 4). An implication of the placement strategy currently in place in the Department of CS/IS at UPE is that separate support mechanisms are required in each of the alternative streams of the introductory programming course (Foxcroft *et al.* 1999). This finding is supported by the recommendation that a strategy to raise the successful completion proportion of students in already oversubscribed introductory programming courses without reducing the quality of the course is that of adjusting the techniques used in the presentation of the course material specifically to cater for those students who are not successful (Wilson *et al.* 1985; Austin 1987).

The lack of widespread acceptance of any specific type of support tool in the introductory programming course (Chapter 3) was thus a secondary factor in the initiation of the current investigation. B#, a visual iconic programming notation and development environment, the design and implementation of which was described in Chapter 5, is proposed in the current investigation as one such support tool and instrument in the investigative methodology on which this chapter focuses.

Any technological educational tool that is used as a support tool for novice programmers in an introductory programming course needs to be quantitatively and qualitatively assessed with respect to the level of support in the areas of novice programmer deficiency as identified in Chapter 2. The previous chapter determined that in terms of its design, B# satisfies all of the novice programmer requirements except for the following (Table 5.4):

- increase in level of motivation when using the programming notation (requirement *R3*); and
- increase in novice programmer performance achievement measured in terms of higher level of accuracy in program solutions in program domain (requirement *R8*).

In order to provide a comprehensive study, this chapter focuses on the research methodology adopted in order to effectively measure the effect of B# with respect to the abovementioned novice programmer requirements *R3* and *R8* (Table 5.4).

A review of the literature suggests that it is widely believed that visual programming notations, the category of programming notations into which B# is classified, offer benefits over textual programming notations (Shu 1988; Schiffer *et al.* 1995; Quinn

2002; Cranor *et al.* undated). Confirmation of the benefits of using B# in the learning environment of an introductory programming course is thus required. This confirmation is especially required with respect to the recommended throughput rate of 75%.

The literature review focussing on programming notations and development environments used in the teaching and learning of programming by novice programmers (Chapter 3) suggests that the assessment of technological support like B# is in terms of benefits to a novice programmer. These benefits are identified as the ease and retention of learning so that the increase in individual performance achievement of a novice programmer in an introductory programming course is evident (Calloni *et al.* 1997; Ramalingam *et al.* 1997; Crews 2001; M$^c$Iver 2001; Carlisle *et al.* 2004).

The resultant small number of comparative studies in programming that determine these benefits can be attributed to the following restrictions (M$^c$Iver 2001):

- lack of formal evaluation strategies;
- large amount of evidence gathered from studies in programming has a tendency to be anecdotal in nature;
- evaluation process for technological support generally requires years of practice using it in the intended environment; and
- in a tertiary educational context, the requirements of introductory courses and curricula make it difficult to compare different forms of technological support in the same course. Comparative studies using different courses, and even the same course in subsequent years, could result in observations that are obscured due to the sufficiently different curricula, contexts and experimental groups.

The methodology described in this chapter is thus especially relevant due to the existence of limited documented empirical research in studies on programming (Calloni *et al.* 1997; Ramalingam *et al.* 1997; Crews 2001; M$^c$Iver 2001; Carlisle *et al.* 2004).

In preparation for the empirical investigation (Chapters 7 and 8), this chapter discusses the introductory programming course's learning environment in the Department of CS/IS at UPE, being the context in which the investigation is conducted (Section 6.2). The methods of data analysis adopted for the purposes of this investigation are described (Section 6.3), including the formulation of hypotheses that are later scrutinised (Chapter 7) and reported on (Chapter 8). This chapter concludes with the observation that every experimental proposal has risks associated with it (Applin 2001). The risks to the design of the current empirical investigation are identified (Section 6.4) and strategies proposed to address each of them.

Based directly on the experimental research methodology described in this chapter, Chapter 7 presents an empirical analysis of the results obtained when using a visual iconic programming notation and development environment within the teaching model of an introductory programming course at tertiary level.

## 6.2 Investigative Study Learning Environment

Evidence of success in the introductory programming course presented by the Department of CS/IS at UPE is a measure of the students' ability to design, code and test a program solution for any variety of introductory problems. This section focuses on the learning environment in which the current study that determines the level of success for introductory programming students using B# as a technological support tool takes place.

In particular, the section commences with a description of the structure of the introductory programming course in the Department of CS/IS at UPE (Section 6.2.1). A discussion of the different materials and instruments used in the investigation follows (Section 6.2.2). The section concludes with an overview of the procedure followed during the course of the current study (Section 6.2.3).

### *6.2.1 Introductory Programming Course Structure at UPE*

A learning framework for an introductory programming course typically consists of **learning resources**, **learning activities** and **learning supports** (Garner 2003). Learning resources consist of the material that provide the content for the course and aid the student in the construction of the mental model with respect to the presented knowledge. Learning activities are the tasks which students are expected to participate in to assist them in the learning process. Learning supports guide the students and provide feedback in a fashion that is responsive and sensitive to individual students. Such a learning framework to support the goal that all students should be able to program on completion of the course (Lister *et al.* 2003) is evident in the introductory programming course at UPE.

Learning Resources

Learning resources for the introductory programming course at UPE include notes provided during lecture learning activities, a prescribed introductory programming textbook and weekly practical task sheets to be completed using the prescribed programming notation and development environment. The practical task sheets attempt to place the algorithms required to be solved within some context. Consequently, story problems have evolved, as is the case at other institutions (Jonassen 2000).

The provided learning resources support the curriculum of the introductory programming course presented by the Department of CS/IS at UPE. The curriculum covers the programming constructs of basic data types, variables, assignments, arithmetic operations, comparison, branching, looping and subroutines, specifically functions and procedures (UPE 2003b). Each programming construct is successively introduced by means of the learning activities of the introductory programming course.

Learning Activities

Two introductory programming educators not directly involved with the current investigation are responsible for facilitating the learning activities of independent groups of students. The duration of the introductory programming course in the Department of CS/IS at UPE is 15 weeks, with the weekly learning activities being distributed as follows:

- 105 minutes of lectures, divided into two sessions, one of 70 minutes duration and the other of 35 minutes duration; and
- 80 minutes of practical exposure in laboratories using the prescribed programming notation and development environment.

Students are also expected to make use of at least a further 2 hours per week for self-study and preparation for practical learning activities.

Similar to other institutions (Shannon 2003), UPE unfortunately does not have sufficient resources to offer separate introductory courses for CS/IS majors and non-CS/IS majors. Consequently, both CS/IS majors, making up approximately 35% of the student population in 2003 in the introductory programming course, and non-CS/IS majors share lectures and practical implementation sessions using generic learning resources[34]. Further, as is the case at other institutions (Jenkins 2001b), the introductory programming course at UPE is perceived as being difficult and this attitude is frequently communicated to new students.

Evidence of the level of difficulty in the course (UPE 2003a) is that the attrition rate in the introductory programming course at UPE of 273 historically first year students during 2002 was observed to be 52% (*n* = 143). This proportion of unsuccessful students is comprised of (Figure 6.1):

[34] This situation changed in the year after that in which the investigation was administered. Due to changes in degree programme requirements at UPE, non-CS/IS majors are no longer required to enrol for the introductory programming course as from 2004 (UPE 2004).

- voluntary course cancellations (2% of attrition total; 1% of total registrations; *n* = 3);
- voluntary changing to the alternate slower paced stream introductory programming course (13% of attrition total; 7% of total registrations; *n* = 19);
- insufficient evidence of performance progress to take the final examination based on a weighted average of continuous assessment results (38% of attrition total; 20% of total registrations; *n* = 55); and
- failure in the final examination upon completion of the course (47% of attrition total; 24% of total registrations; *n* = 66).

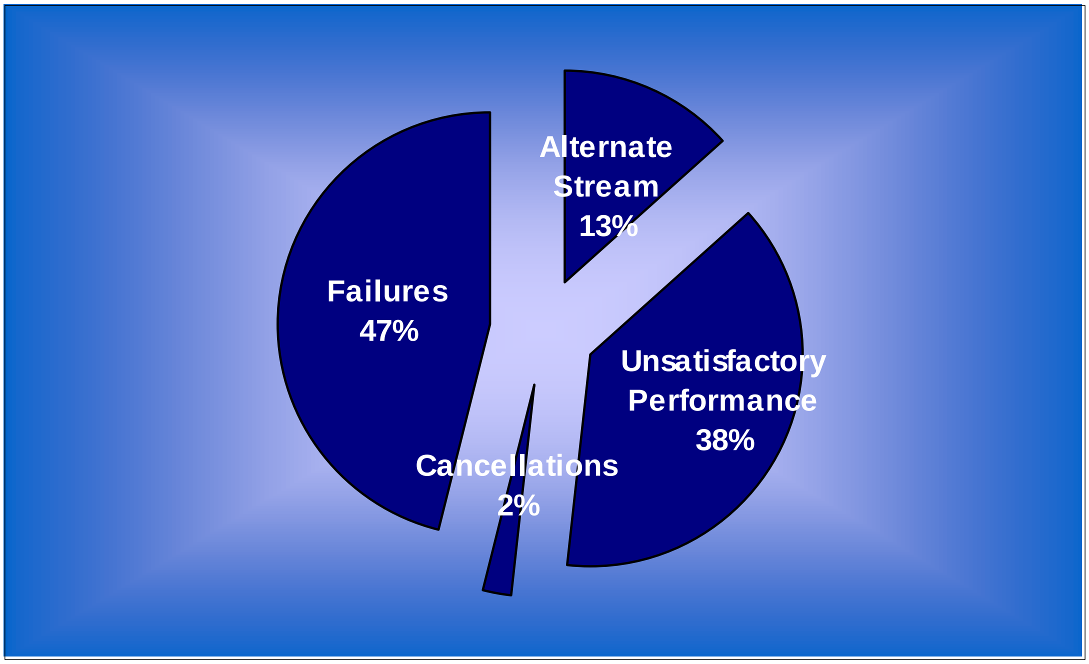


*Figure 6.1: Attrition Proportions in an Introductory Programming Course*

During weekly lecture learning activities, the introductory programming educator introduces a new programming construct to the students (for example, conditional programming construct), illustrating it by means of relevant examples. During the weekly practical learning activity, the students are expected to solve problems that are either novel or similar to those demonstrated during the previous week's lecture learning activities using the appropriate learning resources. The application of the learning activities corresponds to that described in the learning framework (Garner 2003).

Learning Supports

Non-technological tools used during the first 2 weeks of the introductory programming course at UPE are flowcharts and pseudo-code. Thereafter, all program solutions are coded using the prescribed programming notation and, during practical learning activities, its associated development environment. The prescribed programming notation and associated development environment used in the current investigation are respectively Pascal and Borland© Delphi™ Enterprise version 6 (Section 5.2.1).

There is, however, a current lack of learning supports of the type defined as being a requirement of a learning framework (Garner 2003), especially with respect to technological support. This limitation in the learning framework is addressed by means of the current investigation.

Since the final outcome of the introductory programming course is evidence of academic learning, upon completion of the introductory programming course at UPE, students receive course grades on the basis of the following weights:

- 60% from a single syntax specific pen-and-paper examination; and
- 40% accumulated from continuous assessment conducted for the duration of the course.

The continuous assessment portion is comprised of the following:

- 15% from two practical syntax specific tests conducted in the laboratories;
- 21% from two syntax specific pen-and-paper tests; and
- 4% from the assessment of a maximum of 5 of a possible 14 programming assignments.

A weighted average of at least 50% is considered evidence of successful completion of the introductory programming course at UPE. The materials used to determine the grades awarded, together with other instruments used in the investigation, are the focus of the next section.

### *6.2.2 Instruments and Materials*

The current experimental investigation makes use of a number of different types of materials and instruments, each for a specific task. This section provides a background on the programming notations and development environments used as the instruments in this study. Thereafter, an overview of the information sharing, information gathering and assessment materials relevant to the study is presented.

#### Development Environment Instruments

The instruments in the comparative study are the two identified programming notations and associated development environments, namely the prescribed commercial textual programming notation and development environment Delphi™ Enterprise (Section 5.3.1) and B# (Section 5.3.2), an experimental visual iconic programming notation and development environment. Selected screenshots of the interface presented by each of these instruments appear in Appendices A and B.

The subjects of the current study are divided into two groups of students, a control and treatment group, each with a similar participant profile and identical sample size (Section 6.3.2). The control group in the current empirical investigation uses only Borland© Delphi™ Enterprise version 6. B# is used by the treatment group concurrently with Borland© Delphi™ Enterprise version 6. The reason for this arrangement is so as not to disadvantage the treatment group since all students in the UPE introductory programming course are required to write the same final examination and exhibit the same level of proficiency in the prescribed commercial programming notation and development environment upon completion of the course.

Delphi™ Enterprise is developed and maintained external to the Department of CS/IS at UPE by Borland in the United States (Borland 2003). B# was developed and maintained over a period of three years as an essential part of the current study. This section consequently provides a brief background on the development process of B#

by a research team in the Department of CS/IS at UPE. The specific issues regarding decisions on the design and implementation of the programming notation and development environment are detailed in Chapter 5.

The experimental iconic programming notation and development environment B# was developed in the Department of CS/IS at UPE as an integral part of the current investigation. During the period 2001 – 2003, the task of implementing the 3 subsequent versions of B# was assigned to honours students as official postgraduate projects (Brown 2001a, b; Thomas 2002a, b; Yeh 2003a, b)[35]. A research task team consisting of the author and the two promoters was formed to support, advise and assist in the development of these development environments. Both promoters were involved in extensive previous related research in predictors of success in computer science (Calitz 1984; Calitz *et al.* 1992; Calitz 1997; Calitz *et al.* 1997) and the selection and placement of introductory programming course students (Greyling 2000; Greyling *et al.* 2002; Greyling *et al.* 2003) respectively.

The task team had weekly meetings during which the development process and progress of B# was discussed. For the purposes of the current study, an early decision was made that it was imperative that a reliable and functioning program notation and development environment be provided at the conclusion of each academic year. This was necessary to ensure that accurate production testing to assess the goodness of fit of B# to the current research be possible. The cost of this decision was that the initial version of the programming notation and development environment (Brown 2001a, b) provided minimal functionality and, despite promising results (Cilliers *et al.* 2002; Cilliers *et al.* 2003), was thus insufficient for extensive empirical testing as required by the current research. The advantage of this decision was, however, that the initial version of B# was designed and implemented in such a way that it could be modified to incorporate the functionality required by subsequent versions of the programming notation and development environment with a minimum amount of rewriting of the existing source code.

[35] Each of these postgraduate students was the recipient of an award and/or commendation from 2 internal and an external examiner in their relevant study year as a result of their B# implementation.

After considering different implementation languages such as Visual Basic, C++ MFC and Java, a decision was made with the first version of B# to use Borland© Delphi™ version 5 (Borland 2000) as the implementation tool. Even though there existed limited Delphi™ expertise within the department, the fact that the implementation tool provided the facility for efficient executables, a fast compiler, easy component development and the existence of a visual component library were some of the reasons for this decision. The implication of this decision was that subsequent versions of B# were also required to be implemented using Borland© Delphi™ to prevent the rendering of all existing source code useless.

Due to the timing of the completed versions of B#, the second version was the instrument used in the current empirical investigation. Concerns related to the design and implementation decisions made for each of the three versions of B# is detailed in Section 5.3.2.

In order to effectively assess the use of each of the aforementioned programming notation and development environment tools as instruments in the context of the current research, a number of materials are required as components of the current investigation. The identification, format and use of each of the relevant materials are the focus of the following section.

### Investigative Study Materials

A number of information sharing, information gathering and assessment materials are required in the context of the current research so as to effectively measure the impact of the investigative instruments on introductory programming course students. A list of the materials relevant to the study together with each one's designated task and target participant group appears in Table 6.1. The material used in the study is primarily comprised of demographic, training as well as formative, qualitative and quantitative assessment material.

*Demographic Material*

The **biographical data** (*M1*) and participative **consent** (*M2*) surveys are administered once prior to the commencement of the experiment (Table 6.1). The biographical data survey is required to determine the characteristics and demographic profile of the participant groups (Streicher 2003). The consent survey is required to ensure that the policy of UPE's Ethical Committee is upheld and is in agreement with recommended experimental research methodology (Berenson & Levine 1999). Examples of these surveys appear in Appendix D.

| | **Material** | **Use** | **Group** |
|---|---|---|---|
| M1 | Biographical data form | Demographic | Control and Treatment |
| M2 | Consent form | | |
| M3 | B# Training Manual (nine weekly documents) | Training | Treatment only |
| M4 | Development environment practical task evaluation forms (eight weekly documents) | Qualitative assessment | |
| M5 | General development environment evaluation questionnaire (single document) | | Control and Treatment |
| M6 | Pen-and-paper tests (two documents) | Quantitative assessment | |
| M7 | Practical tests (two documents) | | |
| M8 | Practical sheets (nine weekly documents) | Formative assessment | |
| M9 | Pen-and-paper examination (single documents) | Quantitative assessment | |
| M10 | Tutor and student attitude to practical learning activity evaluation form (single document) | Qualitative assessment | |

*Table 6.1: Materials used in Investigative Study*

*Formative Assessment Material*

A total of nine weeks of the introductory programming course at UPE is committed to the administering of the experiment. In each of these weeks, participants in both the control and treatment groups receive an identical weekly take-home **practical sheet** (*M8*) consisting of a set of between 3 and 5 tasks that are to be practiced using the programming development environment(s) appropriate to the group (Table 6.1). The formative practical tasks are provided as learning exercises for a more effective learning experience (Roumani 2002) than traditional lecture learning activities. The tasks in each practical sheet are consequently selected and presented in such a way as to provide the opportunity of practice of the programming concepts introduced in the previous week's lecture learning activity. The required deliverable for all tasks is a program solution written in the procedural, or imperative, programming paradigm in a format compatible with that developed using the Delphi™ Enterprise supported textual programming notation. A maximum timeframe of a week is set for the completion of a single week's practical tasks.

Story problems are used to place the required practical task in some kind of context, this technique being typical of introductory programming courses (Jonassen 2000). The result is that the values required to solve the task are embedded within a short narrative or scenario. Participants are consequently required to make a choice of the most suitable programming method for solving the problem, extract the values from the narrative and use them when solving the task problem.

Each individual task in a practical sheet has been designed to incorporate as many of the previous week's programming concepts as possible. In this way, regardless of the speed at which an individual participant progresses in the practical learning activity, the objective is that each participant will have the opportunity of practicing the required programming concepts at least once per weekly practical sheet, assuming the completion of at least a single task (program solution) per practical sheet. The computation of a maximum, minimum and average of multiple integer values is indicative of the level of sophistication of program solutions expected as deliverables from weekly practical tasks.

*Training Material*

In the case of the treatment group, a relevant section of the **B# training manual** (*M3*) accompanies the weekly practical sheet (Table 6.1). This is required so that the participants in the treatment group have the necessary programming notation and development environment manipulation skills when using B# to solve their practical tasks. To encourage an accurate mental model of the B# programming notation and development environment, screen shots are used extensively in the training manual with text being minimised, this technique having been successfully used in a similar programming study (Shih *et al.* 1993). The practical sheets designed for the treatment group consistently require that the first task be completed using B#, the second using Delphi™ Enterprise, and subsequent tasks in a programming notation and development environment of the individual participant's choice. The practical sheets and accompanying B# training manual designed for the treatment group appear in Appendix C.

*Qualitative Assessment Material*

At the start of each practical learning activity, all participants of the treatment group are requested to complete a qualitative survey consisting of a **development environment practical task evaluation form** (*M4* in Table 6.1). The purpose of this survey is to determine the preferred programming notation and development environment for particular practical tasks of the previous week's practical learning activity, together with reasons for the participant's preference. The development environment practical task evaluation forms appear in Appendix D.

Five weeks into the experiment, a **tutor and student practical learning activity attitude questionnaire** (*M10*) is administered to the tutors and participants of both the control and treatment groups (Table 6.1). The goal of this survey is to determine the general attitude and motivation of the tutors and participants to the practical learning activity, which is dependent upon the programming notation and associated development environments in use as instruments in the particular practical session. The tutor and student practical learning activity attitude questionnaires appear in Appendix D.

Approximately 6 weeks into the experiment, a **general development environment evaluation questionnaire** (*M5*) is administered to the participants of both the control and treatment groups (Table 6.1). The purpose of this questionnaire is to determine the general attitude and motivation of the participants to the programming notation and development environment(s) being used. The general development environment evaluation questionnaire appears in Appendix D.

The format of the questionnaire is one of 7 open-ended questions. Participants are required to provide answers to the questions in the context of the programming notation and development environments that they are currently using in the introductory programming course. To encourage as comprehensive responses as possible, the answer sheets circulated to participants have large spaces for written responses. No questions appear on the answer sheets. The questionnaire is administered by the author during a combined meeting session of participants in both the treatment and control groups. Each question is consecutively displayed on an overhead display and the participants are given approximately 3 – 5 minutes to respond to each question. Only once there is an indication on the part of the participants that they are ready for the next question, is it displayed for their written response. The administering of the questions continues in this format until the entire questionnaire has been covered.

The approach of administering this survey in response to preliminary observations that further qualitative data is required to explain the initial observations is in accordance with recommended qualitative research methodology (Ely *et al.* 1995). This survey is administered in response to the observation that further qualitative data is required as a result of the quantitative data collection and analysis of the development environment practical task evaluation forms and early programming performance achievement measures.

*Quantitative Assessment Material*

Programming achievement measures in studies on programming typically include the measurement of (Irons 1981):

- programming skill and knowledge acquisition;
- comprehension of a given problem;
- composition of a program solution;
- debugging to locate errors in a program solution; and
- the modification of a program solution.

Consequently, programming achievement measures which are used in many computer programming achievement investigations have been identified as the tasks of reading program solutions, writing program solutions and program solution writing laboratory exercises (Austin 1987). Accordingly, quantitative assessment in the current investigation is performed by means of the summative assessment materials of syntax- and problem-solving based **pen-and-paper** (*M6* in Table 6.1) and **practical laboratory tests** (*M7* in Table 6.1) as well as a **pen-and-paper examination** (*M8* in Table 6.1).

All participants in the current study are expected to be proficient in both the PASCAL programming notation and Delphi™ Enterprise development environment upon successful completion of the introductory programming course and are expected to be successful in the final Departmental examination. In anticipation of a valid statistical comparison component in the study, the contents of these assessment materials are, wherever possible, identical across the control and treatment groups. Samples of the assessment materials appear in Appendix D.

The pen-and-paper tests as well as the pen-and-paper examination contain no B# programming notation or development environment specific questions and are thus restricted in content to PASCAL programming notation specific questions. Practical tests containing identical practical tasks are customised to suite each experimental group's requirements with respect to programming notation and development environment.

A total of two pen-and-paper tests take place, one at the start of the experiment and the second 7 weeks into the experiment. The specific contents and aim of each problem in the first of the pen-and-paper tests is given in Table 6.2. Table 6.2 highlights that the goal of the first pen-and-paper test is solely to establish the problem-solving expertise of the participants using one of the non-technological tools (flowcharts and pseudo-code) covered in the introductory programming course. No programming notation syntax related tasks are assessed.

| Question (Section D.1.1) | Problem | Goal | Expected Response | Proportion of Test |
|---|---|---|---|---|
| Question 1 | Develop solutions to problems using flowcharts and/or pseudo-code | Solution composition | Flowchart/ pseudo-code solution | 40% |
| Question 2 | Isolate and identify inputs, outputs and process for a given solution in the form of a flowchart/ pseudo-code | Solution comprehension | Identification and description of inputs, outputs and process | 20% |
| Question 3 | Modify existing solution | • Solution comprehension<br>• Solution composition | Adapted solution in form of a flowchart/ pseudo-code | 40% |

*Table 6.2: Problems and Associated Goals of First Pen-and-paper Test*

The successful comprehension of a program solution in the pen-and-paper test requires a participant's ability to read the program solution and demonstrate an understanding of what the program solution does in terms of the specified use of programming constructs present in the solution (Bloom *et al.* 1956; Lister *et al.* 2003). The successful composition of a program solution requires a participant's ability to demonstrate correct application of concepts in an appropriate program solution for which no format is explicitly specified (Bloom *et al.* 1956).

The specific contents and aim of each problem in the second pen-and-paper test is given in Table 6.3. Table 6.3 shows that the goal of the second pen-and-paper test is to determine the problem-solving expertise of the participants within the context of programming notation syntax-based tasks. The same abilities of program solution comprehension and composition, now applicable to the PASCAL textual programming notation, are required to be exhibited by participants to whom the second pen-and-paper test is administered. The second pen-and-paper test specifically determines whether participants have the ability to determine the effects of PASCAL programming statements as well as assemble PASCAL programming notation primitives in such a way that a required result is achieved (Rader *et al.* 1998).

| Question (Section D.1.2) | Problem | Goal | Expected Response | Proportion of Test |
|---|---|---|---|---|
| Section A | Multiple choice questions | Textual programming notation comprehension | Results of tracing, debugging and analysis of program solution extracts | 48% |
| Section B | Create program solution extracts | Program solution composition using textual programming notation | PASCAL textual programming notation program solutions | 52% |

*Table 6.3: Problems and Associated Goals of Second Pen-and-paper Test*

The final pen-and-paper examination mirrors the structure of the second pen-and-paper test. The only difference between the assessments is in the proportions dedicated to program solution comprehension (47%) and composition (53%) in the examination. The goals of all of the pen-and-paper assessments are in accordance with those documented in a similar programming investigation (Kaasbøll 1998).

Objective testing forms a large part of both the second pen-and-paper test and the final examination. This type of testing takes the form of multiple choice questions (Table 6.3) in each of the specified assessment materials. The goal of the multiple choice questions is to establish participant knowledge about textual programming notation syntax and program solution behaviour and thus assesses a participant's comprehension of given program solution segments (McCracken *et al.* 2001). In the remaining portion of the assessment material shown in Table 6.3, participants are required to create program solution segments as evidence of ability to generate working solutions to given problems.

Two performance-based assessment (McCracken *et al.* 2001) programming exercise (practical) tests are administered to the participants of both the control and treatment groups. These tests are administered in a fashion similar to that described by Daly *et al*. (2004).

In both tests participants are required to individually generate working and tested program solutions to given problems in a programming notation and development environment within the time allotted. The short assignments occur during fixed-length laboratory sessions under controlled conditions to reduce the opportunity for plagiarism.

The first practical test determines the problem-solving expertise of each participant with respect to the composition of correct syntax-based program solutions to tasks in the programming development environment(s) allotted to the group. The specific contents and aim of each problem in the practical test customised for participants of the treatment group is given in Table 6.4. The difficulty level of the tasks is introductory. Algorithms indicative of the level of complexity of program solutions expected include a count or list of factors of an entered integer. Control group participants are required to complete all program solutions for problems in the test using Delphi™ Enterprise and its supported textual programming notation.

| Question (Section D.2.1) | Problem | Goal | Expected Response | Proportion of Test |
|---|---|---|---|---|
| Task 1 | Syntax-based program composition | • Problem-solving<br>• * Program composition achievement using B# | Correct B# program solution | 40% |
| Task 2 | Syntax-based program composition | • Problem-solving<br>• Program composition achievement using textual programming notation supported by Delphi™ Enterprise | Correct PASCAL program solution | 60% |

* Control group participants are required to show evidence of achievement in this problem using the PASCAL textual programming notation supported by Delphi™ Enterprise.

*Table 6.4: Problems and Associated Goals of First Practical Test (Treatment group)*

The second practical test involves tasks of introductory to intermediate level of difficulty in an introductory programming course. Algorithms indicative of the level of complexity of program solutions expected include using a number of subroutines to implement simple algorithms such as a computation of a maximum, minimum or average of an entered list of numerical values.

One of the tasks in the second practical test is dedicated to determining the achievement of participants with respect to textual programming constructs syntax and semantics in the context of the PASCAL textual programming notation. The other tasks determine the syntax-based problem-solving expertise of each participant with respect to the specific programming development environments of the associated group. One of the latter tasks permits participants in the treatment group to individually select a programming notation and associated development environment in which to solve the problem, thereby indicating their preference of programming notation and development environment. The specific contents and aim of each problem in the test for participants in the treatment group is given in Table 6.5.

| Question (Section D.2.2) | Problem | Goal | Expected Response | Proportion of Test |
|---|---|---|---|---|
| Question 1 | Locate and eliminate textual programming notation syntactical errors | • Program solution comprehension<br>• Textual programming notation skills<br>• Achievement using Delphi™ Enterprise | Correct PASCAL program solution | 14% |
| Question 2 | Syntax-based problem-solving | * Program solution composition achievement using B# | Correct B# program solution | 14% |
| Question 3 | Syntax-based problem-solving | * Program solution composition achievement using preferred programming notation and development environment | Correct program solution (either PASCAL or B#) | 36% |
| Question 4 | Syntax-based problem-solving | Program solution composition achievement using PASCAL textual programming notation and Delphi™ Enterprise | Correct PASCAL program solution | 36% |

* Control group participants are required to show evidence of achievement in this problem using the PASCAL textual programming notation supported by Delphi™ Enterprise.

*Table 6.5: Problems and Associated Goals of Second Practical Test (Treatment group)*

In all of the assessments requiring the creation of program solution extracts to problems, the assessment of the program solution representation includes the assessment of data structure decision and correct programming technique, as well as the presentation of the program solution extract in an appropriate form, as is the case in a similar documented study on programming (McCracken *et al.* 2001).

An overview of the process followed in applying each of the materials and instruments described in this section, and in collecting and analysing the relevant data in the current investigation is the focus of the section that follows.

### *6.2.3 Procedure for Data Collection*

The data for the current experimental study consists of pre-treatment measures that allow for the statistical comparison of non-random groups as well as intermediary and post-treatment measures. The intermediary and post-treatment quantitative measures are comprised of performance achievement measurements determined in terms of the accuracy achieved in completing tasks. The qualitative measures comprise of the level of participant attitude and motivation towards the programming notations and development environments being compared.

Data for the study is collected at 9 distinct stages of the 15-week introductory programming course (Figure 6.2), with the methods of collection being the analysis of participants' submitted practical tasks, questionnaires, pen-and-paper and practical tests as well as a single final pen-and-paper examination.

The experimental study requires the initial assignment of participants to treatment and control groups. Consequently, this process necessitates an initial rigid balancing of the number of potential participants in each of the strata (Section 6.3) within each experimental group in anticipation of valid statistical comparison. Thereafter, the investigation requires the creation and administering of a number of materials aimed at the participants in both the control and treatment groups. The identification and objective of each of these materials is detailed in the previous section (Section 6.2.2).

The sequence of the administering of each of the materials detailed in Section 6.3 at various stages within the treatment period in the 15 week introductory programming course is illustrated by columns C and D in Figure 6.2. The area framed in dark blue indicates the portion of the introductory programming course during which the treatment occurs.

| A | B | C | D |
|---|---|---|---|
| **Week** | **Introductory Programming Concept** | **Repetitive Material (Table 6.1)** | **Single Material (Table 6.1)** |
| **1** | **Problem-solving** | | Demographic Material administered (*M1*) |
| **2** | **Problem-solving** | | |
| **3** | **Problem-solving** | | Consent Form administered (*M2*) |
| **4** | **Variables, data types, Input/Output** | Administering of practical sheets (*M8*)<br>Distribution of B# training manual (*M3*)<br>Development environment practical task evaluation (*M4*) | |
| **5** | **Conditional programming constructs** | | Problem-solving theoretical assessment (*M6*) |
| **6** | **Looping programming constructs** | | Tutor and student attitude to practical learning activity evaluation form (*M10*) |
| **7** | **Looping programming constructs** | | Introductory level practical assessment (*M7*) |
| **8** | **Looping programming constructs** | | |
| **9** | **Structured programming** | | |
| **10** | **Subroutines** | | Syntax-based pen-and-paper assessment (*M6*) |
| **11** | **Subroutines** | | General development environment evaluation (*M5*) |
| **12** | **Subroutines** | | |
| **13** | **Error-proofing and debugging** | | Intermediary level practical assessment (*M7*) |
| **14** | **String manipulation** | | |
| **15** | **Processing text files** | | |
| | | | Final Examination (*M9*) |

*Figure 6.2: Administering of Material*

All demographic questionnaires are administered and captured for the purposes of experimental group profile analysis on commencement of the introductory programming course. Non-technological tools (flowcharts and pseudo-code) form the initial learning support (weeks 1 – 3), with the actual treatment period (indicated by blue shading in Figure 6.2) commencing in week 4 and continuing until week 12. Participants in both experimental groups sign consent forms to give their permission to participate in the study and for the confidential distribution for the purposes of research of any data collected during the course of the investigation. During the treatment period, participants in the treatment and control groups receive identical course content during traditional lecture learning activities. At no time during lecture learning activities is there any explanation of or reference to the B# iconic programming notation or development environment.

Further, both experimental groups receive identical practical tasks covering the programming concepts listed in column B of Figure 6.2. The programming tasks are required to be completed on a weekly basis and collaborative work is encouraged. In the design of the practical tasks, it is ensured that the tasks are simple problems which require fairly short program solutions, but which at the same time encourage the use of various programming constructs. The only way to guarantee that participants will attempt any practical task is to make its marks contribute towards the final grade in the course (McCracken *et al.* 2001). Each participant is thus randomly selected on a maximum of 5 occasions during the introductory programming course to demonstrate completed practical tasks to the course instructor. The assessment of these demonstrations contributes to the final course grade (Section 6.2.1).

The treatment group practical sheets are customised to include relevant sections of the B# training manual, this being the only way in which information specific to B# is shared with the participants (Appendix C). Similar to the study by Calloni *et al.* (1994), treatment group participants are required to use both B# and Delphi™ Enterprise for programming assignments, and submit both formats of files for assessment purposes when required. Treatment group participants are also required to complete weekly development environment practical task qualitative evaluation forms. By experimental design, treatment group participants consequently use B# in parallel with Delphi™ Enterprise for a period of 9 weeks, being 67% of the total introductory programming course duration.

The quantitative assessment instruments (Appendix D) required for the collection of participant performance achievement data are indicated in column D of Figure 6.2 in reverse shading. It is ensured that the assessment tasks mirror the level of complexity evident in the collaborative practical tasks already completed by the participants, as is the case in a similar programming study (Chamillard & Braun 2000). The performance achievement level for each of the assessments is measured in terms of the accuracy level achieved in completing the given tasks (Blackwell 2001) and focuses on evidence of problem-solving and programming notation skills. Participants are required to solve problems within allotted time periods and are not permitted to make use of any other instructional aids, nor require responses to questions related to the program domain (Chamillard *et al.* 2000).

Problems in the assessment materials, where appropriate, are predominantly presented in the form of the PASCAL textual programming notation and Delphi™ Enterprise development environment representation since all participants, regardless of treatment applied, are expected to develop proficiency with the PASCAL programming notation and Delphi™ Enterprise development environment upon introductory course completion.

The grades awarded in the assessments are continuous variables representing a number between 0 and 100. Each of the participant assessment submissions is graded by the author and moderated by the course instructors in order to eliminate any differences in the application of the grading criteria, being primarily the examination of program solution and problem-solving approach. A high score in an assessment is an indication of a participant's high level of achievement in the relevant assessment material.

The performance and qualitative data is available on all of the participants who have correctly identified themselves on submitted tasks and questionnaires and who are still registered for the introductory programming course after completion of the final assessment, the pen-and-paper examination. The data available is analysed using quantitative and qualitative techniques, which are described in detail in the following section.

## 6.3 Method of Data Analysis

The current study is primarily a quantitative and qualitative behavioural study of novice programmers in an introductory programming course. The goal of the study is to determine whether novice programmer performance is independent of the programming notation used as the technological delivery method in the teaching and learning environment for an introductory programming course at tertiary level. The independent variables identified form a subset of those of performance time, error rate, retention time, accuracy and subjective preference acknowledged in a previous study on programming (Maryland 2001).

The statistical analysis methodology applied in the current empirical analysis of quantitative data is adopted from the hypothesis testing methodology described in Berenson *et al*. (1999). The statistical analysis methodology firstly requires the formulation of hypotheses to be scrutinised, as well as the specification of the level of significance ($\alpha$) against which the statistical measurements are to be compared. This section thus formulates the hypotheses to be tested (Section 6.3.1).

Allocation of students to the various learning activities in the introductory programming course is assumed to be random since the students pre-register for lecture and practical learning activities without the knowledge that a controlled experiment is planned. The participant population of the controlled experiment as well as the mechanisms implemented to select a sample for each of the control and treatment groups essential for the current investigation is described (Section 6.2.2). This section also includes a discussion of the data analysis techniques applicable to the current investigative study (Section 6.3.3).

### *6.3.1 Formulation of Hypotheses*

In the formulation of particular hypotheses to be quantitatively and qualitatively analysed, the treatment group refers to historically first year introductory programming course subjects whose programming assignments are concerned with the use of a visual iconic programming notation and development environment, B#, concurrently with a traditional commercial textual programming notation and development environment, Delphi™ Enterprise. The control group refers to historically first year introductory programming course subjects who write programs from first principles for each task using only a traditional commercial textual programming notation and development environment, Delphi™ Enterprise.

In particular, the following hypothesis (Berenson *et al.* 1999) is thus formulated for examination and tested for significance at the 95% percentile ($\alpha = 0.05$):

$H_0$: *Academic performance in an introductory programming course is independent of programming notation and development environment.*

$H_1$: *Academic performance in an introductory programming course is dependent on programming notation and development environment.*

The independent variables in the current study are achievement measures for all of the tasks in two pen-and-paper tests, two practical tests, a single pen-and-paper examination, as well as weighted class and final grades for each participant in the study, giving a total of 28 independent variables. Consequently, the null hypothesis ($H_0$) above is refined to produce the following sub-hypotheses:

$H_{0.1}$: *The average mark achieved in an introductory programming course is independent of programming notation and development environment.*

$H_{1.1}$: *The average mark achieved in an introductory programming course is dependent on programming notation and development environment.*

and

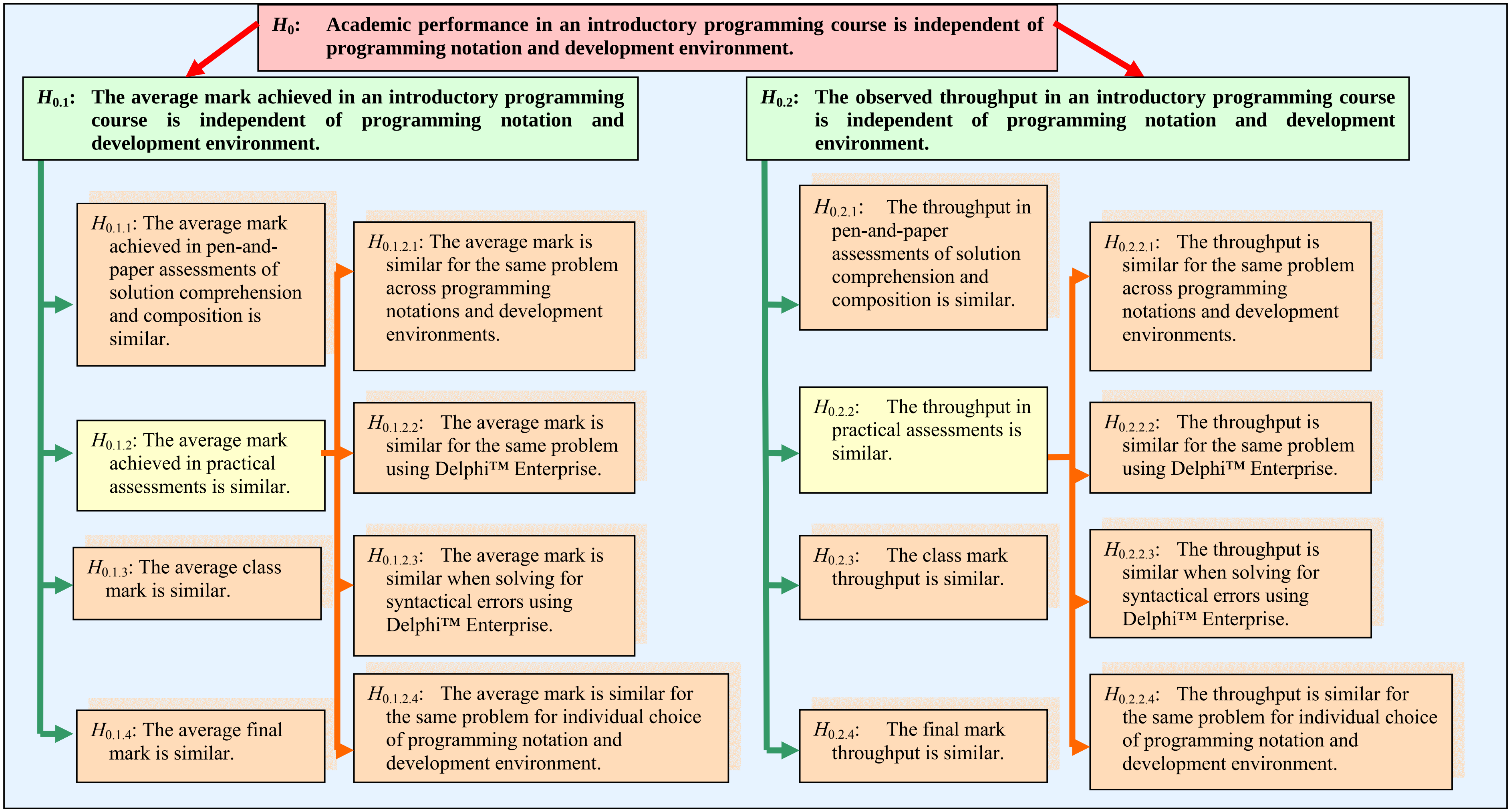
$H_{0.2}$: *The observed throughput in an introductory programming course is independent of programming notation and development environment.*

$H_{1.2}$: *The observed throughput in an introductory programming course is dependent on programming notation and development environment.*

Further refinement of the null hypotheses ($H_{0.1}$ and $H_{0.2}$) above produces the hypotheses shown in Figure 6.3 which individually scrutinise the equivalence of participant achievement in the areas of solution comprehension, solution composition in the comparable programming notation and development environments as well as PASCAL textual programming notation retention. For the purposes of simplicity, only the refined null hypotheses are shown in Figure 6.3.

For completeness, the full complement of both null and alternative hypotheses relevant to the current statistical analysis appears in Appendix H. The 14 leaf null hypotheses shaded in orange in Figure 6.3 are the hypotheses that are scrutinised in the current research. These are the hypotheses identified as $H_{0.1.1}$, $H_{0.1.2.1}$, $H_{0.1.2.2}$, $H_{0.1.2.3}$, $H_{0.1.2.4}$, $H_{0.1.3}$, $H_{0.1.4}$, $H_{0.2.1}$, $H_{0.2.2.1}$, $H_{0.2.2.2}$, $H_{0.2.2.3}$, $H_{0.2.2.4}$, $H_{0.2.3}$ and $H_{0.2.4}$.

The next phase of the hypothesis testing methodology requires that the sample on which the analysis is to be performed be defined. The following section describes the derivation of the sample appropriate to the current research.



**$H_0$: Academic performance in an introductory programming course is independent of programming notation and development environment.**

**$H_{0.1}$: The average mark achieved in an introductory programming course is independent of programming notation and development environment.**

$H_{0.1.1}$: The average mark achieved in pen-and-paper assessments of solution comprehension and composition is similar.

$H_{0.1.2}$: The average mark achieved in practical assessments is similar.

$H_{0.1.2.1}$: The average mark is similar for the same problem across programming notations and development environments.

$H_{0.1.2.2}$: The average mark is similar for the same problem using Delphi™ Enterprise.

$H_{0.1.2.3}$: The average mark is similar when solving for syntactical errors using Delphi™ Enterprise.

$H_{0.1.2.4}$: The average mark is similar for the same problem for individual choice of programming notation and development environment.

$H_{0.1.3}$: The average class mark is similar.

$H_{0.1.4}$: The average final mark is similar.

**$H_{0.2}$: The observed throughput in an introductory programming course is independent of programming notation and development environment.**

$H_{0.2.1}$: The throughput in pen-and-paper assessments of solution comprehension and composition is similar.

$H_{0.2.2}$: The throughput in practical assessments is similar.

$H_{0.2.2.1}$: The throughput is similar for the same problem across programming notations and development environments.

$H_{0.2.2.2}$: The throughput is similar for the same problem using Delphi™ Enterprise.

$H_{0.2.2.3}$: The throughput is similar when solving for syntactical errors using Delphi™ Enterprise.

$H_{0.2.2.4}$: The throughput is similar for the same problem for individual choice of programming notation and development environment.

$H_{0.2.3}$: The class mark throughput is similar.

$H_{0.2.4}$: The final mark throughput is similar.

*Figure 6.3: Refinement of Hypotheses*

### *6.3.2 Participant Population and Sample Size*

In an accurate experimental design, attention is required to be paid to the manner in which participants for the study are selected (Whitley 1997). The design of the current study requires the ability to identify the participants from within the student body registered for the introductory programming course for the 2003 academic year, the participation for which UPE's Ethic's Committee gave approval in the previous academic year (Appendix G).

The students that are considered as potential participants are historically first year students, this deliberation being in line with that of prior similar research at UPE (Calitz 1997; Greyling 2000). Consequently, no repeating registrations or registrations after the first year of registration at UPE are included as participants. Repeating students as well as those students who are not historically first year students, even though first time registrations for the introductory programming course, cannot be compared with students entering the university for the first time. This is due to the existence of a certain amount of bias due to exposure to a tertiary level learning environment in the former category of students.

Similar prior studies in programming have indicated that introductory course programmers can be identified as a heterogeneous population (Whitley 1997). The experience level can vary from first exposure with computers to highly competent. This fact can create difficulties when effects of independent variables are hidden by the variances within the participant population. A further factor influencing the identification of potential participants in the study is that if students are requested to volunteer for a programming exercise, anyone who perceives themselves as being poor in programming is unlikely to choose to participate (McCracken *et al.* 2001). Consequently, a particular method is applied in the design of the experiment forming the focus of the current investigation, specifically in connection with the identification of the participants.

In the current study, the participants are preferably required to be randomly assigned to treatment and control groups resulting in a between-groups experimental design (Whitley 1997). In an educational environment like UPE, the random assignment of participants to groups is impractical since UPE is a campus where students, in many cases, and especially in the Department of CS/IS, decide their own course timetable for learning activities.

Consequently, for the purposes of the current investigation, allowance is made for the use of control and treatment groups being acknowledged as independent course sections, as is reported in a similar study (Applin 2001). All historically first year registered students for the introductory course are obligated to take part in the study in either the control or treatment populations. While neither the participants nor the treatment is randomly assigned, each of the control and treatment groups is stratified sample based (Berenson *et al.* 1999).

The preference for a stratified sample based analysis is based on the fact that this type of sample provides an efficient way of ensuring a representation of students across the entire population. In turn, this ensures greater precision in the estimates of the underlying population parameters and also ensures homogeneity of students within each stratum. The stratified samples for the current study are based on discrete pre-test measures within the independent course sections, namely the control and treatment groups.

The discrete pre-test measures are the results, or predicted marks, obtained upon the participants undertaking UPE's placement test, the background and implementation of which was discussed in Chapter 4 (Section 4.3). The pre-test applied to the participants in the study measures a related aspect of the learning material for an introductory programming course but not knowledge of the material itself. The pre-test is considered valid and reliable in that the test itself has been researched and studied over a period of time and statistical measures are available for it (Greyling 2000; Greyling *et al.* 2002; Greyling *et al.* 2003).

The participants of each of the control and treatment groups are grouped according to strata based on the discrete pre-test measure of predicted mark for the introductory programming course at UPE, the characterization of the strata being described in Definition 6.1. A historically first year student registered for the introductory programming course in either of the control or treatment populations is a participant of a particular stratum if the student's predicted mark falls within the range specified for the relevant stratum. Each stratum is uniquely identifiable within the control and treatment populations. Each participant is similarly uniquely identifiable within the entire introductory programming course population.

**Participant $j \in$ Stratum$_i^m$ $\Leftrightarrow$**

**_pre-test measure_(Participant _j_) $\in$ [_36+5i … 40+5i_] $\wedge$**

**_experimental group_(Participant _j_) = _m_,**

**where $i = 1, 2, \ldots, 12$ and $m \in$ {_treatment group, control group_}**

*Definition 6.1: Membership of Participants in Strata*

Due to a minimum pre-test measure of 40% (predicted mark) being applied in the selection and placement model at UPE for the introductory programming course (Greyling *et al.* 2003), the current empirical analysis consequently comprises of a total of 12 strata per experimental group. These stratum identifiers together with appropriate discrete pre-test measure ranges are listed in Table 6.6.

| **Stratum Identifier**<br>**(_m_ ∈ {_treatment group, control group_})** | **Predicted mark range** |
|---|---|
| Stratum$_1^m$ | 41, 42, …, 45 |
| Stratum$_2^m$ | 46, 47, …, 50 |
| Stratum$_3^m$ | 51, 52, …, 55 |
| Stratum$_4^m$ | 56, 57, …, 60 |
| Stratum$_5^m$ | 61, 62, …, 65 |
| Stratum$_6^m$ | 66, 67, …, 70 |
| Stratum$_7^m$ | 71, 72, …, 75 |
| Stratum$_8^m$ | 76, 77, …, 80 |
| Stratum$_9^m$ | 81, 82, …, 85 |
| Stratum$_{10}^m$ | 86, 87, …, 90 |
| Stratum$_{11}^m$ | 91, 92, …, 95 |
| Stratum$_{12}^m$ | 96, 97, …, 100 |

*Table 6.6: Strata in Empirical Study*

The sample size (*n*) of the participants in each of the treatment and control groups in the current investigation is equal in value and given by Definition 6.2. To maintain a balance between the sample sizes in the strata of the control and treatment groups, the size of the sample for stratum *i* for each group is restricted to be the minimum of the population sizes of stratum *i* in each of the experimental groups. Each stratum *i* in each of the treatment and control groups consists of $n_{\text{Stratum}i}$ participants randomly selected from the stratum population of students in each group who complete the introductory programming course and as a consequence also have complete sets of the required assessment performance achievement measures. The total number of participants in the current study is thus 2*n*.

$$\boldsymbol{n = \sum_{i=1}^{12} n_{\text{Stratum}i}, \ \text{where } n_{\text{Stratum}i} = \mathit{minimum}(\mathit{count}(\text{Stratum}_i^{\mathit{treatment\ group}}), \mathit{count}(\text{Stratum}_i^{\mathit{control\ group}}))}$$

*Definition 6.2: Sample Size*

The participants in the current investigation receive no reward other than the grades for the introductory programming course that they in any case would be entitled to in the absence of the current research.

Once the performance achievement measures and qualitative data for the participants in the sample have been collected, various techniques are applied in order to express a decision in terms of the focus of the study. The following section consequently focuses on the data analysis techniques applicable to the current research.

### *6.3.3 Data Analysis Techniques*

Prior to the application of any data analysis technique to the collected data, all data is required to be thoroughly prepared for examination. The procedure of data preparation consists of the coding, transcribing and cleansing of the data as well as with the process of discarding of any outliers.

The following are the rules adopted for the current study regarding the possible discarding of data collected for the participants in each of the treatment and control groups of the current investigation:

- Discard all quantitative data for a participant in the treatment group who elects to withdraw from the study. In such an event, conduct a face-to-face interview with the participant in order to determine the reasons for withdrawing from the study for the purposes of collecting qualitative data as to the participant's attitude and motivation towards the programming notations and development environments being used.
- Discard all quantitative data for a participant who does not have a complete set of data. This includes all participants who
  - are successful in a Departmental competency test to assess the level of prior learning and are thus not required to register for the introductory programming course in 2003;
  - elect to change registration to that of the slower paced alternative introductory programming course; or
  - cancel registration for the introductory programming course.
- Discard all quantitative data for a participant who has been awarded any kind of supplementary assessment in the introductory programming course, irrespective of whether the assessments are for acceptable leave of absence or for re-assessment purposes. The reason for this decision is that due to the timing of the supplementary assessment, the assessment material has the potential to differ from the original assessment materials in content. Further, the participants exposed to supplementary assessment have the advantage of being given a second chance. Consequently, the data cannot be used in statistical comparison without the existence of a possibility of skewing the statistical analysis.
- Assume that

$$n_1 = \mathit{count}(\text{Stratum}_i^{\mathit{treatment\ group}}) \text{ and}$$
$$n_2 = \mathit{count}(\text{Stratum}_i^{\mathit{control\ group}}), \quad \text{where } i = 1, 2, \ldots, 12$$

> If $n_1$ > $n_2$, discard all quantitative data for ($n_1$-$n_2$) randomly selected participants from the treatment group, otherwise if $n_2$ > $n_1$, discard all quantitative data for ($n_2$-$n_1$) randomly selected participants from the control group.

The Department of CS/IS at UPE implements a policy whereby all students in the introductory programming course who fail to obtain a minimum class mark of 40%, made up of a weighted average of the practical tasks assessments, two practical and two pen-and-paper tests, are not permitted to write the final examination. Due to the fact that these students have in fact completed yet are not successful in the introductory programming course, EXCEL's (Microsoft Corporation 2002) **FORECAST** function is used to predict both the examination and final grades for the relevant participants for inclusion in the data collection process.

Analysis of all the resulting data collected for the current study concurrently takes place in two ways, namely using quantitative and qualitative approaches. The following two subsections discuss the techniques relevant to each of these approaches.

### Quantitative Statistical Techniques and Test Statistics

Two types of test statistics are applicable to the quantitative analysis of academic performance in the current study. These tests statistics are a computed mean being the average mark achieved by participants in the study and a computed proportion being the pass rate (or observed throughput) achieved by participants in the study. The following statistical techniques are thus appropriate to the current study (Berenson *et al.* 1999):

- Pooled-variance two-tailed t-test for the testing of the difference in the two average marks computed for each of the control and treatment groups; and
- $\chi^2$-test for homogeneity of proportions using a contingency table for testing the equality of the pass rates computed for each of the control and treatment groups.

The first of the statistical techniques, which is a common method used in programming studies for the comparison of two means (Chamillard *et al.* 2000), assumes that the samples are drawn from underlying normal populations with equal variances. In the case of the total sample size (Definition 6.2) being less than the recognised large sample size[36], it is required that the assumption of normality be assessed.

In order to assess the reasonableness of the assumption of normality and the equality of the variances in the treatment and control groups, STATISTICA (StatSoft Inc. 2001), a data analysis software package, is used for exploratory data analysis. In particular STATISTICA is used to compute the standard descriptive statistics for the performance achievement measures used to compute the average mark for the comparison of the two means.

Visual confirmation for the fit of a theoretical distribution to the observed data is done by examining the probability-probability plot (P-P plot). In a P-P plot, the observed cumulative distribution function is plotted against the theoretical cumulative distribution function. If the theoretical cumulative distribution approximates the observed distribution well, then all points in the plot should fall onto the diagonal line. If deemed necessary, the P-P plot facility of STATISTICA is used to visually confirm whether the two groups in the study are drawn from underlying normal populations with equal variances. Thereafter, the pooled-variance two-tailed t-test for testing the difference between two means is applied to test the leaf hypotheses appearing on the left hand side of Figure 6.3 ($H_{0.1.1}$, $H_{0.1.2.1}$, $H_{0.1.2.2}$, $H_{0.1.2.3}$, $H_{0.1.2.4}$, $H_{0.1.3}$, $H_{0.1.4}$), namely those hypotheses that have been refined from hypothesis $H_{0.1}$.

The $\chi^2$-test for homogeneity of proportions uses a 2×2 cross-classification table resulting from a survey of the pass rates of participants in the control and treatment groups. Using this statistical technique, the equivalence of the pass rates at 1 degree of freedom at the 95% percentile ($\alpha = 0.05$; critical value of $\chi^2$-statistic = 3.841) is tested. The $\chi^2$-test for homogeneity of proportions is applied to test the leaf

[36] Law of Large Numbers and the Central Limit Theorem (Larson 1974).

hypotheses appearing on the right hand side of Figure 6.3 ($H_{0.2.1}$, $H_{0.2.2.1}$, $H_{0.2.2.2}$, $H_{0.2.2.3}$, $H_{0.2.2.4}$, $H_{0.2.3}$, $H_{0.2.4}$), namely those hypotheses that have been refined from hypothesis $H_{0.2}$. STATISTICA is the data analysis tool used in the computations required for both the t- and $\chi^2$-tests.

Besides these quantitative statistical techniques and test statistics, qualitative data analysis is also relevant to the current investigation. The analysis technique applicable to this approach in the investigation is the focus of the following subsection.

Qualitative Techniques

Qualitative analysis of data typically occurs concurrently with the qualitative data collection process (Ely *et al.* 1995) resulting from oral and written interviews, surveys, essays and observations (Merriam 1998). Qualitative analysis of data is useful for analysing data in order to determine tendencies and trends (Dee Medley 2001), specifically for the discovering of hypotheses in the form of models that describe the findings (Merriam 1998). Techniques of qualitative analysis thus provide a facility to discover patterns within the context of an environment under investigation. The analysis process therefore involves the establishment of an initial set of categories that arise from and give significance to the specific data collected.

As the data is collected, it is coded and structured into categories or bins (Ely *et al.* 1999). This process, the outcome of which is textual data, serves to organise the qualitative data into some meaningful context. The qualitative data contributions for the current investigation originate predominantly from the participants in the treatment group since these participants are in the unique position of being able to judge the two programming notations and development environments being compared. The participants in the treatment group are thus the most likely to contribute to the understanding of concerns raised in the current study (Merriam 1998).

During the data collection and analysis, it may become apparent that further data be required to be collected to confirm or explain observations. This fact further emphasises the need to perform the qualitative data collection and analysis in parallel. The majority of the qualitative data collection for the current investigation is done by means of pen-and-paper based surveys. This method is preferred due to the need for a large volume of data to be collected and analysed within the shortest possible time. Further, this technique allows for a larger sample of participants to be effectively processed. In appropriate situations during the current investigation, data collection is done by means of one-on-one interviews to complement the qualitative research process.

The final analysis of qualitative data involves the search for and determination of themes from the identified categories. A theme is defined (Ely *et al.* 1995; Ely *et al.* 1999) as a statement of meaning that

- runs through all or most of the pertinent data; or
- one that carries heavy emotional or factual impact.

Themes can be identified as the explicit or implied attitudes towards an observed behaviour. Thematic analysis is consequently used to present the findings assembled from the various qualitative surveys administered in the current research. The thematic analysis is complemented by frequency counts in order to compare the quantitative loadings in the identified categories and themes. A further need for the inclusion of frequency counts is that of the presentation of a quantitative category profile for each of the treatment and control groups. Where appropriate in the current investigation, further quantitative statistical analysis in the form of rankings and standard descriptive statistics complements the presentation of the frequency counts.

During the design of the experimental study for the current investigation, a number of risks are evident. The discussion of the identified risks and strategies proposed to address each of them is the focus of the following section.

## 6.4 Risks to Investigative Study

Every experimental proposal has risks associated with it (Applin 2001). Consequently, cognisance of risks relevant to the current empirical investigation is made in this section. The risks identified are associated with sample size, elimination of bias in performance achievement measures, method employed in the administering of practical learning activities and the formal assessment of practical performance within the context of laboratory sessions.

### *6.4.1 Sample Size*

A major concern in the current investigation is that of sample size due to a predicted small maximum population size of between 200 – 250 students in the introductory programming course and historically high withdrawal of participants not only voluntarily from the experiment but also from the course as a whole (Figure 6.1). Strategies to address this possibility incorporate:

- the inclusion of continuous qualitative data collection and analysis in the current research; and
- a process of participant allocation that initially maximises and equalises the population sizes of the strata from which the samples are drawn for each of the treatment and control groups.

The first strategy of qualitative analysis has been discussed in detail in Section 6.3.3. The second strategy entails the initial allocation of introductory programming course students to control and treatment groups at commencement of the treatment period ensuring that each group's population is similar in volume, with approximately 50% of the introductory programming course student population being allocated to each group. In this way, an assumption is made of the equal probability of attrition from either group. It is further noted in an earlier related programming study (Applin 2001) that tight experimental controls can warrant valid statistical analysis with samples as small as 15 per group. With this minimum sample size in mind, the definition of the strata (Definition 6.1) is customised during the data analysis process as required.

Another risk identified is the fact that the assessment of performance achievement of sample participants in the introductory programming assessment materials might be subject to bias. This risk is addressed in the following section.

### *6.4.2 Bias in Performance Achievement Measures*

A potential exists for bias in performance achievement measures to be evident on the part of the assessors who draft the assessments as well as the assessors who measure the performance achievement. Two strategies are proposed to address this particular area of risk.

Firstly, all measuring of performance achievement for participants in the study is conducted by the author, being the primary investigator in the current research. In this way a consistent approach to the assessment process is ensured and the risk of inconsistent interpretation of assessment criteria is minimised, if not eliminated entirely.

Secondly, all material in the introductory programming course is subject to a stringent moderation process. All training and assessment material drafted by the introductory programming course instructors is moderated by both the author and a departmental academic external to the introductory programming course, namely one of the promoters of the report of the investigation. Both of these individuals have in the recent past themselves been introductory programming course instructors in the Department of CS/IS at UPE for at least 4 years each. Similarly, all material drafted by the author for use in the gathering of data for the investigation, is subject to moderation by the introductory programming course instructors as well as the previously identified promoter. Further, all measures of performance achievement are subject to moderation from the introductory programming course instructors as well the investigation promoter mentioned earlier.

A further potential for evidence of bias in performance achievement measures exists on the part of the participants in the treatment group. Ethics dictates that the treatment group participants are exposed to both the B# as well as the Delphi™

Enterprise programming notation and development environments. The basis for this decision is to prevent unfair discrimination against a certain population of students in the current study. The decision is also due to the requirements of the introductory programming course outcome that all students who successfully complete the introductory programming course at UPE should exhibit evidence of proficiency in both the PASCAL programming notation and the Delphi™ Enterprise programming development environment. The potential for the emergence of a less positive attitude and motivation on the part of the treatment group participants is thus closely monitored by means of the qualitative study described in the previous section.

Performance achievement measures are an assessment of the existence of introductory programming skills. Consequently, practical learning activities are required for the development of skills in the programming notations and development environments used as the instruments in the current study. The use of this type of learning activity has its own associated risks, each of which are reviewed in the section that follows.

### *6.4.3 Administering Practical Learning Activities*

In the application of practical learning activities, algorithms are placed in some kind of context by means of story problems (Jonassen 2000). The practical tasks thus are in the form of values required to solve an algorithm embedded within a brief narrative or scenario. Consequently, participants in the current study are required to extract the values from the narrative and use them in the most appropriate programming constructs to solve the problem. The process described is a more complex cognitive process than merely reading and comprehending a given procedural algorithm.

Risks with the approach described above are that the scenario contexts for the practical problems, due to their introductory level and small program solution requirements, are often superficial and uninteresting to students. The result is that when students attempt to transfer practiced narrative problem skills to other problems, the focus is on surface features often resulting in the recollection of familiar solutions from sometimes unrelated previously solved problems (Jonassen 2000). Students often do not successfully comprehend the underlying principles and abstract concepts and are thus not able to transfer the ability to solve one kind of problem to problems with the same structure but different features. In addressing this risk, collaboration of participants during practical learning activities is encouraged. Further, senior students in the role of trained introductory programming course tutors are available for consultation in both the program and problem domains in the approximate proportion of 1:13 (tutor:students) during practical learning activities.

Practical learning activities are highly vulnerable to plagiarism. There is a forced acceptance of the existence of the characteristic on the part of the instructors as a direct by-product of collaborative learning. However, random assessment of student practical tasks by the introductory programming course instructors is an attempt to discourage the practice and plays a part in reducing the practice but provides no evidence that it is eliminated.

A further concern related to practical learning activities is that insufficient time is available in a single practical learning activity for the completion of practical tasks by some portions of the introductory programming course student population. The situation specifically disadvantages those students not able to spend adequate time on the tasks due to limited hardware accessibility or other degree programme commitments. In order to address this concern, a maximum timeframe of a week is provided for the completion of any particular week's practical tasks for the purposes of the current research.

Participants in the treatment group are exposed to a further risk of knowledge overload. In addition to performing all the tasks required for the purposes of the introductory programming course, the treatment group participants are also expected to master the skills of an additional programming notation and development environment, namely B#. The only learning support available in this regard is in the form of paper resources which takes the form of pictorial instructions that illustrate how to use the environment to create sample B# program solutions. These learning resources complement the weekly practical sheets and thus consist of 9 weekly handouts of 75 pages in total (an average of 8 pages per handout with the largest being 27 pages and the smallest being 3 pages in length).

In order to become proficient in B#, treatment group participants are expected to study this learning resource over and above any other introductory programming course commitments. No additional time is provided for this task. It is obvious to deduce, then, that treatment group participants might be disadvantaged in the course of the current investigation with respect to time available for completing required tasks and are thus vulnerable to poor achievement performance. In an attempt to address and control this risk, B# is designed to be as intuitive as possible, specifically exhibiting consistency with respect to the implementation of programming constructs (Chapter 5).

Treatment group participants are also required to complete programming notation and development environment evaluation surveys throughout the treatment period. A risk exists that since the treatment group are consequently at all times aware of being involved in an experiment, such awareness is sufficient to bias the results of the investigative study (Applin 2001). In an attempt to address this risk, the maximum volume of qualitative data is required to be collected from the practical development environment surveys and analysed on a weekly basis, thereby making provision for an instantaneous reactive qualitative response to any observed trend.

Furthermore, the longer a treatment period of an experiment persists, the more likely the treatment and control groups are to notice each other resulting in a possible distortion in the results of the study. The actual treatment period of 9 weeks makes the current study vulnerable to this risk. In order to reduce the effects of this risk, participants in the control and treatment groups are allocated to distinct practical learning activity sessions. Consequently, no practical learning activity session contains participants from both groups. Further, participants are discouraged from attending practical learning activities dedicated to the group that is alternative to their own by means of a rigorous roll call approach.

Formal assessment of programming notation and development environment skills is administered by means of practical tests which take place within stringently monitored laboratory conditions. This category of assessment is also prone to risks, which are the focus of the next section.

### 6.4.4 Formal Assessment of Practical Performance in Laboratory Sessions

Any assessment of academic ability is likely to be considerably more cognitively demanding and participants taking tests in laboratory sessions may specifically be affected by computer anxiety (McDonald 2002). Consequently, a concern with the practical assessment of programming skills in the relevant programming notations and development environments is that practical tests could be perceived as being unfair to those students who experience test anxiety when performing under time pressure (McCracken *et al.* 2001). Experiences of anxiety tend to reduce the capacity of working memory, which has the potential to negatively impact on the achievement performance measures of the participants in the study, and ultimately on the statistical equivalence of the measures.

In an attempt to reduce the impact of this risk to the experiment, two independent practical tests are administered. The final performance achievement of participants relies thus on a weighted average of the independent practical tests. Further, the problems posed in the practical tests are similar in structure to those for which program solutions are required during practical learning activities.

Limited hardware resources in the Department of CS/IS at UPE necessitate the existence of multiple successive practical test sessions per practical test assessment activity. In order to eliminate the occurrence of plagiarised program solutions during single sessions, multiple question papers per individual practical test session are implemented. Consequently, performance achievement measures used in the comparative analysis do not necessarily have the exact same question paper as their source. It remains a difficult process to measure the equivalence of different practical tasks across question papers.

In order to reduce the impact of different question papers on the comparative analysis, a moderation process similar to that described earlier (Section 6.4.2) is in place to judge equivalence of tasks across differing question papers as accurately as possible within the existing physical constraints. To complement this process in the interests of a more accurate comparative analysis, selected questions are combined in such a way that the practical test questions themselves are discreetly duplicated across the treatment and control groups without the occurrence of the duplication of an entire question paper.

## 6.5 Conclusion

Studies in programming similar to the process described in this chapter which have investigated visual programming notations have compared them to textual programming notations with mixed results[37]. The lack of confirmatory analysis can be attributed to a number of problems generally experienced with such comparative studies (Section 6.1). The methodology of the current investigation addresses these problems with the application of a formal evaluation strategy of hypothesis testing. Furthermore, anecdotal evidence gathered is applied to a strategy of thematic analysis. The context in which the current investigation takes place is within the same course during the same year and using the same curriculum. This is necessary to prevent any obscuring of results observed.

The lack of existing confirmatory analysis in related studies requires that the current investigation examines the hypothesis that novice programmer academic performance is independent of the programming notation and development environment used as a technological delivery method in the learning environment for an introductory programming course at tertiary level. This investigation is the culmination of a long process of research in the Department of CS/IS at UPE.

The hypothesis testing method adopted in the current investigation compares the academic performance achievement differences between two experimental groups of participants, the control and treatment groups. The investigation is for statistically significant variation in performance that is expected to result from the use of comparable programming notations and development environments assigned to each group. The longitudinal study described in this chapter makes use of two programming instruments. These instruments are Delphi™ Enterprise, a traditional commercial textual programming notation and B#, an experimental iconic programming notation. Each instrument has an associated development environment.

[37] (Green *et al.* 1991; Koelma *et al.* 1992; Moher *et al.* 1993; Lord 1994; Schiffer *et al.* 1995; Whitley 1997; Blackwell *et al.* 1999a; Deek *et al.* 2002; Cooper *et al.* 2003; Sanders *et al.* 2003a)

An observed statistically significant variation in measured academic performance in the favour of the experimental iconic programming notation and its associated development environment would support the finding that programming in the form of B# flowchart program solutions influences the learning of programming.

Performance achievement of participants in the study is assessed in terms of averages of individual performance achievement measures and proportional rate of success per group. In anticipation of a valid statistical comparison, quantitative data relevant to the measures of novice programmer performance are collected by means of a number of assessment materials, which are administered during a treatment period covering 67% of the duration of the introductory programming course at UPE during 2003. Other material administered includes a demographic and various qualitative surveys, as well as a consent form since ethics dictates that all human subjects be informed that an experiment is being conducted and that the participants in the study permit the use of the data collected during the experiment period. Analysis of the qualitative data collected produces a better understanding of the larger picture in the environment under investigation. This analysis technique is suited to research projects involving smaller groups of participants, which is the predicted case of the current investigation.

The described nature of the introductory programming course environment at UPE results in the observation of a number of potential risks to the current investigative study, the greatest of these being a potentially seriously limiting sample size as well as the application of multiple programming notations and development environments concurrently for ethical reasons. A high tendency towards a large attrition rate in the introductory programming course at UPE is a potential contribution to a reduced sample size which could negatively impact the envisaged quantitative data analysis. Ethics dictates the necessity of the treatment group to make use of both instruments in order to prevent any unfair discrimination against them in terms of the curriculum requirements of the introductory programming course at UPE. The resulting situation possesses a potential to impact on the attitude and motivation of participants in the treatment group, thereby skewing performance measurements.

In an attempt to respond to any potentially less positive findings, a qualitative study is administered concurrently with the comprehensive quantitative study. The primary purpose of the qualitative study is to determine the attitude and motivation of participants in the treatment group.

The sound academic methodology reported on in this chapter motivates the necessity for a stratified based sample of participants. The preference for this type of sample is in order to ensure a representation of students across the entire introductory programming course population in the Department of CS/IS at UPE. The empirical investigation methodology also identifies a number of analysis techniques for use in addition to that of initial descriptive statistics.

| | **Material** | **Test Statistic** | **Data Analysis Technique** |
|---|---|---|---|
| M4 | Development environment practical task evaluation forms<br>(eight weekly documents) | None | Thematic analysis |
| M5 | General development environment evaluation questionnaire<br>(single document) | None | Thematic analysis |
| M6 | Pen-and-paper tests<br>(two documents) | Mean | Pooled-variance two-tailed t-test |
| | | Proportion | $\chi^2$-test for homogeneity of proportions |
| M7 | Practical tests<br>(two documents) | Mean | Pooled-variance two-tailed t-test |
| | | Proportion | $\chi^2$-test for homogeneity of proportions |
| M9 | Pen-and-paper examination<br>(single documents) | Mean | Pooled-variance two-tailed t-test |
| | | Proportion | $\chi^2$-test for homogeneity of proportions |
| M10 | Tutor and student attitude to practical learning activity evaluation form<br>(single document) | None | Thematic analysis |

*Table 6.7: Mapping of Test Statistics and Data Analysis Techniques to Data Collected*

The main analysis techniques decided upon are the pooled-variance two-tailed t-test, $\chi^2$-test for homogeneity of proportions and thematic analysis. Each of these techniques has been selected in terms of their analysis suitability to the different kinds of data being collected. A mapping of the relevant test statistics and data analysis techniques to the material collected during the current empirical investigation is shown in Table 6.7. The biographical form (*M1*) is used to characterise the participants in the study. The consent form (*M2*), B# training manual (*M3*) and practical sheets (*M8*) are primarily information sharing materials. All of these materials are thus not subject to any of the selected data analysis techniques.

The results of the application of all the different data analysis methods described in this chapter are presented in the next chapter (Chapter 7), with a discussion of the results appearing in Chapter 8.

# Chapter 7

# Results of Investigation

## 7.1 Introduction

Novice programmer requirements for technological support in the learning environment of an introductory programming course have been derived from an extensive literature study and are presented in Chapter 2 (Table 2.1). Evidence in available literature suggests that there remains a need for technological support in the learning environment of an introductory programming course that satisfies these identified requirements (Chapter 3). In response to this challenge, B#, an iconic programming notation and development environment was recently developed in the Department of CS/IS at UPE (Chapter 5).

A subsequent conclusion of Chapter 5 is that B# satisfies the majority of the novice programmer requirements in terms of its design and implementation (Table 5.4 duplicated in Table 7.1). The methodology for determining B#'s support for the remaining novice programmer requirements (*R3* and *R8* in Table 7.1) is presented in Chapter 6.

| Requirements for Novice Programmer Technological Support | Supported |
|---|---|
| ***R1***: Elimination of finer implementation details typically found at the superficial learning level of the program domain | ✓ |
| ***R2***: Increased level of program solution comprehension at the in-depth learning level of the program domain | ✓ |
| ***R3***: Increase in level of motivation when using the programming notation | **To be determined by the current study** |
| ***R4***: Designed specifically for use by novice programmers | ✓ |
| ***R5***: Provision of visual techniques to aid comprehension process at the in-depth learning level of the program domain | ✓ |
| ***R6***: Support for reduced mapping between the problem and program domains | ✓ |
| ***R7***: Increased focus on problem-solving | ✓ |
| ***R8***: Increase in novice programmer performance achievement measured in terms of higher level of accuracy in program solutions in program domain | **To be determined by the current study** |

*Table 7.1: Support for Novice Programmer Requirements by B#*

In light of the preceding discussion, the following hypothesis (duplicated from Sections 6.3.1 and 6.3.2) has been formulated in Chapter 6 for examination and testing for significance at the 95% percentile ($\alpha = 0.05$) in order to determine B#'s support for novice programmer requirement *R8* while controlling for predicted mark by means of a stratified sample based analysis:

$H_0$: *Academic performance in an introductory programming course is independent of programming notation and development environment.*

$H_1$: *Academic performance in an introductory programming course is dependent on programming notation and development environment.*

The focus of this chapter is on the data collection process and computation of sample values of test statistics according to the methodology described in Chapter 6 to provide data for deliberation (Chapter 8) on the formulated hypothesis. The chapter reports on the observed results of an empirical study conducted during 2003 on novice programmers in an introductory programming course in the Department of CS/IS at UPE.

An observed statistically significant variation in measured performance in the favour of B# would support the finding that the use of the B# iconic programming notation and development environment is beneficial to novice programmers in terms of academic performance achievement.

The chapter discusses the process followed in categorising participants into appropriate stratified samples (Section 7.2) according to predicted mark (Chapter 4), acknowledging that the total sample size for the study is an identified risk (Section 6.4.1). The results of the quantitative (Section 7.3) and qualitative (Section 7.4) analysis techniques applied to the selected sample of participants and described in the previous chapter are presented. Further analysis of surveys administered (Section 7.5) are presented to supplement the quantitative and qualitative analysis. The primary purpose of this chapter is thus to perform a comparative analysis and make statistical decisions.

## 7.2 Selection of Participants

The selection of participants in the current investigation is at risk of large scale attrition, as has also been the case in the past (Figure 6.1). As a result, the defined strata (Definitions 6.1 and 6.2) are modified in order to maximise the total sample size of participants allocated to each of the control and treatment groups (Section 7.2.1). Furthermore, due to the nature of the focus of the current investigation being on the determination of motivation and academic performance, any potential bias due to demographic (Section 7.2.2) and course instructor (Section 7.2.3) influence needs to be determined.

### *7.2.1 Sample Size*

Table 7.2 shows the initial population size (N) and potential sample size (*n*) for each of the strata in the treatment and control groups, taking the attrition (D) in the introductory programming course during 2003 into consideration. The treatment group constitutes 46% (*n* = 96) of the total population of potential participants in the study. The cells that are shaded green indicate the actual sample size for each stratum according to Definition 6.2. The column under which the green shaded cells fall (treatment or control group) indicates the experimental group that influenced the decision of sample size per stratum. The final sample size per stratum in each experimental group is taken to be the minimum of the sample sizes of the treatment and control groups for that particular stratum (Definition 6.2). The total sample size for each of the treatment and control experimental groups is 58 participants, giving a total of 116 participants in the study.

| | | **Treatment Group** | | | **Control Group** | | | **Sample Size** | |
|---|---|---|---|---|---|---|---|---|---|
| **SId** | **P** | **N** | **D** | ***n*** | **N** | **D** | ***n*** | **Treatment** | **Control** |
| $S_1$ | 41 – 45 | 7 | 3 | **4** | 6 | 2 | 4 | 4 | 4 |
| $S_2$ | 46 – 50 | 10 | 2 | **8** | 11 | 2 | 9 | 8 | 8 |
| $S_3$ | 51 – 55 | 13 | 7 | **6** | 20 | 6 | 14 | 6 | 6 |
| $S_4$ | 56 – 60 | 19 | 4 | **15** | 24 | 6 | 18 | 15 | 15 |
| $S_5$ | 61 – 65 | 19 | 8 | **11** | 36 | 8 | 28 | 11 | 11 |
| $S_6$ | 66 – 70 | 24 | 10 | 14 | 15 | 3 | **12** | 12 | 12 |
| $S_7$ | 71 – 75 | 4 | 2 | **2** | 3 | 0 | 3 | 2 | 2 |
| $S_8$ | 76 – 80 | 0 | 0 | **0** | 0 | 0 | 0 | 0 | 0 |
| $S_9$ | 81 – 85 | 0 | 0 | **0** | 0 | 0 | 0 | 0 | 0 |
| $S_{10}$ | 86 – 90 | 0 | 0 | **0** | 0 | 0 | 0 | 0 | 0 |
| $S_{11}$ | 91 – 95 | 0 | 0 | **0** | 0 | 0 | 0 | 0 | 0 |
| $S_{12}$ | 96 – 100 | 0 | 0 | **0** | 0 | 0 | 0 | 0 | 0 |
| | Total | 96 | 36 | 60 | 115 | 27 | 88 | 58 | 58 |
| | Proportion of **N** | 100% | 37% | 63% | 100% | 23% | 77% | 60% | 50% |

***Key to abbreviations***

| | | | |
|---|---|---|---|
| **SId** | Stratum identifier (Definition 6.1) | | |
| **P** | Discrete range of predicted marks | **N** | Population size |
| **D** | Discarded samples size | ***n*** | Potential sample size |

*Table 7.2: Sample Derivation*

The high frequency of nil values in strata $S_8$ – $S_{12}$ is directly due to the placement model currently in place in the Department of CS/IS at UPE (Section 4.3). The final mark predicted for any participant in 2003 did not exceed that of 72%.

Due to the resulting post-attrition small sample sizes per individual stratum (Table 7.2), the strata are combined to form three discrete logical strata, each with a more practical sample size (Applin 2001). A similar approach of 3 discrete logical groups is adopted in previous related research conducted by the Department of CS/IS at UPE (Calitz 1997; Greyling 2000).

The identifiers designated to each of these logical strata are namely *high risk stratum*, *medium risk stratum* and *low risk stratum*. The mapping of these logical strata to the strata defined by Definition 6.1 and shown in Table 7.2 is given in Table 7.3. Using this technique of merging of strata to counteract the risk of inconsequential sample sizes (Section 6.4.1), the sample size of each of the treatment and control groups is maximised to 59 participants, resulting in a total of 118 participants in the study (comprising of 62% of treatment population and 51% of control population).

| | | | **Treatment Group** | | | **Control Group** | | | **Sample Size** | |
|---|---|---|---|---|---|---|---|---|---|---|
| **LSId** | **SId** | **P** | **N** | **D** | ***n*** | **N** | **D** | ***n*** | **Treat-ment** | **Con-trol** |
| High risk | $S_1$<br>$S_2$ | 41–50 | 17 | 5 | **12** | 17 | 4 | 13 | 12 | 12 |
| Medium risk | $S_3$<br>$S_4$<br>$S_5$ | 51–65 | 51 | 19 | **32** | 80 | 20 | 60 | 32 | 32 |
| Low risk | $S_6$<br>$S_7$<br>$S_8$<br>$S_9$<br>$S_{10}$<br>$S_{11}$<br>$S_{12}$ | 66–100 | 28 | 12 | 16 | 18 | 3 | **15** | 15 | 15 |
| | | Total | 96 | 36 | 60 | 115 | 27 | 88 | 59 | 59 |
| | | Proportion of **N** | 100% | 37% | 63% | 100% | 23% | 77% | 62% | 51% |

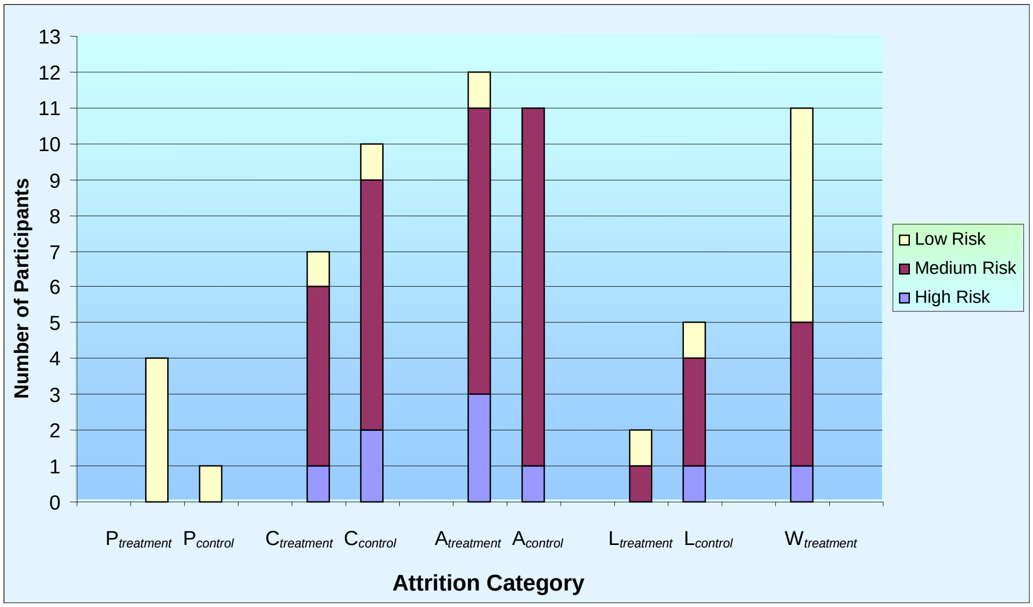
***Key to abbreviations***

| | | | |
|---|---|---|---|
| **LSId** | Logical stratum identifier | **SId** | Stratum identifier (Definition 6.1) |
| **P** | Discrete range of predicted marks | **N** | Population size |
| **D** | Discarded samples size | ***n*** | Potential sample size |

*Table 7.3: Adjusted Sample Derivation*

Data for participants are discarded if they fall into any of 5 categories judged to be relevant to the current investigation (Section 6.3.3). These categories are namely recognition of prior learning, cancellation of registration, changing registration to a slower paced introductory programming course, leave of absence and voluntary withdrawal from the investigation.

Decomposition into these categories of attrition (D) of the population participants for each of the treatment and control groups is illustrated by Figure 7.1. Attrition due to voluntary withdrawal from the study ($W_{treatment}$) is only applicable to participants in the treatment group since the control group is exposed to only the prescribed technological support for UPE's introductory programming course. Participants in the control group are thus not eligible for voluntary withdrawal from the investigative study.



***Key to abbreviations*** ***(m ∈ {treatment, control})***

| | | | |
|---|---|---|---|
| $\mathbf{P}_m$ | Recognition of prior learning | $\mathbf{C}_m$ | Voluntary cancellations |
| $\mathbf{A}_m$ | Alternative course stream | $\mathbf{L}_m$ | Leave of absence |
| $\mathbf{W}_m$ | Voluntary withdrawals | | |

*Figure 7.1: Classification of Attrition of Participants in Revised Strata*

It is clear from Figure 7.1 that the bulk of participant attrition in both the treatment and control groups is due to two of the five categories. These two categories are the voluntary cancellation of registration for the introductory programming course ($C_{treatment}$ and $C_{control}$) and the voluntary changing of registration to the alternative slower paced introductory programming course offered by the Department of CS/IS at UPE ($A_{treatment}$ and $A_{control}$). It is also clear from Figure 7.1 that the bulk of participant attrition is from the medium risk stratum.

### *7.2.2 Demographic Profile*

All of the 118 participants identified to take part in the investigation in either of the treatment or control groups consented to take part in the empirical study. Analysis of the demographic material administered (*M1* in Table 6.1) resulted in the biographical profile of the sample participants per experimental group as shown in Table 7.4.

| | | **Treatment Group (*n* = 59)** | **Control Group (*n* = 59)** | **$\chi^2$-test statistic** | **p-value** |
|---|---|---|---|---|---|
| **Gender** | **Male** | 56% (*n* = 33) | 62% (*n* = 37) | 0.56 | 0.454 |
| | **Female** | 44% (*n* = 26) | 38% (*n* = 22) | | |
| **Home Language** | **English** | 44% (*n* = 26) | 56% (*n* = 33) | 1.66 | 0.198 |
| | **Afrikaans** | 25% (*n* = 15) | 20% (*n* = 12) | 0.43 | 0.511 |
| | **isiXhosa** | 22% (*n* = 13) | 16% (*n* = 10) | 0.49 | 0.486 |
| | **Other** | 9% (*n* = 5) | 8% (*n* = 4) | 0.12 | 0.729 |
| **Contact with other technology** | **VCR** | 57% (*n* = 34) | 42% (*n* = 25) | 2.75 | 0.098 |
| | **ATM** | 61% (*n* = 36) | 51% (*n* = 30) | 1.24 | 0.266 |
| | **Tape recorder** | 56% (*n* = 33) | 51% (*n* = 30) | 0.31 | 0.580 |
| | **CD player** | 85% (*n* = 50) | 92% (*n* = 54) | 1.30 | 0.255 |
| | **TV Games** | 24% (*n* = 14) | 27% (*n* = 16) | 0.18 | 0.672 |
| | **Cellular phone** | 83% (*n* = 49) | 94% (*n* = 55) | 2.92 | 0.088 |
| | **DSTV** | 62% (*n* = 37) | 42% (*n* = 25) | 4.89 | 0.027 |
| **Computer skills** | **General use** | 95% (*n* = 56) | 85% (*n* = 50) | 3.34 | 0.068 |
| | **Typing** | 90% (*n* = 53) | 81% (*n* = 48) | 1.72 | 0.190 |
| | **Use of mouse** | 97% (*n* = 57) | 100% (*n* = 59) | 2.03 | 0.154 |
| | **Windows objects** | 93% (*n* = 55) | 95% (*n* = 56) | 0.15 | 0.697 |
| **Prior computer experience** | | 70% (*n* = 41) | 71% (*n* = 42) | 0.04 | 0.840 |

*Table 7.4: Demographic Profile of Participants and Results of Tests for Equality of Proportions (p < 0.01)*

Research has indicated that the demographic variables listed in Table 7.4 are indicative of success in an introductory programming course[38]. In order to establish whether these variables might significantly obscure the motivational and academic performance results observed in the current investigation, an evaluation of the demographic profile of each experimental group is required.

[38] (Sauter 1986; Evans *et al.* 1989; Howell 1993; Haliburton 1998; Newman *et al.* 1999; Hagan *et al.* 2000; Morrison *et al.* 2001; Wilson *et al.* 2001; Boyle *et al.* 2002; Rountree *et al.* 2002; Rowell *et al.* 2003; Streicher 2003)

Application of the $\chi^2$-test for the homogeneity of proportions at the 99% percentile ($p < 0.01$) indicates that the number of participants in each of the treatment and control groups is similar for each of the demographic items appearing in Table 7.4. The three most prominent language groups are those of English, Afrikaans and isiXhosa. These three are also nationally recognised as the most prominent language groups in the geographical region in which UPE is physically located.

Evidence of participant contact with other technology indicates that popular technology amongst the participants in both experimental groups is that of CD players and cellular phones. The majority of both the treatment (70%) and control (71%) group participants indicated that they, at the time of the commencement of the investigative study, worked on a computer at least once a week (prior computer experience).

Furthermore, at least 81% of all participants in both experimental groups assessed their individual ability of working with a variety of computer skills as being reasonable to excellent. The skills surveyed were general use of a computer, typing skills, the use of a mouse as well as the use of standard Windows objects such as buttons, check boxes and radio buttons. The detailed analysis of this demographic aspect appears together with the detailed analysis of the other aspects of the demographic questionnaire in Appendix I.

A negligible amount of the entire complement of treatment and control group participants (Table 7.4) revealed an attitude of experiencing difficulty with using a computer (2%; $n = 1$ in both cases). Computer facilities were available at least once a week for use by the majority of the treatment (81%) and control (86%) group participants. Both the treatment and control group participants indicated a preference for using email and internet facilities above software packages like word processing and spreadsheet packages. None of the treatment group indicated a significant preference for composing program solutions, whereas 4% of the control group indicated that they created program solutions on a regular basis. These variations in proportions are not evaluated as being significant ($p < 0.01$).

The statistical evaluation techniques applied to the identified demographic variables confirm that there are no significant differences (p < 0.01) that can be attributed to any of these demographic variables.

### *7.2.3 Academic Profile*

Participants from both experimental groups are distributed between two distinct lecture learning activities, with each lecture learning activity being facilitated by a distinct course instructor. The proportion of each experimental group's participants under the facilitation of the distinct instructors is given in Table 7.5. Application of the $\chi^2$-test for the homogeneity of proportions at the 99% percentile (p < 0.01) indicates that the number of participants in each of the treatment and control groups is similar for each of the distinct instructors.

| | | **Treatment Group ($n$ = 59)** | **Control Group ($n$ = 59)** | **$\chi^2$-test statistic** | **p-value** |
|---|---|---|---|---|---|
| **Course Facilitation** | **Instructor1** | 68% ($n$ = 40) | 71% ($n$ = 42) | 0.16 | 0.689 |
| | **Instructor2** | 32% ($n$ = 19) | 29% ($n$ = 17) | | |

*Table 7.5: Academic Profile of Participants and Result of Test for Equality of Proportions (p < 0.01)*

| | **Treatment Group** | | | **Control Group** | | |
|---|---|---|---|---|---|---|
| | **Instructor1** | **Instructor2** | **p-value** | **Instructor1** | **Instructor2** | **p-value** |
| **N** | 40 | 19 | | 42 | 17 | |
| **Mean mark** | 58% | 61% | 0.492 | 57% | 49% | 0.159 |
| **Minimum mark** | 31% | 29% | | 26% | 23% | |
| **Maximum mark** | 87% | 89% | | 89% | 78% | |
| **Standard deviation** | 13% | 18% | | 20% | 16% | |

*Table 7.6: Descriptive Statistics categorised by Instructor and Experimental group and Results of Independent Sample T-test*

Descriptive statistics for the final marks obtained by the participants in each of the experimental groups and categorised for each distinct course instructor appear in Table 7.6. Independent sample t-tests (Table 7.6) verify that there are no significant differences (p < 0.01) in the final marks obtained that could be attributed to instructor influence.

## 7.3 Quantitative Analysis

Due to this elimination of bias due to demographic (Table 7.4) and course instructor influence (Tables 7.5 and 7.6), the participants of each experimental group (control and treatment), regardless of demographic profile and allocated course instructor, is analysed as a single group in the investigative study. This section presents the comparative quantitative analysis of performance achievement measures observed for all of the tasks in the quantitative assessment materials (Table 6.1) administered to each participant in the treatment and control groups.

A total of 28 independent variables are selected from the quantitative analysis materials administered to the participants and identified as being relevant for quantitative analysis in the current investigation (Section 6.2.2). Table 7.7 codes each and associates with each of these variables a category of analysis in anticipation of the deliberation to follow in Chapter 8. The timing in relation to the treatment period of each performance achievement measure is also provided in Table 7.7. The assessment materials from which the variables are derived appear in Appendix D.

Table 7.7 classifies each of the independent variables into *composition* and/or *comprehension* analysis categories. The *composition* category refers to tasks that assess a participant's ability to compose novel program solutions in a given programming notation. The *comprehension* category refers to tasks that assess a participant's ability to comprehend program solutions in a given programming notation. These two analysis categories are used in the interpretation (Chapter 8) of the results presented in this chapter. Independent variables that do not fall into any of these analysis categories are those that are the weighted averages of each of the assessment materials (suffixed with **Tot**) as well as the **Class** and **Final** marks. The **Class** and **Final** mark variables also do not have a timing associated with them due to the fact that they are weighted averages computed in terms of the other independent variables (Section 6.2.1).

| Variable (Appendix D) | | Material (Section 6.2.2) | Analysis Category | Timing (Table 6.2) |
|---|---|---|---|---|
| Th1Q1 | Question 1 | Problem-solving pen-and-paper assessment (***M6***) | Composition | 2 weeks into treatment |
| Th1Q2 | Question 2 | | Comprehension | |
| Th1Q3 | Question 3 | | Comprehension and Composition | |
| Th1Tot | Weighted average of Th1Q1, Th1Q2, Th1Q3 | | Problem-solving total | |
| Pr1Q1 | Question 1 | Introductory level practical assessment (***M7***) | Composition using B# programming notation | 4 weeks into treatment |
| Pr1Q2 | Question 2 | | Composition using textual programming notation | |
| Pr1Tot | Weighted average of Pr1Q1, Pr1Q2 | | Introductory practical assessment total | |
| Th2MC | Multiple choice questions | Syntax-based pen-and-paper assessment (***M6***) | Comprehension | 7 weeks into treatment |
| Th2Q1 | Question 1 | | Composition | |
| Th2Q2 | Question 2 | | Composition | |
| Th2Cmp | Weighted average of Th2Q1, Th2Q2 | | Pen-and-paper composition total | |
| Th2Tot | Weighted average of Th2MC, Th2Q1, Th2Q2 | | Pen-and-paper total | |
| Pr2Q1 | Question 1 | Intermediary level practical assessment (***M7***) | Comprehension using textual programming notation syntax | 1 week after completion of treatment |
| Pr2Q2 | Question 2 | | Composition using B# programming notation | |
| Pr2Q3 | Question 3 | | Composition using individual choice of programming notation | |
| Pr2Q4 | Question 4 | | Composition using textual programming notation | |
| Pr2Tot | Weighted average of Pr2Q1, Pr2Q2, Pr2Q3, Pr2Q4 | | Intermediary practical total | |
| Class | | Class mark (Section 6.2.1) | | |
| ExMC | Multiple choice questions | Final pen-and-paper examination (***M9***) | Comprehension | > 3 weeks after completion of treatment |
| ExQ1 | Question 1 | | Composition | |
| ExQ2 | Question 2 | | Composition | |
| ExQ3 | Question 3 | | Comprehension | |
| ExQ4 | Question 4 | | Composition | |
| ExQ5 | Question 5 | | Composition | |
| ExCmpr | Weighted average of ExMC, ExQ3 | | Examination comprehension total | |
| ExCmp | Weighted average of ExQ1, ExQ2, ExQ4, ExQ5 | | Examination composition total | |
| ExTot | Weighted average of ExMC, ExQ1, ExQ2, ExQ3, ExQ4, ExQ5 | | Examination total | |
| Final | | Final mark (Section 6.2.1) | | |

*Table 7.7: Coding of Independent Variables*

Descriptive statistics for the performance achievement variables measured for the full complement of each of the experimental groups appear in Table 7.8. For the purposes of completeness, Table 7.8 also includes the descriptive statistics for the expected final mark as predicted by the placement model currently implemented at UPE (Section 4.3).

| | **Mean** | | **Minimum** | | **Maximum** | | **Standard Deviation** | |
|---|---|---|---|---|---|---|---|---|
| **Variable ($n_{treatment} = n_{control} = 59$)** | **Treat-ment** | **Control** | **Treat-ment** | **Control** | **Treat-ment** | **Control** | **Treat-ment** | **Control** |
| Th1Q1 | 48% | 47% | 5% | 10% | 95% | 100% | 19% | 20% |
| Th1Q2 | 81% | 79% | 40% | 40% | 100% | 100% | 13% | 15% |
| Th1Q3 | 73% | 74% | 25% | 45% | 100% | 100% | 19% | 17% |
| Th1Tot | 65% | 64% | 42% | 36% | 93% | 93% | 12% | 13% |
| Pr1Q1 | 50% | 54% | 0% | 0% | 100% | 100% | 35% | 38% |
| Pr1Q2 | 48% | 41% | 0% | 0% | 100% | 100% | 35% | 35% |
| Pr1Tot | 48% | 46% | 0% | 0% | 100% | 100% | 31% | 32% |
| Th2MC | 61% | 60% | 25% | 25% | 83% | 88% | 15% | 16% |
| Th2Q1 | 55% | 49% | 14% | 0% | 93% | 93% | 21% | 25% |
| Th2Q2 | 25% | 30% | 0% | 0% | 100% | 92% | 29% | 28% |
| Th2Cmp | 41% | 40% | 8% | 0% | 93% | 91% | 19% | 25% |
| Th2Tot | 50% | 50% | 20% | 16% | 84% | 83% | 15% | 18% |
| Pr2Q1 | 50% | 53% | 0% | 0% | 100% | 100% | 25% | 28% |
| Pr2Q2 | 64% | 59% | 0% | 0% | 100% | 100% | 34% | 36% |
| Pr2Q3 | 60% | 49% | 4% | 0% | 100% | 100% | 26% | 33% |
| Pr2Q4 | 60% | 49% | 0% | 0% | 100% | 100% | 28% | 36% |
| Pr2Tot | 59% | 51% | 1% | 0% | 90% | 99% | 23% | 30% |
| Class | 58% | 55% | 22% | 19% | 86% | 87% | 15% | 20% |
| ExMC | 73% | 66% | 36% | 42% | 97% | 97% | 16% | 17% |
| ExQ1 | 25% | 25% | 0% | 0% | 83% | 100% | 18% | 21% |
| ExQ2 | 79% | 71% | 33% | 0% | 100% | 100% | 17% | 25% |
| ExQ3 | 37% | 32% | 0% | 0% | 100% | 100% | 33% | 32% |
| ExQ4 | 55% | 49% | 0% | 0% | 100% | 100% | 34% | 41% |
| ExQ5 | 50% | 51% | 0% | 0% | 100% | 100% | 27% | 29% |
| ExCmpr | 66% | 60% | 32% | 34% | 95% | 95% | 16% | 18% |
| ExCmp | 51% | 49% | 15% | 13% | 94% | 99% | 20% | 24% |
| ExTot | 60% | 55% | 29% | 26% | 95% | 91% | 16% | 19% |
| Final | 59% | 55% | 29% | 23% | 89% | 89% | 15% | 19% |
| Predicted | 59% | 59% | 41% | 42% | 72% | 72% | 8% | 8% |

*Table 7.8: Descriptive Statistics for Full Treatment and Control Groups*

| | **Treatment Group** | **Control Group** |
|---|---|---|
| **Full complement** | 0.46 ** (*n* = 60) | 0.57 ** (*n* = 88) |

** significant where p < 0.01

*Table 7.9: Correlation between Predicted and Observed Final Marks*

In Table 7.9, the correlations between the observed final marks and the final marks as predicted by the placement model currently implemented in the Department of CS/IS at UPE (Section 4.3) are shown for each experimental group. The correlations of both experimental groups for the full complement of participants are significant ($p < 0.01$). Consequently, the results confirm previous studies (Greyling *et al.* 2002; Greyling *et al.* 2003) that there exists a strong association between the predicted and observed final marks for participants in each of the experimental groups.

The test statistics identified in Chapter 6 as being applicable to the quantitative analysis of academic performance in the current investigation are a computed mean (observed average mark) and a computed proportion (observed throughput) (Section 6.3.3). The identified statistical techniques appropriate to the current study are:

- a pooled-variance two-tailed t-test for the testing of the difference in the two average marks computed for each of the treatment and control groups while controlling for predicted mark by means of a stratified sample based analysis (Section 7.3.1); and
- a $\chi^2$-test for homogeneity of proportions using a contingency table for testing the equality of the throughput computed for each of the treatment and control groups while controlling for predicted mark by means of a stratified sample based analysis (Section 7.3.2).

The results presented in this section have been computed using the STATISTICA (StatSoft Inc. 2001) data analysis tool.

### *7.3.1 Testing for Differences between Pairs of Means*

The pooled-variance two-tailed t-test is used to examine and test for significance at the 95% percentile ($\alpha = 0.05$) the following hypothesis (duplicated from Section 6.3.1) while controlling for predicted mark by means of a stratified sample based analysis:

$H_{0.1}$: *The average mark achieved in an introductory programming course is independent of programming notation and development environment.*

$H_{1.1}$: *The average mark achieved in an introductory programming course is dependent on programming notation and development environment.*

Since the sample size of each experimental group exceeds *n* = 30 (Larson 1974)[39], the assumption of normality holds for each of the treatment (*n* = 59) and control (*n* = 59) groups, and the pooled-variance two-tailed t-test is therefore applicable (Section 6.3.3). No further analysis to determine the existence of normality is thus required.

| | **Treatment Group** | | |
|---|---|---|---|
| | **Predicted Mean** | **Observed Mean** | **p-value** |
| **Full Complement**<br>**(*n* = 60)** | 59% | 60% | 0.763 |
| **High Risk Stratum**<br>**(*n* = 12)** | 47% | 51% | 0.267 |
| **Medium Risk Stratum**<br>**(*n* = 32)** | 59% | 57% | 0.544 |
| **Low Risk Stratum**<br>**(*n* = 16)** | 68% | 71% | 0.375 |

| | **Control Group** | | |
|---|---|---|---|
| | **Predicted Mean** | **Observed Mean** | **p-value** |
| **Full Complement**<br>**(*n* = 88)** | 59% | 55% | 0.031* |
| **High Risk Stratum**<br>**(*n* = 13)** | 47% | 38% | 0.008** |
| **Medium Risk Stratum**<br>**(*n* = 60)** | 59% | 54% | 0.024* |
| **Low Risk Stratum**<br>**(*n* = 15)** | 68% | 70% | 0.727 |

* significant where $p < 0.05$

** significant where $p < 0.01$

*Table 7.10: T-test Computed Values for Predicted and Observed Final Marks*

Table 7.10 presents for the full complement of participants as well as for each stratum for each of the experimental groups the computed t-test values to determine the existence of any significant variation between the observed and predicted average final marks. The predicted average final marks are identical for each stratum across

[39] Law of Large Numbers and Central Limit Theorem.

the two experimental groups. For the treatment group, no significant variation between the predicted and observed average final marks is evident ($p < 0.05$). This situation is, however, not replicated for the control group.

A statistically significant variation in predicted and observed average final marks is observed in the control group (Table 7.10). For the high risk (predicted 47%; observed 38%) and medium risk (predicted 59%; observed 54%) strata, as well as the full complement (predicted 59%; observed 55%) of control group participants, the observed average final mark is significantly less ($p < 0.05$) than the average final mark predicted by the placement model.

Each of the following subsections respectively examine and elaborate on the results of the pooled-variance two-tailed t-test on the observed differences between the means for the treatment and control groups. The discussion focuses on the **full complement** as well as each of the **high**, **medium** and **low** risk strata of each of the treatment and control experimental groups.

Full Complement

Table 7.11 presents only the independent variables that exhibit significant differences ($p < 0.05$) for the full complement of participants for each of the experimental groups. The comprehensive results of the pooled-variance two-tailed test for all 28 independent variables for the full complement of the experimental groups appear in Appendix L.

| **Variable**<br>**($n_{treatment} = n_{control} = 59$)** | **Treatment Group Mean** | **Control Group Mean** | **p-value** |
|---|---|---|---|
| Pr2Q3 | 60% | 49% | 0.041* |
| ExMC | 73% | 66% | 0.028* |
| ExQ2 | 79% | 71% | 0.046* |
| ExCmpr | 66% | 60% | 0.044* |

* significant where $p < 0.05$

*Table 7.11: T-test Computed Values for Independent Variables: Full Complement of Participants*

The results of the pooled-variance two-tailed t-test on the full complement of the treatment and control samples verify that there are significant differences ($p < 0.05$) in 4 of the 28 variables (Appendix L). The first of these variables (**Pr2Q3**) forms part of the introductory level practical assessment. This question required the treatment group participants to implement the required program solution in a programming notation and development environment of their choosing, namely either B# or Delphi™ Enterprise. Only 4 of the 59 participants elected to implement the program solution using B#. The reasons for this are deliberated in Chapter 8 (Section 8.3) with respect to the supplementary qualitative analysis (Section 7.4).

The second variable (**ExMC**) forms part of the final pen-and-paper examination. This section of the examination assesses the level of comprehension of simple small PASCAL textual programming notation program solution extracts. The third variable (**ExQ2**), also part of the final pen-and-paper examination, required the participants to create a program solution using the PASCAL textual programming notation supported by Delphi™ Enterprise. The fourth variable (**ExCmpr**) is a combination of the performance achievement measures **ExMC** and **ExQ3** (Table 7.7). This variable, also part of the final pen-and-paper examination, assesses the level of comprehension of a more advanced PASCAL textual programming notation program solution extract.

Based on the computed t-test statistics in Table 7.11, it can be concluded that with respect to the full complement of participants in the treatment and control groups, hypothesis $H_{0.1}$ (Section 6.3.1) cannot be rejected except in the cases of the 4 variables noted above.

In the case of variable **Pr2Q3**, hypothesis $H_{0.1.2.4}$ (Figure 6.3) is rejected. At the 95% level of confidence, the average mark achieved for a program solution based on individual choice of programming notation and development environment is thus dependent on the programming notation and development environment forming part of the learning environment. It is apparent from the sample measurements that there exists a significant difference ($p < 0.05$) in observed means for this variable (treatment group 60%; control group 49%).

Based on the observed means, in the cases of the variables **ExMC** (treatment group 73%; control group 66%), **ExQ2** (treatment group 79%; control group 71%) and **ExCmpr** (treatment group 66%; control group 60%) hypothesis $H_{0.1.1}$ (Figure 6.3) is similarly rejected. It is therefore concluded at the 95% level of significance that the average mark achieved in an introductory programming course for these 3 variables is dependent on programming notation and development environment.

<u>High Risk Stratum</u>

Table 7.12 presents only the independent variables that exhibit significant differences ($p < 0.05$) for the high risk stratum of participants for each of the experimental groups. The comprehensive results of the pooled-variance two-tailed test for all 28 independent variables for the high risk strata of the experimental groups appear in Appendix L.

The results of the pooled-variance two-tailed t-test on the high risk stratum treatment and control samples verify that there are significant differences ($p < 0.05$) in 9 of the 28 variables (Appendix L). Of these 9 variables, 6 show stronger significance at the 99% level of significance ($p < 0.01$), indicated by ** in Table 7.11.

Table 7.12 also lists a brief description of each variable's aim within the scope of the administered assessment materials (Section 6.2.2) as well as specifies the hypotheses which are rejected as a result of the observed significance.

Based on the computed t-test statistics, it can be concluded that with respect to the high risk stratum sample of participants in the treatment and control groups, hypothesis $H_{0.1}$ cannot be rejected except in the cases of the 9 variables appearing in Table 7.12. It is therefore concluded that there exists a significant variation ($p < 0.05$) in the observed values for these 9 variables and that the average mark achieved in an introductory programming course for these variables is dependent on programming notation and development environment.

| **Variable ($n_{\text{treatment}} = n_{\text{control}} = 12$)** | **Aim** | **Treat-ment Group Mean** | **Control Group Mean** | **p-value** | **Hypothesis Rejected (Figure 6.3)** |
|---|---|---|---|---|---|
| Pr2Q1 | Assess comprehension of PASCAL syntax in Delphi™ Enterprise | 54% | 30% | 0.015* | $H_{0.1.2.3}$ |
| Pr2Q3 | Assess composition of program solution using programming notation and programming development environment of choice (either B# or Delphi™ Enterprise) | 60% | 20% | 0.001** | $H_{0.1.2.4}$ |
| Pr2Q4 | Assess composition of program solution using PASCAL syntax in Delphi™ Enterprise | 56% | 16% | 0.002** | $H_{0.1.2.2}$ |
| Pr2Tot | Average measure of performance achievement in practical assessment. | 57% | 22% | 0.001** | $H_{0.1.2}$ |
| Class | Average measure of performance achievement in respect of weighted averages of pen-and-paper and practical assessments (Section 6.2.1). | 52% | 36% | 0.005** | $H_{0.1.3}$ |
| ExQ2 | Assess composition of program solution using PASCAL programming notation | 71% | 53% | 0.034* | $H_{0.1.1}$ |
| ExQ3 | Assess comprehension of program solution in PASCAL programming notation | 29% | 8% | 0.006** | $H_{0.1.1}$ |
| ExQ4 | Assess composition of program solution using PASCAL programming notation | 44% | 13% | 0.006** | $H_{0.1.1}$ |
| Final | Average measure of performance achievement in respect of weighted averages of class mark and final exam mark (Section 6.2.1). | 51% | 39% | 0.017* | $H_{0.1.4}$ |

* significant where $p < 0.05$
** significant where $p < 0.01$

*Table 7.12: T-test Computed Values for Independent Variables: High Risk Stratum*

### Medium Risk Stratum

Table 7.13 presents only the independent variables that exhibit significant differences ($p < 0.05$) for the medium risk stratum of participants for each of the experimental groups. The comprehensive results of the pooled-variance two-tailed test for all 28 independent variables for the medium risk strata of the experimental groups appear in Appendix L.

| **Variable**<br>**($n_{treatment} = n_{control} = 32$)** | **Treatment Group Mean** | **Control Group Mean** | **p-value** |
|---|---|---|---|
| ExMC | 73% | 64% | 0.036* |

* significant where $p < 0.05$

*Table 7.13: T-test Computed Values for Independent Variables: Medium Risk Stratum*

The results of the pooled-variance two-tailed t-test on the medium risk stratum treatment and control samples verify that there is a significant difference ($p < 0.05$) in only 1 of the 28 variables. This variable (**ExMC**) forms part of the final pen-and-paper examination. This section of the examination assesses the level of comprehension of simple small PASCAL textual programming notation program solution extracts.

Based on the computed t-test statistic in Table 7.13, it can be concluded that with respect to the medium risk stratum sample of participants in the treatment and control groups, hypothesis $H_{0.1}$ cannot be rejected except in the case of this single identified variable.

In the case of the variable **ExMC**, hypothesis $H_{0.1.1}$ is rejected. It is therefore concluded at the 95% level of significance that the average mark achieved by medium risk stratum participants in an introductory programming course for this variable is dependent on programming notation and development environment.

<u>Low Risk Stratum</u>

Table 7.14 presents only the independent variable that exhibits a significant difference ($p < 0.05$) for the low risk stratum of participants for each of the experimental groups. The comprehensive results of the pooled-variance two-tailed test for all 28 independent variables for the low risk strata of the experimental groups appear in Appendix L.

| **Variable**<br>**($n_{treatment} = n_{control} = 15$)** | **Treatment Group Mean** | **Control Group Mean** | **p-value** |
|---|---|---|---|
| Pr2Q1 | 50% | 71% | 0.034* |

* significant where $p < 0.05$

*Table 7.14: T-test Computed Values for Independent Variables: Low Risk Stratum*

The results of the pooled-variance two-tailed t-test on the low risk stratum treatment and control samples verify that there is a significant difference ($p < 0.05$) in only 1 of the 28 variables. This variable (**Pr2Q1**) forms part of the intermediary level practical assessment. This question of the assessment determines the skill level with respect to the detection and correction of PASCAL programming notation syntax errors using Delphi™ Enterprise as the development environment.

Based on the computed t-test statistic in Table 7.14, it can be concluded that with respect to the low risk stratum sample of participants in the treatment and control groups, hypothesis $H_{0.1}$ cannot be rejected except in the case of this single identified variable.

In the case of the variable **Pr2Q1**, hypothesis $H_{0.1.2.3}$ is rejected. It is therefore concluded at the 95% level of significance that the average mark achieved by low risk stratum participants in an introductory programming course for this variable is dependent on programming notation and development environment.

This section examined hypothesis $H_{0.1}$ and its associated sub-hypotheses (Figure 6.3) and determined that the influence of B# is predominantly significant in the high risk stratum of participants in the current investigation with respect to average marks achieved. The following section examines and tests hypothesis $H_{0.2}$ in order to measure the variance in observed throughput for the treatment and control groups for each of the 28 independent variables identified as being relevant to the current investigation.

### *7.3.2 Testing for Homogeneity of Proportions*

The $\chi^2$-test for homogeneity of proportions uses a 2×2 cross-classification table resulting from a survey of the observed throughput of participants in the treatment and control groups to examine and test for significance at the 95% percentile ($\alpha = 0.05$) the following hypothesis (duplicated from Section 6.3.1):

$H_{0.2}$: *The observed throughput in an introductory programming course is independent of programming notation and development environment.*

$H_{1.2}$: *The observed throughput in an introductory programming course is dependent on programming notation and development environment.*

Table 7.15 compares the measured predicted and observed throughput for each of the experimental groups. Throughput is that proportion of participants who achieved an average final mark of at least 50%. By implication of the predicted mark ranges (all at least 50%), the expected throughput rate for each of the medium and low risk strata in each of the experimental groups is 100%.

| | **Treatment Group** | | | |
|---|---|---|---|---|
| | **Predicted Throughput** | **Observed Throughput** | **$\chi^2$-test statistic** | **p-value** |
| **Full Complement (*n* = 60)** | 88% | 75% | 3.56 | 0.059 |
| **High Risk Stratum (*n* = 12)** | 42% | 67% | 1.51 | 0.219 |
| **Medium Risk Stratum (*n* = 32)** | 100% | 66% | 13.28 | 0.001** |
| **Low Risk Stratum (*n* = 16)** | 100% | 100% | 0.00 | 1.000 |

| | **Control Group** | | | |
|---|---|---|---|---|
| | **Predicted Throughput** | **Observed Throughput** | **$\chi^2$-test statistic** | **p-value** |
| **Full Complement (*n* = 88)** | 91% | 68% | 21.14 | 0.000** |
| **High Risk Stratum (*n* = 13)** | 38% | 23% | 0.72 | 0.395 |
| **Medium Risk Stratum (*n* = 60)** | 100% | 62% | 28.45 | 0.000** |
| **Low Risk Stratum (*n* = 15)** | 100% | 93% | 1.03 | 0.309 |

** significant where $p < 0.01$

*Table 7.15: $\chi^2$-test Computed Values for Predicted and Observed Throughput*

The observed throughput for the full complement of each experimental group is less than that expected with the control group's observed throughput being significantly less ($p < 0.01$). In the medium risk stratum of both experimental groups, a greater proportion of the treatment group participants passed (66% versus 62%), even though both observed values were significantly below that expected ($p < 0.01$).

Each of the following sub-sections respectively examine and elaborate on the results of the $\chi^2$-test for homogeneity of proportions on the full complement as well as each of the high, medium and low risk strata of each of the treatment and control experimental groups.

Full Complement

Table 7.16 presents only the independent variables that exhibit significant differences in proportions ($p < 0.05$) for the full complement of participants for each of the experimental groups. The comprehensive results of the $\chi^2$-test for homogeneity of proportions for all 28 independent variables for the full complement of the experimental groups appear in Appendix L.

The results of the $\chi^2$-test for homogeneity of proportions on the full treatment and control samples verify that there are significant differences in proportions ($p < 0.05$) in 6 of the 28 independent variables (Appendix L). Of these 6 variables, the variables **ExQ2** and **ExCmpr** (indicated by **) show stronger significance at the 99% level of significance ($p < 0.01$).

Table 7.16 lists the independent variables that show evidence of significance in the full sample (treatment and control groups). A brief description of each variable's aim within the scope of the administered assessment materials (Section 6.2.2) is also listed in Table 7.16. Table 7.16 also specifies the hypotheses which are rejected as a result of the observed significance.

Based on the computed $\chi^2$-test statistics, it can be concluded that with respect to the full sample of participants in the treatment and control groups, hypothesis $H_{0.2}$ cannot be rejected except in the cases of the 6 variables appearing in Table 7.16. In the cases of these variables it is therefore concluded at the 95% level of significance that the throughput achieved by the full complement of participants in the experimental groups in an introductory programming course is dependent on programming notation and development environment.

| **Variable** | **Aim** | **Throughput Rate** | | **$\chi^2$-test statistic** | **p-value** | **Hypothesis Rejected (Table 6.3)** |
|---|---|---|---|---|---|---|
| | | **Treatment Group (*n* = 60)** | **Control Group (*n* = 88)** | | | |
| Pr2Q3 | Assess composition of program solution using programming notation and programming development environment of choice (either B# or Delphi™ Enterprise) | 68%<br>(*n* = 41) | 49%<br>(*n* = 43) | 5.510 | 0.019* | $H_{0.2.2.4}$ |
| Pr2Q4 | Assess composition of program solution using PASCAL syntax in Delphi™ Enterprise | 67%<br>(*n* = 40) | 50%<br>(*n* = 44) | 4.038 | 0.045* | $H_{0.2.2.2}$ |
| Pr2Tot | Average measure of performance achievement in practical assessment. | 72%<br>(*n* = 43) | 55%<br>(*n* = 48) | 4.420 | 0.036* | $H_{0.2.2}$ |
| ExMC | Assess comprehension of small PASCAL program solution extracts | 92%<br>(*n* = 55) | 80%<br>(*n* = 70) | 3.990 | 0.046* | $H_{0.2.1}$ |
| ExQ2 | Assess composition of program solution using PASCAL programming notation | 97%<br>(*n* = 58) | 72%<br>(*n* = 63) | 15.040 | 0.000** | $H_{0.2.1}$ |
| ExCmpr | Assess comprehension of PASCAL program solution extracts. the variables ExMC and ExQ3. | 83%<br>(*n* = 50) | 61%<br>(*n* = 54) | 8.240 | 0.004** | $H_{0.2.1}$ |

* significant where $p < 0.05$
** significant where $p < 0.01$

*Table 7.16: $\chi^2$-test Computed Values : Full Complement of Participants*

High Risk Stratum

Table 7.17 presents only the independent variables that exhibit significant differences in proportions ($p < 0.05$) for the high risk stratum of participants for each of the experimental groups. The comprehensive results of the $\chi^2$-test for homogeneity of proportions for all 28 independent variables for the high risk strata of the experimental groups appear in Appendix L.

The results of the $\chi^2$-test for homogeneity of proportions on the high risk stratum samples verify that there are significant differences ($p < 0.05$) in 7 of the 28 variables (Appendix L). Of these 7 variables, 5 variables (indicated by **) show stronger significance at the 99% level of significance ($p < 0.01$).

Table 7.17 lists the variables that show evidence of significance in the high risk stratum sample, together with a brief description of each variable's aim within the scope of the administered assessment materials (Section 6.2.2). Table 7.17 also specifies the hypotheses which are rejected as a result of the observed significance.

Based on the computed $\chi^2$-test statistics, it can be concluded that with respect to the full sample of participants in the treatment and control groups, hypothesis $H_{0.2}$ cannot be rejected except in the cases of the 7 variables appearing in Table 7.17. In the cases of these variables it is therefore concluded at the 95% level of significance that the throughput achieved by the high risk stratum of participants in the experimental groups in an introductory programming course is dependent on programming notation and development environment.

| Variable | Aim | Throughput Rate |  | $\chi^2$-test statistic | p-value | Hypothesis Rejected (Table 6.3) |
|---|---|---|---|---|---|---|
|  |  | Treatment Group (*n* = 12) | Control Group (*n* = 13) |  |  |  |
| Pr2Q3 | Assess composition of program solution using programming notation and programming development environment of choice (either B# or Delphi™ Enterprise) | 75%<br>(*n* = 9) | 15%<br>(*n* = 2) | 9.000 | 0.003** | $H_{0.2.2.4}$ |
| Pr2Q4 | Assess composition of program solution using PASCAL syntax in Delphi™ Enterprise | 67%<br>(*n* = 8) | 8%<br>(*n* = 1) | 9.420 | 0.002** | $H_{0.2.2.2}$ |
| Pr2Tot | Average measure of performance achievement in practical assessment. | 75%<br>(*n* = 9) | 15%<br>(*n* = 2) | 9.000 | 0.003** | $H_{0.2.2}$ |
| Class | Average measure of performance achievement in respect of weighted averages of pen-and-paper and practical assessments (Section 6.2.1). | 75%<br>(*n* = 9) | 8%<br>(*n* = 1) | 11.779 | 0.003** | $H_{0.2.3}$ |
| ExQ2 | Assess composition of program solution using PASCAL programming notation | 92%<br>(*n* = 11) | 38%<br>(*n* = 5) | 7.667 | 0.006** | $H_{0.2.1}$ |
| ExQ5 | Assess composition of program solution using PASCAL programming notation | 42%<br>(*n* = 5) | 8%<br>(*n* = 1) | 3.949 | 0.047* | $H_{0.2.1}$ |
| Final | Average measure of performance achievement in respect of weighted averages of class mark and final exam mark (Section 6.2.1). | 67%<br>(*n* = 8) | 23%<br>(*n* = 3) | 4.810 | 0.028* | $H_{0.2.4}$ |

* significant where $p < 0.05$
** significant where $p < 0.01$

*Table 7.17: $\chi^2$-test Computed Values : High Risk Stratum*

Medium Risk Stratum

Table 7.18 presents only the independent variable that exhibits a significant difference in proportion ($p < 0.05$) for the medium risk stratum of participants for each of the experimental groups. The comprehensive results of the $\chi^2$-test for homogeneity of proportions for all 28 independent variables for the medium risk strata of the experimental groups appear in Appendix L.

| Variable | Number of Passes | | $\chi^2$-test statistic | p-value |
|---|---|---|---|---|
| | Treatment Group ($n = 32$) | Control Group ($n = 60$) | | |
| ExQ2 | 97%<br>($n = 31$) | 75%<br>($n = 45$) | 6.951 | 0.008** |

** significant where $p < 0.01$

*Table 7.18: $\chi^2$-test Computed Values : Medium Risk Stratum*

The results of the $\chi^2$-test for homogeneity of proportions on the medium risk stratum samples verify that there are significant differences ($p < 0.05$) in only 1 of the 28 variables (Appendix L). This variable (indicated by **) shows stronger significance at the 99% level of significance ($p < 0.01$). The variable **ExQ2** forms part of the pen-and-paper final examination and assesses the composition of PASCAL programming notation program solution extracts.

Based on the computed $\chi^2$-test statistic, it can be concluded that with respect to the medium risk stratum sample of participants in the treatment and control groups, hypothesis $H_{0.2}$ cannot be rejected except in the case of the identified variable. In the case of this variable, hypothesis $H_{0.2.1}$ is rejected. It is therefore concluded at the 95% level of significance that the throughput for this variables is dependent on programming notation and development environment.

Low Risk Stratum

Table 7.19 presents only the independent variable that exhibits a significant difference in proportion ($p < 0.05$) for the low risk stratum of participants for each of the experimental groups. The comprehensive results of the $\chi^2$-test for homogeneity of proportions for all 28 independent variables for the low risk strata of the experimental groups appear in Appendix L.

| **Variable** | **Number of Passes** | | **$\chi^2$-test statistic** | **p-value** |
|---|---|---|---|---|
| | **Treatment Group (*n* = 15)** | **Control Group (*n* = 16)** | | |
| Th2Q2 | 31%<br>(*n* = 5) | 73%<br>(*n* = 11) | 5.490 | 0.019* |

* significant where $p < 0.05$

*Table 7.19: $\chi^2$-test Computed Values : Low Risk Stratum*

The results of the $\chi^2$-test for homogeneity of proportions on the low risk stratum samples verify that there is a significant difference ($p < 0.05$) in only 1 of the 28 variables (Appendix L). The variable (**Th2Q2**) forms part of the syntax based pen-and-paper assessment. The variable assesses the composition of PASCAL programming notation program solution extracts.

Based on the computed $\chi^2$-test statistics in Table 7.19, it can be concluded that with respect to the low risk stratum sample of participants in the treatment and control groups, hypothesis $H_{0.2}$ cannot be rejected except in the case of the identified variable. In the case of this variable, hypothesis $H_{0.2.1}$ is rejected. It is therefore concluded at the 95% level of significance that the throughput for this variable is dependent on programming notation and development environment.

Besides the results of quantitative statistical techniques and test statistics presented in this section, qualitative data analysis is also relevant to the current study. The results of the analysis technique applicable to the qualitative analysis approach are presented in the following section.

## 7.4 Qualitative Analysis

The qualitative analysis of data relevant to the current investigation that determines the impact of a visual iconic programming notation and associated development environment occurred concurrently with the qualitative data collection process. The data collection process consisted primarily of surveys and supplementary personal interviews between participants and the author.

The qualitative analysis occurs in three distinct and independent areas, namely to determine:

- treatment group participant attitude to weekly practical learning activities (Section 7.4.1);
- treatment and control participant attitude to the programming notation(s) and development environment(s) used (Section 7.4.2); and
- motivation for withdrawal from the investigation collected by means of the conducting of supplemental interviews with treatment group participants who voluntary withdrew from the study (Section 7.4.3).

The results of this section have been analysed using the technique of thematic analysis. EXCEL (Microsoft Corporation 2002) is the data analysis tool used to assist in the process.

### *7.4.1. Thematic Analysis of Weekly Practical Learning Activity*

Qualitative data was collected on a weekly basis by means of practical learning activity reflection surveys administered to participants in the treatment group only (Appendix J). This data collection process took place during the majority of the 9 weekly practical learning activities that formed part of the treatment period (Figure 6.2).

The primary purpose of each survey was to determine the treatment group participant attitude towards the programming notations and development environments being used during practical learning activities. The surveys were administered on a weekly basis in order to determine any trend or change in attitude as the treatment period progressed.

Each survey requested the participant to provide the name of the development environment used to implement the program solution for the practical learning activity of the previous week for which (s)he had free choice (Appendix C). The participant was then requested to provide reasons for making the particular choice of development environment.

Figure 7.2 illustrates a comparison of preferred development environment categorised by programming development environment for each practical learning activity survey conducted. Between 54 and 83 of the population of 96 potential participants in the treatment group contributed to each of the weekly surveys.

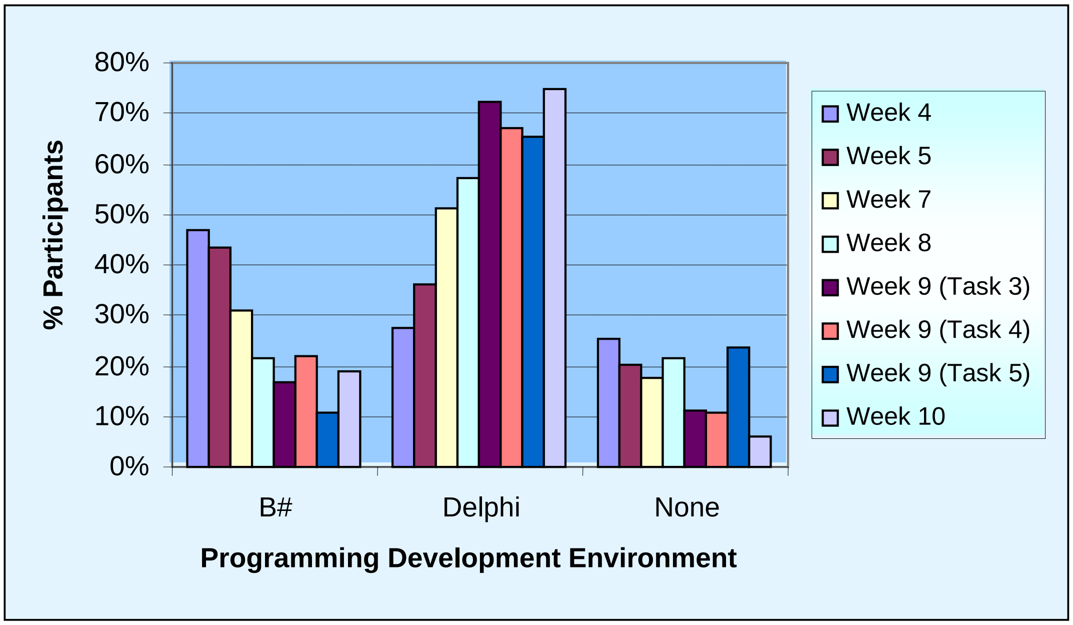


*Figure 7.2: Distribution of Preferred Development Environment*

Each week, a number of the participants indicated that they were unable to complete the surveyed practical task (label *None* in Figure 7.2). It is evident from Figure 7.2 that initially a greater number of participants preferred to implement program solutions using B# (weeks 4 and 5). The reasons for the change in preference to Delphi™ Enterprise are evident in the following thematic analysis of the reasons for choosing the conventional PASCAL textual programming notation and associated development environment over B#.

| | | **Theme: Usability** | | **Theme: Motivation** | |
|---|---|---|---|---|---|
| | **Preferred Instrument (Section 6.2.2)** | **Sub-theme:** ***Easy to use*** | **Sub-theme:** ***Enhances comprehension*** | **Sub-theme:** ***Extrinsic motivation*** | **Sub-theme:** ***Inaccessibility*** |
| **Week 4** | Delphi™ Enterprise (*n* = 23) | | | ✓<br>(35%; *n* = 8) | ✓<br>(30%; *n* = 7) |
| | B# (*n* = 39) | ✓<br>(85%; *n* = 33) | ✓<br>(26%; *n* = 10) | | |
| **Week 5** | Delphi™ Enterprise (*n* = 25) | | | ✓<br>(40%; *n* = 10) | ✓<br>(64%; *n* = 16) |
| | B# (*n* = 30) | ✓<br>(87%; *n* = 26) | ✓<br>(17%; *n* = 5) | | |
| **Week 7** | Delphi™ Enterprise (*n* = 38) | ✓<br>(47%; *n* = 18) | | | ✓<br>(55%; *n* = 21) |
| | B# (*n* = 23) | ✓<br>(74%; *n* = 17) | ✓<br>(17%; *n* = 4) | | |
| **Week 8** | Delphi™ Enterprise (*n* = 35) | ✓<br>(31%; *n* = 11) | | ✓<br>(31%; *n* = 11) | |
| | B# (*n* = 13) | ✓<br>(54%; *n* = 7) | ✓<br>(23%; *n* = 3) | | |
| **Week 9 (Task 3)** | Delphi™ Enterprise (*n* = 39) | ✓<br>(26%; *n* = 10) | | | ✓<br>(38%; *n* = 15) |
| | B# (*n* = 9) | ✓<br>(100%; *n* = 9) | | | |
| **Week 9 (Task 4)** | Delphi™ Enterprise (*n* = 37) | ✓<br>(27%; *n* = 10) | | | ✓<br>(41%; *n* = 15) |
| | B# (*n* = 12) | ✓<br>(92%; *n* = 11) | | | |
| **Week 9 (Task 5)** | Delphi™ Enterprise (*n* = 36) | ✓<br>(28%; *n* = 10) | | | ✓<br>(44%; *n* = 16) |
| | B# (*n* = 6) | ✓<br>(100%; *n* = 6) | ✓<br>(17%; *n* = 1) | | |
| **Week 10** | Delphi™ Enterprise (*n* = 51) | ✓<br>(41%; *n* = 21) | | ✓<br>(31%; *n* = 16) | |
| | B# (*n* = 13) | ✓<br>(92%; *n* = 12) | ✓<br>(15%; *n* = 2) | | |

*Table 7.20: Thematic Analysis : Practical Learning Activities*

A total of 26 categories of responses were derived from the data collected from 8 surveys of the preferred programming notation and development environment used during practical learning activities (Appendix J). These categories were refined to 7 sub-themes, one of which was designated as being a sub-theme to collect unexpected responses. The other 6 sub-themes were further refined to 2 themes, namely **motivation** and **usability**.

A summary of the thematic analysis of the data collected as a result of the surveys is presented in Table 7.20. This summary shows only the most prominent themes and sub-themes (denoted by a ✓) that emerged for each of preferred programming notation and development environments as the treatment period progressed. A detailed analysis for each theme, sub-theme and category appears in Appendix J.

It should be noted that any particular participant, by means of a single survey response, could contribute to the frequency count of multiple sub-themes and themes. Multiple responses to the same sub-theme by a single participant in any particular survey are, however, considered as a single distinct response to the relevant sub-theme in the frequency counts shown in Table 7.20. The timing of the surveys indicated by week identifiers in Table 7.20 corresponds to the description of the procedure used to collect the data for the current investigation (Section 6.2.3 and Table 6.2).

The most prominent theme emerging from participants who preferred to compose program solutions using Delphi™ Enterprise was that of motivation, being specifically **extrinsic motivation** (in 4 of the 8 surveys) and **inaccessibility** of the B# programming notation and development environment (in 6 of the 8 surveys). As from the third survey (week 7), responses included the fact that creating program solutions in Delphi™ Enterprise was easier (in 6 of the 8 surveys), providing support for the **usability** theme.

The *"extrinsic motivation"* sub-theme included responses related to the fact that the PASCAL textual programming notation supported by Delphi™ Enterprise was used in lecture learning activities and that this was the programming notation that would form the major portion of the final examination for the introductory programming course.

The sub-theme of *“inaccessibility”* was due to the fact that many participants preferred to complete their practical tasks off-campus and they were not in possession of a personal copy of B#, whereas they had a personal copy of Delphi™ Enterprise at their disposal.

As the treatment period progressed, a greater proportion of the participants elected to create program solutions using Delphi™ Enterprise (Figure 7.2 and column *Preferred Instrument* in Table 7.20). The theme of “*usability”* reasons cited for this included that the development environment was easy to use since it was not as rigid as B#. Delphi™ Enterprise grew in popularity towards the end of the treatment period at the stage when the creation of user defined functions and procedures were being practiced. The reason cited was that B# was cumbersome in this regard.

B# was consistently cited as being easy to use (in all 8 surveys) and described as a development environment that assisted in the comprehension of programming constructs (in 6 of the 8 surveys), highlighting a theme of **usability**.

In order to determine the attitude of all participants in both the treatment and control groups towards the presented programming notations and development environments, data was collected from a further survey and qualitatively analysed. The thematic analysis of this independent survey is presented in the next section.

### *7.4.2. Thematic Analysis of General Evaluation of Programming Notation and Development Environment*

A single survey to evaluate the attitude of participants to each of the programming notations and development environments was administered to participants in both experimental groups (Appendix K). This data collection process took place once during a lecture learning activity approximately 6 weeks into the treatment period (Figure 6.2). The pen-and-paper based survey was administered by the author in a fashion that encouraged the participants to respond as comprehensively as possible.

The survey consisted of 7 distinct questions (Table 7.21), with 3 of them being administered to participants of the treatment group only.

| Survey Question | Experimental group |
|---|---|
| ***Q1***: If you had the option to do the entire course, lectures and exams included, using only one of B# or Delphi, which would you choose? Give reasons for your answer. | Treatment and Control |
| ***Q2***: How did you experience solving practicals in B#? | Treatment |
| ***Q3***: How did you experience solving practicals in Delphi? | Treatment and Control |
| ***Q4***: What did you specifically like and dislike about solving practicals in B#? | Treatment |
| ***Q5***: What did you specifically like and dislike about solving practicals in Delphi? | Treatment and Control |
| ***Q6***: In what ways do you feel that using B# benefited and/or hindered you while doing your practicals? | Treatment |
| ***Q7***: In what ways do you feel that using Delphi benefited and/or hindered you while doing your practicals? | Treatment and Control |

*Table 7.21: Programming Notation and Development Environment Survey Questions*

A total of 77 treatment group and 116 control group participants participated in the survey. The purpose of each question in the survey and the thematic analysis of the responses recorded are presented in this section.

Question 1

The purpose of the first question of the survey is to determine whether the participants of an experimental group had a preference for any particular programming notation and associated development environment. Of the 77 treatment group participants, 28 (36%) indicated that B# was their preferred environment. The remaining treatment group participants preferred Delphi™ Enterprise (*n* = 49).

A total of 27 categories of responses were derived as being reasons for the preference of a particular programming notation and development environment (Appendix K). These categories were refined to 9 sub-themes, one of which was designated as being a sub-theme to collect unexpected responses. The other 8 sub-themes were further refined to 2 themes, namely **motivation** and **usability**.

A summary of the thematic analysis of the responses collected from treatment group participants (*n* = 77) for Question 1 (Table 7.21) appears in Table 7.22. This summary shows only the most prominent theme and sub-themes that emerge from the analysis of the responses collected for this question. A detailed analysis for these and the other theme, sub-themes and categories appears in Appendix K.

| | **Theme: Usability** | | |
|---|---|---|---|
| **Preferred Instrument (Section 6.2.2)** | **Sub-theme:**<br>***Easy to use*** | **Sub-theme:**<br>***Enhances comprehension*** | **Sub-theme:**<br>***Restricted functionality of B#*** |
| Delphi™ Enterprise<br>(*n* = 49) | ✓<br>(49%; *n* = 24) | | ✓<br>(43%; *n* = 21) |
| B#<br>(*n* = 28) | ✓<br>(79%; *n* = 22) | ✓<br>(64%; *n* = 18) | |

*Table 7.22: Thematic Analysis : Question 1*

The most prominent theme emerging from treatment group participants who selected Delphi™ Enterprise was that of usability, being specifically **ease of use** and **restricted functionality of B#**. A major reason for treatment group participants preferring Delphi™ Enterprise to B# is due to the restricted functionality of B#, especially with respect to the manner in which sub-routine implementation is supported (Table 7.22). The most prominent theme emerging from treatment group participants who selected B# was also that of usability, being specifically **ease of use** and that B# **enhances comprehension** of programming concepts.

Of the 116 control group participants surveyed, a small proportion (5%; *n* = 6) indicated the B# is their preferred programming notation and development environment. Furthermore, 37% (*n* = 43) of the responses indicated that the respondent has at least discussed B# with a third party, with some of the responses indicating definite independent experimentation with B# (Appendix K).

Question 2

The second question of the survey attempted to discover the manner in which treatment group participants experienced using B#. A total of 45 categories of responses were derived (Appendix K). These categories were refined to 8 sub-themes, one of which was designated as being a sub-theme to collect unexpected responses. The other 7 sub-themes were further refined to 2 themes, namely **motivation** and **usability**.

A summary of the thematic analysis of the responses collected from treatment group participants (*n* = 77) for Question 2 (Table 7.21) appears in Table 7.23. This summary shows only the most prominent themes and sub-themes that emerge from the analysis of the responses collected for this question. The summary is analysed in terms of the preferred programming notation and development environment as indicated by the data collected in Question 1. A detailed analysis for these themes, sub-themes and categories appears in Appendix K.

| | **Theme: Usability** | | **Theme: Motivation** |
|---|---|---|---|
| **Preferred Instrument (Section 6.2.2)** | **Sub-theme:** ***Easy to use*** | **Sub-theme:** ***Restricted functionality of B#*** | **Sub-theme:** ***Intrinsic motivation*** |
| Delphi™ Enterprise (*n* = 49) | ✓<br>(53%; *n* = 26) | ✓<br>(55%; *n* = 27) | |
| B# (*n* = 28) | ✓<br>(64%; *n* = 18) | | ✓<br>(43%; *n* = 12) |

*Table 7.23: Thematic Analysis : Question 2*

The most prominent theme emerging from treatment group participants who selected Delphi™ Enterprise was that of usability, being again specifically **ease of use** and **restricted functionality of B#** (Table 7.23). Two prominent themes emerge from treatment group participants who selected B#. These are usability (specifically **ease of use**) and motivation (**intrinsic**). These respondents enjoyed using B# and found that their programs worked more often resulting in a sense of achievement.

Question 3

The third question of the survey attempted to discover the manner in which all participants experienced using Delphi™ Enterprise. A total of 44 categories of responses were derived (Appendix K). These categories were refined to 7 sub-themes, one of which was designated as being a sub-theme to collect unexpected responses. The other 6 sub-themes were further refined to 2 themes, namely **motivation** and **usability**.

A summary of the thematic analysis of the responses collected from treatment group (*n* = 77) and control group (*n* = 116) participants for Question 3 (Table 7.21) appears in Table 7.24. This summary shows only the most prominent themes and sub-themes that emerge from the analysis of the responses collected for this question. The summary for treatment group participants is analysed in terms of the preferred programming notation and development environment as indicated by the data collected in Question 1. A detailed analysis for these themes, sub-themes and categories appears in Appendix K.

| | | **Theme: Usability** | | | **Theme: Motivation** |
|---|---|---|---|---|---|
| **Preferred Instrument (Section 6.2.2)** | | **Sub-theme:**<br>***Easy to use*** | **Sub-theme:**<br>*Difficult to use* | **Sub-theme:**<br>***Not visual*** | **Sub-theme:**<br>***Intrinsic motivation*** |
| **Treatment** | Delphi™ Enterprise (*n* = 49) | | ✓<br>(35%; *n* = 17) | | ✓<br>(45%; *n* = 22) |
| | B# (*n* = 28) | | ✓<br>(68%; *n* = 19) | | ✓<br>(29%; *n* = 8) |
| **Control** | Delphi™ Enterprise (*n* = 116) | ✓<br>(45%; *n* = 52) | | ✓<br>(34%; *n* = 39) | |

*Table 7.24: Thematic Analysis : Question 3*

The most prominent themes emerging from treatment group participants (Table 7.24), regardless of the preferred programming notation and development environment, were that of usability (Delphi™ Enterprise is **difficult to use**, but is **intrinsically motivating**). A single prominent theme emerges from control group participants. This theme is usability (specifically **ease of use** and the fact that Delphi™ Enterprise is textual and not **visual**).

Questions 4 and 6

Questions 4 and 6 determine specific favouritisms and aversions (Question 4), as well as benefits and hindrances (Question 6) to treatment group participants when using B#. A total of 27 categories of responses were derived for Question 4 and 18 categories of responses for Question 6 (Appendix K). These categories were refined to 11 sub-themes, one of which was designated as being a sub-theme to collect unexpected responses. The remaining 10 sub-themes were further refined to 2 themes, namely **motivation** and **usability**.

| | **Question 4** | | | | |
|---|---|---|---|---|---|
| | **Positive responses** | | | **Negative responses** | |
| | **Theme: Usability** | | | **Theme: Usability** | |
| **Preferred Instrument (Section 6.2.2)** | **Sub-theme:** ***Easy to use*** | **Sub-theme:** ***Enhances comprehension*** | **Sub-theme:** ***Reduced level of detail*** | **Sub-theme:** ***Restricted functionality of B#*** | **Sub-theme:** ***Difficult to implement subroutines*** |
| Delphi™ Enterprise (*n* = 49) | ✓<br>(24%; *n* = 12) | ✓<br>(47%; *n* = 23) | | ✓<br>(51%; *n* = 25) | ✓<br>(10%; *n* = 5) |
| B# (*n* = 28) | | ✓<br>(43%; *n* = 12) | ✓<br>(36%; *n* = 10) | ✓<br>(43%; *n* = 12) | ✓<br>(25%; *n* = 7) |

| | **Question 6** | | |
|---|---|---|---|
| | **Positive responses** | | **Negative responses** |
| | **Theme: Usability** | | **Theme: Motivation** |
| **Preferred Instrument (Section 6.2.2)** | **Sub-theme:** ***Easy to use*** | **Sub-theme:** ***Enhances comprehension*** | **Sub-theme:** ***Extrinsic motivation*** |
| Delphi™ Enterprise (*n* = 49) | ✓<br>(6%; *n* = 3) | ✓<br>(35%; *n* = 17) | ✓<br>(20%; *n* = 10) |
| B# (*n* = 28) | ✓<br>(7%; *n* = 2) | ✓<br>(50%; *n* = 14) | ✓<br>(21%; *n* = 6) |

*Table 7.25: Thematic Analysis : Questions 4 and 6*

A summary of the thematic analysis of the responses collected from treatment group participants (*n* = 77) for Questions 4 and 6 (Table 7.21) appears in Table 7.25. This summary shows only the most prominent themes and sub-themes, categorised as positive (favouritisms/benefits) and negative (aversions/hindrances) responses for each question, that emerge from the analysis of the responses collected for this question. The summary for treatment group participants is further analysed in terms of the preferred programming notation and development environment as indicated by the data collected in Question 1. A detailed analysis for these themes, sub-themes and categories appears in Appendix K.

Treatment group participants who preferred Delphi™ Enterprise liked the fact that B# was **easy to use** and assisted with the **comprehension** of programming concepts (Table 7.25). Treatment group participants who preferred B# liked the fact that B# **reduced the level of detail** required to be entered and that it assisted with the **comprehension** of programming concepts. All treatment group participants disliked B# for reasons of **restricted functionality** and **difficult implementation of subroutines**. Furthermore, all treatment group participants felt that they benefited from B# in that it was easy to use and assisted with the comprehension of programming concepts. The major hindrance to treatment group participants is identified as **extrinsic motivation**, being specifically that B# as a programming notation and development environment is not examinable and is also not industry related.

Questions 5 and 7

Questions 5 and 7 determine specific favouritisms and aversions (Question 5), as well as benefits and hindrances (Question 7) to all participants when using Delphi™ Enterprise. A total of 32 categories of responses were derived for Question 5 and 26 categories of responses for Question 7 (Appendix K). These categories were refined to 9 sub-themes, one of which was designated as being a sub-theme to collect unexpected responses. The remaining 8 sub-themes were further refined to 2 themes, namely **motivation** and **usability**.

A summary of the thematic analysis of the responses collected from treatment group participants (*n* = 77) for Questions 4 and 6 (Table 7.21) appears in Table 7.26. This summary shows only the most prominent themes and sub-themes, categorised as positive (favouritisms/benefits) and negative (aversions/hindrances) responses for each question, that emerge from the analysis of the responses collected for this question. The summary for treatment group participants is further analysed in terms of the preferred programming notation and development environment as indicated by the data collected in Question 1. A detailed analysis for these themes, sub-themes and categories appears in Appendix K.

| | | Question 5 | | | | |
|---|---|---|---|---|---|---|
| | | **Positive responses** | | | **Negative responses** | |
| | | **Theme: Usability** | | | **Theme: Usability** | |
| | **Preferred Instrument (Section 6.2.2)** | **Sub-theme:** ***Easy to use*** | **Sub-theme:** ***Enhances comprehension*** | **Sub-theme:** ***Full functionality*** | **Sub-theme:** ***Difficult to use*** | **Sub-theme:** ***Did not assist with comprehension*** |
| **Treatment** | Delphi™ Enterprise (*n* = 49) | ✓ (24%; *n* = 12) | ✓ (22%; *n* = 11) | | ✓ (37%; *n* = 18) | |
| | B# (*n* = 28) | | ✓ (29%; *n* = 8) | ✓ (21%; *n* = 6) | ✓ (54%; *n* = 15) | |
| **Control** | Delphi™ Enterprise (*n* = 116) | | ✓ (28%; *n* = 33) | | | ✓ (28%; *n* = 33) |

| | | Question 7 | | | | | |
|---|---|---|---|---|---|---|---|
| | | **Positive responses** | | | | **Negative responses** | |
| | | **Theme: Usability** | | | **Theme: Motivation** | **Theme: Usability** | **Theme: Motivation** |
| | **Preferred Instrument (Section 6.2.2)** | **Sub-theme:** ***Easy to use*** | **Sub-theme:** ***Enhances comprehension*** | **Sub-theme:** ***Full functionality*** | **Sub-theme:** ***Extrinsic motivation*** | **Sub-theme:** ***Difficult to use*** | **Sub-theme:** ***Extrinsic motivation*** |
| **Treatment** | Delphi™ Enterprise (*n* = 49) | ✓ (12%; *n* = 6) | ✓ (12%; *n* = 6) | | ✓ (20%; *n* = 10) | | |
| | B# (*n* = 28) | | | ✓ (14%; *n* = 4) | | | ✓ (21%; *n* = 6) |
| **Control** | Delphi™ Enterprise (*n* = 116) | | ✓ (16%; *n* = 19) | | | ✓ (6%; *n* = 7) | |

*Table 7.26: Thematic Analysis : Questions 5 and 7*

Treatment group participants who preferred Delphi™ Enterprise liked the fact that Delphi™ Enterprise was **easy to use** and assisted with the **comprehension** of programming concepts (Table 7.26). Treatment group participants who preferred B# liked the fact that Delphi™ Enterprise assisted with the **comprehension** of programming concepts and that it provided **full functionality** for programming concepts. Control group participants liked the fact that Delphi™ Enterprise assisted with the **comprehension** of programming concepts, yet at the same time responded that Delphi™ Enterprise did not assist in this regard. All treatment group participants disliked Delphi™ Enterprise because they found it **difficult to use**.

Treatment group participants who preferred B# (Table 7.26) felt that they benefited from the **full functionality** of Delphi™ Enterprise, but were hindered by the fact that it was the examinable programming notation and development environment (**extrinsic motivation**). Treatment group participants who preferred Delphi™ Enterprise (Table 7.26) felt that they benefited from the fact that Delphi™ Enterprise was **easy to use**, **assisted in the comprehension** of programming concepts and was examinable (**extrinsic motivation**). Control group participants (Table 7.26) likewise felt that Delphi™ Enterprise was **assisted in the comprehension** of programming concepts, but at the same time felt that it was **difficult to use**.

On a subject voluntarily withdrawing as a participant of the treatment group, a personal interview was conducted by the author to determine the motivations for withdrawal. The patterns discovered as a result of these interviews are presented in the following section.

### *7.4.3. Thematic Analysis of Personal Interviews*

The proportion of the 96 participants in the treatment group who requested to withdraw from the current investigation is computed to be approximately 11% ($n$ = 11). Voluntary withdrawal was experienced as being sporadic, but tended to escalate towards the end of the treatment period with the occurrence of impending final assessments.

Thematic analysis of the personal interviews conducted during the early stages of the treatment period determined that the participants who requested to withdraw from the current study considered themselves to be experienced in the creation of program solutions using a textual programming notation and associated development environment.

Typical responses elicited during the early interviews were:

> *"I did Delphi at school"*
> *"I have taught myself how to program using* .....(any of a number of textual programming notations)*"*

Some participants did respond during these early interviews that they might have found B# valuable if used during the early stages of the process of them learning to program.

Thematic analysis of the personal interviews conducted during the later stages of the treatment period determined that the participants who requested to withdraw from the current study were experiencing stress associated with workload not necessarily directly related to that of the introductory programming course.

Typical responses elicited during these interviews were:

> *"I don't need to answer any B# question in the tests and exam"*
> *"Working in two packages is a waste of time"*
> *"I won't be able to use B# when I get a job so I don't need to learn it now"*

Another theme that emerged was that Delphi™ Enterprise was considered to be a more stimulating development environment. A typical response that supported this theme was:

> *"I enjoy using Delphi more because it challenges me"*

In addition to the personal interviews, data for supplementary analysis relevant to the current investigation was collected by means of a survey to determine the experience of participants and tutors in terms of atmosphere in a particular practical learning activity. The results of the analysis of these surveys are presented in the following section.

## 7.5 Supplementary Analysis

The survey (Appendix J) to determine participant and tutor experience of the atmosphere in a practical learning activity was administered in week 8 (5th week of the treatment) of the introductory programming course (Figure 6.2). Participants in both the treatment and control groups were requested to respond, as well as tutors facilitating the practical learning activities for each of the experimental groups. Table 7.27 categorises the positive and negative response values per survey item in the survey administered to participants in both the treatment and control groups.

| Survey Item | Positive response values | Negative response values |
|---|---|---|
| SI1: I enjoyed this practical learning activity session | 4, 5 | 1, 2 |
| SI2: I feel that I have learnt something during this practical learning activity session | 4, 5 | 1, 2 |
| SI3: I feel that I have achieved something this practical learning activity session | 4, 5 | 1, 2 |

*Table 7.27:Participant Practical Learning Activity Atmosphere Survey*

Results of a $\chi^2$-test for homogeneity of proportions for the experimental groups are presented in Table 7.28. The results verify that there are no significant differences (p < 0.05) in the way that treatment group and control group participants perceive the atmosphere in a practical learning activity session.

| Survey Item | Proportion of Positive Responses | | $\chi^2$-test statistic | p-value |
|---|---|---|---|---|
| | Treatment Group ($n$ = 60) | Control Group ($n$ = 88) | | |
| SI1 | 33% | 40% | 1.060 | 0.304 |
| SI2 | 62% | 62% | 0.000 | 1.000 |
| SI3 | 57% | 50% | 0.980 | 0.321 |

*Table 7.28: $\chi^2$-test Computed Values : Participant Practical Learning Activity Atmosphere Survey*

Table 7.29 categorises the positive and negative response values per survey item in the survey administered to tutors of both the treatment and control groups. Results of a $\chi^2$-test for homogeneity of proportions for the tutors of the experimental groups are presented in Table 7.30. The results verify that there are significant differences (p <

0.05) in the way that treatment group and control group tutors perceive the atmosphere in a practical learning activity session.

| Survey Item | Positive response values | Negative response values |
|---|---|---|
| TI1: Most of my assistance seemed to be on the programming environment and not on programming problems | 1, 2 | 4, 5 |
| TI2: I felt constructive when I assisted the students | 4, 5 | 1, 2 |
| TI3: The general atmosphere in the laboratory was calm | 4, 5 | 1, 2 |
| TI4: I enjoyed assisting during this practical session | 4, 5 | 1, 2 |
| TI5: I feel that the students managed the practical and were productive | 4, 5 | 1, 2 |

*Table 7.29:Tutor Practical Learning Activity Atmosphere Survey*

| Survey Item | Proportion of Positive Responses | | $\chi^2$-test statistic | p-value |
|---|---|---|---|---|
| | Treatment Group | Control Group | | |
| TI1 | 67% | 45% | 9.820 | 0.002 ** |
| TI2 | 78% | 82% | 0.500 | 0.480 |
| TI3 | 67% | 64% | 0.200 | 0.655 |
| TI4 | 89% | 91% | 0.220 | 0.637 |
| TI5 | 44% | 27% | 6.310 | 0.012 * |

* significant where $p < 0.05$
** significant where $p < 0.01$

*Table 7.30: $\chi^2$-test Computed Values : Tutor Practical Learning Activity Atmosphere Survey*

Tutors in the control group experienced that much of their time while facilitating the practical learning activity was consumed in solving problems related to Delphi™ Enterprise in the program domain rather than in the problem domain. A significantly larger proportion of the treatment group tutors responded that they experienced that the students being facilitated managed the practical tasks set and that the students were productive. These students used B# concurrently with Delphi™ Enterprise during practical learning activities.

## 7.6 Conclusion

In acknowledgement of the risk associated with small sample sizes, the strata as defined in Chapter 6 (Definitions 6.1 and 6.2) are consolidated and modified to incorporate only three discrete logical strata, namely a high risk, medium risk and low risk strata (Table 7.3). High risk participants are those who have been predicted as being low-ability achievers (or at risk of being successful) in terms of academic performance in an introductory programming course at UPE.

All subsequent quantitative analysis was applied to the treatment and control groups as full complements as well as to each of the comparative identified logical strata within each experimental group. A total of 4 statistical tests are thus performed per statistical testing technique. Qualitative analysis was applied to the treatment and control groups as independent samples.

A total of 118 introductory programming students participated for the entire duration of the study associated with the quantitative analysis of data, 59 in the treatment group and 59 in the control group. For each experimental group, 12 were identified as being high risk (with a predicted final mark of 41% – 50%), 32 medium risk (with a predicted final mark of 51% – 65%) and 15 low risk (with a predicted final mark of > 65%). A similar demographic and academic profile existed amongst participants in each of the treatment and control groups (Tables 7.4, 7.5 and 7.6).

The elimination of bias in respect of course instructor and demographical profile between the treatment and control experimental groups (Tables 7.4, 7.5 and 7.6) serves to reinforce the foundation of academic performance achievement and motivational profile on which the current study focuses.

The computed correlation between the predicted final marks and the observed final marks for both the treatment and control groups (Table 7.9) confirms the existence of a strong association as previously determined in related work (Section 4.3). No significant variation from the predicted average final mark was observed for participants in the treatment group. It was however observed that participants in the control group performed significantly worse than expected in all strata, including the full complement of participants but excluding the low risk stratum (Table 7.10).

A significant variation between the predicted and observed throughput was evident in the treatment group for only the medium risk stratum (Table 7.15). Similar significance was observed for both the full complement and medium risk stratum in the control group. The treatment group participants recorded a throughput of 75%, achieving the recommended target.

| | | **Full Complement** | **High Risk** | **Medium Risk** | **Low Risk** | **Distinct variables** |
|---|---|---|---|---|---|---|
| **t-test** | Treatment group | 14% ($n = 4$) | 32% ($n = 9$) | 4% ($n = 1$) | 0% ($n = 0$) | 39% ($n = 11$) |
| | Control group | 0% ($n = 0$) | 0% ($n = 0$) | 0% ($n = 0$) | 4% ($n = 1$) | 4% ($n = 1$) |
| **$\chi^2$-test** | Treatment group | 21% ($n = 6$) | 25% ($n = 7$) | 4% ($n = 1$) | 0% ($n = 0$) | 36% ($n = 10$) |
| | Control group | 0% ($n = 0$) | 0% ($n = 0$) | 0% ($n = 0$) | 4% ($n = 1$) | 4% ($n = 1$) |
| **Distinct variables** | Treatment group | 21% ($n = 6$) | 36% ($n = 10$) | 7% ($n = 2$) | 0% ($n = 0$) | 46% ($n = 13$) |
| | Control group | 0% ($n = 0$) | 0% ($n = 0$) | 0% ($n = 0$) | 7% ($n = 2$) | 7% ($n = 2$) |

*Table 7.31: Frequency of significant variables per strata and experimental group*

A total of 28 independent variables were identified as being relevant to the quantitative data analysis process of the current investigation (Table 7.7). Each variable was categorised according to an in-depth level of learning characteristic (Chapter 2), namely comprehension and/or composition of program solutions. Statistically significant variations in measurements in the favour of the treatment group were observed in a total of 13 of these variables (Table 7.31). The most prominent incidences of observed statistically significant variations were evident in the high risk stratum participants.

In the high-risk stratum, 36% ($n = 10$) of the independent variables were observed to exhibit statistically significant variations (Table 7.12 and Table 7.17). Incidences of observed statistically significant variations were also evident in the other strata as well as on the experimental groups as full complements, but to a lesser degree.

The observed statistically significant variations in the high risk strata were evident in both the pooled-variance two-tailed t-test for the testing of the differences between average marks obtained, as well as the $\chi^2$-test for testing the equality of observed throughput.

| **High Risk Stratum : Predicted Mark Range 41% - 50%** | | | | | | | |
|---|---|---|---|---|---|---|---|
| | | **t-test measures** | | | **$\chi^2$-test measures** | | |
| **Variable** | **Analysis Category** | **B# Mean ($n$ = 59)** | **DE Mean ($n$ = 59)** | **p-value** | **B# Through-put ($n$ = 12)** | **DE Through-put ($n$ = 13)** | **p-value** |
| Pr2Q1 | Comprehension using textual programming notation syntax | 54% | 30% | 0.015 * | | | |
| Pr2Q3 | Composition using individual choice of programming notation | 60% | 20% | 0.001 ** | 75% ($n$ = 9) | 15% ($n$ = 2) | 0.003 ** |
| Pr2Q4 | Composition using textual programming notation | 56% | 16% | 0.002 ** | 67% ($n$ = 8) | 8% ($n$ = 1) | 0.002 ** |
| Pr2Tot | Intermediary practical total | 57% | 22% | 0.001 ** | 75% ($n$ = 9) | 15% ($n$ = 2) | 0.003 ** |
| Class | Class mark (Section 6.2.1) | 52% | 36% | 0.005 ** | 75% ($n$ = 9) | 8% ($n$ = 1) | 0.001 ** |
| ExQ2 | Composition | 71% | 53% | 0.034 * | 92% ($n$ = 11) | 38% ($n$ = 5) | 0.006 ** |
| ExQ3 | Comprehension | 29% | 8% | 0.006 ** | | | |
| ExQ4 | Composition | 44% | 13% | 0.006 ** | | | |
| ExQ5 | Composition | | | | 42% ($n$ = 5) | 8% ($n$ = 1) | 0.047 * |
| Final | Final mark (Section 6.2.1) | 51% | 39% | 0.017 * | 67% ($n$ = 8) | 23% ($n$ = 3) | 0.028 * |

* significant where $p < 0.05$
** significant where $p < 0.01$

***Key to abbreviation***
**DE** Delphi™ Enterprise

*Table 7.32: Significant Variables : High Risk Stratum*

As a result of the computed t-test statistics, it can be concluded that in the case of the identified independent variables for the high risk strata of participants (Table 7.32), hypothesis $H_{0.1}$ (Section 6.3.1 and Figure 6.3) is rejected. The implication is that for these variables, the average mark achieved by the participants is found to be dependent on programming notation and development environment. The average marks measured in these variables for high risk participants in the treatment group significantly exceeded the average marks measured for high risk participants in the control group ($p < 0.05$).

Results of the computed $\chi^2$-test statistics lead to the conclusion that in the case of the identified independent variables (Table 7.32), hypothesis $H_{0.2}$ (Section 6.3.1 and Figure 6.3) is rejected. The implication is that the observed throughput is found to be dependent on programming notation and development environment. The observed throughput in these variables for high risk participants in the treatment group significantly exceeded the observed throughput for high risk participants in the control group ($p < 0.05$).

In terms of the results summarised in Table 7.32, the hypothesis $H_0$ (Section 6.3.1 and Figure 6.3) is thus rejected for the variables **Pr2Q3**, **Pr2Q4**, **Pr2Tot**, **Class**, **ExQ2** and **Final**. For these variables, the empirical study has shown that academic performance for high risk students in an introductory programming course is dependent on programming notation and development environment. The academic performance measured in these variables for high risk participants in the treatment group significantly exceeded the academic performance measured for high risk participants in the control group ($p < 0.05$).

Qualitative analysis deduced that the most prominent themes emerging from participants who preferred to create program solutions using B# were that B# is easy to use and assists in the comprehension of programming constructs (Table 7.20). The themes emerging most prominently amongst participants who preferred to use Delphi™ Enterprise in the creation of program solutions related to the perceived inaccessibility of B# and extrinsic motivation. Participants often motivated their preference for Delphi™ Enterprise in terms of its support for the PASCAL textual programming notation that is examinable and is referred to regularly in the lecture learning activities of the introductory programming course at UPE. The same tendency was apparent in the thematic analysis of a survey to determine treatment and control group participant attitude to each of the programming notations and development environments (Tables 7.22 – 7.26).

Supplementary qualitative analysis of a survey administered to practical learning activity tutors concludes that much of the tutors' time during control group practical learning activity sessions is dedicated to answering questions related to the Delphi™ Enterprise development environment (Table 7.30). The tutors also felt that B# students managed the practical tasks and were productive. This opinion differed significantly from that expressed by tutors of the control group practical learning activities.

The observed statistically significant variations in measured performance in favour of B#, as well as the emerging themes resulting from thematic analysis of surveys administered, support the finding that the use of the B# iconic programming notation and development environment is beneficial to novice programmers, especially those who have been identified as being low-ability achievers in terms of academic performance.

# Chapter 8

# Interpretation of Results

## 8.1 Introduction

Previous studies in the use of flowcharts and iconic programming notations in an introductory programming course have demonstrated benefits to novice programmers in terms of increased accuracy in the composition of program solutions (Pandey *et al.* 1993; Calloni *et al.* 1997; Crews 2001; M$^{c}$Iver 2001; Carlisle *et al.* 2004). The volume of available literature on different categories of experimental programming notations and development environments[40], however, suggests that no single category of experimental programming notation has been widely adopted for use as technological support in the learning environment of an introductory programming course. The categories of experimental programming notations and development environments reviewed in Chapter 3 also do not comprehensively satisfy the framework of novice programmer requirements for such technological support as determined from the literature study documented in Chapter 2.

[40] (Bonar *et al.* 1990; Lyons *et al.* 1993; Calloni *et al.* 1994, 1995; Studer *et al.* 1995; Liffick *et al.* 1996; Calloni *et al.* 1997; Cockburn *et al.* 1997; Crews *et al.* 1998; Blackwell *et al.* 1999a; Good 1999; Cooper *et al.* 2000; Garner 2000; Blackwell 2001; Dagiano *et al.* 2001; Materson *et al.* 2001; Navarro-Prieto *et al.* 2001; Chamillard *et al.* 2002; De Raadt *et al.* 2002; Fergusson 2002; Quinn 2002; Burrell 2003)

Furthermore, documented evidence of scientific experimental verification in the use of experimental programming notations and development environments in an introductory programming course is lacking. Consequently, B#, a visual iconic programming notation incorporating a flowchart program solution representation for program solution composition, has been developed.

Evidence of B# support for the framework of novice programmer requirements for technological support in the learning environment of an introductory programming course is apparent for 6 of the 8 requirements (Chapter 5). The specific requirements met by the design and implementation of B# are sumarised in Table 8.1 (duplicated from Table 5.4). The methodology for determining B#'s support for the remaining 2 requirements (*R3* and *R8*) is presented in Chapter 6.

| Requirements for Novice Programmer Technological Support | Supported |
|---|---|
| ***R1***: Elimination of finer implementation details typically found at the superficial learning level of the program domain | ✓<br>(Section 8.2.1) |
| ***R2***: Increased level of program solution comprehension at the in-depth learning level of the program domain | ✓<br>(Section 8.2.2) |
| ***R3***: Increase in level of motivation when using the programming notation | **To be determined by the current study**<br>(Section 8.3.2) |
| ***R4***: Designed specifically for use by novice programmers | ✓<br>(Section 8.2.3) |
| ***R5***: Provision of visual techniques to aid comprehension process at the in-depth learning level of the program domain | ✓<br>(Section 8.2.4) |
| ***R6***: Support for reduced mapping between the problem and program domains | ✓<br>(Section 8.2.5) |
| ***R7***: Increased focus on problem-solving | ✓<br>(Section 8.2.6) |
| ***R8***: Increase in novice programmer performance achievement measured in terms of higher level of accuracy in program solutions in program domain | **To be determined by the current study**<br>(Section 8.3.3) |

*Table 8.1: Support for Novice Programmer Requirements by B#*

Chapter 7 presents a quantitative comparative analysis of measures and qualitative thematic analysis in order to make statistical decisions with regard to the hypotheses formulated in Chapter 6. These hypotheses relate to the determining of whether observed performance achievements and motivation for participants in the study are independent of programming notation and development environment.

The results reported on in Chapter 7 identify 13 of the 28 variables (Table 7.31) as showing statistically significant variations in measured performance ($p < 0.05$). The preliminary findings are that the observed statistically significant variations in measured performance in favour of B#, as well as the emerging themes resulting from thematic analysis of surveys conducted, support the finding that the use of the B# visual iconic programming notation and development environment is beneficial to a specific class of novice programmers (Table 7.32). This class of novice programmers are those who have been identified by means of a validated computerised selection battery as being low-ability achievers in terms of academic performance in an introductory programming course (Greyling 2000; Greyling *et al.* 2002; Greyling *et al.* 2003).

Comprehensive support by B# for all of the novice programmer requirements identified in Chapter 2 would be indicative of B# being classified as a successful technological learning environment in an introductory programming course. The focus of this chapter is consequently on the comprehension and evaluation of the design and implementation decisions made during the development of B# (Chapter 5) as well as the application of the investigative methodology (Chapters 6 and 7) in terms of the focus of the current investigation. The following objectives of this study are addressed by this chapter (Table 1.1):

- to argue the suitability of B# as a programming notation and development environment for novice programmers;
- to deliberate on the performance achievement level of novice programmers depending on the technological learning environment exposed to; and

- to determine the impact of a specific category of programming notation and development environment (iconic versus textual) on novice programmer motivation in an introductory programming course at tertiary level.

The chapter argues that B# supports all of the novice programmer requirements as determined in Chapter 2. Support for novice programmer requirements at the design and implementation level is argued from the findings presented in Chapter 5 (Section 8.2). The results presented in Chapter 7 form the basis for the argument in support of the novice programmer requirements with regard to the level of motivation and academic performance achievement (Section 8.3). The implications of the findings presented suggest significant theoretical (Section 8.4) and instructional benefits (Section 8.5) for novice programmers.

## 8.2 Evidence of Novice Programmer Requirements Support in B# Design and Implementation

The total cognitive load on a novice programmer in an introductory programming course can be minimised by addressing the extraneous cognitive load (Garner 2001). Extraneous cognitive load is reduced by a programming notation and development environment being sensitive to the manner in which novice programmers function in the program domain. In response to this challenge, B# was developed not to replace conventional textual programming notations, but to complement them in an integrated visual development environment in an attempt to enhance the learning experience in an introductory programming course.

The development of the B# visual iconic programming notation and development environment is influenced by the novice programmer requirements identified from an extensive review of available literature and presented in Chapter 2. This section argues support for each of the novice programmer requirements *R1*, *R2* and *R4 – R7* (Table 8.1) in terms of the discussion on the design and implementation of B# appearing in Chapter 5.

### *8.2.1 Requirement R1: Elimination of finer implementation details at superficial level of learning*

The reduction of the level of detail at the superficial level of learning (*R1* in Table 8.1) is supported by a programming notation supporting a small set of foundation constructs. B# provides support for the sequence, selection, iteration and variable programming constructs (Section 5.3.1), as recommended in available literature[41].

Furthermore, program solutions in B# are composed using a flowchart representation, which typically support minimal syntax (Crews 2001; McKinney 2003). The flowchart representation of B# comprises of connected icons, each one associated with a distinct programming construct (Section 5.3.4). The icon dialogue feature of B# provides for the implementation of error free programming constructs (Section 5.3.6). In this way, support is provided for a minimum number of unproductive errors (Cockburn *et al.* 1997; McIver 2000). Elimination of the finer implementation details is further enhanced by B# with a feature that associates multiple statements of a conventional textual programming notation with a single icon in the corresponding flowchart program solution representation.

### *8.2.2 Requirement R2: Increased level of program solution comprehension at the in-depth level of learning*

Technological support in the program domain that shows the program solution and execution thereof in an integrated fashion enhances the level of program solution comprehension at the in-depth level of learning (Wright *et al.* 2000). B# supports this requirement by means of its tracing facility (Section 5.3.5).

Further support for increased comprehension of program solutions is with the application of secondary notation to encourage the transfer of programming knowledge[42]. B# integrates multiple concurrent representations of a program solution, one in the form of an iconic flowchart and the other in the form of an equivalent textual programming notation, applying appropriate techniques of secondary notation in both forms of the program solution (Section 5.3.5).

[41] (Mayer 1981; Shu 1985; Brusilovsky *et al.* 1994; Calloni 1998; Blackwell *et al.* 2000; M$^c$Iver 2000; Howell 2003; Warren 2003)

[42] (Cant *et al.* 1995; Hendrix *et al.* 1998; Blackwell *et al.* 2000; Lister 2000; Applin 2001)

### *8.2.3 Requirement R4: Designed specifically for use by novice programmers*

Related studies in programming have determined that the control flow paradigm is the preferred programming paradigm of novice programmers[43]. This programming paradigm is naturally supported by the flowchart representation of a program solution in B# (Section 5.3.4).

Furthermore, B# is used in a learning environment that applies flowcharts in the solving of problems prior to the learning of programming constructs. Novice programmers are thus familiar with the flowcharting technique. This is in accordance with the findings of related studies in programming that have determined that prior experience impacts on the level of success of technological support in an introductory programming course (Chang *et al.* 1999; Wright *et al.* 2000). Other studies have determined that the use of unfamiliar terminology is one of the major impediments in an introductory programming course (Perkins *et al.* 1988; Pane *et al.* 2000). In response to these findings, B# uses the familiar flowcharting terminology in the composition of program solutions (Section 5.3.4).

Related studies in programming also recommend the technique of scaffold learning to be supported by technological support for novice programmers in an introductory programming course (De Koning *et al.* 2000; Wright *et al.* 2000). B# supports this recommendation with the implementation of menu picking as well as the typing in of data in icon dialogues (Section 5.3.6).

### *8.2.4 Requirement R5: Provision of visual techniques to aid comprehension process at the in-depth level of learning*

Students of introductory programming courses have been categorised as being visual learners (Felder 1993; Chamillard *et al.* 2002; Felder 2002). This categorisation is confirmed by the observation that visual images feature prominently in novice programmer solutions to programming problems[44]. In accordance with these findings,

[43] (Adelson 1984; Corritore *et al.* 1991; Good *et al.* 1999; Good 1999; Oberlander, Brna *et al.* 1999; Oberlander, Cox *et al.* 1999; Chattratichart *et al.* 2000, 2002)

[44] (Mayer 1981; Mayer *et al.* 1986; Shih *et al.* 1993; Dillon *et al.* 1994; Cockburn *et al.* 1997; Green 1997; Astrachan 1998; Ginat 2001; Pane *et al.* 2001)

the primary visual feature of B# is that of the representation of program solutions by means of a flowchart of iconic images (Section 5.3.4).

The purpose of each icon is to indicate the presence of a particular programming construct within a program solution (Dale 1998; Blackwell *et al.* 2000; Blackwell *et al.* 2001; Chang *et al.* undated). Special attention was paid during the design and implementation of B# to the selection of an appropriate and meaningful icon representation for each distinct programming construct (Section 5.3.3).

Other visual techniques recommended in technological support in the learning environment of an introductory programming course include:

- multiple representations of program solutions (Cockburn *et al.* 1997; Blackwell *et al.* 2000; Wright *et al.* 2000, 2002);
- the exposure to high-quality examples of program solutions (Fincher 1999; Kolling *et al.* 2001);
- implementation of secondary notation to enhance the purpose of program solutions (Kernighan *et al.* 1974; Soloway *et al.* 1982);
- association of textual programming notation extract with appropriate iconic image (Mayer 1981; Astrachan 1998; Howell 2003); and
- to restrict the choices available in the development environment (Pane *et al.* 2002).

B# supports the juxtaposibility feature by presenting alongside one another **multiple mutually consistent versions of the same program solution**, one in a visual iconic form and the other in the form of a conventional textual programming notation (Section 5.3.5). The latter form of program solution is always an example that illustrates **good textual programming notation practice** in terms of syntax, semantics, structure and style. The use of **secondary notation** is incorporated to enhance the readability of the textual programming notation program solution.

Code-highlighting is used in B# to **associate textual programming notation extracts with the equivalent icon** in the flowchart program solution (Section 5.3.5).

This encourages the "chunking" and recognition of the associated textual programming notation extract for the transfer of textual programming notation programming knowledge. Furthermore, the B# programming development environment makes use of context-sensitive views to **guide the novice programmer to make accurate and appropriate choices while functioning in the program domain** (Section 5.3.5). The appropriate design and implementation of B#'s visual capabilities therefore aid the comprehension of program solutions at the in-depth level of learning.

### *8.2.5 Requirement R6: Reduced mapping between the problem and program domains*

Reduction of the cognitive load resulting from the association of the mental model of a solution in the problem domain to one in the program domain is recommended by support for the visualisation of the actual program solution behaviour (Wright *et al.* 2002). B# supports this requirement with the multiple representations of a program solution (iconic and textual programming notations) as well as by means of the tracing facility (Section 5.3.5).

Furthermore, the programming paradigm supported by B#'s programming notation, namely that of control flow, matches the procedural programming paradigm required by the curriculum of the introductory programming course offered by UPE. In this way, the mental model of solutions to problems presented in the introductory programming course is closely mapped to the representation of program solutions in B#.

### *8.2.6 Requirement R7: Increased focus on problem-solving*

Technological support in the learning environment of an introductory programming course is required to focus on problem-solving (Calloni *et al.* 1994; Calloni 1998; M$^c$Iver 2000; Wright *et al.* 2002; Sanders *et al.* 2003a, b). B# supports this requirement with the elimination of mundane syntactical issues typically required by textual programming notations (Section 5.3.6).

Further support evident in B# is the animation of program solutions by means of the tracing feature (Section 5.3.5). In this way, the solution to a problem is illustrated in terms of its behaviour, thereby focussing on the problem-solving aspects of learning to program.

## 8.3 Evidence of Novice Programmer Requirements Support in Empirical Study

Expectations of improved student throughput in an introductory programming course necessitate the modification of the introductory programming course teaching model. The inclusion of B# as technological support in the learning environment of such a course is an attempt to improve not only the throughput, but also the average individual performance achievement of students in the course.

This section discusses the limitations of the experimental design presented in this thesis (Section 8.3.1). In light of the acknowledged limitations, support for the remaining novice programmer requirements (*R3* and *R8* in Table 8.1) relating to motivation and performance is evaluated and argued in terms of the observed results reported on in Chapter 7. This argument is presented in Sections 8.3.2 and 8.3.3.

### *8.3.1 Limitations of the Experimental Design*

A number of risks associated with the design of the current investigation are identified in Chapter 6 (Section 6.4). Strategies to address each of these risks are proposed. Despite the implementation of the proposed strategies during the conducting of the experiment, several limitations of the experimental design should be borne in mind when reading and interpreting the results presented in Chapter 7. These limitations are specifically:

- treatment group participants were required to master two programming notations and development environments due to ethical reasons;
- lack of formal teaching of B#;
- requirement that all participants write identical theoretical tests and final examination, and therefore B# would not be examinable. All participants

were made aware of this arrangement at commencement of the treatment period;

- training manual for B# in form of hard copy only;
- lack of scientific criteria for the measurement of equivalence between programming problems in practical repeated measures assessments;
- impact of stress in students when participating in time-dependent assessments;
- comparative quantitative measurement of only program solution accuracy; and
- focus of current investigative study on academic performance achievement and motivation only.

The primary limitation in the design of the experiment is that treatment group participants were expected to master an additional programming notation and development environment, B#, without the benefit of formal assistance. Participants in both experimental groups were expected to master the Delphi™ Enterprise development environment and the PASCAL textual programming notation supported by it.

The B# programming notation, unlike the PASCAL textual programming notation, was not required to be examined in any pen-and-paper based assessments. Since the primary motivation in any examinable course is the extrinsic reward of marks, it is acceptable to expect that participants in the treatment group may have been negatively motivated by this fact (VanLengen & Maddux 1990). Participants in the treatment group regularly commented about the difficulty of having to learn two development environments concurrently, especially with one of them being short term and non-examinable (Section 7.4).

Further, assistance for learning B# was provided by means of paper documentation since the version of B# used in the empirical investigation had no on-line help facility (Cilliers *et al.* 2004a). This situation caused treatment group participants to obviously spend more time becoming accustomed to B#. It is difficult to estimate how much this difficulty may have impacted the results, but it is certain that it would have disadvantaged the treatment group. Without this complication, the performance achievement measures of the treatment group may have been even better than that observed and reported on in Chapter 7.

One of the performance achievement measures taken into consideration was that of practical assessments. There currently exists a lack of rigid criteria for comparability between programming problems. Because the current experiment used a repeated measures design due to hardware limitations, different problem sets were used for some of the same performance achievement assessments (Section 6.4.4). While all of the problems were brief and simple program solutions which assessed similar programming constructs, no criterion of comparability exists for use to argue about their equivalence for experimental purposes. There is also no criterion against which the level of difficulty of the problems can be measured for the purposes of equivalence for experimental use. The current investigative study, however, makes use of moderators experienced in the practices of UPE's introductory programming course to certify the equivalence of problems (Section 6.4.4).

Each assessment measured the performance of participants at a particular instant in the introductory programming course. All of the assessments from which data was collected for quantitative analysis, had the restriction of time placed upon them. The pressure of completing an assessment within such a pre-specified period of time is a source of stress for many students (Chamillard *et al*. 2000). It is difficult to identify as well as estimate the extent and impact on the observed results of such stress.

In addition to the measuring of performance achievement in terms of accuracy as performed in the current study, the measurement of speed of completion is also relevant to studies in programming (Blackwell *et al.* 1999a). It should be noted that the current investigation provides no technique to accurately monitor the time taken to complete assessments and thus no conclusions can be made in this regard.

Related studies in programming have indicated that demographic issues such as home language, gender and prior computing experience impact on the level of success in an introductory programming course[45] (Section 4.2.1). Any observed measurements are not expected to be partial to the influence from the acknowledged demographic issues due to the observed similar demographic profiles in each of the treatment and control experimental groups (Table 7.4). The focus of the current study remains on motivational (Section 8.3.2) and academic performance (Section 8.3.3) issues only.

### *8.3.2 Requirement R3: Increase in level of motivation*

As the student population in an introductory programming course becomes more diverse, so do the individual student motivations for taking the course. Anecdotal evidence is that students participate in learning activities in a strategic manner (Jenkins 2001a). This section focuses on the level of motivation observed in the experimental group participants during the course of the current investigation. The outcome of the qualitative data collection and analysis conducted as a component of the current investigation (Section 7.4) is indicative of the level of motivation of novice programmers using the B# programming notation and associated development environment (*R3* in Table 8.1).

Treatment group participants were required on a weekly basis to solve a mandatory portion of the practical learning activity tasks using B#, and another mandatory portion using Delphi™ Enterprise (Section 6.2.2). The final portion of each week's practical learning activity tasks was required to be solved in the programming notation and development environment of each individual treatment group participant's choice. Qualitative analysis of the data collected as a result of weekly surveys of treatment group participants on individual choice of programming notation and associated development environment indicates that B# is perceived as (Table 7.20):

- being easy to use; and
- a tool that enhances the comprehension of programming concepts.

[45] (Sauter 1986; Howell 1993; Haliburton 1998; Newman *et al.* 1999; Hagan *et al.* 2000; McKenna 2000; Carter *et al.* 2001; Morrison *et al.* 2001; Boyle *et al.* 2002; Quaiser-Pohl *et al.* 2002; Rountree *et al.* 2002; Rowell *et al.* 2003)

These perceptions in favour of B# are, however, contradicted by the concurrent observation that Delphi™ Enterprise is ultimately the programming notation and development environment of choice amongst the treatment group participants. An early observation in the study was that B# was the programming notation and development environment of choice amongst treatment group participants (Section 7.4.1). This fact is indicative of the participants initially being more comfortable with using B#, but becoming increasingly more comfortable with using Delphi™ Enterprise (Figure 7.2) as the treatment period progressed.

The observation of the increase in level of comfort with using Delphi™ Enterprise, as well as the significant variation in performance achievement measures observed amongst the treatment group participants, confirms a finding in a related study that determined that a student's comfort level was the best indicator of success in an introductory programming course (Wilson *et al*. 2001). Treatment group participants indicated an ever increasing level of comfort in using Delphi™ Enterprise during the treatment period incorporating obligatory exposure of B# on a regular basis (Figure 7.2).

The observation of increasing level of comfort amongst treatment group participants in using Delphi™ Enterprise is confirmed by the fact that only 4 of the 59 treatment group participants (7%) selected B# as their preferred programming notation and development environment in a specific formal assessment task (Section 7.3.1). The remaining treatment group and entire complement of control group participants attempted to solve equivalent versions of the assessment task using Delphi™ Enterprise. Despite the fact that so few treatment group participants elected to solve the assessment task using B#, the observed measurements for the particular assessment task (**Pr2Q3** in Table 7.11) clearly show that a significantly superior level of performance achievement ($p < 0.05$) was evident amongst treatment group participants (mean 60%; $n$ = 59) when compared with the performance achievement measurements for control group participants (mean 49%; $n$ = 59).

The previously mentioned observation of increased performance achievement suggests evidence that the treatment administered positively affected the performance achievement measured. This substantial increase in performance achievement for treatment group participants using Delphi™ Enterprise to solve an assessment task can thus *only be attributed to continued mandatory exposure to B#*.

Primary reasons for treatment group participants selecting Delphi™ Enterprise over B# were analysed to be (Table 7.20):

- the perception that B# is not easily accessible by means of personal copies; and
- the fact that B# (as a programming notation) is not examinable whereas the PASCAL textual programming notation supported by Delphi™ Enterprise is.

Both of the above are themes of extrinsic motivation (Section 7.4.1), and have no direct bearing on a participant's experience with B#'s programming notation and associated development environment. In fact, interpretation of the themes, even though directly related to not selecting B# as the preferred programming notation and development environment, can also be interpreted as being non-negative motivation towards the use of B#.

The first of the above mentioned themes is in fact untrue since for the entire duration of the treatment period (9 weeks), each treatment group participant was afforded the opportunity of collecting a personal copy of B# from the author at a cost equivalent to the cost of the CD on which the software was located. Only 15 of the 59 participants (25%) in the treatment group made use of this opportunity. This theme could therefore also be interpreted as implying that should the B# software have required less effort to obtain, it might have been the programming notation and development environment of choice by treatment group participants.

The second of the above mentioned themes confirms findings by VanLengen *et al.* (1990) and Jenkins (2001a) who independently determined that the level of motivation of students is related to the extrinsic compensation of marks awarded. Another interpretation of this theme could therefore be that should B# have been examinable, it might have been the programming notation and development environment of choice by treatment group participants. The latter of the above mentioned themes was also regularly quoted as a reason for treatment group participants voluntarily withdrawing from the current investigation (Section 7.4.3).

Participants who had completed a prior programming course found less value in B#, confirming the findings of Sanders *et al.* (2003a; 2003b) in a related study. Even so, treatment group participants voluntarily withdrawing from the investigative study rated B# a useful programming notation and development environment for novice programmers (Section 7.4.3).

In a wider survey of participant attitude to the programming notations and development environments used, more treatment group participants preferred Delphi™ Enterprise to B# (Section 7.4.2). Despite this observation, these participants rated B# as being easy to use. The limiting factor was identified as being the restricted functionality of B#. All treatment group participants, regardless of their preferred programming notation and development environment, rated Delphi™ Enterprise as being difficult to use. The control group participants, however, rated Delphi™ Enterprise as being easy to use. This difference can possibly be attributed to the fact that the control group used only Delphi™ Enterprise in their learning activities.

The contents and frequency of regular learning activities of the treatment and control groups were identical. The B# visual iconic programming notation was never specifically discussed with participants in the treatment group. These participants were independently required to gain some level of comprehension about the workings of the B# development environment from the hard copy manuals provided on a weekly basis as well as the system itself without the benefit of an instructor’s explanation. Almost certainly, formal demonstrations of the B# programming notation and development environment would have facilitated a greater understanding of the system and impacted on the level of motivation observed.

Despite the observation that participant attitude towards B# was not entirely positive, no conclusive evidence in this regard can be presented. There does, however, exist quantitative evidence that suggests a positive attitude towards B#.

### *8.3.3 Requirement R8: Increase in performance achievement*

One of the novice programmer requirements for technological support in the learning environment of an introductory programming course is that such support should encourage an increase in novice programmer performance achievement (*R8* in Table 8.1). Related studies in programming that measure performance achievement using alternative technological support in an introductory programming course suggest evidence of a measured increase in performance achievement (Calloni *et al.* 1997; Ramalingam *et al.* 1997; Crews 2001; M$^{c}$Iver 2001; Carlisle *et al.* 2004). Level of performance achievement is commonly measured in terms of the accuracy of program solutions developed.

In one of these studies, the comprehension of program solutions in the procedural (control flow) and object-oriented (dataflow) programming paradigms is compared. In this study Ramalingam *et al.* (1997) compared performance achievement by measuring the number of errors observed in program solution comprehension. Novice programmers demonstrated superior comprehension in control flow program solutions (fewer errors observed).

A comparison in terms of frequency and type of errors observed in the imperative paradigm using a mini-language (GRAIL) and LOGO was conducted by M$^{c}$Iver (2001). A significant variation in observed rates was evident for errors related to each of the superficial and in-depth levels of learning ($p < 0.01$; $n = 13$ in each experimental group). M$^{c}$Iver concluded that participants using the GRAIL mini-language benefited in terms of observed error rate (mean of 13.62 versus 31.08 for syntax errors; mean of 9.54 versus 17.77 for logical errors).

In a related study, Crews (2001) compared the accuracy of program solutions developed by novice programmers using a visual flowcharting programming notation with that of program solutions developed using a conventional textual programming notation. The study concluded that the benefit in terms of accuracy in the flowchart program solution representation was more evident in advanced program solutions ($p < 0.05$). Crews' study also observed a significant variation in the means in the favour of the flowcharting programming notation with respect to the comprehension

($p < 0.05$) and composition ($p < 0.05$) of program solutions in terms of 90% ($n = 9$) of the identified programming concept criteria deemed relevant to the study. The criteria involved the novice programmer correctly identifying and using programming concepts relevant to the particular program solutions. Typical programming concept criteria used in Crews' study were the reading of input, writing of output and computation of an average. This evidence of increased academic performance achievement with a visual programming notation confirms the findings of an earlier related study.

The earlier study by Calloni *et al.* (1997) reported evidence of a higher assessment score (4% – 8%) for introductory programming students using only a visual iconic programming notation, BACCII$^{©}$ when compared with introductory programming students using only a conventional textual programming notation. BACCII$^{©}$ generates conventional textual programming notation program solutions from an iconic programming notation program solution.

Significant variation ($p < 0.05$) was observed in average scores achieved in the common written examination (mean of 78% BACCII$^{©}$ group; 69% conventional textual programming notation group) and final course grade (mean of 78% BACCII$^{©}$ group; 73% conventional textual programming notation group). Further analysis of the written examination by the researchers revealed that the students who had been using BACCII$^{©}$ showed a higher comprehension of conventional textual programming notation syntax despite not having composed program solutions in the textual programming notation directly.

Verification of the findings of these and other (Pandey *et al.* 1993; Carlisle *et al.* 2004) previous studies in programming is dependent upon the results presented in Chapter 7. The outcome of the quantitative data collection and analysis conducted as a component of the current investigation (Section 7.3) is therefore indicative of the performance achievement level of novice programmers using the B# programming notation and associated development environment as an instructional tool in an introductory programming course.

Assessments providing the data for the quantitative analysis generally contained problems for program solutions embedded within descriptive scenarios (Appendix D). A related study by McCracken *et al.* (2001) determined that the most difficult part of such assessments may be for the students to abstract the problem to be solved from the given description. No evidence of such a similar situation within the current investigative study was observed or specifically evaluated. Consequently, the comprehension and evaluation of the performance achievement level is done in terms of two measures, namely by means of observed average marks and observed throughput.

Participants in all strata of the treatment group performed as expected, recording final mark averages similar to those predicted (Table 7.10). The average predicted final marks were identical for all strata across the experimental groups. Despite this fact, participants in all strata of the control group excepting the low risk stratum, performed substantially worse than expected ($p < 0.05$). Furthermore, the expected number of participants was successful in the 2003 introductory programming course in all strata of the treatment group excepting the medium risk stratum (Table 7.15). A total of 75% of the treatment group and 68% of the control group participants were successful in the course. The treatment group achieved the recommended throughput target of 75% (Department of Education 2001; UPE 2002; Wesson 2002). The expected throughput was similar for each stratum across the experimental groups. A substantial number of participants in the full complement and medium risk stratum of the control group, however, were not successful in the course ($p < 0.05$).

The measurements in observed average marks between the experimental groups (Section 7.3.1) lead to interpretations similar to those presented in related studies (Calloni *et al.* 1997; Ramalingam *et al.* 1997; M$^{c}$Iver 2001). The lower standard deviation (Table 7.8) observed in the current investigation for the final mark of treatment group participants (15%; $n = 59$) when compared with that of control group participants (19%; $n = 59$) suggests evidence of more uniform learning occurring in the treatment group. Confirmation of this finding is evident in that the identical trend is reproduced in all variables identified as showing significant variations in recorded measurements (Tables 7.8, 7.11, 7.12, 7.13 and 7.14).

Significant advantage with respect to average marks obtained while using B# as technological support is observed in the full complement of treatment group participants with respect to program solution comprehension and composition in the PASCAL textual programming notation and associated Delphi™ Enterprise development environment (Tables 7.7 and 7.11). This finding is particularly evident amongst the group of students who have been identified as being of low-ability (high risk stratum) in the introductory programming course (Table 7.12).

Similarly, significant advantage with respect to throughput while using B# as technological support is observed in the full complement of treatment group participants with respect to program solution comprehension and composition in the PASCAL textual programming notation and associated Delphi™ Enterprise development environment (Tables 7.7 and 7.16). This finding is again particularly evident amongst the group of students who have been identified as being of low-ability in the introductory programming course (Table 7.17). Sections 7.3 and 7.4 thus provide evidence of successful in-depth learning as well as increased throughput being achieved by treatment group participants who have comprehended and composed program solutions in the form of flowchart representations.

The results of the current experiment using flowchart representation for program solutions supports Crew's (2001) hypothesis and confirms his findings that introductory programming students are more successful at processing algorithm-based problems represented as flowcharts rather than as a sequence of textual programming notation instructions. The current empirical investigation confirms that significant benefits exist in terms of program solution accuracy when comprehending and composing program solutions in the form of a flowchart.

Further, all measured significance in performance achievement in the favour of B# occurs after the conclusion of the treatment period (Tables 6.2, 7.7, 7.11, 7.12, 7.13, 7.16, 7.17 and 7.18). *This observation suggests evidence of program solution comprehension and composition in B# impacting programming concept knowledge retention and transfer to comparable problems in a conventional textual programming notation.*

The full complement of treatment group participants provided measurements of performance achievement and throughput in components of the final pen-and-paper examination (Appendix D) that were significantly superior ($p < 0.05$) to those measured for control group participants (Tables 7.11 and 7.16). This examination occurred at least 5 weeks after termination of the treatment period. This trend was also replicated in both the high and medium risk strata (Tables 7.12, 7.13, 7.17 and 7.18). In the high risk stratum (Tables 7.12 and 7.17), the intermediary level practical assessment, occurring 1 week after treatment termination, also provided evidence of significant differences in average marks achieved and throughput observed ($p < 0.05$).

## 8.4 Theoretical Implications

As a result of the evaluation of the findings of the investigative study, a number of theoretical implications of the current empirical investigation are evident. These theoretical implications are specifically:

- design, implementation and utilisation of a visual iconic programming notation and associated development environment (Section 8.4.2) that satisfies the framework of novice programmer requirements derived in Chapter 2; and
- enhancement of existing experimental methodology in the use of programming notations and development environments in the learning environment of an introductory programming course (Section 8.4.2).

This section elaborates on each of the above theoretical implications.

### *8.4.1 Design, Implementation and Utilisation of a Visual Iconic Programming Notation and Development Environment*

Technological support in the learning environment of an introductory programming course is crucial to the process of novice programmers learning to program. The literature review documented in Chapter 2 emphasises that novice programmers require technological support that encourages the development of skills in program solution comprehension at both the superficial and in-depth levels of learning, but especially at the latter level. Many different types of educational programming notations and development environments have been proposed in response to this need[46].

Despite the fact that the use of imagery in a programming notation and development environment is recommended based on studies of the cognitive model of the novice programmer (Chapter 2), an extensive literature review reveals that no single type of programming notation and associated development environment has been widely accepted for use in introductory programming courses as an alternative to that of a conventional textual programming notation and its associated development environment (Chapter 3). There consequently remains a requirement for a programming notation and associated development environment to sufficiently support the mental model of the novice programmer in an introductory programming course.

In response to the challenge, B#, a visual iconic programming notation and development environment, has been proposed in this thesis as the program domain in the learning environment of an introductory programming course. The design and implementation decisions for B# are documented in Chapter 5 with direct reference to support for 6 of the 8 novice programmer requirements derived in Chapter 2 (*R1*, *R2*, *R4 – R7* in Table 8.1).

The results presented in Section 7.3 and the associated deliberation on these results (Section 8.3) indicate that B# as a visual iconic programming notation, together with its associated development environment, encourages the learning of programming concepts at the in-depth level. The positive effect of the longer-term assimilation of programming concept knowledge at the in-depth level of learning is apparent in that significant differences are observed in the first week after termination of the treatment. This effect remains significantly visible up to 6 weeks after the conclusion of the treatment, especially for those students who have been identified as low-ability achievers.

[46] (Bonar *et al.* 1990; Lyons *et al.* 1993; Calloni *et al.* 1994, 1995; Studer *et al.* 1995; Liffick *et al.* 1996; Calloni *et al.* 1997; Cockburn *et al.* 1997; Crews *et al.* 1998; Blackwell *et al.* 1999a; Good 1999; Cooper *et al.* 2000; Garner 2000; Blackwell 2001; Dagiano *et al.* 2001; Materson *et al.* 2001; Navarro-Prieto *et al.* 2001; Chamillard *et al.* 2002; De Raadt *et al.* 2002; Fergusson 2002; Quinn 2002; Burrell 2003)

Furthermore, the treatment group participants were subject to extra cognitive load. This is due to the treatment group being laboured with an additional programming notation, development environment and relevant training manuals. Despite the resulting additional cognitive load, the improved results suggest evidence of B# in fact reducing the total cognitive load of novice programmers by addressing the extraneous load using a visual iconic programming notation. This observation is again especially evident in the measurements recorded for students who have been identified as being of low-ability in an introductory programming course. The suggestion that introductory programming students are visual learners (Chapter 2) who benefit from imagery imbedded within program solutions is thus confirmed by the current empirical analysis of the impact of a visual iconic programming notation.

Using a process of thematic analysis of responses to surveys conducted, an increase in the level of motivation is evident when using a visual iconic programming notation and associated development environment (*R3* in Table 8.1). The increase is evident from the perception that such a programming notation is easy to use as well as encourages the comprehension of programming concepts. Evidence suggests that the extrinsic motivation of marks awarded in assessments requiring program solutions only in the form of a conventional textual programming notation negatively impacts on the attitude towards the use of the visual iconic programming notation within an introductory programming course. Despite this observation, confirmation for the increased level of motivation when using a visual iconic programming notation is argued by deduction on the qualitative analysis of surveys conducted.

The integrated use of a visual iconic programming notation and associated development environment benefits students who have been identified as being of low-ability, as well as those who have been identified as being of medium risk, but to a lesser extent. The benefits are specifically that these students record a better performance achievement measure and substantially more of them pass the introductory programming course when compared with the observed throughput of the participants using only a conventional textual programming notation.

The theoretical implications of the design and implementation of B# as well as the current empirical investigation into the use of B# as technological support in the learning environment of an introductory programming course are therefore:

- a visual iconic programming notation and development environment that integrates a flowchart representation for the composition of program solutions with a conventional textual programming notation supports the mental model of low-ability novice programmers while learning to program;
- the incorporation of a visual iconic programming notation and development environment into an introductory programming course encourages a significant increase in individual academic performance achievement, especially of low-ability novice programmers; and
- the incorporation of a visual iconic programming notation and development environment into an introductory programming course encourages a significant increase in throughput, especially of low-ability novice programmers.

The evidence presented permits researchers in programming to focus on visual iconic programming notations and their associated development environments as appropriate technological support in the learning environment of an introductory programming course.

Despite these obvious theoretical implications, the investigative research methodology applied in the current empirical investigation augments the methodologies used in related studies in programming.

### *8.4.2 Enhancement of Existing Empirical Methodology*

It is common in studies on programming to compare single attributes, for example, a single programming construct, rather than an entire programming notation and development environment (Budd & Pandey 1995; Allen *et al.* 1996). This approach neglects to verify the appropriateness of the programming notation and development environment to the task of learning to program.

Furthermore, evidence of a measured increase in academic performance achievement is suggested in related comparative studies of learning to program (Calloni *et al.* 1997; Ramalingam *et al.* 1997; Crews 2001). These studies, while successfully comparing the observed average marks, neglect to compare the observed throughput of each of the control and treatment groups. Furthermore, the studies disregard the fact that certain sections of the participant population might benefit from the treatment more/less than other sections.

The investigative research methodology presented in the current study successfully addresses these deficiencies. A stratified sample-based between-groups statistical analysis is applied to the participants of both a control and treatment group. The specific statistical techniques applied are the measurement of the variance between the observed means as well as the equivalence of observed throughput for corresponding strata in each of the experimental groups.

The theoretical implications of the application of this investigative research methodology are therefore:

- inclusion of a further relevant aspect for comparative purposes, namely observed throughput; and
- isolation of distinct sections of the participant population in which the benefit of the treatment is more/less evident.

The investigative methodology presented permits researchers in programming to similarly increase the scope of their investigations with the incorporation of an additional relevant aspect for the purpose of a more comprehensive comparative study. Furthermore, the application of a validated computerised selection and placement battery for introductory programming students (Greyling *et al.* 2002; Greyling *et al.* 2003) enables the identification and isolation of a variety of distinct sections of the participant population (strata) upon which the comparative study can be conducted.

## 8.5 Instructional Implications

The composition of program solutions using a textual programming notation is suggested as the most appropriate technique for learning to program (Shih *et al.* 1993). The current empirical investigation suggests that the exclusive use of such a textual programming notation in the comprehension and composition of program solutions in the program domain makes learning to program difficult. *Support for a textual programming notation by means of continued integrated exposure over a period of time to a visual iconic programming notation like B# encourages more accurate program solutions to be composed as well as the retention and transfer of programming concepts to comparable problems in the textual programming notation*.

The manner in which B# connects imagery to the correct corresponding textual programming notation program solution extracts benefits introductory programming students. The visible exposure to accurate textual programming notation program solutions results in the future comprehension and composition of more accurate textual programming notation program solutions in the program domain.

Furthermore, the B# visual iconic programming notation and associated development environment supports the conceptual teaching of programming constructs as the method for learning to program. This support is provided by a reduction on the incidence of primitive programming constructs typically learnt at the superficial level of learning.

Despite the dangers associated with learning an additional programming notation and associated development environment, evidence suggests that the inclusion of a visual iconic programming notation in the teaching model of an introductory programming course benefits novice programmers, especially those who have been identified as being of low-ability. A major distinct advantage is the positive impact such a programming notation and associated development environment has on the existing challenge of increasing and maintaining satisfactory throughput in an introductory programming course at tertiary level.

## 8.6 Conclusion

Despite the existence of limitations in the experimental design acknowledged as being potentially discriminating to treatment group participants, the concrete model provided by B# tends to improve the performance achievement of low-ability (high risk strata) participants. This concrete model, however, has a much smaller effect for higher-ability (medium and low risk strata) participants.

The participants who received the program solution representations of B# first for any programming concept during practical learning activities were presumably able to use the flowchart model presented while encoding the concepts provided by the corresponding PASCAL textual programming notation. Interpretation of the empirical analysis suggests evidence of high-ability students in an introductory programming course already possessing their own useful concrete models for learning to program, and consequently benefiting on a lesser scale from the use of B#.

In the comparative analysis of the predicted and observed average final marks, each stratum in the treatment group performed at least as well as the corresponding stratum in the control group. Most of the treatment group strata measurements, however, exceeded those predicted. This finding is imitated in the comparative analysis of the predicted and observed throughputs.

The overall improved results for participants in the treatment group can only be attributed to the treatment, namely the use of B# as support during practical learning activities over a period of 9 weeks (65% of the course duration). During the practical learning activities, the treatment participants learnt to compose accurate textual programming notation program solutions. These program solutions exhibited accuracy in terms of structure, syntax and semantics due to the consistent and continued exposure of well-written textual programming notation program solutions during the treatment period.

Analysis of the measurements obtained from the evaluation materials provide evidence to conclude that B# is beneficial to introductory programming students despite the fact that it might not be the programming notation and development environment of choice. The results of the current study suggest evidence of the following significant benefits in the favour of the visual iconic programming notation and development environment, B#:

- easy to use;
- enhances the comprehension of programming concepts;
- assists in the development of a higher comfort level in a corresponding textual programming notation visually supported by B#;
- useful tool for novice programmers;
- encourages more uniform learning;
- uses flowchart representations to encourage successful in-depth learning for the comprehension and composition of program solutions;
- encourages the retention and transfer of programming knowledge;
- reduces the total cognitive load on introductory programming students by means of a reduction in extraneous load in terms of the use of an appropriate programming notation and associated development environment;
- uses imagery, confirming evidence that introductory programming students are visual learners; and
- functions well as instructional support for a conventional textual programming notation.

The deductive argument presented in this chapter suggests evidence that the use of B# increases the level of motivation in novice programmers. It cannot, however, be conclusively deduced that B# would be the programming notation and development environment of choice in the absence of the observed extrinsic motivation in the favour of the prescribed textual programming notation and development environment.

Empirical evidence implies that novice programmer performance achievement measured in terms of higher level of accuracy in program solutions composed in the program domain is improved with the use of B#, specifically for low-ability students. Consequently, B# is observed to provide comprehensive support (summarised in Table 8.2) for all 8 of the novice programmer requirements identified in Chapter 2 as being required to be features of technological support in the learning environment of an introductory programming course at tertiary level. B#, a visual iconic programming notation is therfore classified as a successful technological learning environment in an introductory programming course.

| **Requirements for Novice Programmer Technological Support** | **Supported** |
|---|---|
| ***R1***: Elimination of finer implementation details typically found at the superficial learning level of the program domain | ✓<br>(Section 8.2.1) |
| ***R2***: Increased level of program solution comprehension at the in-depth learning level of the program domain | ✓<br>(Section 8.2.2) |
| ***R3***: Increase in level of motivation when using the programming notation | ✓<br>(Section 8.3.2) |
| ***R4***: Designed specifically for use by novice programmers | ✓<br>(Section 8.2.3) |
| ***R5***: Provision of visual techniques to aid comprehension process at the in-depth learning level of the program domain | ✓<br>(Section 8.2.4) |
| ***R6***: Support for reduced mapping between the problem and program domains | ✓<br>(Section 8.2.5) |
| ***R7***: Increased focus on problem-solving | ✓<br>(Section 8.2.6) |
| ***R8***: Increase in novice programmer performance achievement measured in terms of higher level of accuracy in program solutions in program domain | ✓<br>**Low-ability students predominantly**<br>(Section 8.3.3) |

*Table 8.2: Support for Novice Programmer Requirements by B#*

The findings offered in this chapter provide researchers of studies in introductory programming a focus in terms of appropriate technological support in the learning environment of an introductory programming course. This is supplemented with evidence of a more comprehensive comparative methodology for the empirical analysis of observed measurements in such studies.

# Chapter 9

# Conclusion of Investigation

## 9.1 Introduction

The implementation of effective methods and strategies that encourage novice programmers to be successful in an introductory programming course results from increasing pressure from national government to improve student throughput at South African tertiary education institutions. Internationally recognised approaches to the problem of unsatisfactory throughput involve either the identification of potentially successful introductory programming candidates or the modification of the introductory programming course teaching model.

The former approach has been comprehensively researched at UPE since 1982. This research resulted in the successful implementation of a validated computerised selection and placement model in the Department of CS/IS at UPE in 2001. The latter approach seeks to increase and maintain satisfactory throughput of introductory programming students by adapting the course presentation techniques. Any modifications in presentation techniques are primarily to support students who have been identified as being of low-ability in an introductory programming course. In response to the challenge posed by the second type of approach, the goal of this investigation is to establish and assess the impact of a visual iconic programming notation in the role of a development environment for an introductory programming course.

In order to validate the selection of a suitable programming notation for novice programmers to be used in the empirical study, the interpretation of a comprehensive literature study results in the following:

- a framework of novice programmer requirements (Chapter 2). This framework emphasises the requirements for programming notations and development environments that support the behaviour of novice programmers when learning to program (Table 2.1 duplicated as the criteria column in Table 9.1).
- the identification and classification of 6 different categories of experimental technological support used in the learning environment of introductory programming courses (Chapter 3). These technological supports are proposed as alternatives to those that support conventional textual programming notations.
- evaluation of the identified categories of introductory programming technological support against the criteria contained in the framework of novice programmer requirements (Chapter 3). It is observed that the category of visual programming notations is the most responsive to the identified novice programmer requirements (Table 3.7 duplicated as column G in Table 9.1).
- design and implementation of a visual iconic programming notation and development environment, B# (Chapter 5).

A subsequent conclusion is that B#, in terms of design and implementation, supports all of the identified novice programmer requirements except for 2 requirements (*R3* and *R8* in Table 9.1), whose support is validated by the empirical analysis section of this thesis. The methodology for the empirical analysis is described (Chapter 6) against the background of the pre-selective technique applied to participants in the study by means of the placement model currently in use at UPE (Chapter 4). The empirical analysis utilises the statistical testing techniques of measuring the differences between observed average marks and observed throughput. These techniques are applied in a between-groups quantitative comparative analysis between two stratified sample-based groups of participants, namely a treatment and control group. Qualitative analysis using the technique of thematic analysis supplements the aforementioned quantitative analysis.

The results of the quantitative and qualitative empirical analysis (Chapter 7) and associated deliberations (Chapter 8) suggest evidence that B# satisfies the entire set of identified novice programmer requirements (Table 9.1). The level of support provided by B# is especially significant for students who have been identified as being of low-ability in an introductory programming course. Modification of the presentation techniques of an introductory programming course with the inclusion of B# as technological support in the learning environment suggests strong evidence of increased throughput. The observed improved throughput attributed solely to the treatment achieves the target of 75% recommended by national government.

This chapter evaluates the objectives as determined in Chapter 1 for the current investigation (Section 9.2). Flowing out of this discussion, the contribution of this thesis to existing research within the context of the use of technological support in the learning environment of an introductory programming course is argued (Section 9.3). The findings of the current investigation suggest a number of instructional implications (Section 9.4), as well as a number of limitations of the research. The main limitations are presented (Section 9.5) together with suggestions for future research (Section 9.6).

## 9.2 Evaluation of Research Objectives

The objectives of this study presented as research questions in Chapter 1 are summarised as follows (derived from Table 1.1):

- establishment of the mental model of novice programmers when learning to program;

- specification of criteria for the selection of an appropriate programming notation and associated development environment for novice programmers;
- identification and categorisation of programming notations and associated development environments used in introductory programming courses at tertiary level;
- evaluation of experimental programming notations and associated development environments according to the selection criteria;
- overview of the contribution of previous research at UPE to the current investigation;
- discussion of the development of the visual iconic programming notation and associated development environment used as the experimental instrument in the study; and
- investigation of the impact of a visual iconic programming notation on performance achievement and motivation of introductory programming students by means of appropriate quantitative and qualitative analysis techniques.

The successful act of programming is the accurate transformation of an appropriate mental representation of a problem statement in the problem domain to a corresponding solution in the form of a particular programming notation in the program domain (Figure 9.1). Novice programmers typically experience a large amount of effort being required during this conversion process.

The large amount of conversion required results in a high cognitive load being placed on the novice programmer. The only way in which the cognitive load while learning to program can be minimised is by means of the reduction of extraneous cognitive load. Extraneous cognitive load is dependent upon the selection of an appropriate programming notation and associated development environment to be used in the program domain. Conventional textual programming notations negatively affect the extraneous cognitive load of novice programmers by means of the high level of precision required in the syntax of such programming notations. The syntax of each programming construct supported by such programming notations needs to be effectively mastered at the superficial level of learning prior to the occurrence of in-depth learning of the programming concept, the latter type of learning being the preferred outcome of an introductory programming course.

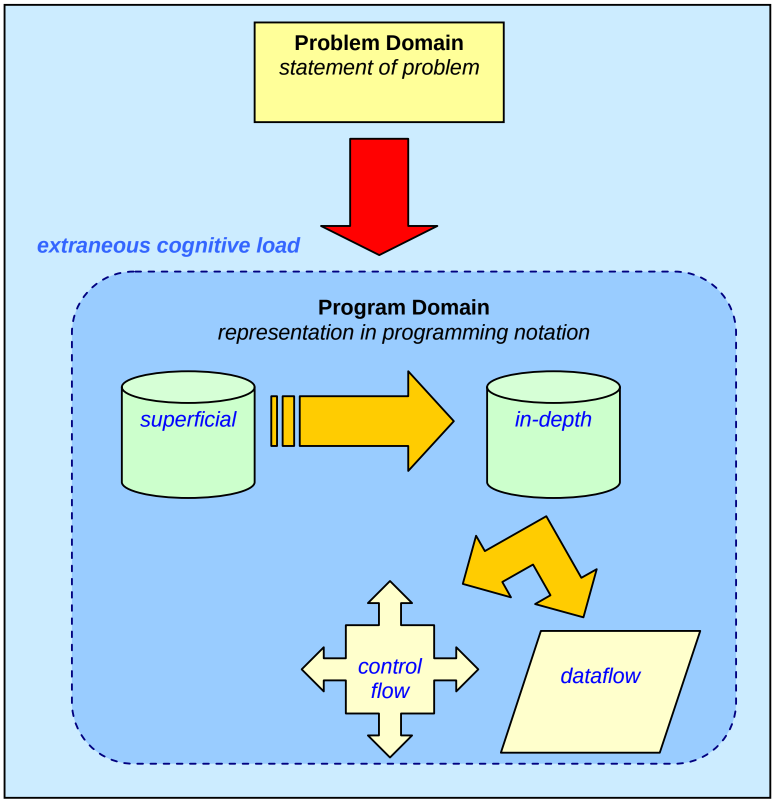


*Figure 9.1: Mental Model of Novice Programmer while Learning to Program*

Evidence of the successful assimilation of programming knowledge is the existence of the in-depth level of learning phases of control flow and dataflow comprehension. The mastering of control flow comprehension permits the novice programmer to derive knowledge of how a program solution functions. Mastering of dataflow comprehension permits the novice programmer to recognise program solution patterns.

The mastering of both of these phases is essential for a novice programmer to successfully transfer knowledge of programming concepts in the composition of novel program solutions for new but similar problems. There is thus a need for appropriate technological support in the learning environment of an introductory course that encourages the mental model of a novice programmer as described and depicted in Figure 9.1.

| Criteria | A | B | C | D | E | F | G | B# |
|---|---|---|---|---|---|---|---|---|
| ***R1***: Elimination of finer implementation details typically found at the superficial learning level of the program domain | ✗ |  | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| ***R2***: Increased level of program solution comprehension at the in-depth learning level of the program domain | ✗ | ✓ | ✓ | ✓ |  | ✓ | ✓ | ✓ |
| ***R3***: Increase in level of motivation when using the programming notation |  | ✓ | ✓ |  |  | ✓ |  | ✓ |
| ***R4***: Designed specifically for use by novice programmers | ✗ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ | ✓ |
| ***R5***: Provision of visual techniques to aid comprehension process at the in-depth learning level of the program domain |  |  | ✓ | ✓ |  |  | ✓ | ✓ |
| ***R6***: Support for reduced mapping between the problem and program domains | ✗ | ✓ |  | ✓ |  |  | ✓ | ✓ |
| ***R7***: Increased focus on problem-solving | ✗ | ✓ |  |  |  | ✓ | ✓ | ✓ |
| ***R8***: Increase in novice programmer performance achievement measured in terms of higher level of accuracy in program solutions in program domain |  | ✓ | ✓ |  | ✓ |  | ✓ | ✓ |

**Legend**
A Conventional IDEs and Textual Programming Notations
B Problem Analysis Supporting Development Environments
C Mini-languages and Micro-worlds
D Pseudo-programming
E Worked Examples and Code Restructuring
F Scripting Languages
G Visual Programming Notations and Development Environments

✗ Not supported

✓ Supported

*Table 9.1: Evaluation of Programming Notations and Associated Development Environments*

Technological support in the learning environment of an introductory programming course should strive to support the 8 criteria derived in Chapter 2 from the analysis of the mental model of a novice programmer while learning to program and listed in Table 9.1. Verification of the use of the criteria is presented in terms of the evaluation of different categories of alternative technological support used in the learning environment of an introductory programming course.

Six distinct categories of alternative technological support used in introductory courses are identified and defined in Chapter 3. These categories appear in the legend of Table 9.1 (B – G). In the evaluation of these categories of technological support proposed as alternatives to conventional textual programming notations and their associated environments, it is at times difficult to separate the effects of the programming notation from the effects of the development environment due to the strong cohesion between the two components. Chapter 3, however, succeeds in highlighting the deficiencies associated with conventional textual programming notations and their development environments that novice programmers experience while learning to program (summarised in column A of Table 9.1).

Each alternative category of technological support is assessed in Chapter 3 with respect to support for the requirements of a novice programmer while learning to program (summarised in columns B – G in Table 9.1). Table 9.1 therefore illustrates the relative suitability of each category to the framework of novice programmer requirements.

In some cases, there is insufficient evidence to conclusively determine the level of support for a particular requirement (indicated by white shading in Table 9.1). The evaluation process, however, suggests evidence that the category of visual programming notations is the most responsive to the requirements of a novice programmer. The design and implementation of B# (Chapter 5), a visual iconic programming notation and programming development environment, is influenced by this finding.

Prior to the use of B# in the empirical study, previous related research conducted by the Department of CS/IS at UPE (Chapter 4) is acknowledged as being relevant to the current investigation for the following reasons:

- the research forms part of the ongoing research at UPE to investigate strategies that improve and maintain satisfactory throughput in introductory programming courses;
- the research involves the identification of potentially successful introductory programming candidates and is thus an implementation of the first type of approach acknowledged as being instrumental in increasing and maintaining satisfactory throughput in introductory programming courses;
- participants in the current investigation have been pre-selected into the introductory programming course by means of a validated computerised placement model implemented as a direct result of the above mentioned research; and
- the computerised placement model is used to determine the stratum into which a participant in the investigative study is classified.

The need for the implementation of the second type of approach that increases and maintains satisfactory throughput in an introductory programming course is prompted by the limited success of the implementation of the model recommended by the previous research conducted at UPE. The second type of approach necessitates the modification of the course presentation techniques for an introductory programming course. The current investigation consequently explores the use of B# as an example of a visual iconic programming notation and development environment as technological support in the learning environment of such a course (Chapter 6).

Argument is presented on the correlation between novice programmer requirements for technological support and issues relevant to the design and implementation of B# (Chapter 5). The empirical analysis of the effect of B# on novice programmers determines that the visual iconic programming notation and its associated development environment positively impacts on both the academic performance achievement and motivation of students (Chapters 7 and 8). The positive affect of B#

is most evident in students who are identified as being of low-ability in the course (predicted mark of 41% – 50%).

The significant improvement in both observed average marks achieved as well as observed throughput can be attributed solely to the inclusion of B# as technological support in the learning environment of the introductory programming course at UPE. *B# is thus evaluated as supporting all eight of the requirements of a novice programmer* (Table 9.1).

## 9.3 Contribution of Thesis

There exists insufficient research and empirical evidence (Chapter 6), both nationally and internationally, to justify the use of non-conventional programming notations and development environments to improve throughput in an introductory programming course. This deficiency in documented research is especially evident in respect of non-textual programming notations like visual iconic programming notations.

Documented research (Chapter 8) on the use of experimental technological support provides observations that have a tendency to be anecdotal in nature. There is consequently a lack of sufficient application of formal evaluation techniques in the assessment of the use of experimental technological support. Furthermore, experimental technological support is not widely used in introductory programming courses since it remains an expensive, time consuming and non-profitable exercise to fully develop such support successfully.

The present research forms an integral part of ongoing attempts at UPE to improve and sustain satisfactory throughput in an introductory programming course. The contributions of this thesis are chiefly apparent in the following areas:

- determination of a framework of novice programmer requirements for technological support in the learning environment of an introductory programming course (Section 9.3.1);

- categorisation and assessment of existing introductory programming experimental programming notations and development environments (Section 9.3.2);
- design and implementation of a visual iconic programming notation and associated development environment that satisfies the framework of novice programmer requirements (Section 9.3.3); and
- enhancement of research methodology and results in existing evidence of empirical studies in the use of programming notations and development environments in the learning environment of an introductory programming course (Section 9.3.4).

Each of the above mentioned contributions address the deficiencies presently existing in current documented research.

### *9.3.1 Framework of Novice Programmer Requirements for Technological Support in the Learning Environment of an Introductory Programming Course*

The literature review presented in Chapter 2 fails to directly produce an acknowledged framework of novice programmer requirements for technological support in the learning environment of an introductory programming course, yet it emphasises that novice programmers require technological support that encourages the development of skills in program solution comprehension at both the superficial and in-depth levels of learning, but especially at the latter level. As a result of the extensive literature study, a number of common characteristics have been identified, consolidated and proposed as novice programmer requirements for technological support in the learning environment of an introductory programming course (Table 9.1).

The proposed set of 8 novice programmer requirements is recommended to be satisfied in the program domain. In satisfying these requirements, a programming notation and development environment reduces the extraneous cognitive load, and thereby the total cognitive load, of novice programmers when learning to program (Figure 9.1). The requirements are also proposed as the assessment criteria against which programming notations and development environments can be evaluated in order to permit instructors of introductory programming courses to make a more informed choice on the programming notation and associated development environment used as technological support in the learning environment of an introductory programming course.

### *9.3.2 Categorisation and Evaluation of Existing Technological Support in the Learning Environment of an Introductory Programming Course*

A large number of experimental programming notations and development environments have been recommended over the past decade as technological support alternative to that of conventional textual programming notations and their associated development environments[47]. This situation suggests evidence of dissatisfaction with the latter category of technological support, namely textual programming notations and their associated development environments.

The quantity of experimental technological support is furthermore indicative of no single alternative category receiving widespread acceptance. An extensive literature survey failed to produce an acknowledged standard for assessment criteria against which technological support in the learning environment of an introductory course can be evaluated.

The categorisation of the existing experimental programming notations and their associated development environments in terms of commonality of features is required prior to the validation of the proposed framework of novice programmer requirements derived in Chapter 2 as assessment criteria for technological support in the learning environment of an introductory programming course. A total of 7 distinct categories are identified and proposed (Table 9.2).

No single category of experimental programming notation and development environment reviewed in Chapter 3 was evaluated as comprehensively satisfying the

[47] (Bonar *et al.* 1990; Lyons *et al.* 1993; Calloni *et al.* 1994, 1995; Studer *et al.* 1995; Liffick *et al.* 1996; Calloni *et al.* 1997; Cockburn *et al.* 1997; Crews *et al.* 1998; Blackwell *et al.* 1999a; Good 1999; Cooper *et al.* 2000; Garner 2000; Stajano 2000; Blackwell 2001; Dagiano *et al.* 2001; Materson *et al.* 2001; Navarro-Prieto *et al.* 2001; Baas 2002; Chamillard *et al.* 2002; De Raadt *et al.* 2002; Fergusson 2002; Gibbs 2002; Quinn 2002; Burrell 2003; Donaldson 2003; Lane 2003; Lane *et al.* 2003; Watts 2003; Carlisle *et al.* 2004; Lane 2004; Lane *et al.* 2004a, b; Mahmoud *et al.* 2004; Zelle undated)

framework for novice programmer requirements for technological support in the learning environment of an introductory programming course (summarised in columns B – G in Table 9.1). This result is partly due to the lack of sufficient documentary evidence.

| Category | Description | Examples (Chapter 3) |
|---|---|---|
| Conventional textual programming notation | Integrated development environment that supports textual programming notation | Delphi™ Enterprise;<br>Visual Studio |
| Problem Analysis Supporting Development Environments | A problem-solving methodology integrated with the program solution development tasks | SOLVEIT;<br>Interrogative programming;<br>Coached program planning |
| Mini-languages and Micro-worlds | Fewer programming commands.<br>Simulates a real or imaginary world | GRAIL; Karel the Robot; Jeroo; Alice |
| Pseudo-programming | Less rigid syntactical rules | MULSPREN;<br>Literate programming |
| Worked Examples and Code Restructuring | Exposure of correct program solutions in the form of a conventional textual programming notation | CORT |
| Scripting Languages | Eliminates the rigidity of data typing usually required by conventional textual programming notations | Python; JavaScript; VBScript |
| Visual Programming Notations and Development Environments | Programming by means of the interactive manipulation of visual expressions such as graphics or icons | LabVIEW; Prograph; SCIL-VP; DataVis; FLINT; Visual Logic; RAPTOR; SFC BACCII©; SIVIL; Youngster; EC |

*Table 9.2: Categories of Technological Support*

The illustration of the application of the framework for novice programmer requirements to assess existing technological support in the learning environment of an introductory programming course serves as an example of a technique to evaluate the appropriateness of any given programming notation and development environment.

### *9.3.3 Design, Implementation and Utilisation of a Visual Iconic Programming Notation and Development Environment*

The findings of the current research suggest evidence that a visual iconic programming notation and development environment exhibiting the properties of B# promote the following benefits:

- reduces the extraneous cognitive load of novice programmers; and
- encourages the assimilation of programming concept knowledge at the in-depth level of learning in a textual programming notation by
    - using flowchart representations for the composition of program solutions; and
    - visually supporting an integrated, accurate and equivalent textual programming notation representation of a flowchart program solution.

This thesis presents strong evidence of the benefits of using a visual programming notation which facilitates learning and does not encumber the novice programmer in terms of extraneous cognitive load. The evidence presented provides instructors of introductory programming courses with confirmation that a visual programming notation is an appropriate technological support in the learning environment of an introductory programming course.

### *9.3.4 Enhancement of Existing Investigative Research Methodology and Results*

The empirical analysis of the current investigation confirms and strengthens the empirical evidence in the existing research (Calloni *et al.* 1997; Ramalingam *et al.* 1997; Crews 2001; Carlisle *et al.* 2004). These studies suggest evidence that visual programming notations enhance the individual performance achievements of students, increasing the average marks achieved. By using a stratified sample-based analysis strategy to isolate and highlight the sections of the novice programmer population that experience substantial benefit from the use of B#, the current investigation enhances the success of the aforementioned related studies in programming. The current investigation thus adopts, modifies and confirms the success of an approach of using a visual programming notation in the learning environment of an introductory programming course.

Furthermore, the current investigation presents findings that not only demonstrate improved average marks, but also improved throughput and motivation of novice programmers, especially for those who have been identified as being of low-ability in an introductory programming course.

## 9.4 Implications of Findings

It is unacceptable that large amounts of money be spent on students of low-ability who are at risk of not being successful in introductory programming courses. Any effort to reduce the volume of students who fail such courses will not only benefit the students, but also the tertiary education institution. One such effort is the modification of the presentation techniques of an introductory programming course.

Technological support is integral to the learning environment of an introductory programming course. The choice of such support has obvious implications, as reported on in this thesis. If a programming notation and associated development environment is required in the longer term but is difficult for students to use and understand, attempts should be made to ease the difficulties. Students of the introductory programming course at UPE are expected to acquire the programming skills associated with a conventional textual programming notation. The difficulties associated with this expectation are reduced with the integration of B# into the learning environment.

The issues relevant to the modification of the introductory programming course presentation techniques with the incorporation of B# as technological support for a required textual programming notation as reported on in this thesis include the following:

- the use of multiple programming notations and associated development environments;

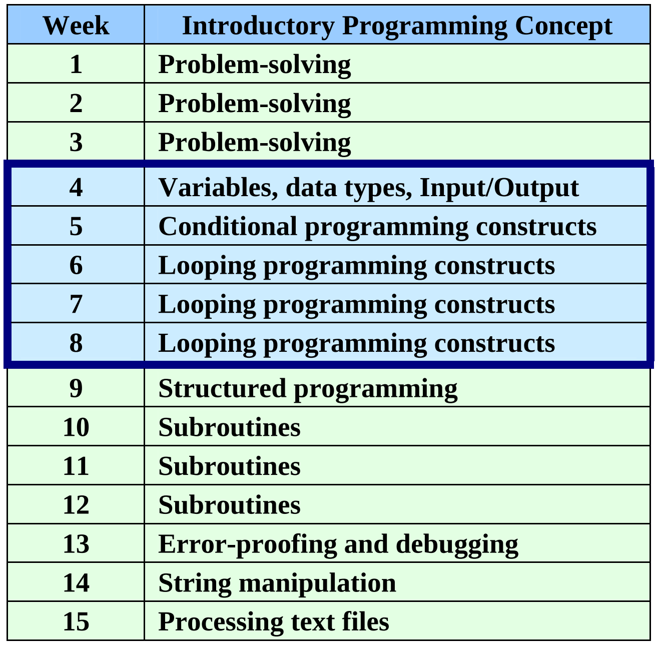
- the lack of interactive training in the use of the B# programming notation and development environment; as well as
- the existence of the extrinsic motivation of marks awarded.

Incorporating a supplemental programming notation and development environment such as B# into the learning environment of an introductory programming course has the potential to overload the students. In order to reduce this risk, rigid management of appropriate tasks administered during practical learning activities is required. This management would incorporate the inclusion of practical learning activity tasks that encourage the transfer of programming concept knowledge from program solutions composed in B# to new but similar problems requiring the composition of program solutions in a conventional textual programming notation. The volume of practical learning activity tasks presented would also have to be monitored closely in order to optimise the use of the allocated time for the introductory programming course.

A related issue is the fact that the current version of B#, although designed to present an intuitive interface, is greatly in need of an interactive interface for the purposes of training students in its use. This interface would replace the current hard copy training manual (Appendix C), be included with the development environment and would consequently be more directly accessible to the students.

Instructors of introductory programming courses appreciate the instructional benefits provided by technological support such as B#. Students, however, do not appreciate these benefits. One reason for this observation is due to the lack of the extrinsic motivation of marks. In order to encourage students of an introductory programming course to use B#, a certain amount of suitable examinable content directly applicable to B# should be identified and examined. An example of such content is requiring students to accurately match textual programming notation program solution extracts to appropriate B# icons.

Further, the use of B# should be mandatory for students who have been identified as being of low-ability in the course, that is, those who achieve a predicted mark in the range 41% – 50% in the implemented computerised placement model prior to entry into the course. In this way, the section of the student population who have been observed as being advantaged by using B# would be encouraged and rewarded for benefiting instructionally from B#.



| Week | Introductory Programming Concept |
|---|---|
| 1 | Problem-solving |
| 2 | Problem-solving |
| 3 | Problem-solving |
| 4 | Variables, data types, Input/Output |
| 5 | Conditional programming constructs |
| 6 | Looping programming constructs |
| 7 | Looping programming constructs |
| 8 | Looping programming constructs |
| 9 | Structured programming |
| 10 | Subroutines |
| 11 | Subroutines |
| 12 | Subroutines |
| 13 | Error-proofing and debugging |
| 14 | String manipulation |
| 15 | Processing text files |

*Figure 9.2: Incorporation of B# into the learning activities of an Introductory Programming Course*

Due to identified deficiencies (Section 7.4.1) of the manner in which the current version of B# implements subroutines (Thomas 2002a), the proposed incorporation of B# into the introductory programming course curriculum in the Department of CS/IS at UPE would initially incorporate support for only the following programming concepts (Figure 9.2):

- variables;
- data types;
- input and output;
- single and multiple branching conditional programming constructs; as well as
- counting, pre- and post-test sentinel looping programming constructs.

The area in Figure 9.2 that is framed in dark blue indicates the portion of the introductory programming course during which B# is used in practical learning activities. B# would therefore still be short-term technological support and would cater for 33% of the total introductory course duration.

The discussion on the implications of the findings has highlighted challenges concerning the modification of an introductory programming course teaching model with the inclusion of B#. If these challenges are not successfully addressed, the benefits of B# as technological support will not be realised. In addition to the challenges presented, certain limitations of the current investigation must also be highlighted.

## 9.5 Limitations of Research

Limitations of this research are mainly related to ethics, limited sample sizes, extrinsic reward of marks and restricted functionality of B#. The following limitations are identified:

- for academic ethical reasons, B# could not be used exclusively as the technological support in the learning environment of the introductory programming course. Both students and subsequent UPE programming courses require the programming skills of a conventional textual programming notation and associated development environment. Therefore, it could not be determined, for example, whether exclusive use of B# could account for a more significant variance in academic performance achievement;
- the introductory programming course at UPE consists of a restricted population size ($N$ = 211 in 2003) from which suitable participants could be selected. The population size is reduced even further by a high attrition rate, resulting in a population of 148 students who were administered all materials required for the investigation. The stratified sample-based between-groups analysis technique applied in the study restricted the sample size even further ($n$ = 59 in each group). Practical issues surrounding the isolation of participants in each of the experimental groups to independent practical learning activities also impacted on the final sample size.

- B# was never the topic of any formal instruction and the programming notation is not examinable in the introductory programming course. A B# model was given prior to the application of programming concepts but never prior to the teaching of the concepts in a lecture learning activity (Appendix C). Further, students are more co-operative when being challenged, and thus a more significant impact could have materialised in the event of the students being formally examined and rewarded for B# programming notation tasks. Furthermore, the current investigation used a repeated measures design for practical assessments administered. Due to a lack of criteria for the arguing of the equivalence of these tasks for experimental purposes, moderation by course instructors was implemented.
- version of B# used in the empirical study implemented subroutines (functions and procedures) in a way that was frustrating to the treatment group participants in the investigation. Despite this fact, strong evidence of improved performance resulted.
- version of B# used in the empirical study had no visual presentation of program solution execution to directly encourage exploratory learning and problem-solving. The version exhibiting this functionality was in development at the time that the study was being conducted. No conclusion in regard to the impact of this feature can thus be made.
- current empirical study measured only the accuracy of program solutions. Related studies in programming measure timing as well as the total number of errors recorded during the development of any particular program solution. No mechanism was in place at the time of the current investigation to measure the time taken to complete program solutions by participants in each of the experimental groups, or to record the total number of errors made during the composition of program solutions. No conclusion could thus be made in this regard.
- results of the current investigation suggest evidence of the transfer and retention of knowledge of programming concepts over a period of time. No mechanism exists for the measurement of this factor, and thus no conclusion can be made in this regard.

While the present study finds distinctive differences in the performance levels of students identified as being at risk in the introductory programming course, its limitations leave many questions which provide directions for future research.

## 9.6 Recommendations for Further Research

In recommending directions for future research, the development of a simplified development environment and interactive dynamic graphical algorithm animation environment as well as the need for further controlled studies in programming are highlighted. Figure 9.3 illustrates the context of the current investigation reported on by this thesis (in orange) in relation to existing research (in grey) and provoked current research at UPE (in green).

There is a need for controlled studies that separate the effects of visual programming notation primitives from the interactive aspects of the development environment. Such studies would be valuable in determining whether the visual aspects of a programming notation add any benefits that an interactive textual programming notation cannot. Research (*Programming Environments* in Figure 9.3) that determines whether a simplified development environment that supports a reduced set of interactive features in order to reduce gratuitous complexity has recently been provoked at UPE as a result of the findings of this thesis (De Jager 2004; Vogts 2004).

Related research into the effects of a dynamic interactive graphical algorithm animation system has also recently been provoked (Yeh 2004). This system permits a student to interactively define data structures, data contents as well as algorithms to the animator system. The effects of the execution process are visualised in a comparative exploratory learning manner in order for the student to make informed decisions as to algorithm efficiency under a variety of conditions (*Algorithm Visualisation* in Figure 9.3).

Looping programming constructs are an acknowledged area of difficulty for many novice programmers. Research that is related to the current study (*Generic programming notation* in Figure 9.3) and involves the development of a programming notation that, amongst other programming constructs, encompasses all three types of looping mechanisms (counter controlled, pre- and post-test) into a single simple looping mechanism has also been provoked (Naudé 2004).

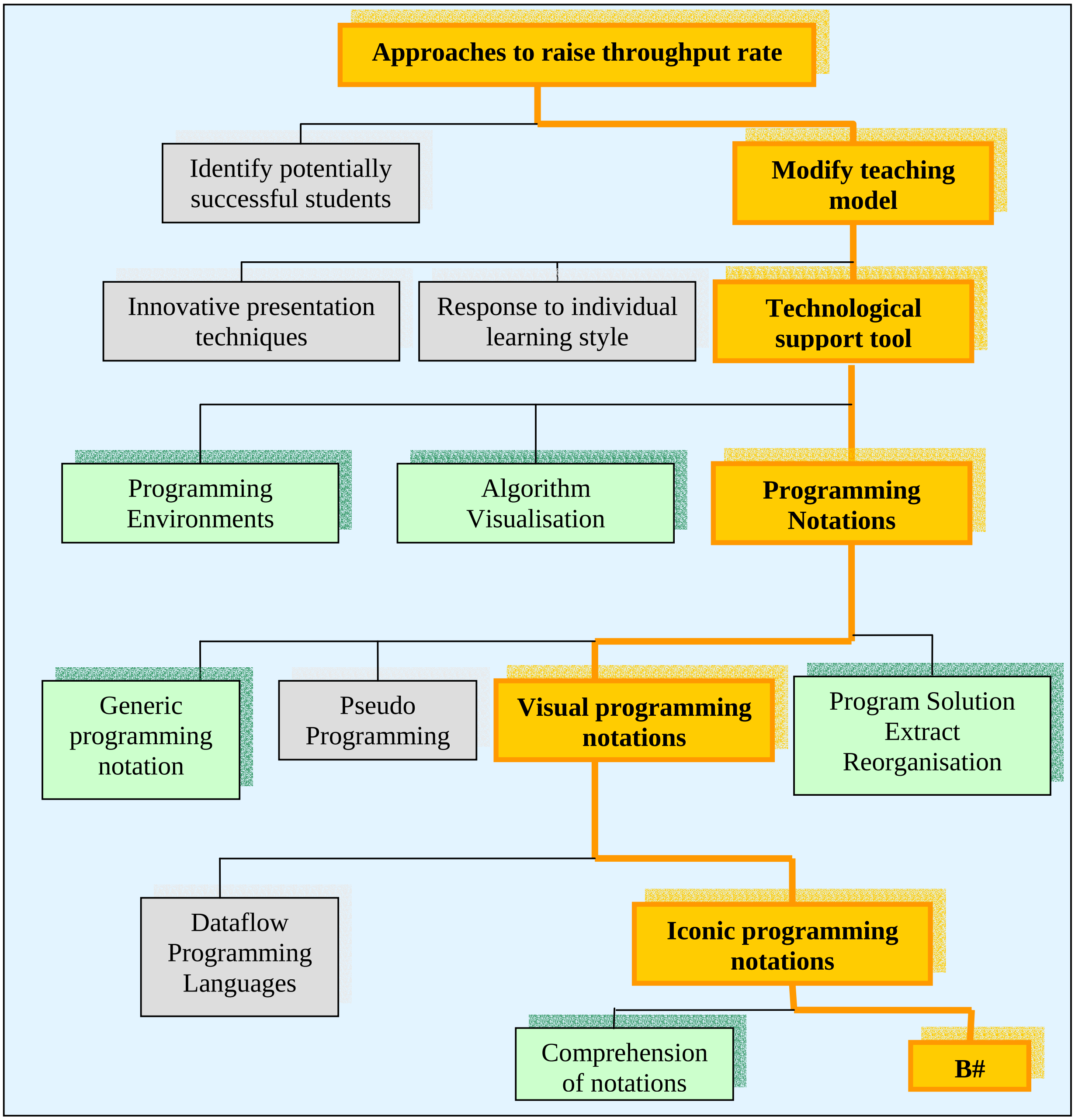

Approaches to raise throughput rate
Identify potentially successful students
Modify teaching model
Innovative presentation techniques
Response to individual learning style
Technological support tool
Programming Environments
Algorithm Visualisation
Programming Notations
Generic programming notation
Pseudo Programming
Visual programming notations
Program Solution Extract Reorganisation
Dataflow Programming Languages
Iconic programming notations
Comprehension of notations
B#

*Figure 9.3: Current Related Research at UPE*

The current research reported on in this thesis has also promoted research into the development of a system (Mamtani 2004) to be used in a future controlled study that evaluates whether novice programmers prefer visual iconic programming notation program solutions to the equivalent textual programming notation program solutions in terms of accuracy of comprehension (*Comprehension of notations* in Figure 9.3). Another system under development (Henning 2004) will provide the means to interactively train and assess introductory programming students in the completion of textual programming notation program solutions, and thereby enhance their comprehension skills at the in-depth level of learning (*Program Solution Extract Reorganisation* in Figure 9.3).

Visual feedback of the execution of program solutions is expected to enhance the understanding of the semantics of program solutions. There is a need for a further controlled study to determine the effect of B#'s tracing facility on introductory programming students using the current version of B# (Yeh 2003a, b). Further, the current study reported on in this thesis has compared B# with a textual programming notation that exhibited the same programming paradigm as that of B#. Comparative studies between comprehension of program solutions in B# and programming notations that exhibit differing paradigms, for example, an object-oriented paradigm, are required to be conducted.

The findings of the current investigation also prompted investigation into evaluation techniques that will assess B#'s suitability for use in the particular context of an introductory programming course. The technique of the cognitive dimensions of notations framework (Green & Blackwell 1998; Blackwell *et al.* 1999b; Green 2000) is used to determine whether the intended activities of the novice programmer are adequately supported by the structure of B# (Cilliers *et al.* 2004a). The main deficiency highlighted by this assessment is that there is a requirement for B# to support interactive learning support by means of an interactive tutorial component. The integration of this component into B# would allow novice programmers to educate themselves about the syntax, semantics and applications of B# and thereby significantly improve the learnability of B#. A controlled study to confirm this observation is required.

Furthermore, there is a need to determine the locus of effect of B#. This controlled study would involve determining the effects of B# when used prior to and after the learning of a conventional textual programming notation. A more conclusive scientific measuring instrument is also required to assess the level of motivation of participants using B#.

A related study would involve using eye-tracking devices to determine how often, if at all, students refer to the automatically generated textual programming notation program solution. This study would also involve determining whether programming concepts are successfully transferred when using B# prior to using an industry accepted programming notation and development environment.

The qualitative analysis technique applied (Chapter 7) indicates an emerging theme that B# is characterised as being easy to use. A study which examines the following hypothesis is therefore encouraged:

$H_0$: *It is easy to use B# when learning to program.*

$H_1$: *It is not easy to use B# when learning to program.*

It would also be interesting to compare the observed subsequent programming course marks for participants in the treatment and control groups of the study reported on in this thesis. While such data would not represent a valid longitudinal empirical study, the historical study might indicate a trend for the long term impact on subsequent programming courses of the use of B# in the introductory programming course.

## 9.7 Summary

The artefacts resulting from this research are:

- a framework for novice programmer requirements for technological support in the learning environment of an introductory programming course (Chapter 2; duplicated in Table 9.1);
- categorisation of experimental programming notations and development environments used as technological support in the learning environment of an introductory programming course (Chapter 3; legend of Table 9.1);

- evaluation of categories of technological support in the learning environment of an introductory programming course in terms of the provided framework (Chapter 3; summarised in Table 9.1);
- a visual iconic programming notation and development environment that supports a framework for novice programmer requirements (Chapter 5);
- investigative methodology for an investigative study in programming (Chapter 6);
- empirical evidence in support of a visual iconic programming notation and development environment (Chapters 7 and 8); and
- proposal for the incorporation of B# into the learning environment of an introductory programming course (Section 9.4).

The framework of novice programmer requirements is used to examine existing technological support as well as support the design of new technological support used by novice programmers while learning to program. It thus provides information for the selection of existing as well as the design of new technological support.

Research that further develops the framework for novice programmer requirements, validates and verifies the application of the framework to technological support in the learning environment of an introductory programming course or adapts and improves on the presented investigative methodology to related studies is provoked.

This thesis has treated programming notations and development environments in the learning environment of an introductory programming course as technological support that sustains the mental model of a novice programmer. The thesis has effectively applied substantial existing research on the cognitive model of the novice programmer as well as that on experimental technological support to the comparison of the effect of technological support in the teaching model of a tertiary level introductory programming course. The increase of throughput to a recommended rate of 75% in the tertiary level introductory programming course at UPE is attributed solely to the incorporation of iconic technological support in the teaching model of the course.

# References

AC Nielsen Research Services (2000): Employer Satisfaction with Graduate Skills.

ACC (1997): ACCUPLACER Program Overview: Coordinator's Guide.

Adelson, B. (1984): When novices surpass experts: The difficulty of a task may increase with expertise. *Journal of Experimental Psychology: Learning, Memory and Cognition*, **10**:483 - 495.

Allen, R.K., Grant, D.D. and Smith, R. (1996): Using Ada as the first Programming language: A Retrospective. In *Proceedings of Software Engineering: Education & Practice*.

Applin, A. G. (2001): Second Language Acquisition and CS1: Is * = = ** ? *ACM SIGCSE Bulletin*, **33**(1):174 - 178.

Astrachan, O. (1998): Concrete Teaching: Hooks and Props as Instructional Technology. *ACM SIGCSE Bulletin*, **30**(3):21 - 24.

Atkinson, R. C. (2001): Achievement versus Aptitude in College Admissions, in *Issues in Science and Technology Online*, **Winter**. Available at http://www.nap.edu/issues/18.2/atkinson.html. [Accessed on 19 September 2002].

Austin, H. S. (1987): Predictors of Pascal Programming Achievement for Community College Students. *ACM SIGCSE Bulletin*, **19**(1):161 - 164.

Baas, B. (2002): Ruby in the CS Curriculum. *Journal of Computing in Small Colleges*.

Bauer, R., Mehrens, W. A. and Visionhaler, J. R. (1968): Predicting Performance in a Computer Programming Course. In *Proceedings of American Educational Research Association Annual Meeting,* Chicago.

Ben-Ari, M. (1998): Constructivism in Computer Science Education. *ACM SIGCSE Bulletin*, **30**(1):257 - 261.

Ben-Ari, M. (2001): Constructivism in Computer Science Education. *Journal of Computers and Mathematics in Science Teaching*, **20**(1):45 - 73.

Berenson, M. L. and Levine, D. M. (1999): *Basic Business Statistics: Concepts and Applications*. 7th Edn, Prentice-Hall International, Inc. ISBN 0-13-081254-4.

Biggs, J. (1999): Teaching for Quality Learning at University. *Society for Research into Higher Education/Oxford University Press*.

Blackwell, A. F. (2001): Pictorial Representation and Metaphor in Visual Language Design. *Journal of Visual Languages and Computing*, **12**:223 - 262.

Blackwell, A. F., Whitley, K. N., Good, J. and Petre, M. (2001): Cognitive Factors in Programming with Diagrams. *Artificial Intelligence Review*(Thinking with Diagrams). Available at http://cl.cam.ac.uk/users/afb21/publications/aire.pdf.

Blackwell, A.F. (1996): Metacognitive Theories of Visual Programming: What do we think we are doing? In *Proceedings of IEEE Symposium on Visual Languages*:240 - 246. Available at http://cui.unige.ch/eao/www/Visual/local/Blackwell96.html.

Blackwell, A.F. and Green, T.R.G. (1999a): Does Metaphor Increase Visual Language Usability? In *Proceedings of IEEE Symposium on Visual Languages*:246 - 253. Available at http://www.cl.cam.ac.uk/~afb21/publications.html.

Blackwell, A.F. and Green, T.R.G. (1999b): Investment of Attention as an Analytic Approach to Cognitive Dimensions. In T. Green, R. Abdullah and P. Brna (Eds.) *Proceedings of 11th Annual Workshop of the Psychology of Programming Interest Group*:24 - 35. Available at http://www.cl.cam.ac.uk/users/afb21/publications/PPIG99.html.

Blackwell, A.F. and Green, T.R.G. (2000): A Cognitive Dimensions questionnaire optimised for users. In A.F. Blackwell and E. Bilotta (Eds.) *Proceedings of 12th Annual Workshop of the Psychology of Programming Interest Group*:137 - 154. Available at http://www.ppig.org/papers/12th-blackwell.pdf.

Bloom, B.S., Englehatt, M.D., Furst, E.J., Hill, W.H. and Kathwohl, D.R. (1956): *Taxonomy of Educational Objectives: Handbook I: Cognitive Domain*. Longmans, Green and Company.

Bonar, J. and Liffick, B.W. (1990): A visual programming language for novices. In *Principles of Visual Programming Systems*. S.K. Chang (Ed.), Prentice-Hall.

Bonar, J. and Soloway, E. (1983): Uncovering Principles of Novice Programming. In *Proceedings of 10th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*:10 - 13.

Borland (2000): Delphi Enterprise Ver. 5.

Borland (2003): Delphi Enterprise Ver. 6.

Boyle, R., Carter, J. and Clark, M. (2002): What Makes Them Succeed? Entry, progression and graduation in Computer Science. *Journal of Further and Higher Education*, **26**(1):3 - 18.

Brown, D. (2001a): *B#: A visual programming tool*. Honours treatise. Department of Computer Science and Information Systems, University of Port Elizabeth. Port Elizabeth, South Africa.

Brown, D. (2001b): B#: A Visual Programming Tool Ver. 1.0.

Brusilovsky, P., Kouchnirenko, A., Miller, P. and Tomek, I. (1994): Teaching programming to novices: a review of approaches and tools. In *Proceedings of ED-MEDIA 94-World Conference on Educational Multimedia and Hypermedia*, Vancouver, British Columbia, Canada.

Buck, D. and Stucki, D. J. (2001): JKarelRobot: A Case Study in Supporting Levels of Cognitive Development in the Computer Science Curriculum. *ACM SIGCSE Bulletin*, **33**(1):16 - 20.

Budd, T.A. and Pandey, R. K. (1995): Never Mind the Paradigm, What About Multiparadigm Languages? *ACM SIGCSE Bulletin*, **27**(2):25 - 30.

Burnett, M. M., Baker, M., Bohus, C., Carlson, P., Yang, S. and Van Zee, P. (1995): Scaling up visual programming languages. *IEEE Computer*.

Burnett, M. M. and Baker, M. J. (1994): *A Classification System for Visual Programming Languages*. Technical Report 93-60-14. Department of Computer Science, Oregon State University, Corvallis.

Burrell, C. (2003): Observing Novice Programmer Skill Development: A Micro-world that supports Object-oriented Programming and Usage Data Visualisation. In *Proceedings of 3rd International Conference on Science, Mathematics and Technology Education*, East London, South Africa.

Butcher, D. F. and Muth, W. A. (1985): Predicting performance in an introductory computer science course. *Communications of the ACM*, **28**(3):263 - 268.

Byrne, P. and Lyons, G. (2001): The effect of student attributes on success in programming. *ACM SIGCSE Bulletin*, **33**(3):49 - 52.

Calitz, A. P. (1984): *Evaluation of the Aptitudes of Prospective Computer Science Students with a view to the development of a Computer Aided Testing Program*. Masters Dissertation. Department of Computer Science and Information Systems, University of Port Elizabeth. Port Elizabeth, South Africa.

Calitz, A. P. (1997): *The Development and Evaluation of a Strategy for the Selection of Computer Science Students at the University of Port Elizabeth*. Doctoral Thesis. Department of Computer Science and Information Systems, University of Port Elizabeth. Port Elizabeth, South Africa.

Calitz, A. P., de Kock, G. de V. and Venter, D. J. L. (1992): Selection Criteria for First Year Computer Science Students. *South African Computer Journal*, **8**:4 - 11.

Calitz, A. P., Watson, M. B. and de Kock, G. de V. (1997): Identification and Selection of Successful Future IT Personnel in a Changing Technological and Business Environment. In *Proceedings of SIGCPR*, San Fransisco, USA.

Calloni, B. A. (1998): BACCII$^{++}$ Ver. 1.50.

Calloni, B. A. (2002): Biography. Available at http://www.stc-online.org/stc2002proceedings/spkr965.cfm or http://comstar.csusa.net/~baccii/biography.html. [Accessed on 17 June 2004].

Calloni, B. A. and Bagert, D. J. (1994): Iconic Programming in BACCII© vs Textual Programming: Which is a Better Learning Environment? *ACM SIGCSE Bulletin*, **26**(1):188 - 192.

Calloni, B. A. and Bagert, D. J. (1995): *Iconic Programming for Teaching the First Year Programming Sequence* [online]. Available at http://fie.engrng.pitt.edu/fie95/2a5/2a53/2a53.htm. [Accessed on 26 September 2002].

Calloni, B. A. and Bagert, D. J. (1997): Iconic Programming proves Effective for Teaching First Year Programming Sequence. *ACM SIGCSE Bulletin*, **28**(1):262 - 266.

Campbell, P. F. and McCabe, G. P. (1984): Predicting the success of freshmen in a computer science major. *Communications of the ACM*, **27**(11):1108 - 1113.

Cant, S. N., Jeffery, D. R. and Henderson-Sellers, B. (1995): A Conceptual Model of Cognitive Complexity of Elements of the Programming Process. *Information and Software Technology*, **37**(7):351 - 362.

Carbone, A., Hurst, J., Mitchell, I. and Gunctone, D. (2001): Characteristics of Programming Exercises that lead to Poor Learning Tendencies: Part II. *ACM SIGCSE Bulletin*, **33**(3):93 - 96.

Cardellini, L. (2002): An Interview with Richard M. Felder. *Journal of Science Education*, **3**(2):62 - 65.

Carlisle, M.C., Wilson, T.A., Humphries, J.W. and Hadfield, S.M (2004): RAPTOR: Introducing Programming to Non-Majors with Flowcharts. In *Proceedings of ACM SIGCSE 35th Technical Symposium on Computer Science Education*. Available at http://www.usafa.af.mil/dfcs/bios/mcc_html/raptor_paper.doc.

Carter, J. and Jenkins, T. (2001): Gender differences in programming? *ACM SIGCSE Bulletin*, **33**(3):173.

CC (2001): *Computing Curriculum 2001*. Available at http://www.acm.org/sigs/sigcse/education/cc2001/cc2001.pdf.

Chamillard, A. T. and Braun, K. A. (2000): Evaluating Programming Ability in an Introductory Computer Science Course. *ACM SIGCSE Bulletin*, **32**(1):212 - 216.

Chamillard, A.T., Moore, J. A. and Gibson, D. S. (2002): Using Graphics in an Introductory Computer Science Course. *Journal of Computer Science Education Online*. Available at http://www.iste.org/sigcs/community/jcseonline/2002/2/chamillard.html.

Chang, B.-W. (1995): Getting Close to Objects. In *Visual Object-Oriented Programming Concepts and Environments*:186. M.M. Burnett, A. Goldberg and T. Lewis (Eds.).

Chang, S. K., Burnett, M. M., Levialdi, S., Marriott, K., Pfeiffer, J. J. and Tanimoto, S. L. (1999): The Future of Visual Languages. In *Proceedings of IEEE Symposium on Visual Languages*:58 - 61.

Chang, S. K., Polese, G., Orefice, S. and Tucci, M. (1994): A Methodology and Interactive Environment for Iconic Language Design. *International Journal of Human-Computer Studies,* **41**(5):683 - 716.

Chattratichart, J. and Kuljis, J. (2000): An Assessment of Visual Representations for the 'Flow of Control'. In A.F. Blackwell and E. Billotta (Eds.) *Proceedings of 12th Workshop of the Psychology of Programming Interest Group*:45 - 60. Available at http://www.ppig.org/papers/12th-chattratichart.pdf.

Chattratichart, J. and Kuljis, J. (2002): Exploring the Effect of Control-Flow and Traversal Direction on VPL Usability for Novices. *Journal of Visual Languages and Computing*, **13**:471 - 500.

Chen, H. and Vecchio, R. P. (1992): Nested IF-THEN-ELSE constructs in end-user computing: Personality and aptitude as predictors of programming ability. *International Journal of Man-Machine Studies*, **36**:843 - 859.

Chmura, G. A. (1998): What abilities are necessary for success in Computer Science? *ACM SIGCSE Bulletin*, **30**(4):55a - 58a.

Christensen, K., Davis, D. and Rundus, D. (2000): A "Boot Camp" for Preparing Freshman Students for Success in a First Course in Computing. In *Proceedings of ASEE Southeastern Section Conference*.

Christians, D. N. (2003): *A Simple IDE for Delphi*. Honours treatise. Department of Computer Science and Information Systems, University of Port Elizabeth. Port Elizabeth, South Africa.

Cilliers, C. B., Calitz, A. P. and Greyling, J. H. (2003): The Development and Evaluation of Introductory Programming Tools. In *Proceedings of 3rd International Conference on Science Mathematics and Technology Education,* East London, South Africa.

Cilliers, C. B., Calitz, A. P. and Greyling, J. H. (2004a): The Application of the Cognitive Dimensions Framework for Notations as an Instrument for the Usability Analysis of an Introductory Programming Tool. *Submitted for publication in a special edition of Alternation Journal on e-Learning*.

Cilliers, C. B., Calitz, A. P. and Greyling, J. H. (2004b): B# : A Programming Tool for Novice Programmers. Scientific Address at Joint IFIP TC.13 and CHI-SA Workshop, University of Port Elizabeth, Port Elizabeth, South Africa. Available at http://www.chi-sa.co.za.

Cilliers, C. B. and Vogts, D. (2002): Initial Experiences of Delivery Methods in a First Year Programming Course. In *Proceedings of Southern African Computer Lecturers Association (SACLA) Annual Conference*.

Cockburn, A. and Churcher, N. (1997): Towards Literate Tools for Novice Programmers. In *Proceedings of Australasian Computer Science Education Conference*, Melbourne, Australia:107 - 116. ACM Press. Available at http://www.cosc.canterbury.ac.nz/~andy/papers/acse97LitProg.pdf.

Cooper, S., Dann, W. and Pausch, R. (2000): Alice: A 3-D Tool for Introductory Programming Concepts. *Journal of Computing in Small Colleges*, **15**(5):108 - 117.

Cooper, S., Dann, W. and Pausch, R. (2003): Teaching Objects-First in Introductory Computer Science. In *Proceedings of ACM SIGCSE.*

Corritore, C.L. and Wiedenbeck, S. (1991): What do novices learn during program comprehension? *International Journal of Human-Computer Interaction*, **3**(2):199 - 222.

Cranor, L. and Apte, A. (undated): *Programs worth 1000 words: Visual Languages bring programming to the masses* [online]. Available at http://info.acm.org/crossroads/xrds1-2/visual.html. [Accessed on 26 August 2002].

Crews, T. (2001): *Using a Flowchart Simulator in a Introductory Programming Course* [online]. Available at http://www.cstc.org/data/resources/213/Visual.pdf. [Accessed on 12 March 2003].

Crews, T. (2003): *Visual Logic* [online]. Available at http://cis1.wku.edu/flint/VisualLogicFall03Demo.zip. [Accessed on 16 August 2003].

Crews, T. and Butterfield, J. (2002): Using Technology to Bring Abstract Concepts into Focus: A Programming Case Study. *Journal of Computing in Higher Education*, **13**(2):25 - 50.

Crews, T. and Ziegler, U. (1998): The Flowchart Interpreter for Introductory Programming Courses. In *Proceedings of 28th Annual Frontiers in Education Conference*, **1**:307 - 312. Available at http://fie.engrng.pitt.edu/fie98/papers/1107.pdf.

CS&IS (2003): *WRA101 (Algorithmics 1.1) Module Guide*. Department of Computer Science and Information Systems, University of Port Elizabeth. Port Elizabeth, South Africa.

Curtis, B. (1981): A Review of Human Factors Research on Programming Languages and Specifications. *ACM Press*:212 - 218.

Dagdilelis, V., Satratzemi, M. and Evangelidis, G. (2002): What they really do? Attempting (once again) to model novice programmers' behavior. *ACM SIGCSE Bulletin*, **34**(3):244.

Dagiano, C., Kahn, K., Cypher, A. and Smith, D. C. (2001): Integrating Learning Supports into the Design of Visual Programming Systems. *Journal of Visual Languages and Computing*, **12**:501 - 524.

Dale, N. (1998): Two Threads from the Empirical Studies of Programmers. *ACM SIGCSE Bulletin*, **30**(4):16a - 17a.

Daly, C. and Waldron, J. (2004): Assessing the assessment of programming ability. In *Proceedings of 35th SIGCSE Technical Symposium on Computer Science Education*:210 - 213.

Danial, S. E. (1985): *Relationships between mathematical ability, attentional and interpersonal style and achievement in a problem-solving oriented computer programming course*. Doctoral thesis. University of Pittsburgh.

Darius, R. (1987): *A validity estimate of a computer aptitude test as it differentiates while controlling for age, sex and ACT scores*. Doctoral thesis. University of Akron.

Davies, S. P. (1990): The nature and development of programming plans. *International Journal of Man-Machine Studies*, **32**:461 - 481.

De Jager, S. (2004): *SimplifIDE*. Honours treatise. Department of Computer Science and Information Systems, University of Port Elizabeth. Port Elizabeth, South Africa.

de Kock, G. de V. (1993): *Implementation of selection model for First Year Programming Students*. Internal departmental correspondence. Department of Computer Science and Information Systems, University of Port Elizabeth. Port Elizabeth, South Africa.

De Koning, K., Bredeweg, B., Breuker, J. and Wielinga, B. (2000): Model-based reasoning about learner behaviour. *Artificial Intelligence*, **117**:173 - 229.

De Raadt, M., Watson, R. and Toleman, M. (2002): Language Trends in Introductory Programming Courses. In *Proceedings of Informing Science + Information Technology Education Joint Conference (I$^n$SITE)*:329 - 337.

De Roure, D., Maclean, S. D. and Glaser, H. (1998): An Extensible Interpreter for Experimentation with the Semantics of Prograph. In *Proceedings of IEEE Symposium on Visual Languages*:76 - 77.

Dee Medley, M. (2001): Using qualitative research software for CS education research. *ACM SIGCSE Bulletin*, **33**(3):141 - 144.

Deek, F P (1999): A Framework for an Automated Problem Solving and Program Development Environment. *Transactions of the Society for Design and Process Science*, **3**(3):1 - 13.

Deek, F. P. and McHugh, J. A. (2002): An Empirical Evaluation of Specification Oriented Language in Visual Environment for Instruction Translation (SOLVEIT): A Problem-Solving and Program Development Environment. *Journal of Interactive Learning Research*, **13**(4):339 - 373. Available at http://www.aace.org/dl/files/JILR/JILR134339.pdf.

Denning, P. J. (1996): The University's Next Challenges. *Communications of the ACM*, **39**(5):27 - 31.

Department of Education (2001): *National Plan for Higher Education*. South African Department of Education. Available at http://education.pwv.gov.za/DoE_Sites/Higher_Education/HE_Plan/section_2.htm.

Dillon, L. K., Kutty, G., Melliar-Smith, P. M., Moser, L. E. and Ramakrishna, Y. S. (1994): Visual specifications for temporal reasoning. *Journal of Visual Languages and Computing*, **5**(1).

Dingle, A. and Zander, C. (2001): Assessing the ripple effect of CS1 language choice. *Journal of Computing in Small Colleges*, **16**(2):85 - 93.

Donaldson, T. (2003): Python as a First Programming Language for Everyone. In *Proceedings of World Conference for Computer Education*. Available at http://www.cs.ubc.ca/wccce/Program03/papers/Toby.html.

Downes, S. (2002): Into the New Millenium: Why do Students Decide to study IT? In *Proceedings of Informing Science + Information Technology Education Joint Conference (I$^n$SITE)*:371 - 372.

Du Toit, L. (1983): Manual for the Survey of Study Habits and Attitudes. Pretoria, South Africa.

DuHadway, L. P., Clyde, S. W., Recker, M. M. and Cooley, D. H. (2002): A Concept-first approach for an Introductory Computer Science Course. *Journal of Computing in Small Colleges*, **18**(2):6 - 16.

Eiselen, R. and Geyser, H. (2003): Factors distinguishing between Achievers and At Risk Students: a qualitative and quantitative synthesis. *South African Journal of Higher Education*, **17**(2):118 - 130.

Ely, M., Anzul, M., Friedman, T., Garner, D. and MacCormack Steinmetz, A. (1995): *Doing Qualitative Research: Circles Within Circles*. Falmer Press.

Ely, M., Vinz, R., Downing, M. and Anzul, M. (1999): *On Writing Qualitative Research: Living by Words*. Falmer Press.

Entwisle, N. (1998): Motivation and Approaches to Learning: Motivating and Conceptions of Teaching. In *Motivating Students*. S. Brown (Ed.), Kogan Page.

Evans, G. E. and Simkin, M. G. (1989): What Best Predicts Computer Proficiency? *Communications of the ACM*, **32**(11):1322 - 1327.

Felder, R.M. (1993): Reaching the Second Tier: Learning and Teaching Styles in College Science Education. *Journal of College Science Teaching*, **23**(5):286 - 290. Available at http://www.ncsu.edu/felder-public/Papers/Secondtier.html.

Felder, R.M. (2002): *Author's Preface to "Learning and Teaching Styles in Engineering Education"* [online]. Available at http://www.ncsu.edu/felder-public/Papers/. [Accessed on 8 December 2003].

Fergusson, K. (2002): *Anti Compiler: An Educational Tool in First Year Programming*. Honours Proposal. Computer Science and Software Engineering, Monash University. Clayton, Australia.

Fincher, S. (1999): What are We Doing When We Teach Programming? In *Proceedings of Frontiers in Education*:12a4-1 to 12a4-5. IEEE Press.

Fix, V., Wiedenbeck, S. and Scholtz, J. (1993): Mental Representations of programs by Novices and Experts. In *Proceedings of Conference on Human Factors in Computing Systems*:74 - 79.

Fowler, G. C. and Glorfield, L. W. (1981): Predicting aptitude in introductory computing: A classification model. *Association for Educational Data Systems Journal*, **14**(2):96 - 109.

Fowler, L., Allen, M., Armarego, J. and Mackenzie, J. (2000): Learning Styles and CASE tools in Software Engineering. In A. Herrmann and M.M. Kulski (Eds.) *Proceedings of 9th Annual Teaching Learning Forum*. Available at http://ccea.curtin.edu.au/tlf/tlf2000/fowler.html.

Foxcroft, C. D. (1997): Establishment of Task Team to investigate Admission Criteria and the Development of an Admissions Test/Entrance Exam. Internal memo, University of Port Elizabeth. Port Elizabeth, South Africa.

Foxcroft, C., Koch, E. and Watson, A. (1999): A developmental focus to student access at UPE: Process and preliminary insights. In *Proceedings of Selection and Admissions Seminar*, Port Elizabeth, South Africa.

Freeman, E., Jagannathan, S. and Gelernter, D. (1996): *In Search of a Simple Visual Vocabulary* [online]. Available at http://www.computer.org/conferences/vl95/html-papers/freeman2/VL.html#MAPprogram. [Accessed on 12 August 2002].

Gamieldien, M. R. (2003): *Activity Logger*. Honours treatise. Department of Computer Science and Information Systems, University of Port Elizabeth. Port Elizabeth, South Africa.

Garner, S. (2000): A Code Restructuring Tool to help Scaffold Novice Programmers. In *Proceedings of International Conference in Computer Education (ICCE)*.

Garner, S. (2001): Cognitive Load Reduction in Problem Solving Domains. In *Proceedings of International Conference in Computer Education (ICCE)*.

Garner, S. (2002): COLORS for Programming: A System to Support the Learning Environment. In *Proceedings of Informing Science and Information Technology Education Joint Conference ($I^n$SITE)*:533 - 542. Available at http://ecommerce.lebow.drexel.edu/eli/2002Proceedings/papers/Garne069COLOR.pdf.

Garner, S. (2003): Learning Resources and Tools to Aid Novices Learn Programming. In *Proceedings of Informing Science + Information Technology Education Joint Conference ($I^n$SITE)*:213 - 222.

George, C. E. (2000): Experiences with Novices: The Importance of Graphical Representations in Supporting Mental Models. In A.F. Blackwell and E. Billotta (Eds.) *Proceedings of 12th Workshop of the Psychology of Programming Interest Group*:33 - 44. Available at http://www.ppig.org/papers/12th-george.pdf.

Ghittori, E., Mosconi, M. and Porta, M. (1998): *Designing and Testing new Programming Constructs in a Data Flow VL* [online]. Available at http://iride.unipv.it/research/papers/98r-dataflow/usabilit.htm. [Accessed on 8 August 2002].

Gibbs, D.C. (2002): A Case Study: An Interactive Introductory Programming Environment using VBScript. In *Proceedings of World Conference on Educational Multimedia, Hypermedia and Telecommunications (ED-MEDIA)*. Available at http://www.uwsp.edu/cis/dgibbs/Conference/EDMEDIA2002/fullpaper.doc.

Gilmore, D. J. and Green, T. R. G. (1984): Comprehension and recall of miniature programs. *International Journal of Man-Machine Studies,* **21**:31 - 48.

Ginat, D. (2001): Metacognitive awareness utilized for learning control elements in algorithmic problem solving. *ACM SIGCSE Bulletin*, **33**(3):81 - 84.

Glenn, I. (2000): The Learning Curve, in *The Money Standard (Standard Bank Publication, South Africa)*:42 - 45.

Goldstein, G. (1987): *The Computer Programmer Aptitude Battery as a Predictor of Achievement in FORTRAN Computer Programming Courses at the Two-year Community College Level*. Doctoral thesis. New York University.

Good, J. A. (1999): VPLs and Novice Program Comprehension: How do Different Languages Compare? In *Proceedings of IEEE Symposium on Visual Languages*.

Good, J. and Brna, P. (1999): Getting a GRiP on the Comprehension of Data-Flow Visual Programming Languages. In T. Green, R. Abdullah and P. Brna (Eds.) *Proceedings of 11th Annual Workshop of the Psychology of Programming Interest Group*. Available at http://www.ppig.org/papers/11th-good.pdf.

Goold, A. and Rimmer, R. (2000): Factors Affecting Performance in First-year Computing. *ACM SIGCSE Bulletin*, **32**(2):39 - 43.

Green, T. R. G. (1989): The cognitive dimensions of notations. In *People and Computers V*:443 - 460. A. Sutcliffe and L. Macaulay (Eds.) Cambridge, Cambridge University Press.

Green, T. R. G. (2000): Instructions and Descriptions: some cognitive aspects of programming and similar activities. In V. Di Gesù, S. Levialdi and L. Tarantino (Eds.) *Proceedings of Working Conference on Advanced Visual Interfaces,* New York:21 - 28. ACM Press. Available at http://www.ndirect.co.uk/~thomas.green/workStuff/Papers/AVI2000.pdf.

Green, T. R. G. and Blackwell, A. F. (1998): *Cognitive Dimensions of Information Artefacts: a tutorial*. Available at http://www.ndirect.co.uk/~thomas.green/workStuff/Papers/.

Green, T.R.G. (1997): *Cognitive Approaches to Software Comprehension: Results, Gaps and Limitations* [online]. Available at http://www.ndirect.co.uk/~thomas,green/workStuff/Papers/LimerickTalk1997/LimerickTalk.html. [Accessed on 28 August 2002].

Green, T.R.G. and Blackwell, A. F. (1996): *Thinking about Visual Programs* [online]. Available at http://cui.unige.ch/eao/www/Visual/local/GreenBlackwell96.html. [Accessed on 14 August 2002].

Green, T.R.G. and Petre, M. (1993): Learning to Read Graphics: Some evidence that "seeing" an Information Display is an Acquired Skill. *Journal of Visual Languages and Computing*, **4**:55 - 70.

Green, T.R.G. and Petre, M. (1996): Usability Analysis of Visual Programming Environments: A 'Cognitive Dimensions' Framework. *Journal of Visual Languages and Computing*, **7**:131 - 174. Available at http://www.idealibrary.com/links/doi/10.1006/jvlc.1996.0009.

Green, T.R.G., Petre, M. and Bellamy, R.K.E. (1991): Comprehensibility of Visual and textual programs: A Test of Superlativism against the 'Match-Mismatch' Conjecture. In *Proceedings of 4th Workshop of Empirical Studies of Programmers*:121 - 146.

Greyling, J. H. (2000): *The Compilation and Validation of a Computerised Selection Battery for Computer Science and Information Systems Students*. Doctoral Thesis. Department of Computer Science and Information Systems, University of Port Elizabeth. Port Elizabeth, South Africa.

Greyling, J. H. and Calitz, A P (2003): The Implementation of a Computerised Placement Battery for First Year IT Courses. In *Proceedings of 3rd International Conference on Science, Mathematics and Technology Education*, East London, South Africa.

Greyling, J. H., Calitz, A. P. and Watson, M. (2002): The Implementation of a Streaming Model in First Year IT Courses. In *Proceedings of Southern African Computer Lecturers Association (SACLA) Annual Conference*.

Hagan, D. and Markham, S. (2000): Does It Help to Have Some Programming Experience Before Beginning a Computing Degree Program? *ACM SIGCSE Bulletin*, **32**(3):25 - 28.

Haliburton, W. (1998): Gender Differences in Personality Components of Computer Science Students: A Test of Holland's Congruence Hypothesis. *ACM SIGCSE Bulletin*, **30**(1):77 - 81.

Hansen, W.J., Bell, B., MacKaskle, G.A., Smedley, G., Kimura, D. and Poswig, J. (1994): *The 1994 Visual Languages Comparison* [online]. Available at http://www-cgi.cs.cmu.edu/~wjh/papers/bakeoff.html. [Accessed on 26 August 2002].

Harman, G. (1994): Student selection and admission to higher education: policies and practices in the Asian region. *International Journal of Higher Education*, **27**(3):313 - 339.

Hendrix, T. D., Cross, J. H., Barowski, L. A. and Mathias, K. S. (1998): Visual Support for Incremental Abstraction and Refinement in Ada 95. In *Proceedings of SIGAda '98*, Washington, D.C.

Hennefeld, J. and Burchard, C. (1998): *Using C++: An Introduction to Programming*. PWS Publishing Company.

Henning, J. (2004): *A Visual Code Reorganisation Tool*. Honours treatise. Department of Computer Science and Information Systems, University of Port Elizabeth. Port Elizabeth, South Africa.

Hickey, T. J. (2004): Scheme-based Web Programming as a basis for a CS0 Curriculum. In *Proceedings of 35th SIGCSE Technical Symposium on Computer Science Education*. Available at http://www.cs.brandeis.edu/~tim/Papers/sigces-cs0.pdf.

Hilburn, T. B. (1993): A Top-Down Approach to Teaching an Introductory Computer Science Course. *ACM SIGCSE Bulletin*, **25**(1):58 - 62.

Hils, D. D. (1999): DataVis: A Visual Programming Language for Scientific Visualization. In *Proceedings of 19th Annual Conference on Computer Science*:439 - 448.

Holland, J. L. (1985): *Making vocational choices: A theory of vocational personalities and work environments*. 2nd Edn, Prentice Hall, Englewood Cliffs, New Jersey.

Howe, D. (2003): *Free On-line Dictionary of Computing (FOLDOC)* [online]. Available at http://wombat.doc.ic.ac.uk/foldoc/. [Accessed on 1 October 2003].

Howell, K. (1993): The Experience of Women in Undergraduate Computer Science: What does the Research Say? *ACM SIGCSE Bulletin*, **25**(2):1 - 7.

Howell, K. (2003): First Computer Languages. *Journal of Computing in Small Colleges*, **18**(4):317 - 331.

Hunt, A. (1998): *An investigation into the Computerisation of the pen and paper version of the Logo Test*. Honours Treatise. Department of Computer Science and Information Systems, University of Port Elizabeth. Port Elizabeth, South Africa.

Huysamen, G. K. (1997): Potential ramifications of admissions testing at South African institutions of higher education. *South African Journal of Higher Education*, **11**(2):65 - 71.

Huysamen, G. K. and Roozendaal, L. A. (1999): Curricular choice and the differential prediction of the tertiary-academic performance of men and women. *South African Journal of Psychology*, **29**(2):87 - 93.

Irons, D. M. (1981): Cognitive Correlates of Programming Tasks in Novice Programmers. *ACM Press*:219 - 222.

Jamal, R. and Ronpage, T. (1996): *Rapid Prototyping Using Graphical Programming Techniques for Real World Applications* [online]. Available at http://conference.kek.jp/PCaPAC99/cdrom/pcapac96/3POSTER/B/B01.doc. [Accessed on 13 February 2003].

Jenkins, T. (2001a): The motivation of students of programming. *ACM SIGCSE Bulletin*, **33**(3):53 - 56.

Jenkins, T. (2001b): Teaching Programming - A Journey from Teacher to Motivator. In *Proceedings of 2nd Annual LTSN-ICS Conference*. Available at http://www.ics.ltsn.ac.uk/pub/conf2001/papers/Jenkins%20paper.pdf.

Jenkins, T. (2002): *On the Difficulty of Learning to Program* [online]. Available at http://www.ics.ltsn.ac.uk/pub/conf2002/jenkins.html or http://www.psy.gla.ac.uk/~steve/localed/jenkins.html. [Accessed on 13 February 2003].

Jenkins, T. (undated): Motivation = Value x Expectancy. *ACM Press*:174.

Jonassen, D. H. (2000): *Toward a Meta-Theory of Problem Solving* [online]. Available at http://tiger.coe.missouri.edu/~jonassen/Design%20Theory.pdf. [Accessed on 13 February 2003].

Kaasbøll, J.J (1998): *Exploring didactic models for programming* [online]. Available at http://www.ifi.uio.no/~jensj/NIK98.pdf. [Accessed on 10 February 2003].

Kerman, M.C. (2002): *Programming and Probem Solving with Delphi™*. Addison Wesley Longman. ISBN 0-201-70844-2.

Kernighan, B.W. and Plauger, P.J. (1974): Programming Style. In *Proceedings of 4th SIGCSE Technical Symposium on Computer Science Education*:90 - 96.

Kimmel, H., Deek, F.P. and O'Shea, M. (1999): *Teaching, Learning, and Technology: Is there a parallel?* [online]. Available at http://www.csun.edu/cod/conf/1999/session0232.htm. [Accessed on 10 February 2003].

Ko, A.J. (2003a): *Preserving Non-Programmers' Motivation with Error-Prevention and Debugging Support Tools* [online]. Available at http://www-2.cs.cmu.edu/~ajko/HCC2003TravelGrant.pdf. [Accessed on 26 August 2003].

Ko, A.J. (2003b): *Project Marmalade* [online]. Available at http://www-2.cs.cmu.edu/~NatProg/marmalade.html. [Accessed on 26 August 2003].

Koelma, D., Van Balen, R. and Smeulders, A. (1992): SCIL-VP: a multi-purpose visual programming environment. *ACM SIGAPP*:1199 - 1198.

Kolling, M. and Rosenberg, J. (2001): Guidelines for Teaching Object Orientation with Java. *ACM SIGCSE Bulletin*, **33**(3):33 - 36.

Konvalina, J., Stephens, L. and Wileman, S. (1983): Identifying Factors Influencing Computer Science Aptitude and Achievement. *Association for Educational Data Systems Journal*, **16**:106 - 112.

Koubek, R. J., LeBold, W. K. and Salvendy, G. (1985): Predicting performance in computer programming courses. *Behaviour and Information Technology*, **4**(2):113 - 129.

Kushan, B (1994): Preparing Programming Teachers. *ACM SIGCSE Bulletin*.

Kutar, M., Britton, C. and Wilson, J. (2000): Cognitive Dimensions: An experience report. In A.F. Blackwell and E. Bilotta (Eds.) *Proceedings of 12th Annual Workshop of the Psychology of Programming Interest Group*:81 - 98. Available at http://www.ppig.org/papers/12th-kutar.pdf.

LaLiberte, D. (1994): *Visual Languages* [online]. Available at http://www.hypernews.org/~liberte/computing/visual.html. [Accessed on 8 August 2002].

Lane, H. C. (2002): Eliciting Pseudocode in Novice Program Design. In *Proceedings of 33rd ACM Technical Symposium on Computer Science Education*. Available at http://www.cs.pitt.edu/~hcl/pubs/sigcsedc02.html.

Lane, H. C. (2003): Development of an Intelligent Tutoring System for Novice Program Design. In *Proceedings of 34th ACM Technical Symposium on Computer Science Education*. Available at http://www.cs.pitt.edu/~hcl/pubs/sigcsedc03.html.

Lane, H. C. (2004): A Preventive Tutoring System for Beginning Programming. In *Proceedings of 35th ACM Technical Symposium on Computer Science Education*. Available at http://www.cs.pitt.edu/~hcl/pubs/sigcsedc04.html.

Lane, H. C. and VanLehn, K. (2003): Coached Program Planning: Dialogue-Based Support for Novice Program Design. In *Proceedings of 34th SIGCSE Technical Symposium on Computer Science Education*:148 - 152. Available at http://www.cs.pitt.edu/~hcl/pubs/cpp-sigcse03.pdf.

Lane, H. C. and VanLehn, K. (2004a): A Dialogue-Based Tutoring System for Beginning Programming. In *Proceedings of 17th International FLAIRS Conference*. Available at http://www.cs.pitt.edu/%7Ehcl/pubs/FLAIRS04LaneH.pdf.

Lane, H. C. and VanLehn, K. (2004b): ProPl : A Dialogue-based Intelligent Tutoring System.

Langley, R. (1992a): Manual : The Career Development Questionnaire. Pretoria, South Africa.

Langley, R. (1992b): Manual: The Life Roles Inventory. Pretoria, South Africa.

Langley, R., Du Toit, R. and Herbst, D. L. (1992): Manual for the Value Scale. Pretoria, South Africa.

Larkin, J. H. and Simon, H. A. (1987): Why a diagram is (sometimes) worth ten thousand words. *Cognitive Science*, **11**:65 - 99.

Larson, H. J. (1974): *Introduction to Probability Theory and Statistical Inference*. 2nd Edn, Wiley International Edition. ISBN 0-471-51791-7.

Lavonen, J. M., Meisalo, V. P., Lattu, M. and Sutinen, E. (2003): Concretising the programming task: a case study in a secondary school. *Journal of Computers and Education*, **40**:115 - 135.

Leppan, R. (2004): *A Comparison of Student Learning Style Usage Patterns in a 4MATted Online Tutorial*. Unpublished Masters dissertation. Department of Computer Science and Information Systems, University of Port Elizabeth. Port Elizabeth, South Africa.

Lidtke, D. K. and Zhou, H. H. (1998): A Top-Down Collaborative Teaching Approach to Introductory Courses in Computer Sciences. *ACM SIGCSE Bulletin*, **30**(3):291.

Liffick, B.W. and Aiken, R. (1996): A novice programmer's support environment. *ACM SIGCSE Bulletin*, **28**(SI):49 - 51.

Lischner, R. (2001): Explorations: Structured Labs for First-Time Programmers. *ACM SIGCSE Bulletin*, **33**(1):154 - 158.

Lister, R. (2000): On Blooming First Year Programming, and its Blooming Assessment. In *Proceedings of Australasian Conference on Computing Education*:158 - 162.

Lister, R. and Leaney, J. (2003): Introductory Programming, Criterion-Referencing, and Bloom. In *Proceedings of 34th SIGCSE Technical Symposium on Computer Science Education*:143 - 147.

Lockard, J. (1986): Computer programming in Schools: What Should Be Taught. *Computers in the Schools*.

Loftin, R. D. (1987): *The evaluation of selected variables as predictors of achievement in computer programming*. Doctoral thesis. Northwestern State University of Louisiana.

Lord, H. D. (1994): Visual programming for visual applications: a new look for computing, in *Object Magazine*, **4**.

Lourens, A. and Smit, I.P.J. (2003): Retention: predicting first-year success. *South African Journal of Higher Education*, **17**(2):169 - 176.

Loxton, L. (1997): Training vital to keeping up in the IT, in *Weekly Mail and Guardian* of 25 July 1997.

Lyons, P. (undated): *Visual Programming Languages Research Papers* [online]. Available at http://www-ist.massey.ac.nz/~plyons/711_html/VPL%20papers.html. [Accessed on 8 August 2002].

Lyons, P., Simmons, C. and Apperley, M. (1993): HyperPascal: A Visual Language to Model Idea Space. In *Proceedings of 13th New Zealand Computer Society Conference*:492 - 508.

Mahmoud, Q., Dobosiewicz, W. and Swayne, D. (2004): Redesigning introductory computer programming with HTML, JavaScript, and Java. In *Proceedings of 35th SIGCSE Technical Symposium on Computer Science Education*:120 -124.

Mamtani, B. (2004): *Web-based Presentation System for Programming Logic Information Flow*. Honours treatise. Department of Computer Science and Information Systems, University of Port Elizabeth. Port Elizabeth, South Africa.

Maryland (2001): *The "Degree Navigator" Nightmare: Taming an Overly Graphical User Interface (Student HCI Online Research Experiments)* [online]. Available at http://www.otal.umd.edu/SHORE2001/degreeNav/page1.html. [Accessed on 16 August 2002].

Materson, T. F. and Meyer, R. M. (2001): SIVIL: A True Visual Programming Language for Students. *Journal of Computing in Small Colleges*, **16**(4):74 - 86.

Mattson, T. (undated-a): *A Cognitive Model for Programming* [online]. Available at http://www.cise.ufl.edu/research/ParallelPatterns/PatternLanguage/Background/Psychology/CognitiveModel.htm. [Accessed on 12 August 2002].

Mattson, T. (undated-b): *Design Rules from the Psychology of Programming* [online]. Available at http://www.cise.ufl.edu/research/ParallelPatterns/PatternLanguage/Background/Psychology/DesignRules.htm. [Accessed on 12 August 2002].

Mayer, R. E (1981): The Psychology of How Novices Learn Computer Programming. *ACM Computing Surveys*, **13**(1):121 - 141.

Mayer, R. E, Dyck, J. L. and Vilberg, W. (1986): Learning to Program and Learning to Think: What's the Connection? *Communications of the ACM*, **29**(7):605 - 610.

Mazlack, L. J. (1980): Identifying Potential to Acquire Programming Skill. *Communications of the ACM*, **23**(1):14 - 17.

McCracken, M., *et al.* (2001): A multi-national, multi-institutional study of assessment of programming skills of first-year CS students. *ACM SIGCSE Bulletin*, **33**(4):125 - 140.

McDonald, A. S. (2002): The impact of individual differences on the equivalence of computer-based and paper-and-pencil educational assessments. *Computers and Education*, **39**:299 - 312.

McGettrick, A.D. and Smith, P. D. (1983): *Graded Problems in Computer Science*. Addison-Wesley Publishers Limited. ISBN 0-201-13787-9.

M$^c$Iver, L. (2000): The Effect of Programming Language on Error Rates of Novice Programmers. In A.F. Blackwell and E. Bilotta (Eds.) *Proceedings of 12th Workshop of the Psychology of Programming Interest Group*:181 - 192.

M$^c$Iver, L. (2001): *Syntactic and Semantic Issues in Introductory Programming Education*. Doctoral Thesis. School of Computer Science and Software Engineering, Monash University. Australia.

M$^c$Iver, L. (2002): Evaluating Languages and Environments for Novice Programmers. In J. Kuljis, L. Baldwin and R. Scoble (Eds.) *Proceedings of 14th Workshop of the Psychology of Programming Interest Group*. Available at http:www.ppig/org/papers/14th-mciver.pdf.

M$^c$Iver, L. and Conway, D. (1999): GRAIL: A Zeroth Programming Language. In *Proceedings of 7th International Conference on Computers in Education*, **2**:43 - 50.

McKenna, P. (2000): Transparent and opaque boxes: do women and men have different computer programming psychologies and styles? *Computers and Education*, **35**:37 - 49.

McKinney, A. L. (2003): A Recent Radical Graphical Approach to Programming. *Journal of Computing in Small Colleges*, **18**(6):28 - 35.

Merriam, S. (1998): *Qualitative Research and Case Study Applications*. Jossey-Bass Publishers.

Meyer, R. M. and Masterson, T (2000): Towards a Better Visual Programming Language: Critiquing Prograph's Control structures. *Journal of Circuits, Systems and Computers*, **15**(5):183 - 196.

Microsoft Corporation (2002): Microsoft Excel Ver. (10.4302.4219) SP-2.

Miller, P. A. (2003): *How South African further education and training learners acquire, recall, process and present information in a digitally enabled environment*. Doctoral Thesis. Department of Curriculum Studies, Faculty of Education, University of Pretoria. Pretoria, South Africa.

Misthry, N., Mkhize, P. and Harypursat, R. (2003): The Role of Information Technology and its Impact on Tertiary Education. In *Proceedings of Conference on Information Technology in Tertiary Education*. Available at http://citte.nu.ac.za/papers/id5.pdf.

Moher, T.G., Mak, D.C., Blumenthal, B. and Leventhal, L.M. (1993): Comparing the comprehensibility of textual and graphical programs: The case of petri nets. In *Proceedings of 5th Workshop of Empirical Studies of Programmers*:137 - 161.

Monare, M. (2004): Now matric exams face official probe, in *Sunday Times* of 4 January 2004.

Morrison, M. and Newman, T. S. (2001): A Study of the Impact of Student Background and Preparedness on Outcomes in CS 1. *ACM SIGCSE Bulletin*, **33(**1):179 - 183.

Mosconi, M. and Porta, M. (2000): Iteration constructs in data-flow visual programming languages. *Journal of Computer Languages*, **26**:67 - 104.

Mostert, J. W. (2002): Pre registration assessment. In *Proceedings of 2nd Pan Commonwealth Forum*, Durban, South Africa. Available at http://www.col.org/pcf2/papers%5Cmostert.pdf.

Mouton, J. (2001): *How to succeed in your Master's & Doctoral Studies: A South African Guide and Resource Book.* Pretoria, South Africa, Van Schaik Publishers.

Nash, J. (2003): *The association of matriculation English scores with the performance of Information Systems majors at the University of Cape Town*. Masters dissertation. Department of Information Systems, University of Cape Town, South Africa.

Naudé, E. and Hörne, T. (2003): Investigating Suggestions for Increasing the Throughput of IS Students. In *Proceedings of Informing Science + Information Technology Education Joint Conference ($I^n$SITE)*:923 - 929.

Naudé, K A (2004): *Designing Programming Languages for Novice Programmers*. Unpublished Masters dissertation. Department of Computer Science and Information Systems, University of Port Elizabeth. Port Elizabeth, South Africa.

Navarro-Prieto, R. and Cañas, J. J. (1999): *Mental Representation and Imagery in Program Comprehension* [online]. Available at http://www.cs.vu.nl/~eace/ECCE9/papers/navarro.pdf. [Accessed on 2 September 2002].

Navarro-Prieto, R. and Cañas, J. J. (2001): Are visual programming languages better? The role of imagery in program comprehension. *International Journal of Human-Computer Studies,* **54**:799 - 829.

Newman, T. S., Weisskopf, M. and Morrison, M. (1999): CS 090: The Case for a Developmental Course in Computer Science. *Journal of Teaching and Learning*, **4**(1):15 - 35.

Nielsen, S. H., Von Hellens, L. A., Greenhill, A. and Pringle, R. (1997): Collectivism and Connectivity: Culture and Gender in Information Technology Education. In *Proceedings of ACM SIGCPR Conference*:9 - 13.

Oberlander, J., Brna, P., Cox, R. and Good, J. (1999): *The GRIP Project, or … The Match-Mismatch Conjecture and Learning to Use Data-Flow Visual Programming Languages* [online]. Available at http:///www.cbl.leeds.ac.uk/~paul/grip.html. [Accessed on 12 August 2002].

Oberlander, J., Cox, R. and Good, J. (1999): *HCRC Project: The match-mismatch conjecture and learning to use data-flow visual programming languages* [online]. Available at http://www.hcrc.ed.ac.uk/site/grip.html. [Accessed on 20 August 2002].

Osipow, S. H. (1986): Career Decision Scale: Manual. Odessa, Florida.

Pandey, R. K. and Burnett, M. M. (1993): Is it easier to write matrix manipulation programs visually or textually? An Empirical Study. In *Proceedings of IEEE Symposium on Visual Languages*:344 - 351. IEEE Computer Society Press. Available at http://www.ppig.org/papers/a2.html.

Pane, J. F. and Myers, B. A. (2000): The influence of the Psychology of Programming on a Language Design: Project Status Report. In A.F. Blackwell and E. Billotta (Eds.) *Proceedings of 12th Workshop of the Psychology of Programming Interest Group*:193 - 208.

Pane, J.F., Myers, B. A. and Miller, L. B. (2002): Using HCI Techniques to Design a More Usable Programming System. In *Proceedings of IEEE Symposia on Human Centric Computing Languages and Environments*.

Pane, J.F., Ratanamahatana, C. A. and Myers, B. A. (2001): Studying the language and structure in non-programmers' solutions to programming problems. *International Journal of Human-Computer Studies*, **54**:237 - 264. Available at http://www.cs.cmu.edu/~pane/IJHCS.html.

Perkins, D. N. and Salomon, G. (1988): *Teaching for Transfer* [online]. Available at http://www.lookstein.org/integration/teaching_for_transfer.htm. [Accessed on 13 February 2003].

Petre, M. and Blackwell, A.F. (1999): Mental Imagery in Program Design and Visual Programming. *International Journal of Human-Computer Studies*, **51**(1):7 - 30. Available at http://www.cl.cam.ac.uk/users/afb21/publications/IJHCS.html.

Pictorius (1998): Pictorius Prograph for Windows Ver. 1.2.

Proulx, V. K. (2000): Programing Patterns and Design Patterns in the Introductory Computer Science Course. *ACM SIGCSE Bulletin*, **32**(1):80 - 84.

Proulx, V. K., Rasala, R. and Fell, H. (1996): Foundations of Computer Science: What are they and how do we teach them? *ACM SIGCSE Bulletin*, **28**(S1):42 - 48.

Quaiser-Pohl, C. and Lehmann, W. (2002): Girls' spatial abilities: Charting the contributions of experiences and attitudes in different academic groups. *British Journal of Educational Psychology*, **72**:245 - 260.

Quinn, A. (2002): An Interrogative Approach to Novice Programming. In *Proceedings of IEEE Symposia on Human Centric Computing Languages and Environments*.

Rader, C., Cherry, G., Brand, C., Repenning, A. and Lewis, C. (1998): Designing Mixed Textual and Iconic Programming Languages for Novice Users. In *Proceedings of IEEE Symposium on Visual Languages*:187 - 194.

Ramalingam, V. and Wiedenbeck, S. (1997): An empirical study of novice program comprehension in the imperative and object-oriented styles. In *Proceedings of 7th Workshop on Empirical Studies of Programmers*:124 - 139.

Reek, M. M (1995): A Top-Down approach to Teaching Programming. *ACM SIGCSE Bulletin*, **27**(1):6 - 9.

Reid, R.J. (2002): *First Course (CS1) Language List, 23rd Edition* [online]. Available at http://www.cs.wvu.edu/~vanscoy/REID23.HTM. [Accessed on 6 September 2004].

Roos, J.D. (1998): *Computerised Mazes Test*. Honours Treatise. Department of Computer Science and Information Systems, University of Port Elizabeth. Port Elizabeth, South Africa.

Roumani, H. (2002): Design Guidelines for the Lab Component of Objects-First CS1. In *Proceedings of SIGCSE*:222 - 226.

Rountree, N., Rountree, J. and Robins, A. (2002): Predictors of Success and Failure in a CS1 Course. *ACM SIGCSE Bulletin*, **34**(4):121 - 124.

Roussev, B. (2003): Teaching Introduction to Programming as Part of the IS Component of the Business Curriculum. In *Proceedings of Informing Science + Information Technology Education Joint Conference (I$^{n}$SITE)*:1353 - 1360.

Rowell, G.H., Perhac, D.G., Hankins, J. A., Parker, B.C., Pettey, C.C. and Iriarte-Gross, J.M. (2003): Computer-related gender differences. In *Proceedings of 34th SIGCSE Technical Symposium on Computer Science Education*:54 - 58.

Salcedo, M. (2003): Faculty and the 21st Century Student in USA Higher Education. *ACM SIGCSE Bulletin*, **35**(2):83 - 87.

Sanders, D. and Dorn, B. (2003a): Classroom Experience with Jeroo. *Journal of Computing in Small Colleges*, **18**(4):308 - 316.

Sanders, D. and Dorn, B. (2003b): Jeroo: A Tool for Introducing Object-Oriented Programming. In *Proceedings of 34th SIGCSE Technical Symposium on Computer Science Education*:201 - 204.

Satratzemi, M., Dagdilelis, V. and Evagelidis, G. (2001): A system for program visualization and problem-solving path assessment of novice programmers. *ACM SIGCSE Bulletin*, **33**(3):137 - 140.

Sauter, V. L. (1986): Predicting computer programming skill. *Computers and Education*, **10**(2):299 - 302.

Scanlan, D. (1989): Structured Flowcharts Outperform Pseudocode: An Experimental Comparison. *IEEE Software*, **6**(5):28 - 36.

Schiffer, S. and Fršhlich, J. H. (1995): Visual Programming and Software Engineering with Vista. In *Visual Object-oriented Programming Concepts and Environments*:201. M.M. Burnett, A. Goldberg and T. Lewis (Eds.), Manning.

Schmucker, K. J. (1996): Rapid Prototyping using Visual programming Tools. In *Proceedings of CHI 96*:359 - 360.

Shannon, C. (2003): Another Breadth-first approach to CS 1 using Python. In *Proceedings of 34th SIGCSE Technical Symposium on Computer Science Education*:248 - 251.

Shih, Y-F. and Alessi, S.M. (1993): Mental models and transfer of learning in computer programming. *Journal of Research on Computing in Education*, **26**(2):154 - 176.

Shneiderman, B. (1983): Direct Manipulation: A step beyond Programming languages. *IEEE Computer*, **16**(8):56 - 69.

Shu, N. C. (1985): Visual Programming Languages: A Perspective and Dimensional Analysis. In *Proceedings of International Symposium on New Directions in Computing*:326 - 334.

Shu, N. C. (1988): A visual programming environment for automatic programming. In *Proceedings of 21st Hawaii International Conference on System Sciences*:662.

Shute, V. (1991): Who is likely to acquire programming skills? *Journal of Educational Computing Research*, **7**:1 - 24.

Siegel, E. V. (1999): Why Do Fools Fall Into Infinite Loops: Singing To Your Computer Science Class. *ACM SIGCSE Bulletin*, **31**(3):167 - 170.

Smith, D. C., Cypher, A. and Tesler, L. (2000): Novice programming comes of age. *Communications of the ACM*, **43**(3):75 - 81.

Soloman, B. A. and Felder, R.M. (undated): *Index of Learning Styles (ILS)* [online]. Available at http://www.ncsu.edu/felder-public/ILSpage.html. [Accessed on 5 November 2003].

Soloway, E. (1986): Learning to program = learning to construct mechanisms and explanations. *Communications of the ACM*, **29**(9):850 - 858.

Soloway, E., Ehrlich, K. and Bonar, J. (1982): Tapping into Tacit Programming Knowledge. In *Proceedings of SIGCHI Conference on Human Factors in Computing Systems*:52 - 57.

Spohrer, J. C. and Soloway, E. (1986): Novice mistakes: are the folk wisdoms correct? *Communications of the ACM*, **29**(7):624 - 632.

Stajano, F. (2000): Python in Education: Raising a Generation of Native Speakers. In *Proceedings of 8th International Python Conference*. Available at http://www.python.org/workshops/2000-01/proceedings/papers/stajano/stajano.pdf.

StatSoft Inc. (2001): STATISTICA (data analysis software system) Ver. 6.

Stephens, L.J., Wileman, A. and Konvalina, J. (1981): Group Differences in Computer Science Aptitude. *Association for Educational Data Systems Journal*:84 - 95.

Streicher, M. (1998): Interactive Learner Ver. 1.0.

Streicher, M. (2003): *The Significance of User Modelling in Designing Distributed Multimedia-based Computerised Tests*. Unpublished Masters dissertation. Department of Computer Science and Information Systems, University of Port Elizabeth. Port Elizabeth, South Africa.

Studer, S. D., Taylor, J. and Macie, K. (1995): Youngster: A Simplified Introduction to Computing: Removing the Details so that a Child may program. *ACM SIGCSE Bulletin*, **27**(1):102 - 105.

Sutherland, L. (1995): *Facilitating the Development of Problem Solving Expertise: The Impact of the Question Analysis Strategy* [online]. Available at http://www.aare.edu.au/95pap/suthl95.184. [Accessed on 1 October 2003].

Tanimoto, S. T. and Glinert, E. P. (1986): Designing Iconic Programming Systems: Representation and Learnability. In *Proceedings of IEEE Workshop on Visual Languages*:54 - 60.

Thomas, J. (2002a): B#: A Visual Programming Tool Ver. 2.0.

Thomas, J. (2002b): *B#: Version 2*. Honours treatise. Department of Computer Science and Information Systems, University of Port Elizabeth. Port Elizabeth, South Africa.

Thomas, L., Ratcliffe, M., Woodbury, J. and Jarman, E (2002): Learning Styles and Performance in the Introductory Programming Sequence. *ACM SIGCSE Bulletin*, **34**(1):33 - 37.

Toombs, K M (1987): *An Investigation into the Effect of Computer Symbolic Modes on Learning Processes*. Doctoral Thesis. University of South Africa, South Africa.

UCAS (2000): *Universities and Colleges Admissions Service for the UK. Technical Report.* [online]. Available at http://www.ucas.ac.uk/ or http://www.ucas.ac.uk/figures/index.html. [Accessed on 12 August 2003].

UPE (2002): Faculty of Science Pass Rate Report. Internal memo. University of Port Elizabeth. Port Elizabeth, South Africa.

UPE (2003a): Academic Records. University of Port Elizabeth. Port Elizabeth, South Africa.

UPE (2003b): Syllabi for Departments in Faculty of Science. University of Port Elizabeth. Port Elizabeth, South Africa.

UPE (2004): Amendment to Faculty of Commerce Rules. University of Port Elizabeth. Port Elizabeth, South Africa.

Urban-Lurain, M. and Weinshank, D. J. (2000): *Is There A Role for Programming in Non-Major Computer Science Courses?* [online]. Available at http://aral.cse.msu.edu/Publications/FIE2000/fie2000.htm. [Accessed on 18 July 2002].

Van Tonder, M. (2003): *A Java Development Tool*. Honours treatise. Department of Computer Science and Information Systems, University of Port Elizabeth. Port Elizabeth, South Africa.

VanLengen, C. A. and Maddux, C. D. (1990): Does Instruction in Computer Programming Improve Problem Solving Ability? *Journal of Information Systems Education*, **2**(2). Available at http://www.gise.org/JISE/Vol1-5/DOESINST.htm.

Ventura, P.R. (2003): *On the Origins of Programmers: Identifying Predictors of Success for an Objects First CS1*. Doctoral thesis. Department of Computer Science and Engineering, Graduate School of the State University of New York at Buffalo.

Vogts, D. (2004): *A Simplified Programming Language for Novice Programmers*. Unpublished Doctoral thesis. Department of Computer Science and Information Systems, University of Port Elizabeth. Port Elizabeth, South Africa.

Waddel, K. C. and Cross, J. H. (1988): Survey of Empirical Studies of Graphical Representations for Algorithms. In *Proceedings of ACM 16th Annual Conference on Computer Science*:696.

Ware, C. (1993): The Foundations of Experimental Semiotics: A Theory of Sensory and Conventional Representation. *Journal of Visual Languages and Computing*, **4**:91 - 100.

Warren, P R (2000): *Using JavaScript to Teach an Introduction to Programming* [online]. Available at http://saturn.cs.unp.ac.za/~peterw/JavaScript/calculator.html. [Accessed on 5 June 2003].

Warren, P R (2001): Teaching programming using scripting languages. *Journal of Computing in Small Colleges*, **17**(2):205 - 216.

Warren, P R (2003): Learning to Program: Spreadsheets, Scripting and HCI. In *Proceedings of Southern African Computer Lecturers Association (SACLA) Annual Conference*.

Watts, T. (2003): A Structured Flow Chart Editor Version 3. In *Proceedings of ACM SIGCSE Faculty Poster*. Available at http://www.cs.sonoma.edu/~tiawatts/SFC/.

Wesson, J.L. (2002): *Pass Rates*. Internal departmental correspondence. University of Port Elizabeth. Port Elizabeth, South Africa.

West, M. and Ross, S. (2002): Retaining females in Computer Science: A new look at a persistent problem. *Journal of Computing in Small Colleges*, **17**(5):1 - 7.

Whitley, K. N. and Blackwell, A. F. (2001): Visual Programming in the Wild: A Survey of LabVIEW Programmers. *Journal of Visual Languages and Computing*, **12**(4):435 - 472.

Whitley, K.N. (1997): Visual Programming Languages and the Empirical Evidence For and Against. *Journal of Visual Languages and Computing*, **8**(1):9 - 142. Available at http://www.cs.dal.ca/~pcox/CSCI6304/references/whitley.pdf.

Wiedenbeck, S., Ramalingam, V., Sarasamma, S. and Corritore, C. L. (1999): A comparison of the comprehension of object-oriented and procedural programs by novice programmers. *Interacting with Computers*, **11**:255 - 282.

Williams, S. and Walmsley, S. (1999): *Discover Delphi: Programming Principles Explained*. Addison-Wesley.

Wilson, B. C. and Shrock, S. (2001): Contributing to Success in an Introductory Computer Science Course: A Study of Twelve Factors. *ACM SIGCSE Bulletin*, **33**(1):184 - 188.

Wilson, J. D. and Braun, G. F. (1985): Psychological Differences in University Computer Student populations. *ACM SIGCSE Bulletin*, **17**(1):166 - 177.

Wirth, N. (1971): The Programming Language Pascal. *Acta Informatica*, **1**:35 - 63.

Wright, T. and Cockburn, A. (2000): Writing, Reading, Watching: A Task-Based Analysis and Review of Learners' Programming Environments. In *Proceedings of International Workshop on Advanced Learning Technologies*. IEEE Computer Society Press. Available at http://www.cosc.canterbury.ac.nz/~andy/papers/iwalt.pdf.

Wright, T. and Cockburn, A. (2002): Mulspren: a MUltiple Language Simulation PRogramming ENvironment. In *Proceedings of IEEE Symposia on Human Centric Computing Languages and Environments*.

Yeh, C. L. (2003a): B#: A Visual Programming and Tracing Tool Ver. 3.0.

Yeh, C. L. (2003b): *Tracing programs in B#*. Honours treatise. Department of Computer Science and Information Systems, University of Port Elizabeth. Port Elizabeth, South Africa.

Yeh, C. L. (2004): *A Framework for Algorithm Animation in Introductory Programming Courses: A Case Study in Sorting Algorithms*. Unpublished Masters dissertation. Department of Computer Science and Information Systems, University of Port Elizabeth. Port Elizabeth, South Africa.

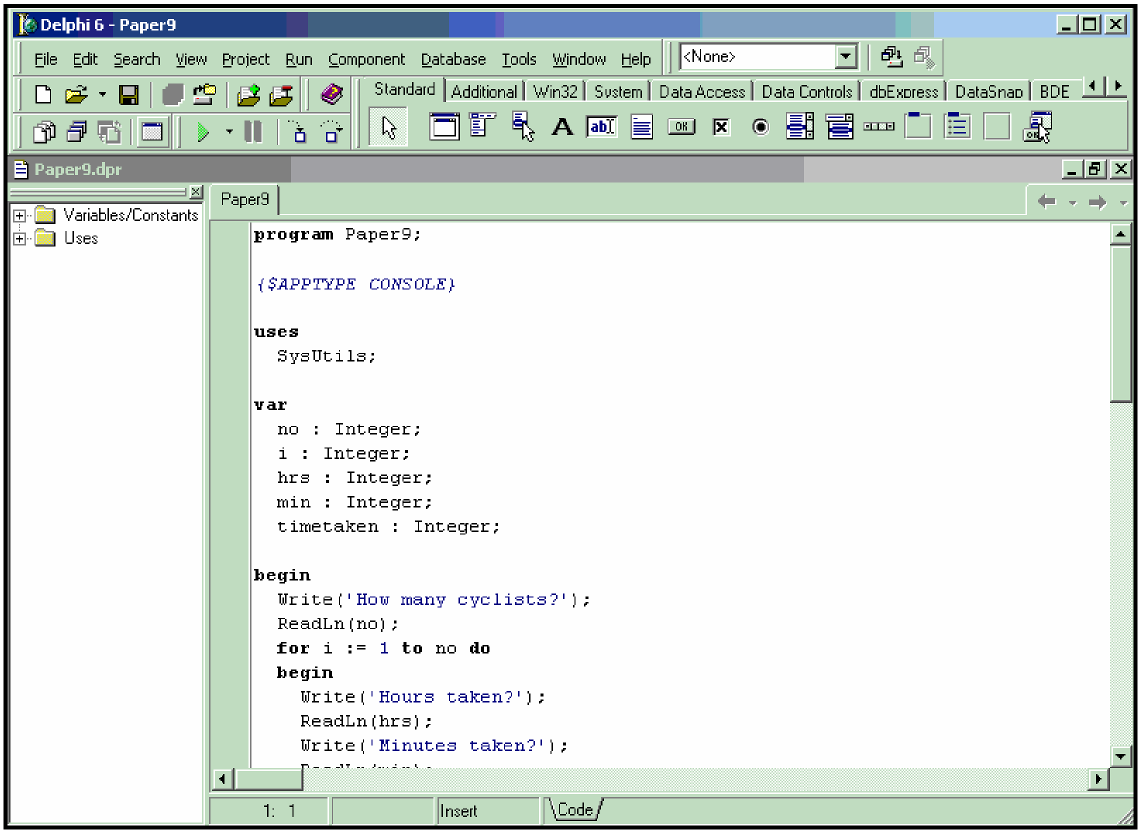
Zaaiman, H., Van der Flier, H. and Thijs, G. D. (1998): Selecting South African higher education students: critical issues and proposed solutions. *South African Journal of Higher Education*, **12**(3):96 - 101.

Zelle, J. M. (undated): *Python as a First Language* [online]. Available at http://mcsp.wartburg.edu/zelle/python/python-first.html. [Accessed on 6 September 2004].

Ziegler, U. and Crews, T. (1999): An Integrated Program development Tool for Teaching and Learning How to Program. *ACM SIGCSE Bulletin*, **31**(1):276 - 280.

# Appendix A

## Commercial Programming Development Environment



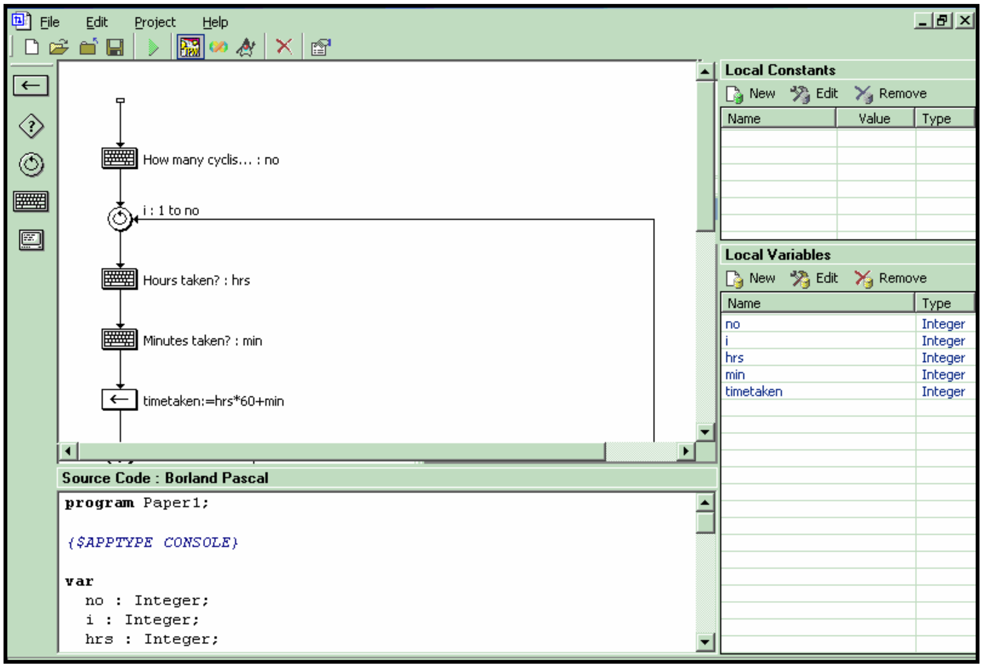
```
program Paper9;

{$APPTYPE CONSOLE}

uses
  SysUtils;

var
  no : Integer;
  i : Integer;
  hrs : Integer;
  min : Integer;
  timetaken : Integer;

begin
  Write('How many cyclists?');
  ReadLn(no);
  for i := 1 to no do
  begin
    Write('Hours taken?');
    ReadLn(hrs);
    Write('Minutes taken?');
```

*Figure A.1: Borland$^{®}$ Delphi ™Enterprise version 6 Textual Programming Environment (Borland 2003)*

# Appendix B

# B# Programming Development Environment

## B.1 Screen Layout



Local Constants

| Name | Value | Type |
|---|---|---|

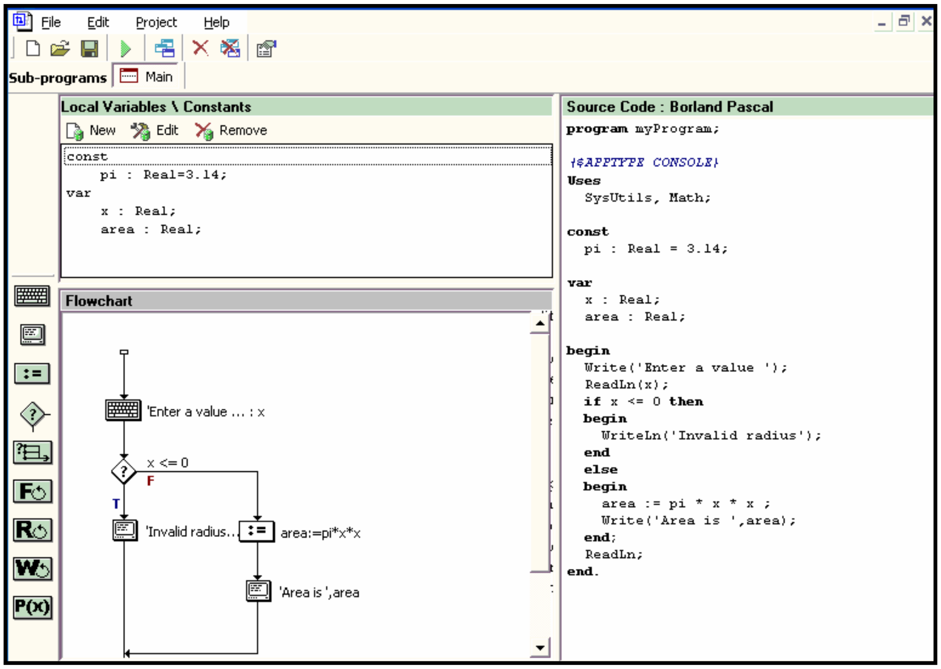
Local Variables

| Name | Type |
|---|---|
| no | Integer |
| i | Integer |
| hrs | Integer |
| min | Integer |
| timetaken | Integer |

Source Code : Borland Pascal

```
program Paper1;

{$APPTYPE CONSOLE}

var
  no : Integer;
  i : Integer;
  hrs : Integer;
```

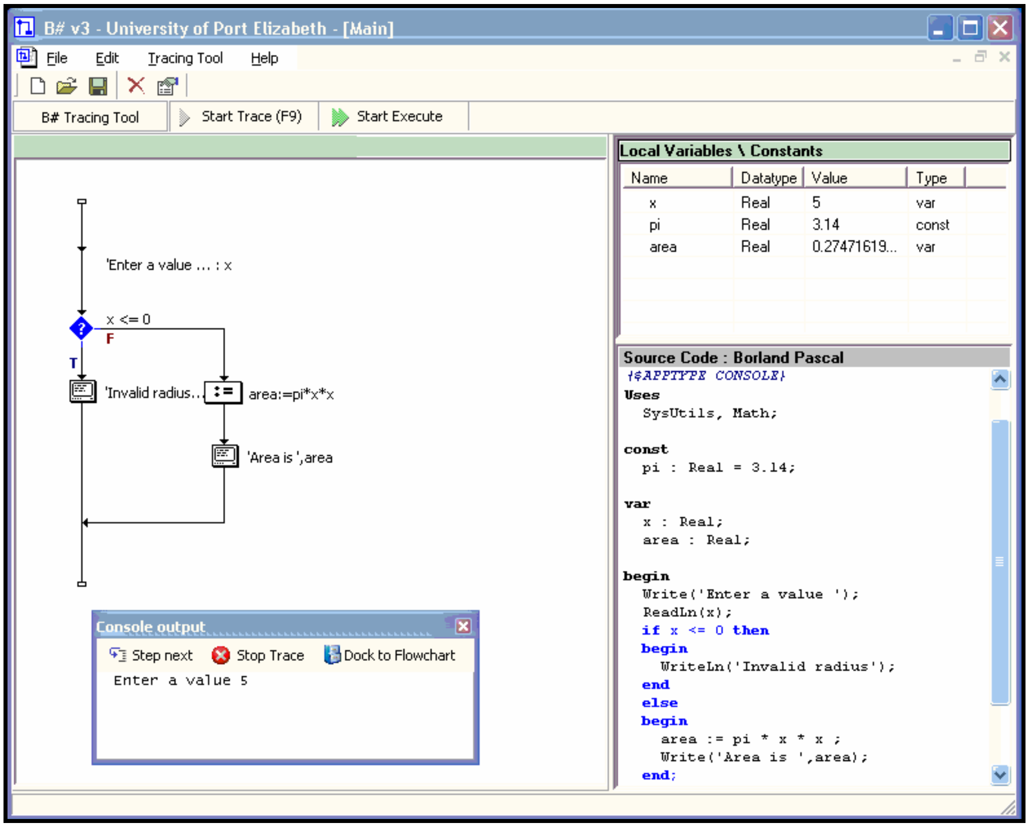
*Figure B.1: B# ver. 1.0 programming environment (Brown 2001b)*



```
const
    pi : Real=3.14;
var
    x : Real;
    area : Real;
```

```
program myProgram;

{$APPTYPE CONSOLE}
Uses
  SysUtils, Math;

const
  pi : Real = 3.14;

var
  x : Real;
  area : Real;

begin
  Write('Enter a value ');
  ReadLn(x);
  if x <= 0 then
  begin
    WriteLn('Invalid radius');
  end
  else
  begin
    area := pi * x * x ;
    Write('Area is ',area);
  end;
  ReadLn;
end.
```

*Figure B.2: B# ver. 2.0 programming environment (Thomas 2002a)*



```
Enter a value 5
```

| Name | Datatype | Value | Type |
|---|---|---|---|
| x | Real | 5 | var |
| pi | Real | 3.14 | const |
| area | Real | 0.27471619... | var |

```
{$APPTYPE CONSOLE}
Uses
  SysUtils, Math;

const
  pi : Real = 3.14;

var
  x : Real;
  area : Real;

begin
  Write('Enter a value ');
  ReadLn(x);
  if x <= 0 then
  begin
    WriteLn('Invalid radius');
  end
  else
  begin
    area := pi * x * x ;
    Write('Area is ',area);
  end;
```

*Figure B.3: B# ver. 3.0 programming environment (Yeh 2003a)*

## B.2 B# ver. 2.0 Icon Dialogues

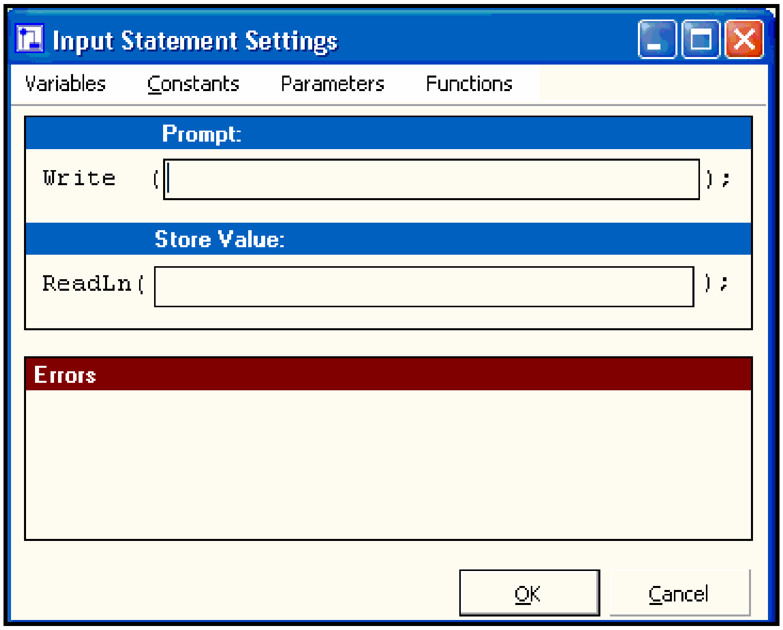
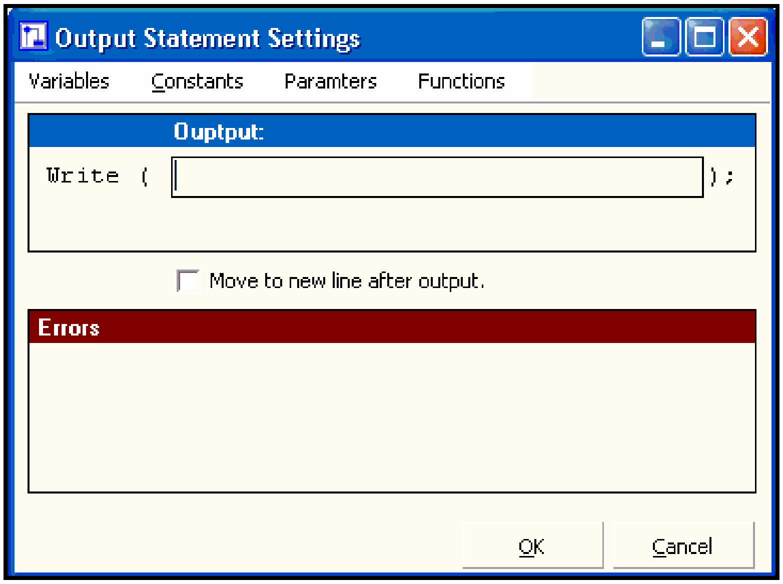


*Figure B.4: Data input programming construct icon dialogue*



□ Move to new line after output.

*Figure B.5: Data output programming construct icon dialogue*

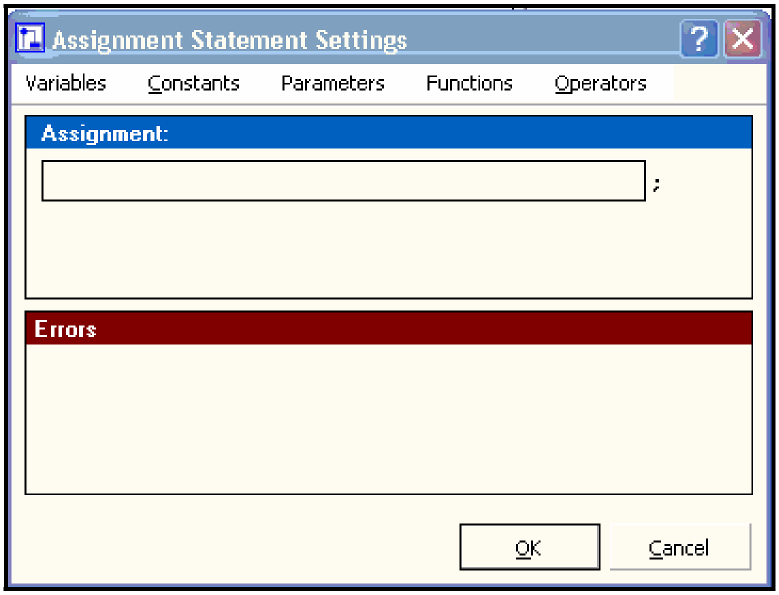
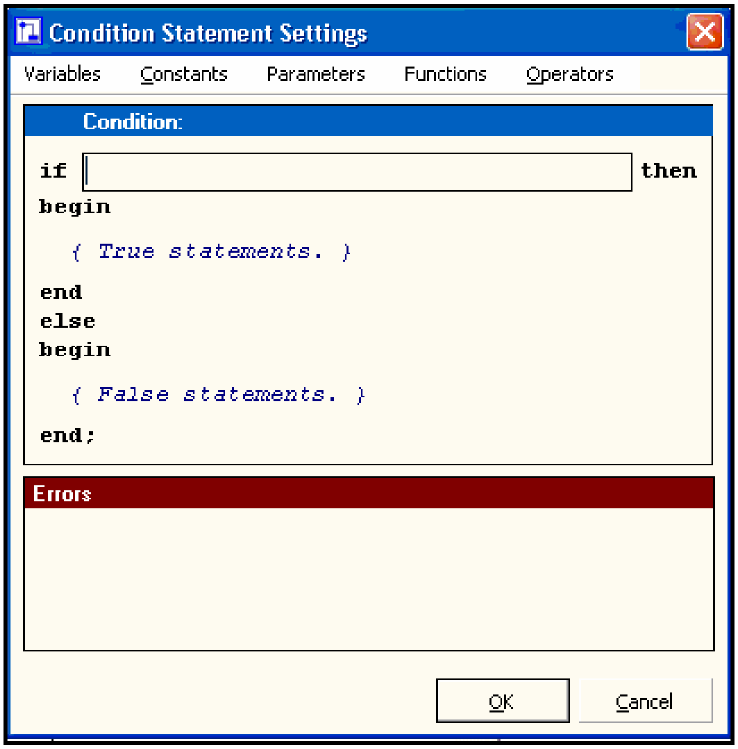
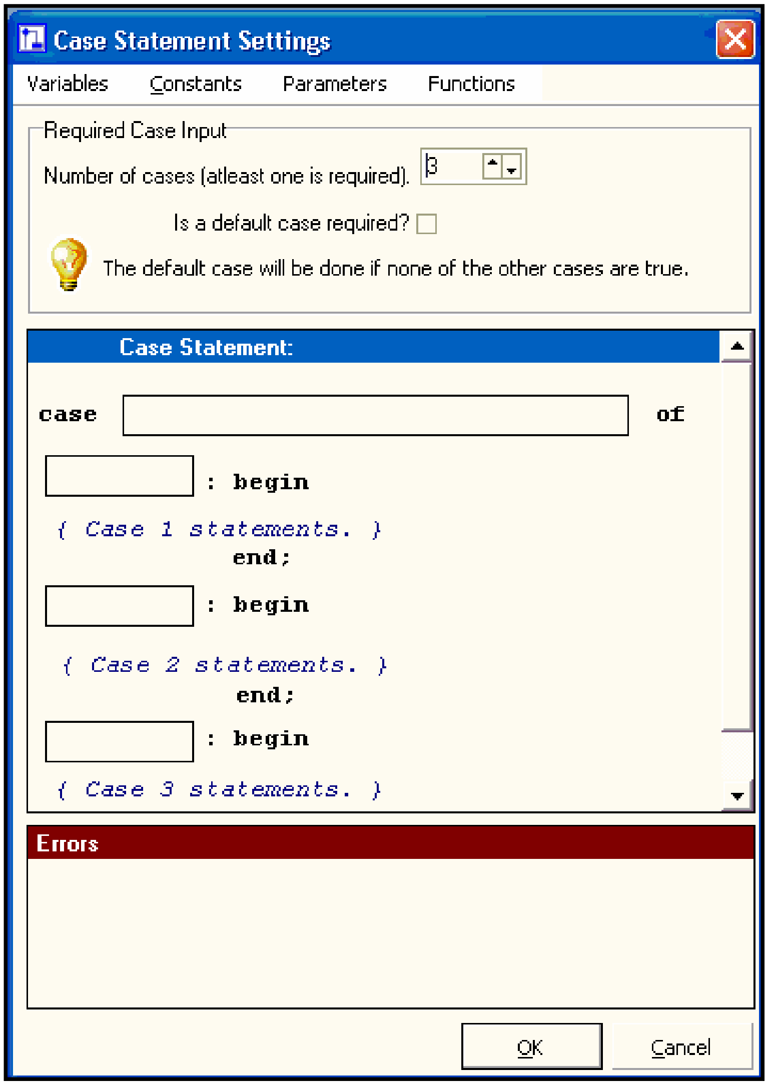


*Figure B.6: Assignment programming construct icon dialogue*



```
if                then
begin
   { True statements. }
end
else
begin
   { False statements. }
end;
```

*Figure B.7: Single branch conditional programming construct icon dialogue*



Required Case Input

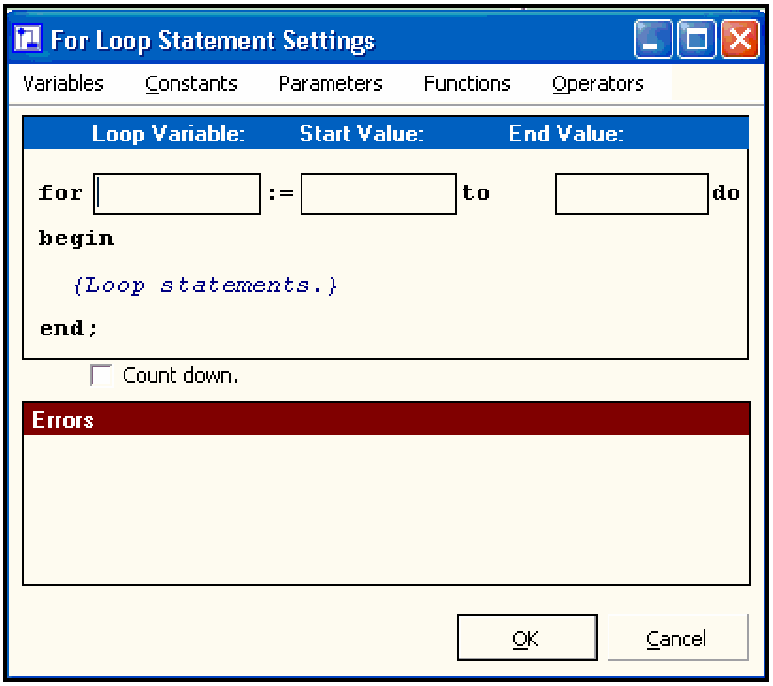
Number of cases (atleast one is required). 3

Is a default case required? □

The default case will be done if none of the other cases are true.

```
case [                    ] of
 [        ] : begin
 { Case 1 statements. }
            end;
 [        ] : begin
 { Case 2 statements. }
            end;
 [        ] : begin
 { Case 3 statements. }
```

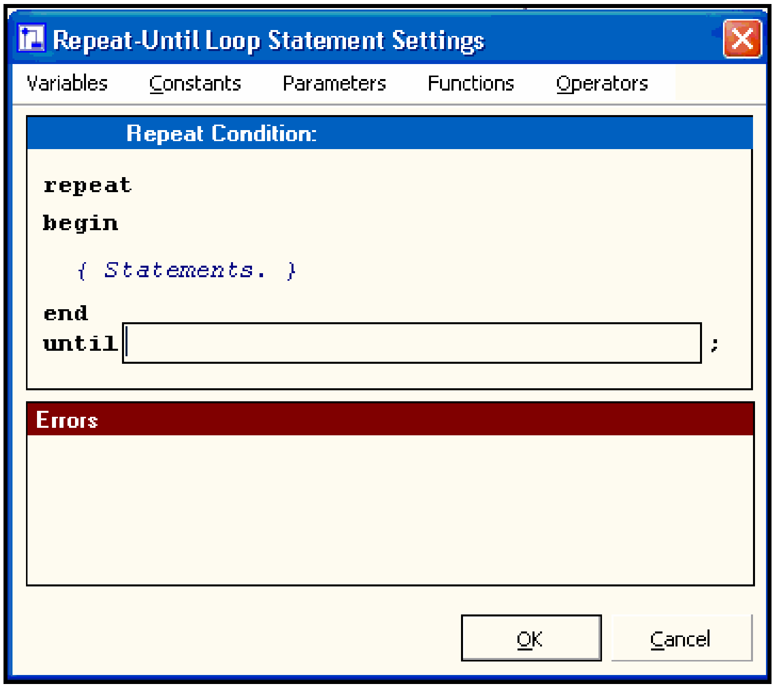
*Figure B.8: Multiple branches conditional programming construct icon dialogue*



```
for [          ] := [          ] to [          ] do
begin
   {Loop statements.}
end;
```

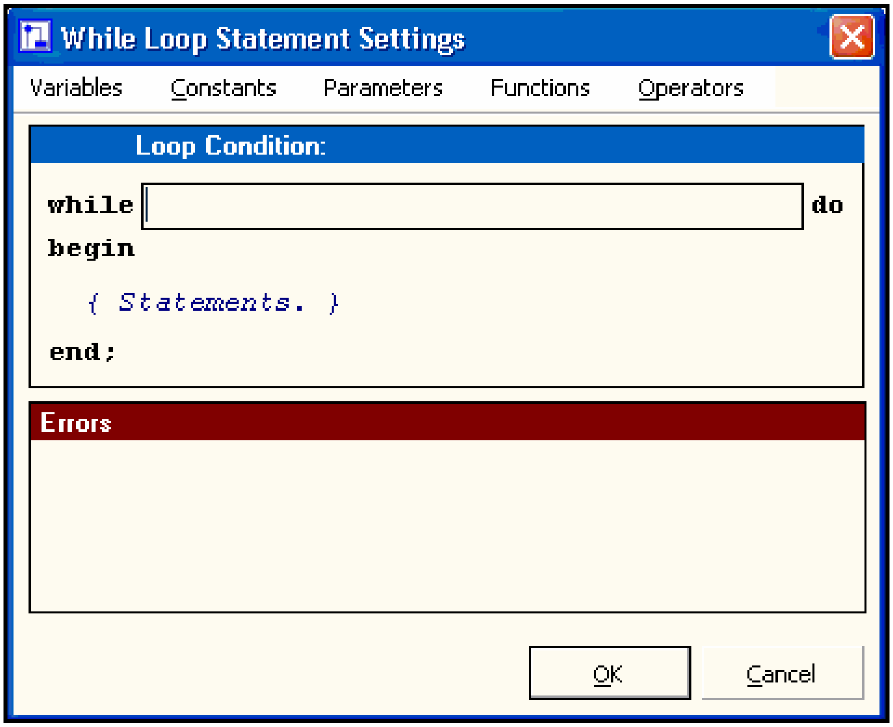
□ Count down.

*Figure B.9: Counter controlled iteration programming construct icon dialogue*



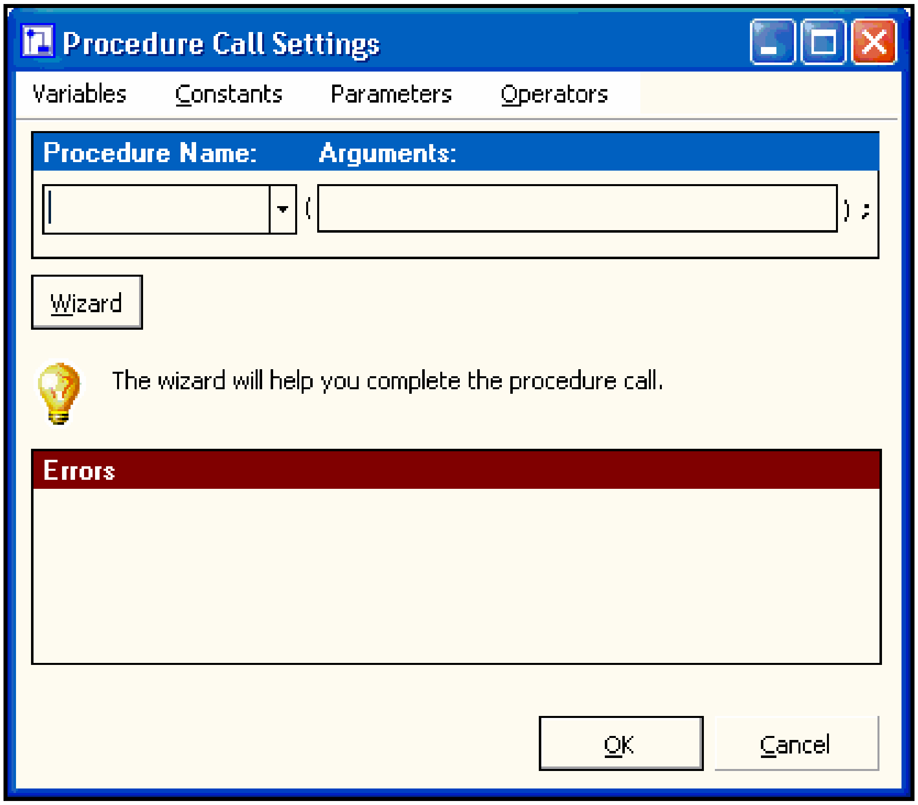
```
repeat
begin
   { Statements. }
end
until [                                        ] ;
```

*Figure B.10: Post-test sentinel controlled iteration programming construct icon dialogue*



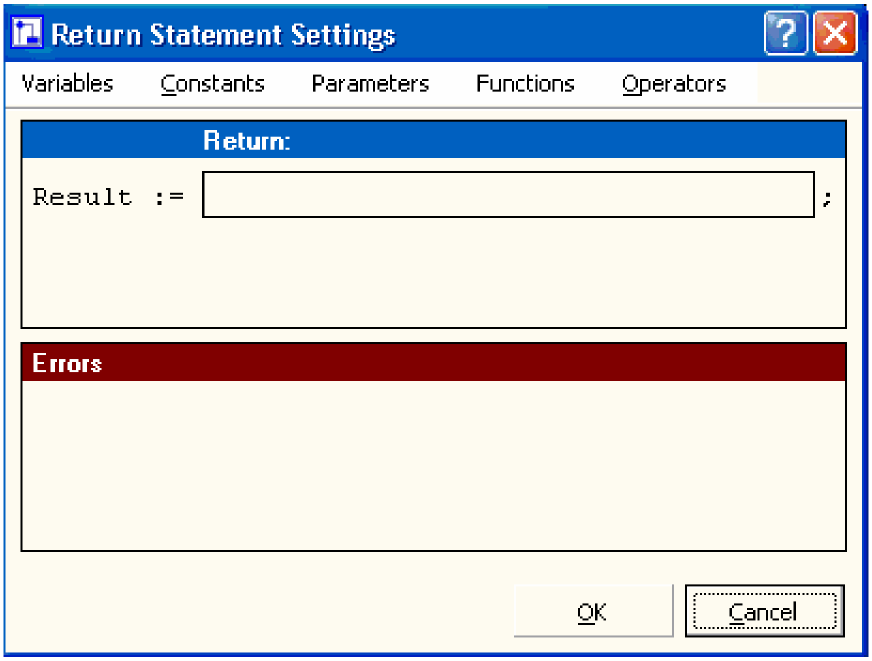
```
while                                   do
begin
   { Statements. }
end;
```

*Figure B.11: Pre-test sentinel controlled iteration programming construct icon dialogue*



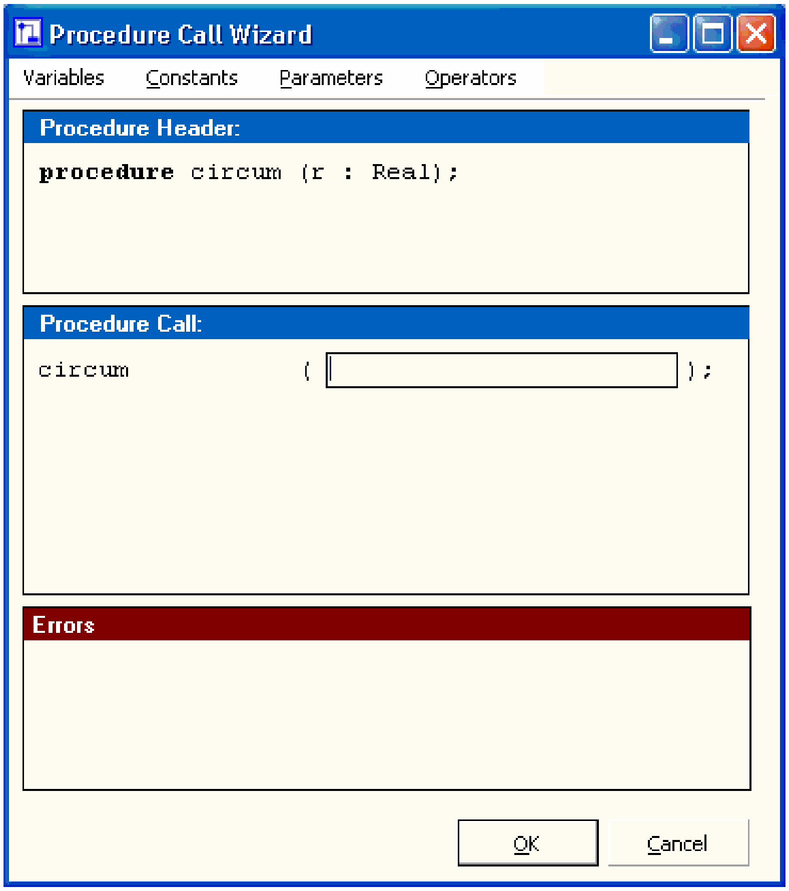
The wizard will help you complete the procedure call.

*Figure B.12: Procedure declaration programming construct icon dialogue*



```
Result :=                              ;
```

*Figure B.13: Return icon dialogue*



```
procedure circum (r : Real);
```

```
circum          (                    );
```

*Figure B.14: Procedure call wizard icon dialogue*

# Appendix C

## Learning Material

The material (M3 and M8) appearing in this appendix was distributed amongst only the subjects under treatment. The material was divided up into 9 weekly sections, a practical per week, with each section covering only those programming constructs that were taught during traditional lectures in the relevant week. A section of the material was consequently distributed on a weekly basis, with the entire duration of the treatment being a period of 9 weeks. The material appearing in this appendix was required to be studied by each subject in the treatment group on their own cognisance.

All page references within a section are relative to the first page of the section.

Subjects who were not under treatment were required to implement the same practicals using only the Delphi™ Enterprise textual programming environment. They were also expected to familiarise themselves with the textual programming environment without instructor interference.

# WRA101 : Practical 3
## Week 4 : 4 - 6 March 2003

*Sections Covered*

Variables and Data Types (pp 76 – 85)
Precedence Rules, Typecasting, Scope, Interactive Input and Output (pp 91 – 104)

*Objectives*

- Introduction to B# (an iconic programming environment)
  - Opening and running an existing solution
  - Creating, saving and running a new solution
- Introduction to Delphi (a textual programming environment)
  - Opening and running an existing solution
  - Creating, saving and running a new solution using the solution generated by B#

*Practical Preparation*

- Using any of the problem solving methodologies studied (flowchart or pseudocode), plan the solutions to each of the problems in Tasks 3, 4 and 5 *before* attending your practical session.
- Copy the file **`F:\Courses\WRA101\Practical 3\Tue\CircleArea.bpf`** under the folder for your usercode in the **`C:\Temp`** directory.

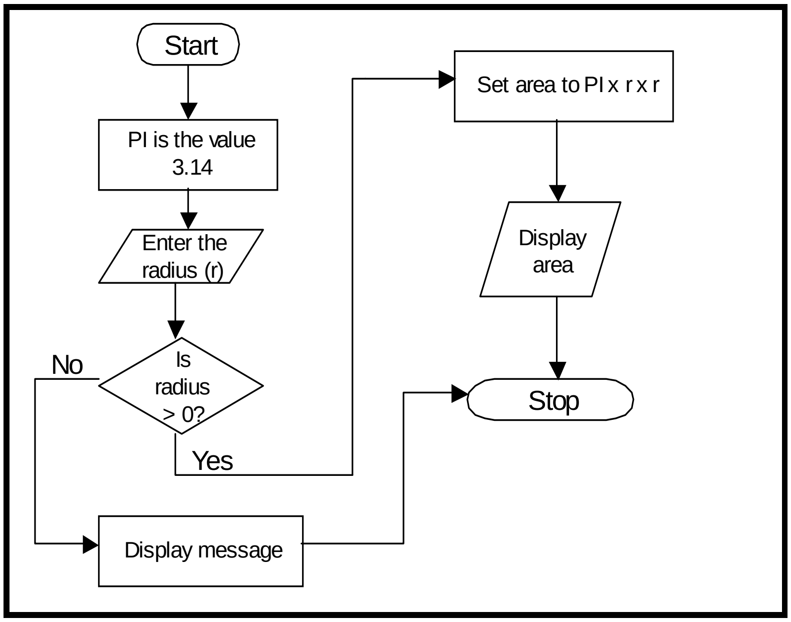
### Compulsory Practical Tasks

### ***Task 1 : Step-by-step introduction to B#***

Consider the following problem.

> *Determine and display the area of a circle if the radius entered is positive (>0), otherwise display an appropriate message.*

A flowchart corresponding to the solution for the above problem is:



The pseudocode corresponding to the problem is:

| ***Constants*** |
|---|
| PI = 3.14 |

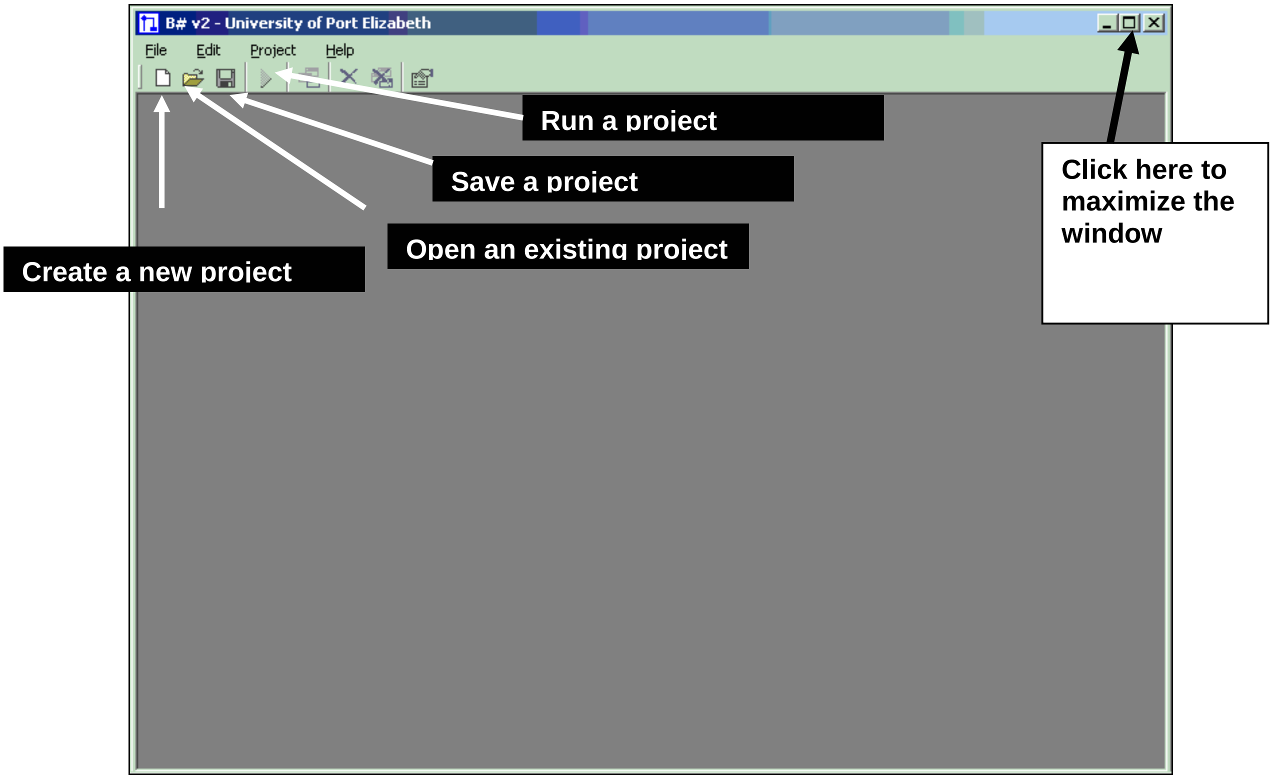
| ***Inputs*** | ***Outputs*** |
|---|---|
| radius | Area of circle or message |

| ***Algorithm*** |
|---|
| 1. Get the radius (r). |
| 2. If r > 0, then proceed to step 3; otherwise proceed to step 5. |
| 3. Calculate the area (area = PI x r x r). |
| 4. Display the area and stop processing. |
| 5. Display an appropriate error message. |

The following illustrates how the software package B# is used to run a solution to the above problem. Follow each of the steps closely. Ensure that you understand what you are doing every step of the way since you will need to imitate these steps in all of your subsequent practical sessions.

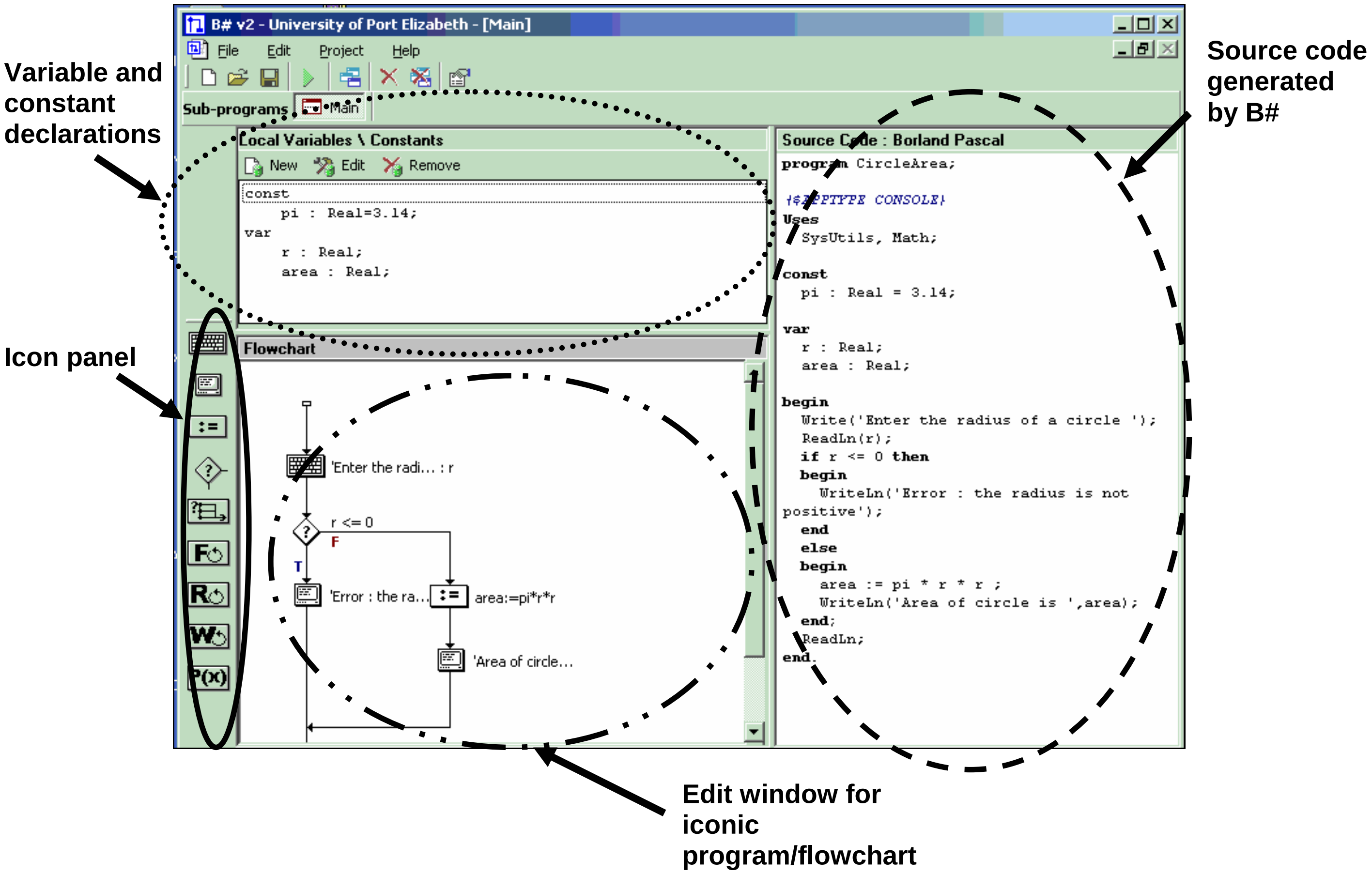
Activate the B# program from your desktop by double-clicking on the B# icon.

The following window is displayed.



## *How to open an existing B# program*

Click on the 📂 icon (*open existing project*) and locate and open the file **`CircleArea.bpf`** under your usercode in the **`C:\Temp`** directory (which you should have copied across from the network drive in preparation for your practical). The following window should then be displayed.


B# v2 - University of Port Elizabeth - [Main]
File
Edit
Project
Help
Sub-programs
Main
Local Variables \ Constants
New
Edit
Remove
Flowchart
'Enter the radi... : r
r <= 0
F
T
'Error : the ra...
area:=pi*r*r
'Area of circle...
Source Code : Borland Pascal
Variable and constant declarations
Icon panel
Source code generated by B#
Edit window for iconic program/flowchart

```
const
    pi : Real=3.14;
var
    r : Real;
    area : Real;
```

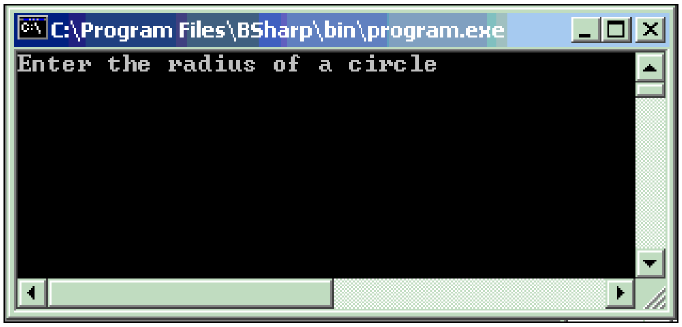
```
program CircleArea;

{$APPTYPE CONSOLE}
Uses
  SysUtils, Math;

const
  pi : Real = 3.14;

var
  r : Real;
  area : Real;

begin
  Write('Enter the radius of a circle ');
  ReadLn(r);
  if r <= 0 then
  begin
    WriteLn('Error : the radius is not
positive');
  end
  else
  begin
    area := pi * r * r ;
    WriteLn('Area of circle is ',area);
  end;
  ReadLn;
end.
```

Note the position and contents of the following panes on your screen.

- The top left hand pane lists all of the constants and variables that have been declared.
- The icon panel at the bottom far left hand side shows all of the programming constructs that are supported by B#. An icon represents each programming construct. During this practical session you will make use of only the top three, namely keyboard input (⌨), screen output (🖥) and assignment (:=).
- The bottom left hand pane is the edit window for the flowchart. This is the pane into which you drag and drop the icons for programming constructs, thereby creating a solution to a given problem.
- The correct source code generated by B# as you create a flowchart is displayed in the right hand pane. B# does not permit you to make any changes here, but get into the habit of always noting what B# places here when you do make a change to your flowchart program.

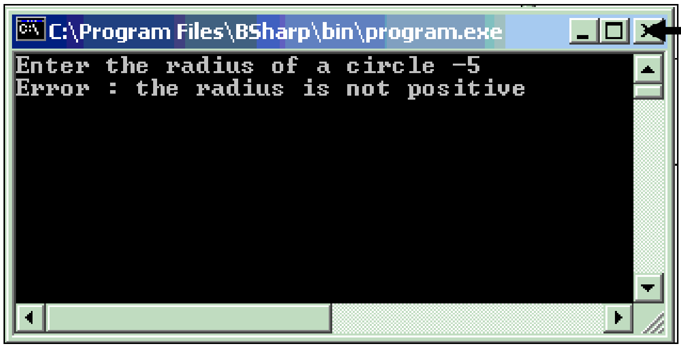
## How to run a B# program

Click on the ▷ icon found along the top of the main window of B# (see page 2). The following DOS window appears:



```
Enter the radius of a circle
```

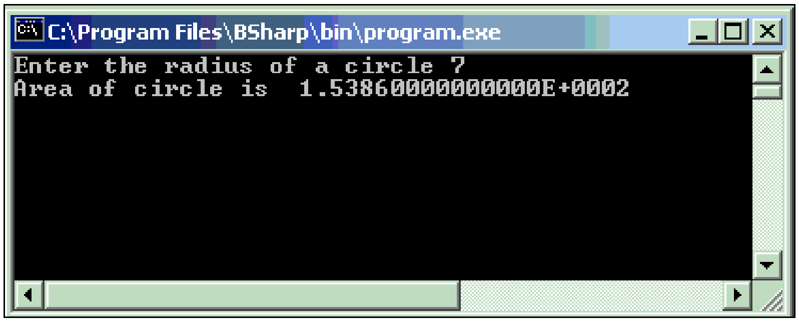
Enter the value **–5** as the radius of the circle and press the enter key (we would expect an appropriate error message to be displayed since **–5 <= 0**).  The following results:



```
Enter the radius of a circle -5
Error : the radius is not positive
```

Close the DOS window by clicking here.

Run the program again, this time entering **7** as the radius of the circle.  The following results:



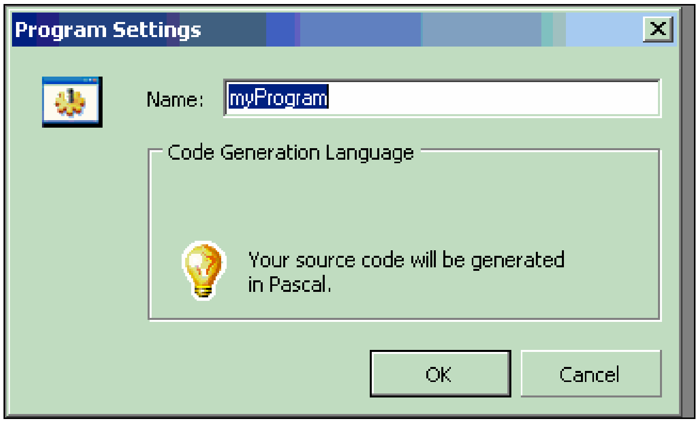
```
Enter the radius of a circle 7
Area of circle is  1.53860000000000E+0002
```

Run the project with the following values for the radius.  In the column ***Area***, write down the value displayed by the program.

| ***Radius*** | ***Area*** |
|---|---|
| 14.343 | |
| 2.175 | |

## *How to create and save a new B# program*

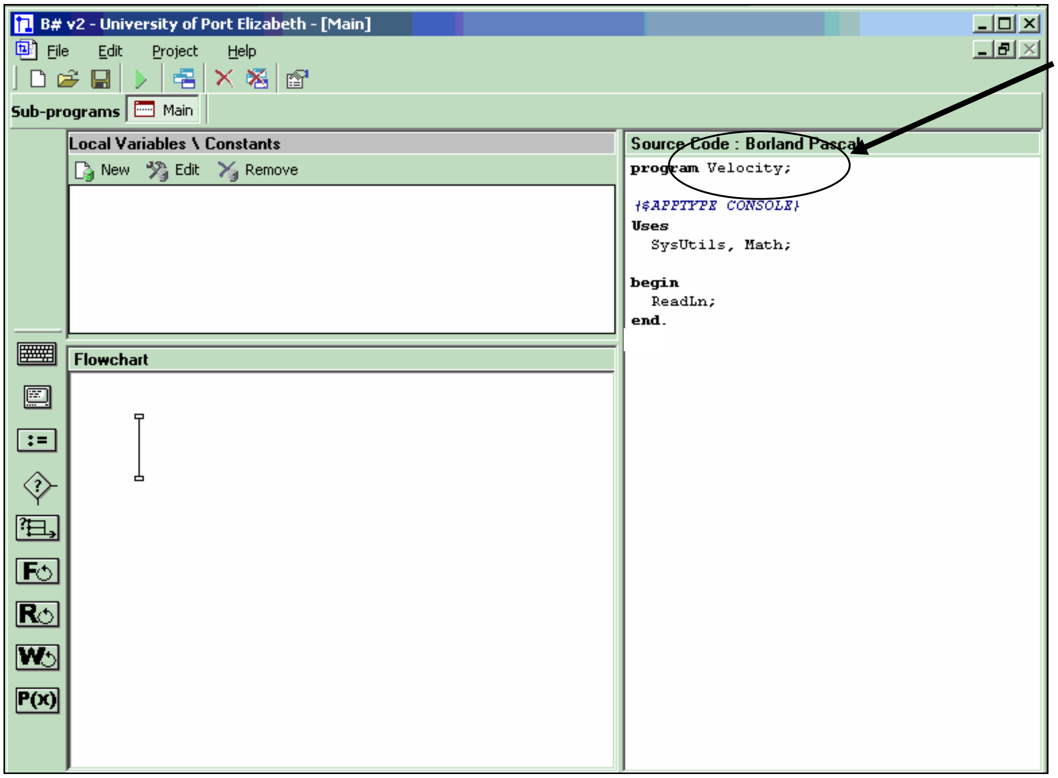
You must first choose to *create a new project* (a project is the name given to the file that contains the solution to the problem).  To do this, click on the 🗋 icon in the top left hand corner of the window (see page 2). The ***Program Settings*** dialogue box then appears in the middle of the window.



Name: myProgram

Your source code will be generated in Pascal.

Here you are provided with the opportunity to give your B# project a name. Give your project the name **`Velocity`** by overwriting the name **`myProgram`**. Click the **`OK`** button. The following window is displayed.



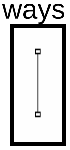
B# v2 - University of Port Elizabeth - [Main]
File
Edit
Project
Help
Sub-programs
Main
Local Variables \ Constants
New
Edit
Remove
Flowchart
Source Code : Borland Pascal

```
program Velocity;

{$APPTYPE CONSOLE}
Uses
  SysUtils, Math;

begin
  ReadLn;
end.
```

**Name of this program**

To save the program, click on the 💾 icon found along the top of the main window of B# (see page 2). This is the first time that the project is being saved, and you will be required to specify the folder in which the program must be saved as well as give the program a name. Store this program with the name **`Velocity`** under your usercode in the **`C:\Temp`** folder.

You will be adding a drawing that resembles a flowchart to the bottom left hand pane, and will also be able to add, edit and delete constants and variables in the top left hand pane. The right hand pane shows the correct Pascal code (which Delphi uses) that is automatically generated as you create your B# flowchart. You are encouraged to always be aware of this developing code as you implement your solution in B#.

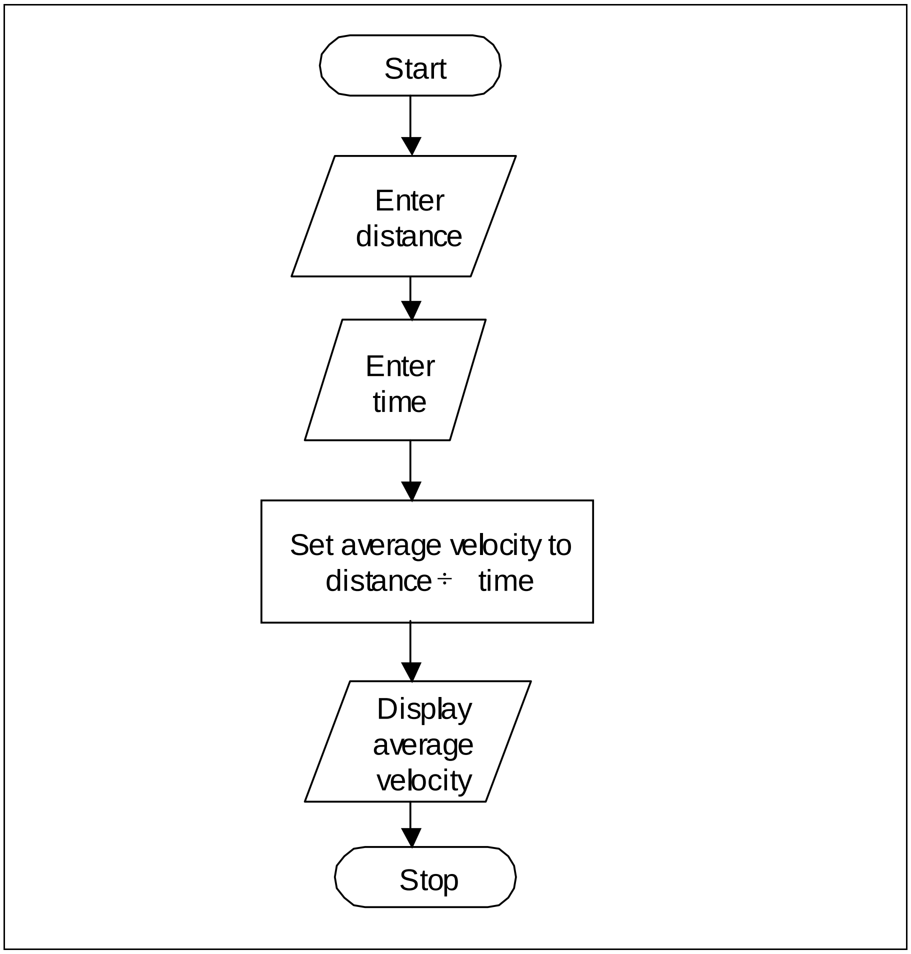
The  in the bottom left hand pane indicates the start (top) and stop (bottom) points of the solution. Anything that forms part of the solution is dragged-and-dropped onto the line between these points.

Consider the following problem.

Compute and display a vehicle's *average velocity* using the formula shown below (*distance* (in km) - and *time* (in hours) are user inputs that are each always assumed to be >0). Make valid assumptions about the data types of variables used:

Average velocity = distance ÷ time

A flowchart corresponding to the solution for the above problem is:



The corresponding pseudo-code is:

| ***Constants*** |
|---|
| |

| ***Inputs*** | ***Outputs*** |
|---|---|
| Distance (in km) | Average velocity |
| Time (in hours) | |

| ***Algorithm*** |
|---|
| 1. Get the distance travelled by the vehicle. |
| 2. Get the time taken to travel the distance. |
| 3. Calculate the average velocity<br>(average velocity becomes distance ÷ time) |
| 4. Display the velocity and stop processing. |

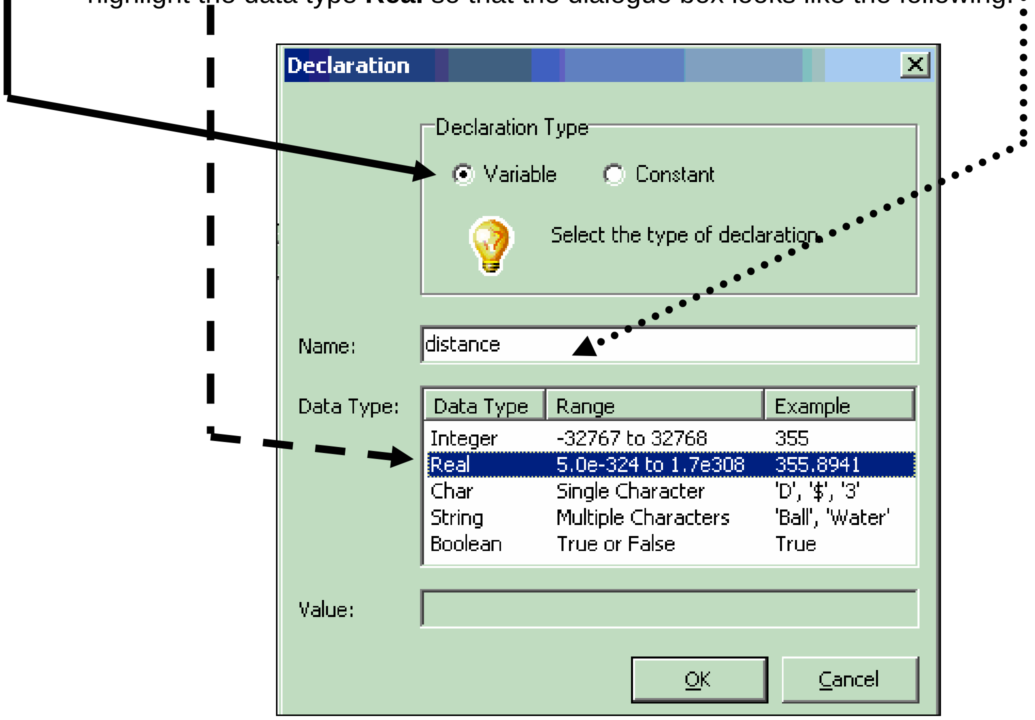
The following guides you in developing the B# solution to this problem.

## Declaring a variable

**Note: Constants are treated the same way as variables regarding declaration and usage.**

At the top of the constant and variable declaration pane (see screen shot on page 3 for assistance), click on the icon to declare a new variable ( New ).

The **Declaration** dialogue box is displayed.
Select the **Variable** radio button; give the variable the name **distance** (type it in) and highlight the data type **Real** so that the dialogue box looks like the following:



Declaration

Declaration Type

● Variable ○ Constant

Select the type of declaration

Name: distance

Data Type:

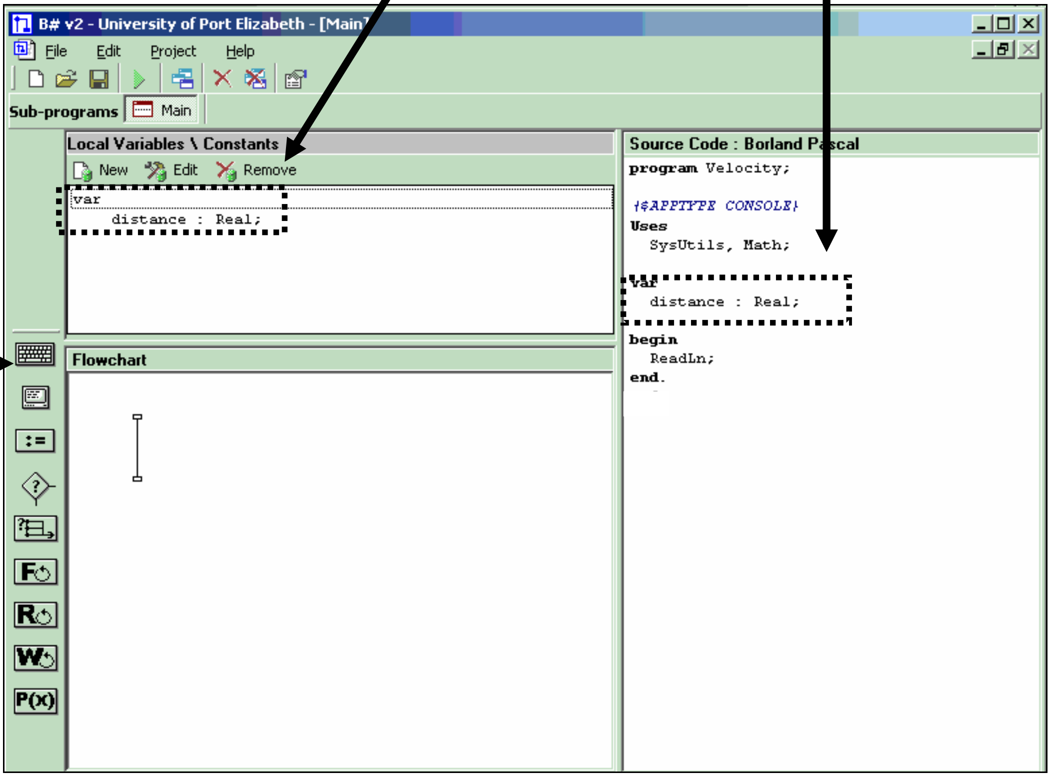
| Data Type | Range | Example |
|---|---|---|
| Integer | -32767 to 32768 | 355 |
| Real | 5.0e-324 to 1.7e308 | 355.8941 |
| Char | Single Character | 'D', '$', '3' |
| String | Multiple Characters | 'Ball', 'Water' |
| Boolean | True or False | True |

Value:

Click on the OK button.

The following window is now displayed.

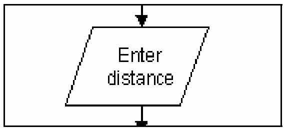
Note how the contents of the **Local Constants / Variables** and **Source Code** panes have changed when compared to the screen shot on page 5.

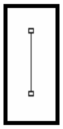

B# v2 - University of Port Elizabeth - [Main]
File
Edit
Project
Help
Sub-programs
Main
Local Variables \ Constants
New
Edit
Remove
Flowchart
Source Code : Borland Pascal

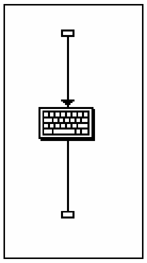
```
var
    distance : Real;
```

```
program Velocity;

{$APPTYPE CONSOLE}
Uses
  SysUtils, Math;

var
  distance : Real;
begin
  ReadLn;
end.
```

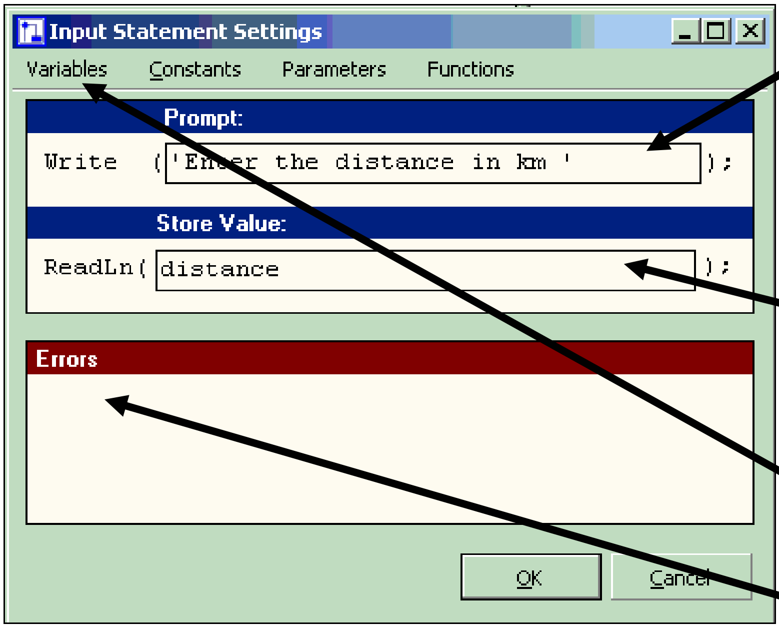
## *Reading input from the user*

To implement the  in the flowchart, which is a prompt to the user to enter a value and reading input from the user, namely the distance travelled by the vehicle, do the following:

Click on the *keyboard input* icon ⌨ found on the icon panel and, holding the left button of the mouse in, drag the icon onto the  positioning it on the line between the start and stop points, and let the mouse button go.  Note the developing diagram in the flowchart pane in the background window:



The ***Input Statement Settings*** dialogue is displayed. Complete it exactly as shown below by adding the indicated text and variable name in the appropriate places:



```
Write  ('Enter the distance in km ');
ReadLn(distance);
```

**Type this text in exactly as shown – including single quotation marks. This will be displayed to the user to request him/her to enter the distance travelled by the vehicle**

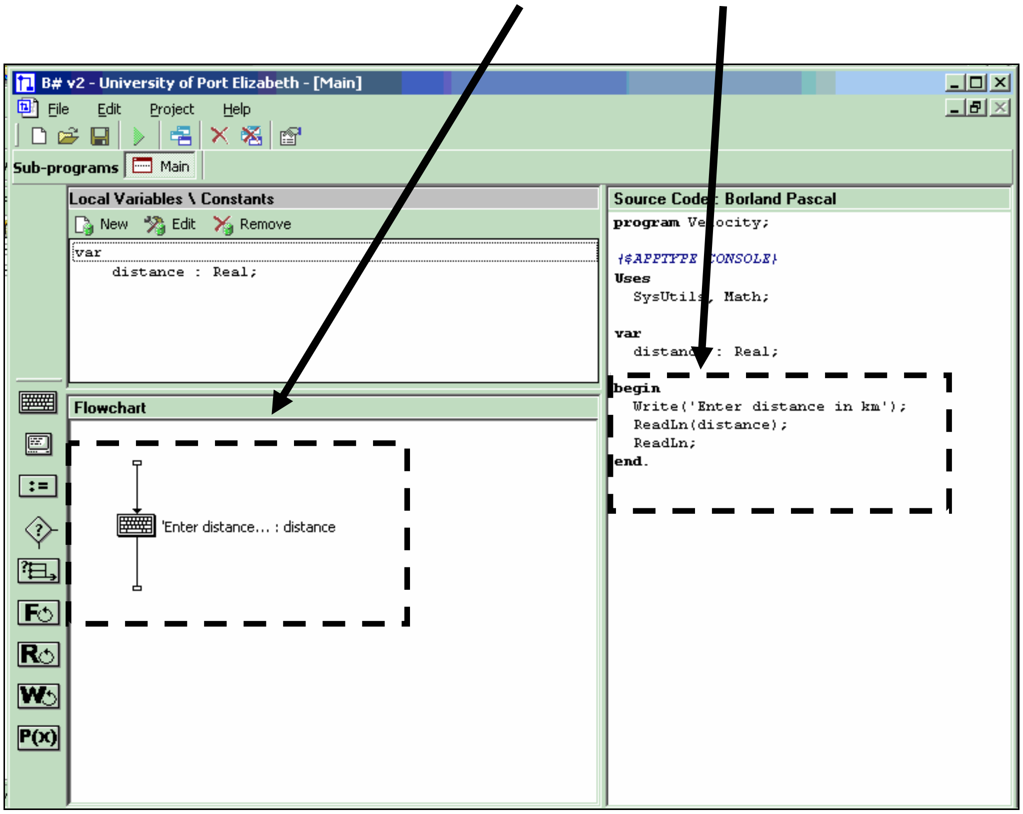
**You can either type in the name of the variable here (if you can remember it), or make a selection from a list of declared variables using this option**

**Any errors that you make will be listed here (for example, typing in an incorrect variable name). Read the error message(s) carefully and make the necessary changes**

Click on the OK button.

The following window is displayed.

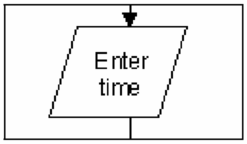
Note the changes in the contents of the **Flowchart** and **Source Code** panes.


B# v2 - University of Port Elizabeth - [Main]
File
Edit
Project
Help
Sub-programs
Main
Local Variables \ Constants
New
Edit
Remove
Flowchart
'Enter distance... : distance
Source Code: Borland Pascal

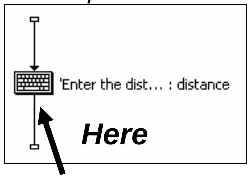
```
var
    distance : Real;
```

```
program Velocity;

{$APPTYPE CONSOLE}
Uses
  SysUtils, Math;

var
  distance : Real;

begin
  Write('Enter distance in km');
  ReadLn(distance);
  ReadLn;
end.
```

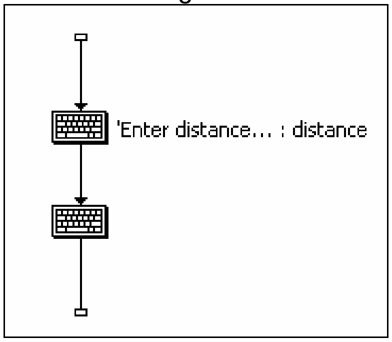
The next part of the solution requires another input value from the user. What follows is an illustration of the same process as above, but in a slightly different order. Once you have mastered the process, B# permits you to perform the process in your preferred manner.

To implement the 

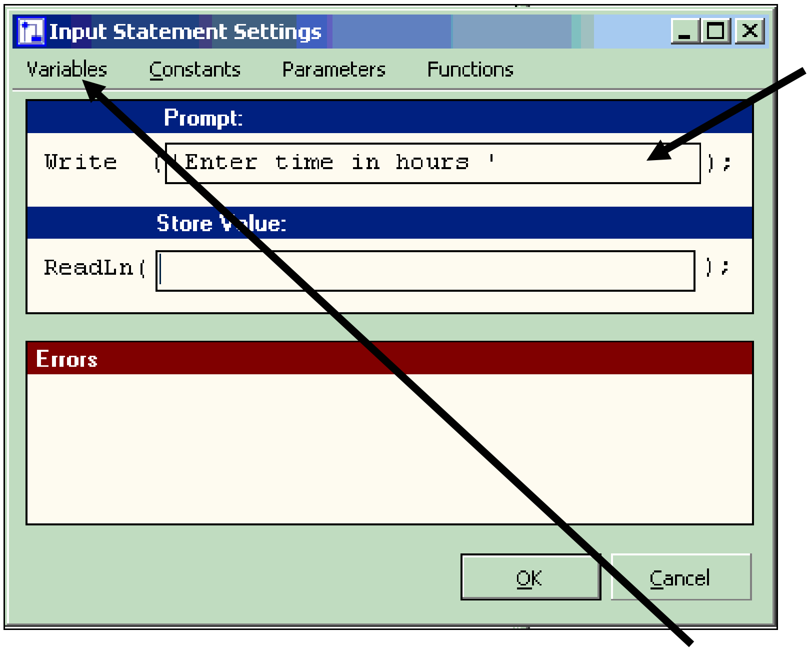
in the flowchart, which is a prompt to the user to enter a value and reading input from the user, namely the time taken by the vehicle to travel the specified distance, do the following:

Click on the *keyboard input* icon found on the icon panel and drag and drop the icon onto the 

positioning it on the line between the existing keyboard input icon and stop point. As you do this, again note the developing diagram in the flowchart pane in the background window:



The **Input Statement Settings** dialogue is displayed.



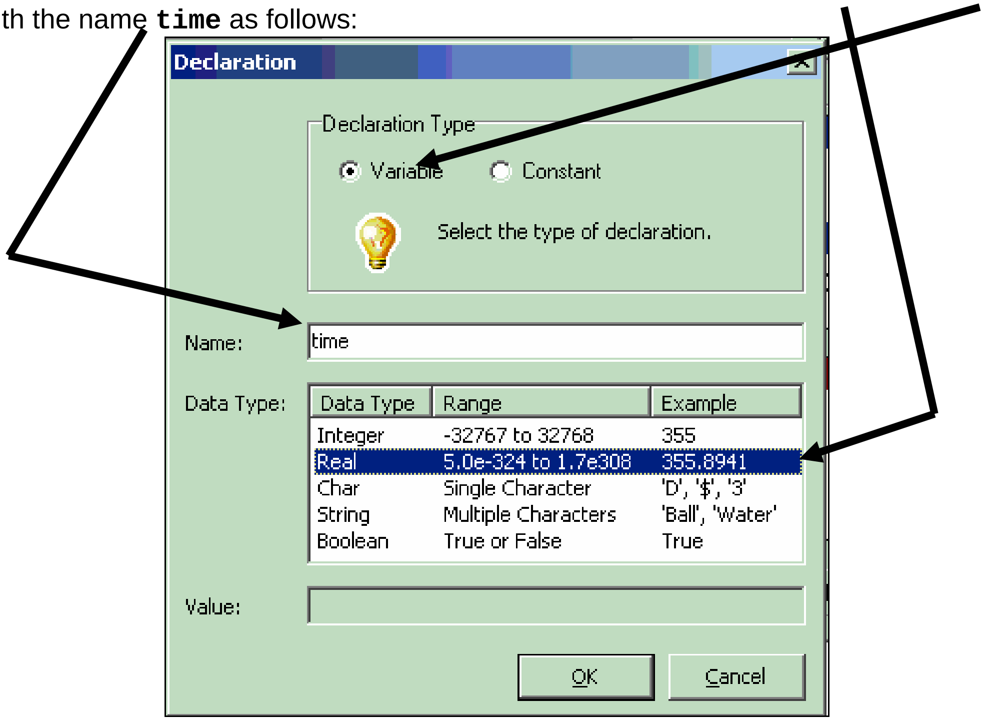
Write ( 'Enter time in hours ' );

ReadLn( );

**Type this text in exactly as shown – including the single quotation marks. This will be displayed to the user to request him/her to enter the time taken for the vehicle to travel the distance**

On the **Input Statement Settings** dialogue, select the **`Variables`** option.

Select **New Variable** from the drop-down menu, and declare a **real variable** with the name **time** as follows:



● Variable ○ Constant

Select the type of declaration.

Name: time

Data Type:

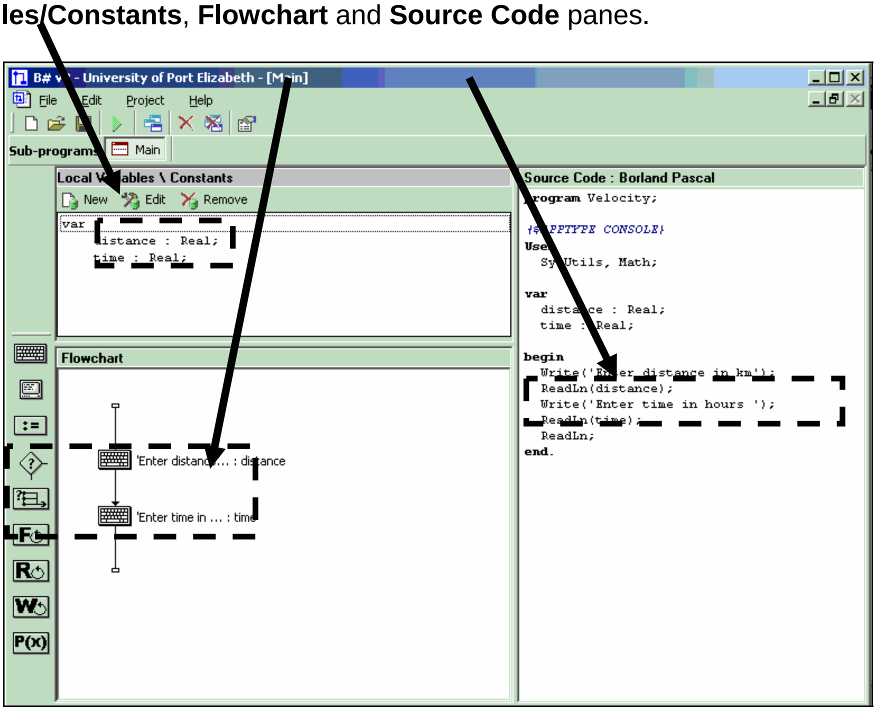
| Data Type | Range | Example |
|---|---|---|
| Integer | -32767 to 32768 | 355 |
| Real | 5.0e-324 to 1.7e308 | 355.8941 |
| Char | Single Character | 'D', '$', '3' |
| String | Multiple Characters | 'Ball', 'Water' |
| Boolean | True or False | True |

Value:

Click on the OK button.

In the **Store Value** field on the displayed **Input Statement Settings** dialogue (like the screen shot on page 10), either select the variable **time** from the list of declared variables, or type in the word **time**, and click on the OK button.

The following window is displayed. Note the contents of the **Local Variables/Constants**, **Flowchart** and **Source Code** panes.


B# v - University of Port Elizabeth - [Main]
File
Edit
Project
Help
Sub-programs
Main
Local Variables \ Constants
New
Edit
Remove
Flowchart
Source Code : Borland Pascal

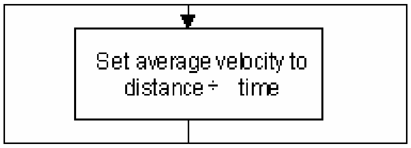
```
var
   distance : Real;
   time : Real;
```

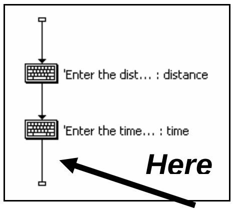
'Enter distance ... : distance

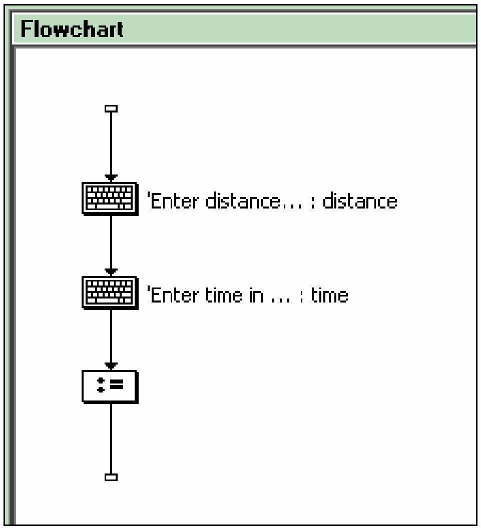
'Enter time in ... : time

```
program Velocity;

{$APPTYPE CONSOLE}
Uses
  SysUtils, Math;

var
  distance : Real;
  time : Real;

begin
  Write('Enter distance in km');
  ReadLn(distance);
  Write('Enter time in hours ');
  ReadLn(time);
  ReadLn;
end.
```

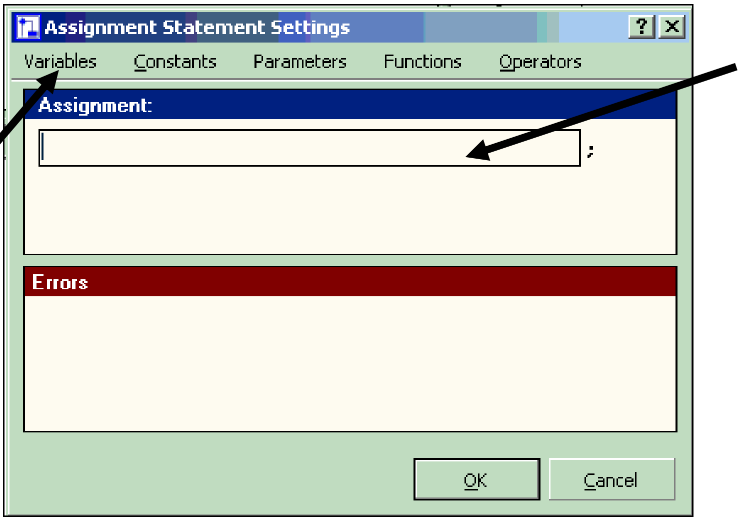
## *Assigning the result of a calculation to a variable*

To implement the  in the flowchart, which is an assignment of the result of the average velocity calculation to a variable, the declaration of a real variable called **ans** (which will record the result of the calculation) is required. This can be done in a way similar to any of two ways illustrated above (see page 7 or once the assignment icon has been added to the flowchart, which will be the way illustrated below).

Drag and drop the *assignment* icon (:=) onto the  positioning it on the line between the lower-most keyboard input icon and stop point. As you do this, again note the developing diagram in the flowchart pane in the background window:



The **Assignment Statement Settings** dialogue is displayed.

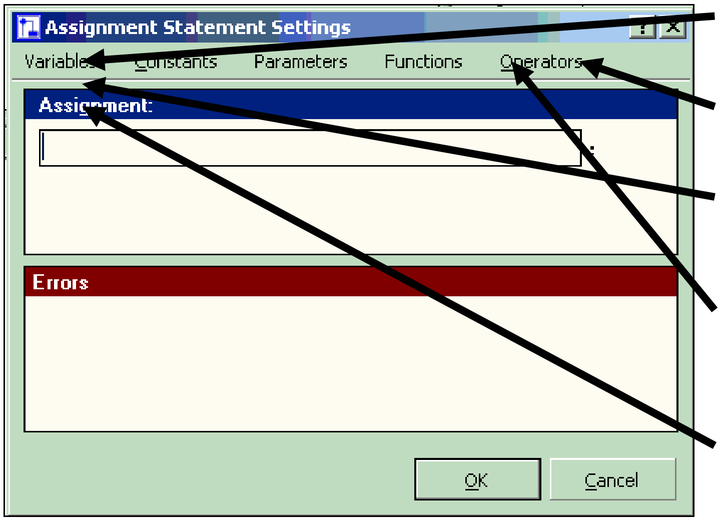


**You may either type in the assignment statement in this field, or allow B# to do it for you. What follows illustrates how B# can make it easy for you.**

In order to declare the new variable **ans**, which will record the value of the average velocity calculation, on the **Assignment Statement Settings** dialogue, select the **Variables** option.

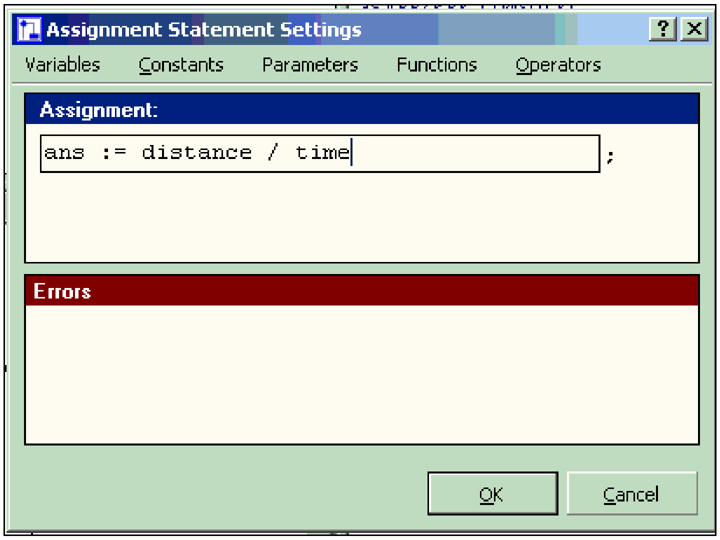
Select **New Variable** from the drop-down menu, and declare a **real variable** with the name **ans** similar to the way that the variable **time** was declared on page 11 (top screen shot).

On the **Assignment Statement Settings** dialogue box, do the following in the correct order:



- Select the **Variables** option, and select the variable **ans**
- Select the **Operators** option and select **:= Assignment**
- Select the **Variables** option, and select the variable **distance**
- Select the **Operators** option, then the **Mathematical** option, and then select **/ Divide**
- Select the **Variables** option, and select the variable **time**

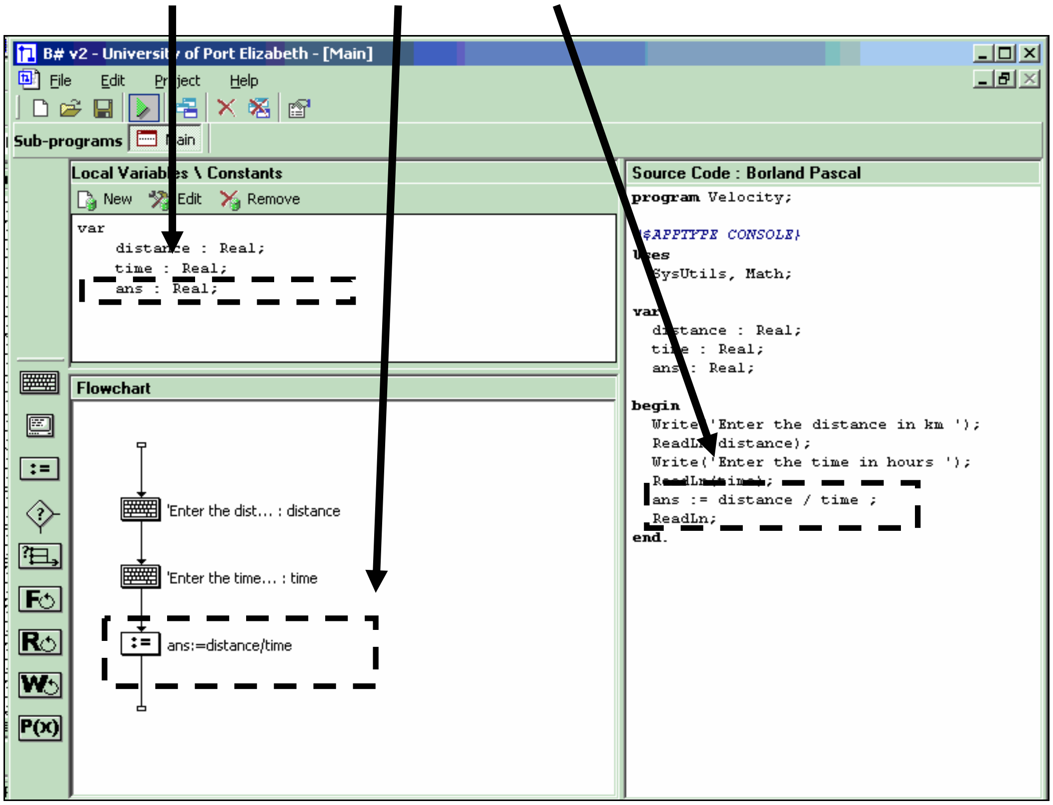
The **Assignment Statement Settings** should now look exactly like the following:



```
ans := distance / time ;
```

Click on the OK button.

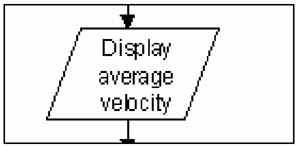
The following window is displayed. Note the contents of the **Local Variables/Constants**, **Flowchart** and **Source Code** panes.


B# v2 - University of Port Elizabeth - [Main]
File
Edit
Project
Help
Sub-programs
Main
Local Variables \ Constants
New
Edit
Remove
Flowchart
'Enter the dist... : distance
'Enter the time... : time
ans:=distance/time
Source Code : Borland Pascal

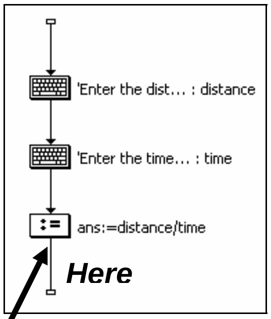
```
var
    distance : Real;
    time : Real;
    ans : Real;
```

```
program Velocity;

{$APPTYPE CONSOLE}
Uses
  SysUtils, Math;

var
  distance : Real;
  time : Real;
  ans : Real;

begin
  Write('Enter the distance in km ');
  ReadLn(distance);
  Write('Enter the time in hours ');
  ReadLn(time);
  ans := distance / time ;
  ReadLn;
end.
```

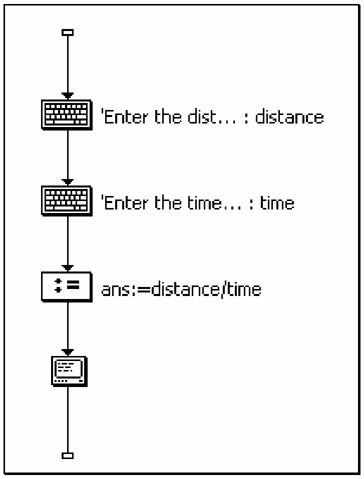
## *Screen Output*



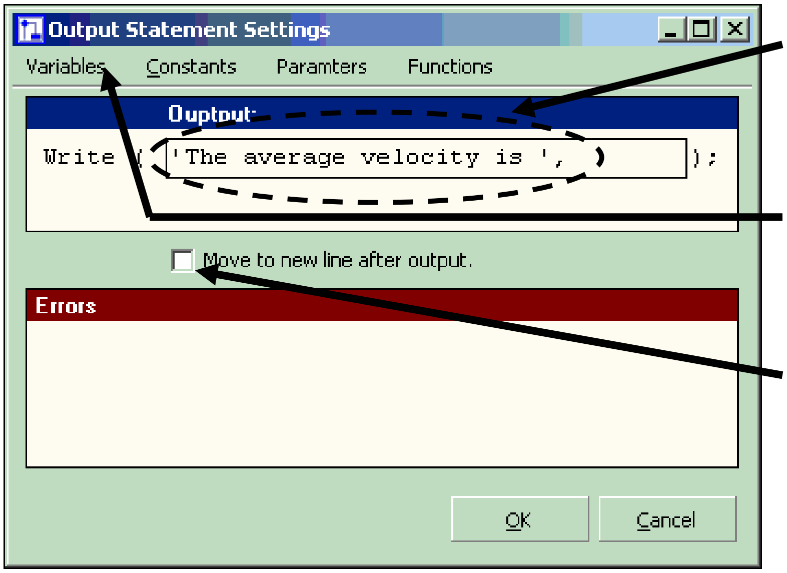
To implement the in the flowchart, which is the output of the contents of a variable on the screen, the following steps are required:



Drag and drop the *screen output* icon (🖥) onto the positioning it on the line between the assignment icon and stop point. As you do this, again note the developing diagram in the flowchart pane in the background window:



The **Output Statement Settings** dialogue is displayed.

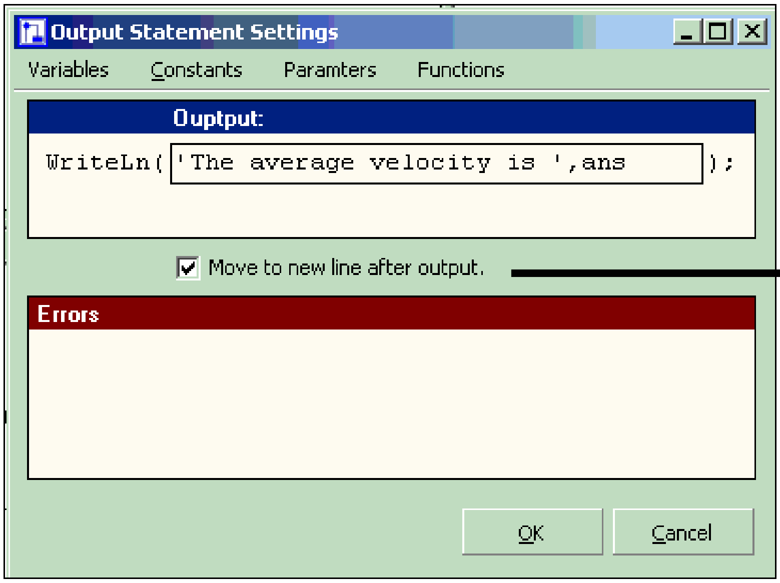


Write ( 'The average velocity is ', );

□ Move to new line after output.

- Type in this text exactly as shown – include the single quotation mark and comma
- Select the **`Variables`** option, and select the variable **`ans`**
- Check this box

The **Output Statement Settings** dialogue box should now look exactly like the following:



```
WriteLn( 'The average velocity is ',ans );
```

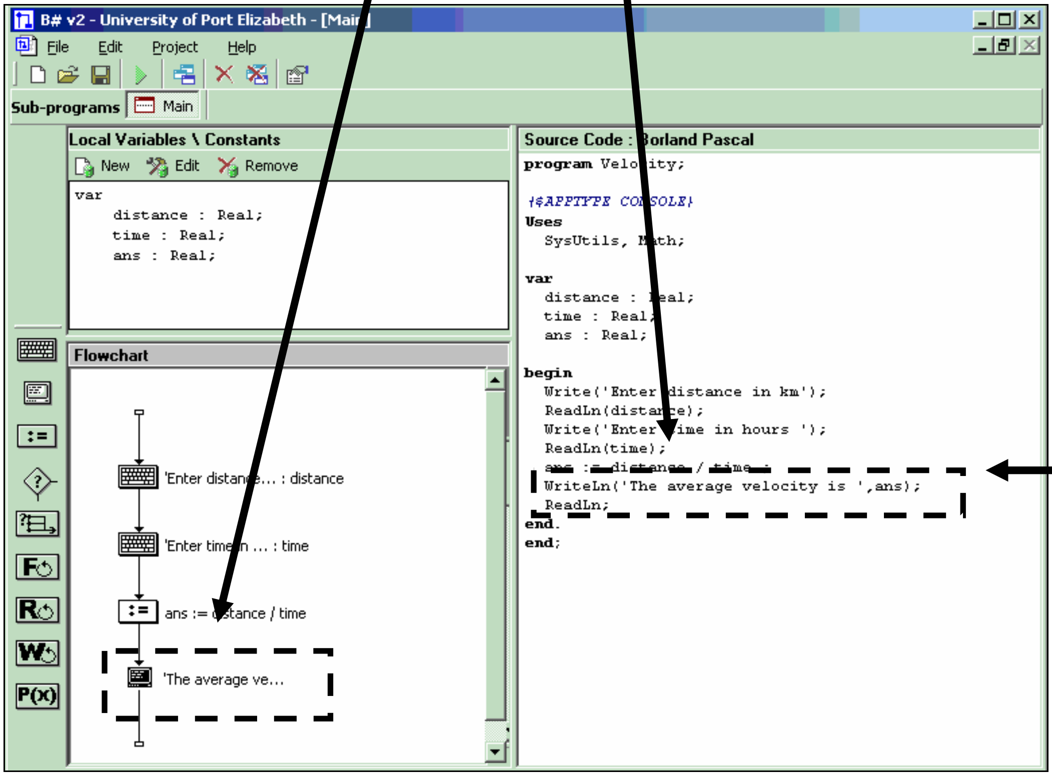
☑ Move to new line after output.

*Note*: Checking this box results in a **`writeln`** statement, and not a **`write`** statement

Click on the OK button.

The following window is displayed.

Note the contents of the **Flowchart** and **Source Code** panes.


B# v2 - University of Port Elizabeth - [Main]
File
Edit
Project
Help
Sub-programs
Main
Local Variables \ Constants
New
Edit
Remove
Flowchart
'Enter distance… : distance
'Enter time in … : time
ans := distance / time
'The average ve…
Source Code : Borland Pascal

```
var
    distance : Real;
    time : Real;
    ans : Real;
```

```
program Velocity;

{$APPTYPE CONSOLE}
Uses
  SysUtils, Math;

var
  distance : Real;
  time : Real;
  ans : Real;

begin
  Write('Enter distance in km');
  ReadLn(distance);
  Write('Enter time in hours ');
  ReadLn(time);
  ans := distance / time ;
  WriteLn('The average velocity is ',ans);
  ReadLn;
end.
end;
```

To save the program, click on the 🖫 icon found along the top of the main window of B# (see page 2).

## How to test a B# program

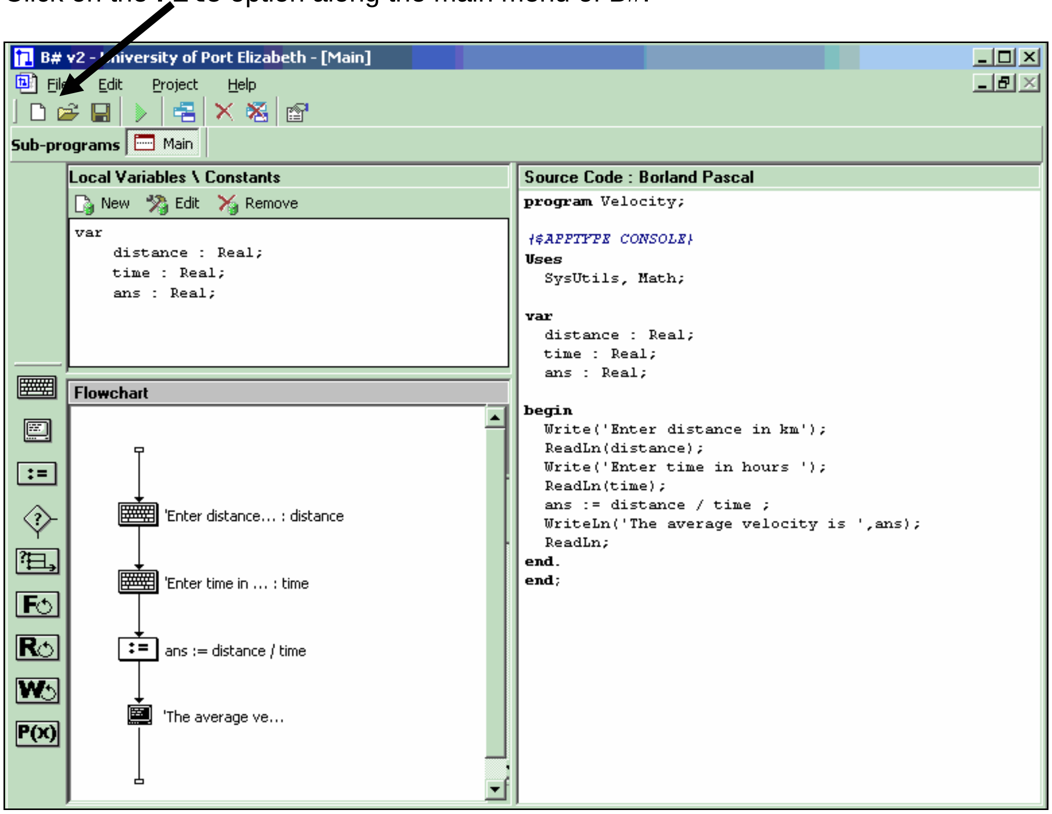
Run the **Velocity** program – if you have forgotten how to run a B# program, refer to page 4.

Test your solution with a distance of 1045.67km and time of 8.75 hours.  Write the answer exactly as displayed in the DOS window in the place below.

## How to export the source code of a B# program for use in Delphi™ Enterprise

Because you will be expected to also program solutions using the Delphi™ Enterprise programming environment, you can make use of B#'s facility to export correct code (since B#'s code is ***ALWAYS*** correct) for use in Delphi.  You may then load the correct code into Delphi and make any necessary changes to create a new program.  This will hopefully result in you producing ***correct*** programs ***more often*** at a ***greater speed*** than if you were not using B#.

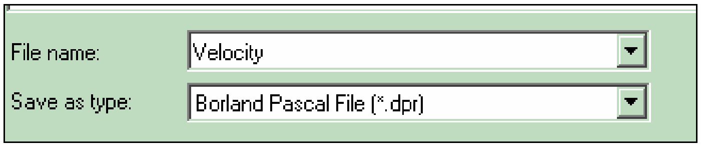
Click on the **File** option along the main menu of B#.


B# v2 - University of Port Elizabeth - [Main]
File
Edit
Project
Help
Sub-programs
Main
Local Variables \ Constants
New
Edit
Remove
Flowchart
'Enter distance... : distance
'Enter time in ... : time
ans := distance / time
'The average ve...
F
R
W
P(X)
Source Code : Borland Pascal

```
var
   distance : Real;
   time : Real;
   ans : Real;
```

```
program Velocity;

{$APPTYPE CONSOLE}
Uses
  SysUtils, Math;

var
  distance : Real;
  time : Real;
  ans : Real;

begin
  Write('Enter distance in km');
  ReadLn(distance);
  Write('Enter time in hours ');
  ReadLn(time);
  ans := distance / time ;
  WriteLn('The average velocity is ',ans);
  ReadLn;
end.
end;
```

Then select the option **Export Code**.  You will be required to specify the folder in which the program must be saved as well as give the program a name (see example below).  Store this program as the name **Velocity** under your usercode in the **C:\Temp** folder.



File name: Velocity

Save as type: Borland Pascal File (*.dpr)

### *How to exit from B#*

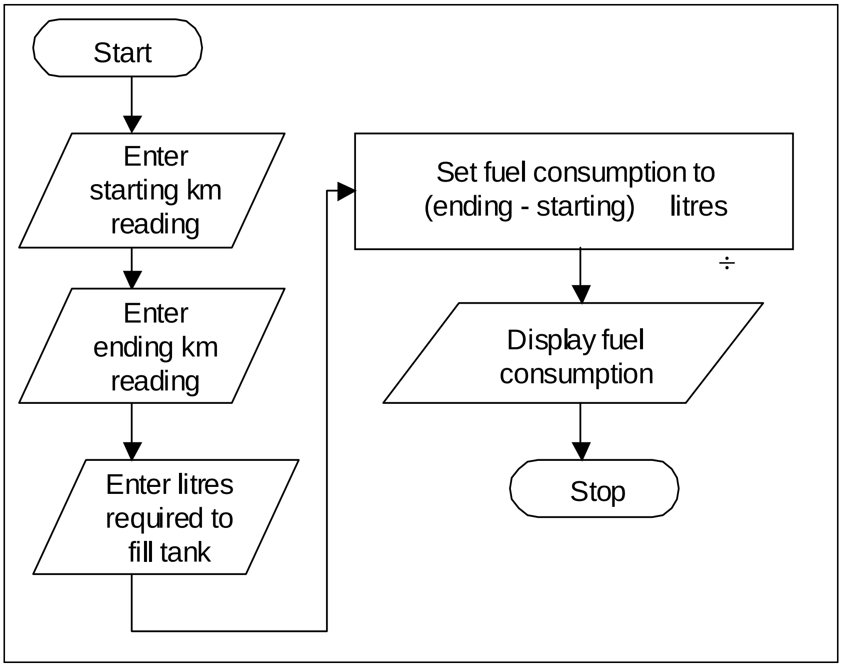
In order to exit from B#, you may either close B#'s window, or click on the **`File`** option along the main menu of B#, and select the **`Exit`** option.

## ***Task 2 : Step-by-step introduction to Delphi™ Enterprise***

Consider the following problem.

> At the beginning of a journey the reading on a car's odometer is S kilometers and the tank is full. After the journey the reading is F kilometers and it takes L litres to fill the tank.
>
> Write a program that obtains values for S, F and L from the user and computes and displays the rate of fuel consumption as kilometers per litre.
>
> Glossary of terms:
>
> Odometer : an instrument that measures the total number of kilometers travelled.
>
> Consumption : usage

A flowchart corresponding to the solution for the above problem is:


Start
Enter starting km reading
Enter ending km reading
Enter litres required to fill tank
Set fuel consumption to (ending - starting) litres
÷
Display fuel consumption
Stop

The corresponding pseudo-code is:

| ***Constants*** |
|---|
| |

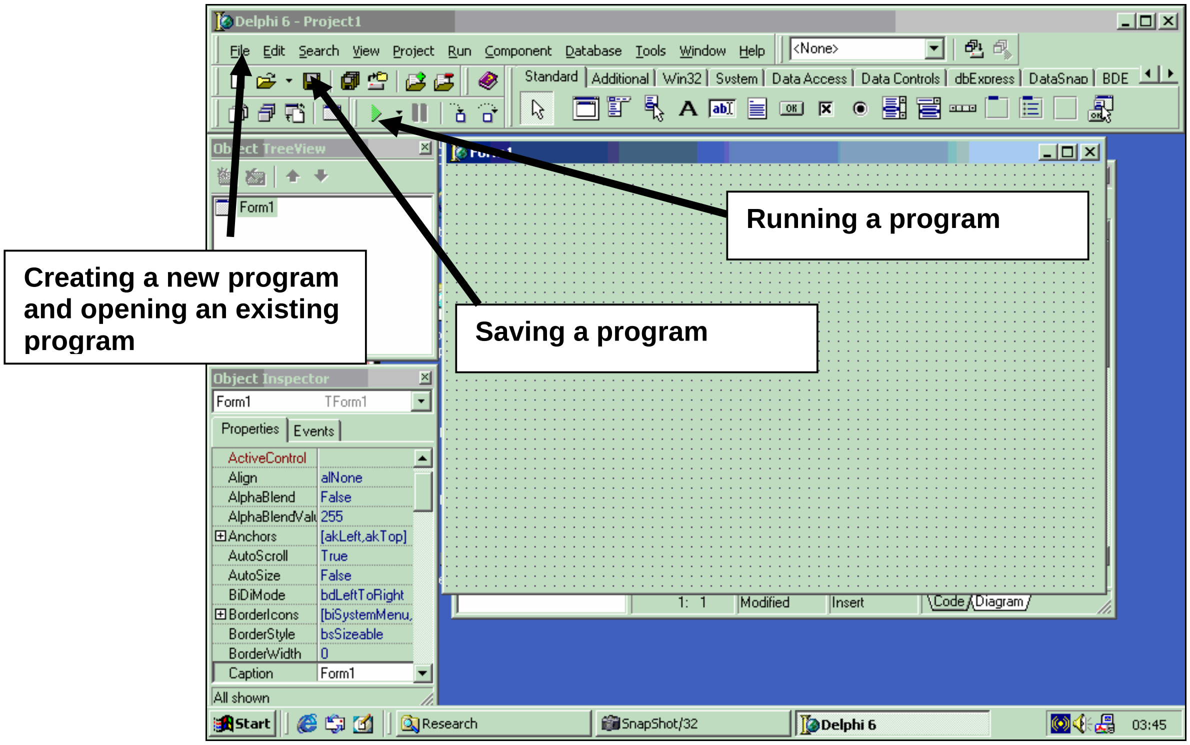
| ***Inputs*** | ***Outputs*** |
|---|---|
| Starting km reading (S) | Fuel consumption (consumption) |
| Ending km reading (F) | |
| Litres to fill tank (L) | |

| ***Algorithm*** |
|---|
| 1. Get the initial odometer reading (S) of the vehicle. |
| 2. Get the final odometer reading (F) of the vehicle. |
| 3. Get the amount of litres (L) required to fill the tank. |
| 4. Calculate the fuel consumption<br>(consumption becomes (F – S) ÷ L) |
| 5. Display the consumption and stop processing. |

The following guides you in developing the Delphi solution to this problem, which looks similar to the velocity problem developed in B# - check this by comparing the above flowchart and/or pseudo-code with those on pages 6 and/or 7 (note that there is just an extra input variable and the calculation differs slightly – the process remains similar).

Activate the Delphi™ Enterprise program from your desktop by double-clicking on the Delphi icon. (You might be prompted by the system to register for Delphi. At this prompt, select that you will do it at a later time, click on the ***Next*** button, and then the ***Exit*** button*)*.

The following window is displayed. The most important concepts have been highlighted.



| | |
|---|---|
| ActiveControl | |
| Align | alNone |
| AlphaBlend | False |
| AlphaBlendVal | 255 |
| ⊞Anchors | [akLeft,akTop] |
| AutoScroll | True |
| AutoSize | False |
| BiDiMode | bdLeftToRight |
| ⊞BorderIcons | [biSystemMenu, |
| BorderStyle | bsSizeable |
| BorderWidth | 0 |
| Caption | Form1 |

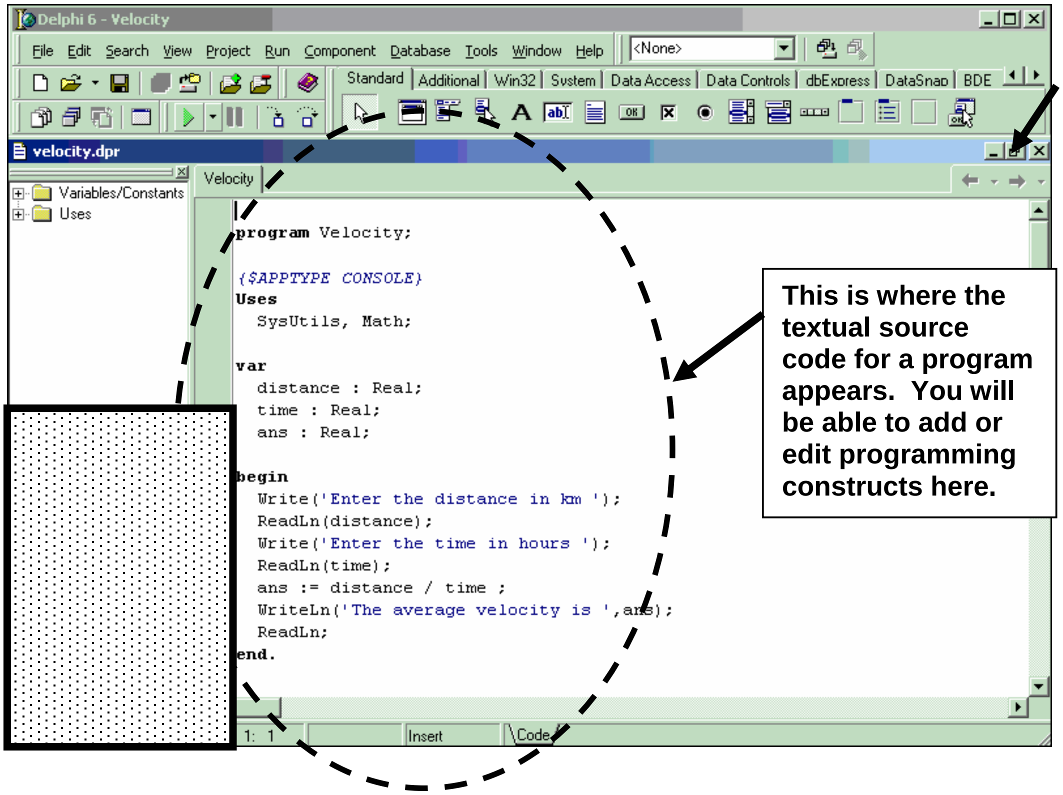
**Creating a new program and opening an existing program**

**Saving a program**

**Running a program**

## *How to open an existing Delphi program*

On Delphi's main menu, select the option **`File`**, then **`Open`** and locate the file that you earlier exported from B#, namely **`Velocity.dpr`** under your usercode in the **`C:\Temp`** folder. The following window should then be displayed.


Delphi 6 - Velocity
File Edit Search View Project Run Component Database Tools Window Help
<None>
Standard Additional Win32 System Data Access Data Controls dbExpress DataSnap BDE
velocity.dpr
Variables/Constants
Uses
Velocity
1: 1
Insert
Code

```
program Velocity;

{$APPTYPE CONSOLE}
Uses
  SysUtils, Math;

var
  distance : Real;
  time : Real;
  ans : Real;

begin
  Write('Enter the distance in km ');
  ReadLn(distance);
  Write('Enter the time in hours ');
  ReadLn(time);
  ans := distance / time ;
  WriteLn('The average velocity is ',ans);
  ReadLn;
end.
```

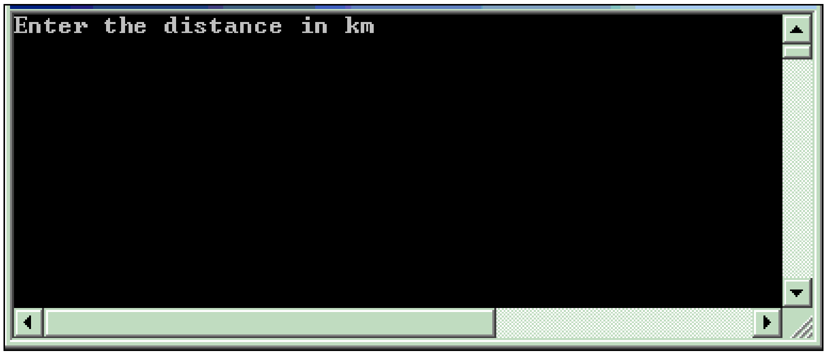
**Click here to maximize the source code pane.**

**This is where the textual source code for a program appears. You will be able to add or edit programming constructs here.**

(*If ever the **Object Inspector** window is displayed, usually in this area, please close it to avoid any confusion*).

## *How to run a Delphi program*

Click on the icon found along the top of the main window of Delphi (see page 19). The following DOS window appears (just like when running a program in B#):

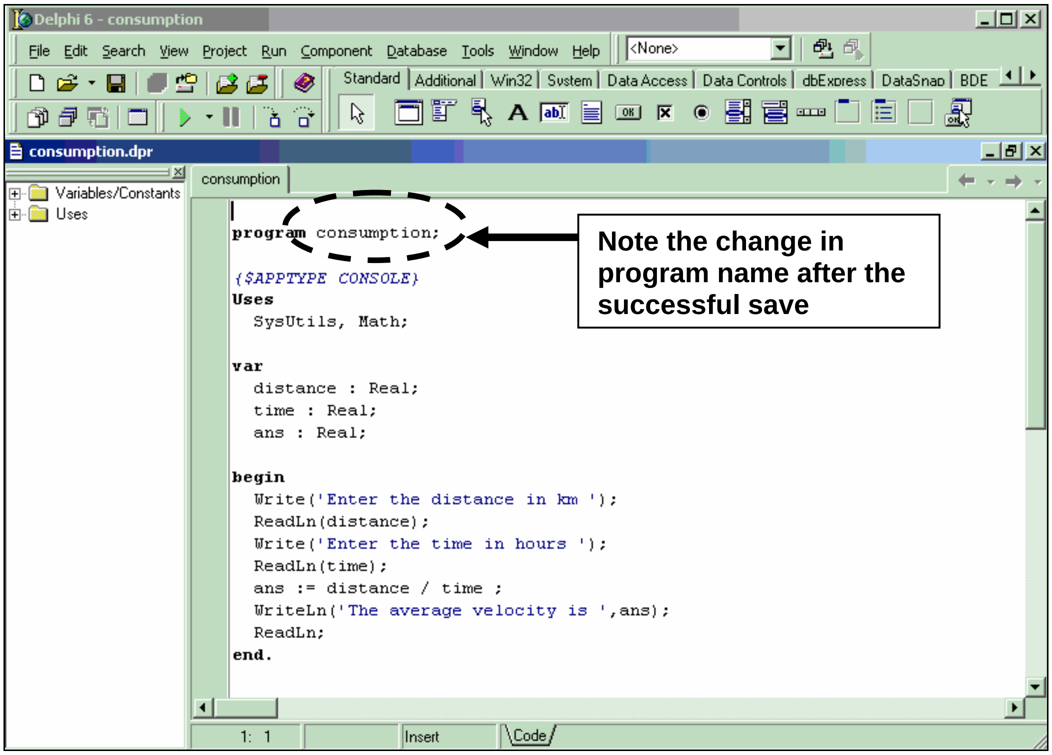


```
Enter the distance in km
```

Enter a distance of 1045.67km and time of 8.75 hours. Compare the answer displayed with your answer recorded on page 17 – they should be identical. Close the DOS window.

## *How to save an existing Delphi program under another name (make a copy of a program)*

On Delphi's main menu, select the option **`File`**, then **`Save As`** and give the file the name **`consumption.dpr`**, saving it under your usercode in the **`C:\Temp`** folder. The following window should then be displayed.

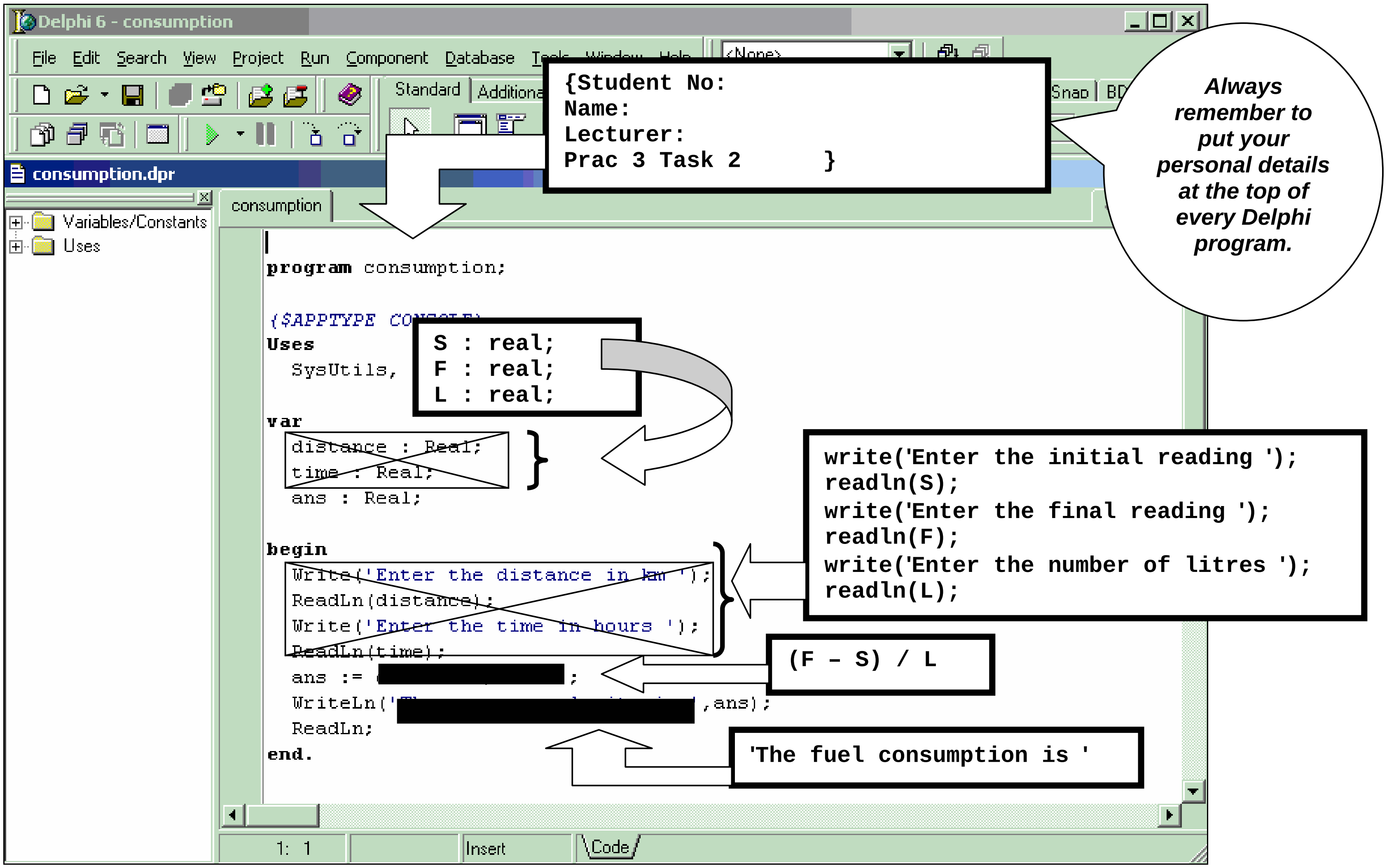

Delphi 6 - consumption
File Edit Search View Project Run Component Database Tools Window Help
<None>
Standard Additional Win32 System Data Access Data Controls dbExpress DataSnap BDE
consumption.dpr
Variables/Constants
Uses
consumption
1: 1
Insert
Code

```
program consumption;

{$APPTYPE CONSOLE}
Uses
  SysUtils, Math;

var
  distance : Real;
  time : Real;
  ans : Real;

begin
  Write('Enter the distance in km ');
  ReadLn(distance);
  Write('Enter the time in hours ');
  ReadLn(time);
  ans := distance / time ;
  WriteLn('The average velocity is ',ans);
  ReadLn;
end.
```

**Note the change in program name after the successful save**

## *How to adapt the code in a Delphi program*

In order to adapt the source code to cater for the fuel consumption problem, make the necessary changes exactly as illustrated on the following page.



```
{Student No:
Name:
Lecturer:
Prac 3 Task 2       }
```

*Always remember to put your personal details at the top of every Delphi program.*

```
program consumption;

{$APPTYPE CONSOLE}
Uses
  SysUtils,

var
  ~~distance : Real;~~
  ~~time : Real;~~
  ans : Real;

begin
  ~~Write('Enter the distance in km ');~~
  ~~ReadLn(distance);~~
  ~~Write('Enter the time in hours ');~~
  ~~ReadLn(time);~~
  ans := [redacted];
  WriteLn('[redacted],ans);
  ReadLn;
end.
```

```
S : real;
F : real;
L : real;
```

```
write('Enter the initial reading ');
readln(S);
write('Enter the final reading ');
readln(F);
write('Enter the number of litres ');
readln(L);
```

```
(F – S) / L
```

```
'The fuel consumption is '
```

### *How to save a program in Delphi*

Click on the 💾 icon found along the top of the main window of Delphi (see page 19). If it is the first time that the project is being saved, you will be required to specify the folder in which the program must be saved as well as give the program a name.

### *How to test a Delphi program*

Run the **`Consumption`** program – if you have forgotten how to run a Delphi program, refer to page 20.

Test your solution with the following values:

| | |
|---|---|
| Initial reading | 40341 |
| Final reading | 40723 |
| Litres | 63 |

Write the answer exactly as displayed in the DOS window in the place below.

### *How to print the source code of a Delphi program*

On Delphi's main menu, select the option **`File`**, then **`Print`**, and click the OK button.

### *How to exit from Delphi*

On Delphi's main menu, select the option **`File`**, then **`Exit`**, or close Delphi's window.

## ***Task 3 : B# solution***

a) In B#, develop a program that computes and displays the result of the equation given below (*p* and *q* are both real numbers and are user inputs). *Note: The solution can always assume that q is never 7*.

$$\frac{5(2+p)}{7-q}$$

b) Test your solution with the following values:

| | |
|---|---|
| p | *5.3726* |
| q | *12.30619* |

Write the result of the equation exactly as displayed in the DOS window in the place below.

## *Task 4 : Delphi solution*

a) In Delphi, develop a program that computes and displays the result of the equation given below ($y$ is a real number and a user input). *Note: The solution can always assume that y is never -6*.

$$\frac{5y^2-8y+3}{12+2y}$$

b) Test your solution with the following value for $y$, namely 7.40325.

Write the result of the equation exactly as displayed in the DOS window in the place below.

## *Task 5 : Using either B# or Delphi (your personal choice)*

a) A pre-school wishes to go on an outing to Bayworld. Parents will be requested to provide transport for 4 children per vehicle. Given the number of children in the pre-school as input, compute and display the number of vehicles that will each have 4 children. Also compute and display the number of children that will be travelling in a vehicle that does not have 4 children in it. For example, if there are 75 children at the pre-school, there will be 18 vehicles with 4 children, and the remaining 3 children will be alone in a vehicle. *Hint: Constants are treated in much the same way as variables regarding declaration and usage in B#.*

b) Test your solution for the following number of children requiring transport to Bayworld. Write the answers exactly as displayed in the DOS window in the appropriate places below.

| ***No. children going*** | ***No. vehicles with 4 children*** | ***No. children remaining*** |
|---|---|---|
| **124** | | |
| **2** | | |
| **17** | | |

c) Make a printout of the source code for this program.

## Optional Practical Tasks

### ***Task 6***

The speed of light, *c*, is approximately 3 x $10^8$ meters per second. Develop a B#/Delphi project that prompts the user to input a time in seconds and then displays the distance that light travels during that time period.

### ***Task 7***

Develop B#/Delphi projects to solve Tasks 1 and 2 of Practical 1.

## *Other useful things that you can do in B#*

### *Debugging a B# Project*

B# detects any errors that you make as you adapt the iconic program. B# will only allow you to continue once you have corrected the error that you have made. In order to determine what the error is that you have made, carefully read the list of errors displayed by B#. You may at any time remove icons from the B# iconic program and then reinsert them somewhere else between the start and stop points.

### *Printing the source code of a B# Project*

B# does not provide a facility for you to print the correct source code generated from the corresponding flowchart. You may, however, export the source code for printing from within Delphi.

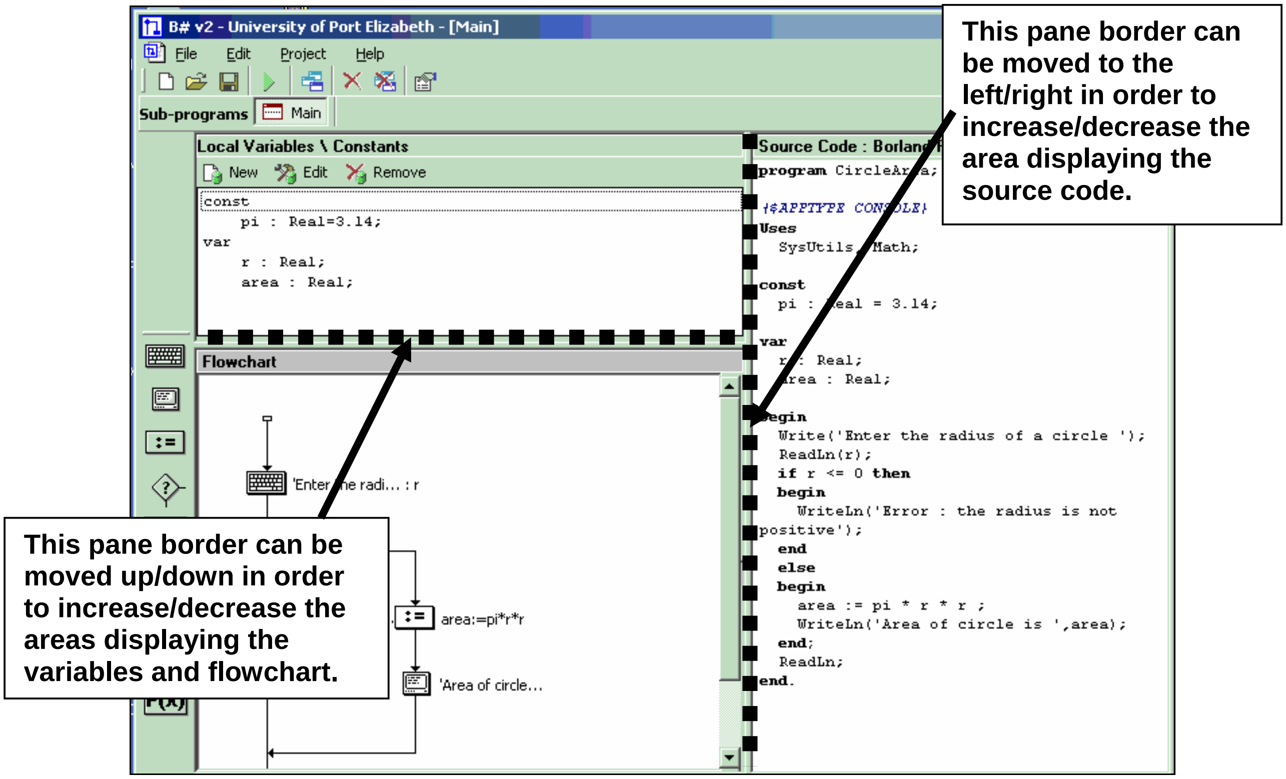
### *Changing an existing B# project*

Click on the 📂 icon found along the top of the main window of B# (page 2). Locate and open the required project. Make your changes to the iconic program according to your amended plan (flowchart or pseudocode).

### *Entering Pascal syntax directly into an iconic program*

In places like the fields shown on the dialogue boxes on pages 9 & 10 (**`Store Value`**), 13 (**`Assignment`**), and 15 (**`Output`**), the correct Pascal syntax may be directly entered without you having to select the appropriate variables and operators.

*Increasing the display area of a B# pane*



```
const
    pi : Real=3.14;
var
    r : Real;
    area : Real;
```

```
program CircleArea;

{$APPTYPE CONSOLE}
Uses
  SysUtils, Math;

const
  pi : Real = 3.14;

var
  r : Real;
  area : Real;

begin
  Write('Enter the radius of a circle ');
  ReadLn(r);
  if r <= 0 then
  begin
    WriteLn('Error : the radius is not
positive');
  end
  else
  begin
    area := pi * r * r ;
    WriteLn('Area of circle is ',area);
  end;
  ReadLn;
end.
```

**This pane border can be moved to the left/right in order to increase/decrease the area displaying the source code.**

**This pane border can be moved up/down in order to increase/decrease the areas displaying the variables and flowchart.**

# WRA101 : Practical 4

## Week 5 : 11 - 13 March 2003

*Sections Covered*

`IF`, nested `IF` and `CASE` statements (pp 128 – 140)

*Objectives*

- To understand the use of the `IF` and `IF…ELSE` statement.
- To understand simple and complex logical expressions.

*Practical Preparation*

- Using either a flowchart or pseudocode, plan the solution for each of the compulsory practical tasks.

***YOU WILL NOT RECEIVE ASSISTANCE DURING THE PRACTICAL SESSION IF YOU DO NOT HAVE YOUR PREPARATION WITH YOU. YOU WILL BE REQUIRED TO PRESENT IT TO THE STUDENT ASSISTANT BEFORE RECEIVING ASSISTANCE.***

## Compulsory Practical Tasks

### *Task 1 : Using B# to solve a problem*

***In B#***, write a program that accepts a student's mark (as an integer which is assumed to be a maximum of 75) and displays the mark in percentage form. The program must also display whether the student has passed or not. The pass mark is 50%.

Write down the output of your program (exactly as it appears on the screen) for the following input values (marks):

| ***Mark (out of 75)*** | ***Program output:*** |
|---|---|
| 65 | |
| 35 | |

### *Task 2 : Using Delphi to solve a problem*

A ***Delphi program*** is required to determine whether a student may continue with WRA102 or not. The program requires two marks, one for WRA101 and one for WRU101, each out of a maximum of 100. If both of these marks are at least 50, the student may continue with WRA102, otherwise he/she may not. *Hint: Adapt the solution for Task 1.*

Write down the output of your program (exactly as it appears on the screen) for the following input values:

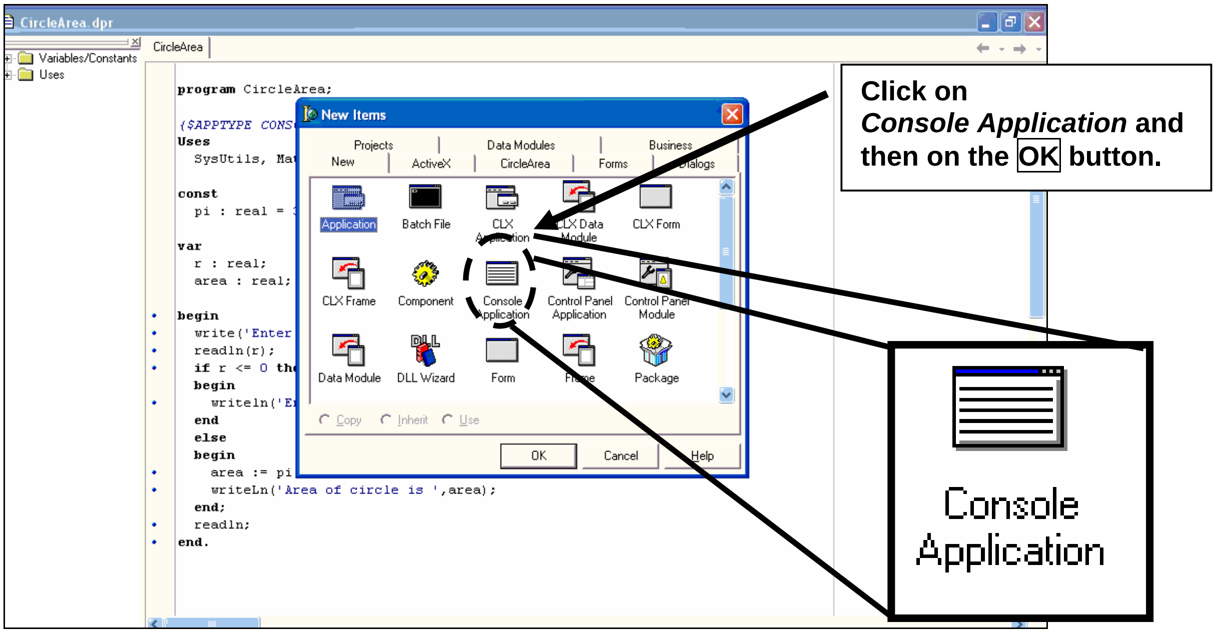
| ***WRA101 Mark*** | ***WRU101  Mark*** | ***Program output:*** |
|---|---|---|
| 55 | 76 | |
| 40 | 55 | |
| 47 | 48 | |

## ***Task 3 : How to create and save a new Delphi program***

Consider the following problem.

> Write a program that accepts values for *x* and *y* and, only where *y* is non-zero, determines the answer to the equation
>
> $$\frac{(y + x^2)}{y}$$
>
> The program must display an appropriate error message if *y* is 0.

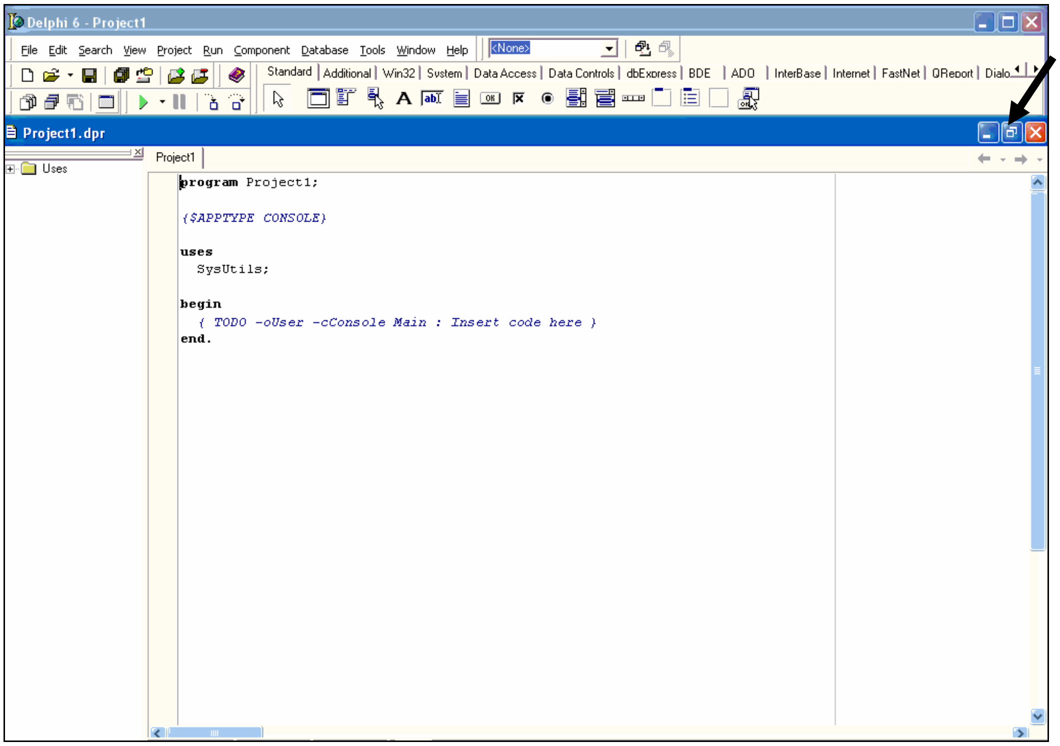
On Delphi's main menu, select the option **`File`**, then **`New`** and **`Other`**.  The following should be displayed.



```
program CircleArea;

{$APPTYPE CONS
Uses
  SysUtils, Mat

const
  pi : real = 3

var
  r : real;
  area : real;

begin
  write('Enter
  readln(r);
  if r <= 0 the
  begin
    writeln('Er
  end
  else
  begin
    area := pi
    writeLn('Area of circle is ',area);
  end;
  readln;
end.
```

**Click on *Console Application* and then on the OK button.**

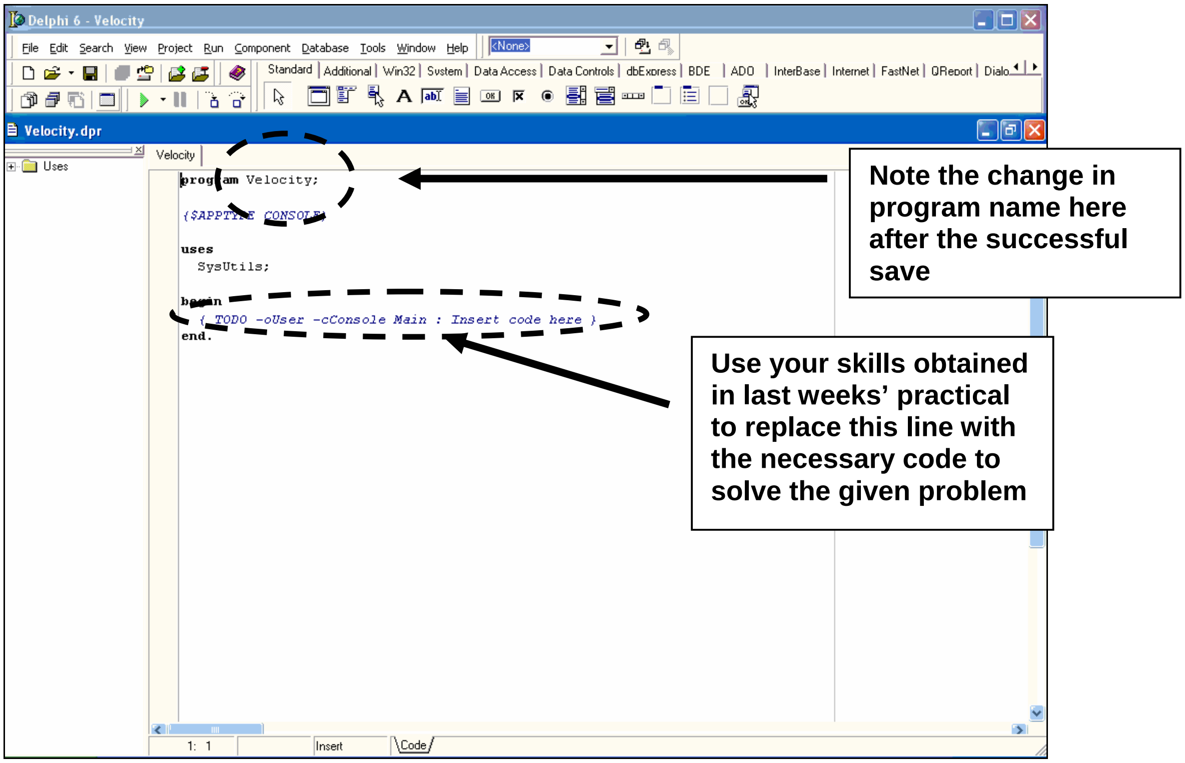
The following window is displayed.


Delphi 6 - Project1
File Edit Search View Project Run Component Database Tools Window Help
<None>
Standard Additional Win32 System Data Access Data Controls dbExpress BDE ADO InterBase Internet FastNet QReport Dialo
Project1.dpr
Uses
Project1

```
program Project1;

{$APPTYPE CONSOLE}

uses
  SysUtils;

begin
  { TODO -oUser -cConsole Main : Insert code here }
end.
```

**Click here to maximize the source code pane.**

To save the program, on Delphi's main menu, select the option **`File`**, then **`Save As`** and give the file a name, saving it under your usercode in the **`C:\Temp`** folder. The following window should then be displayed.


Delphi 6 - Velocity
File Edit Search View Project Run Component Database Tools Window Help
<None>
Standard Additional Win32 System Data Access Data Controls dbExpress BDE ADO InterBase Internet FastNet QReport Dialo
Velocity.dpr
Uses
Velocity
1: 1
Insert
Code

```
program Velocity;

{$APPTYPE CONSOLE}

uses
  SysUtils;

begin
  { TODO -oUser -cConsole Main : Insert code here }
end.
```

**Note the change in program name here after the successful save**

**Use your skills obtained in last weeks' practical to replace this line with the necessary code to solve the given problem**

Write down the output of your program (exactly as it appears on the screen) for the following input values:

| ***x*** | ***Y*** | ***Program output:*** |
|---|---|---|
| 2.157 | 0 | |
| 6.9017 | 3.0345 | |

## ***<u>Task 4</u>***

The Department of Computer Science and Information Systems sells diskettes to students. The cost of each diskette depends on the number of diskettes purchased.

| ***No. of diskettes purchased*** | ***Cost per diskette*** |
|---|---|
| 1 | R5.67 |
| 2 | R4.98 |
| 3 | R4.54 |
| Otherwise | R4.23 |

Write a ***<u>B# program</u>*** that accepts the number of diskettes purchased by a student and displays the total cost of the diskettes.

Write down the output of your program (exactly as it appears on the screen) for the following input values:

| ***Diskettes bought*** | ***Program output:*** |
|---|---|
| 2 | |
| 8 | |

## ***<u>Task 5</u>***

Write a *<u>Delphi program</u>* that reads in the lengths of 3 lines that form the sides of a triangle, and classifies the triangle according to the following:

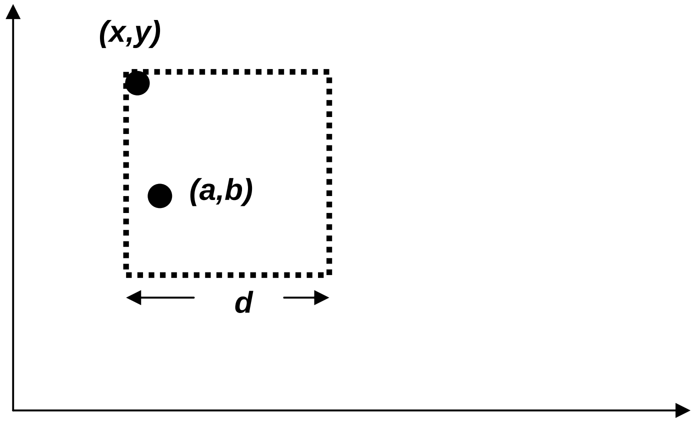
- equilateral (“gelyksydig”) – all sides are equal in length
- isosceles (“gelykbenig”) – two of the sides are the same length
- scalene (“ongelyksydig”) – no equal sides

(*Hint: Refer to* http://www.math.com/school/subject3/lessons/S3U2L2GL.html *if you wish to be reminded what each of these triangles look like*)

Appropriate messages should be displayed.

## ***Task 6***

Using ***either B# or Delphi***, write a program which reads in the co-ordinates (*x,y*) for the top left hand corner of a square, the co-ordinates for another point (*a, b*) and the length of the side of the square (*d*).



The program must determine whether or not the point (*a,b*) is inside the square or not. An appropriate message must be displayed.

## Optional Practical Tasks

## ***Task 7***

Write a program which asks the user for a student's mark (as an integer), and then displays one of the following three messages according to the mark he has obtained:

| | | |
|---|---|---|
| **0 – 49** : Fail | **50 – 74** : Pass | **75 – 100** : Pass with distinction |

An appropriate error message should be displayed for any other value entered.

## ***Task 8***

Write a program that accepts two integers and displays the larger of the two. If they are the same, an appropriate message must be displayed.

## ***Task 9***

Write a program that accepts a student's symbol, and then displays an equivalent mark for that symbol, according to the following table:

| | |
|---|---|
| A | 90 |
| B | 70 |
| C | 60 |
| D | 50 |
| E | 40 |

If the user enters any invalid symbol, an appropriate message must be displayed.

## ***Task 10***

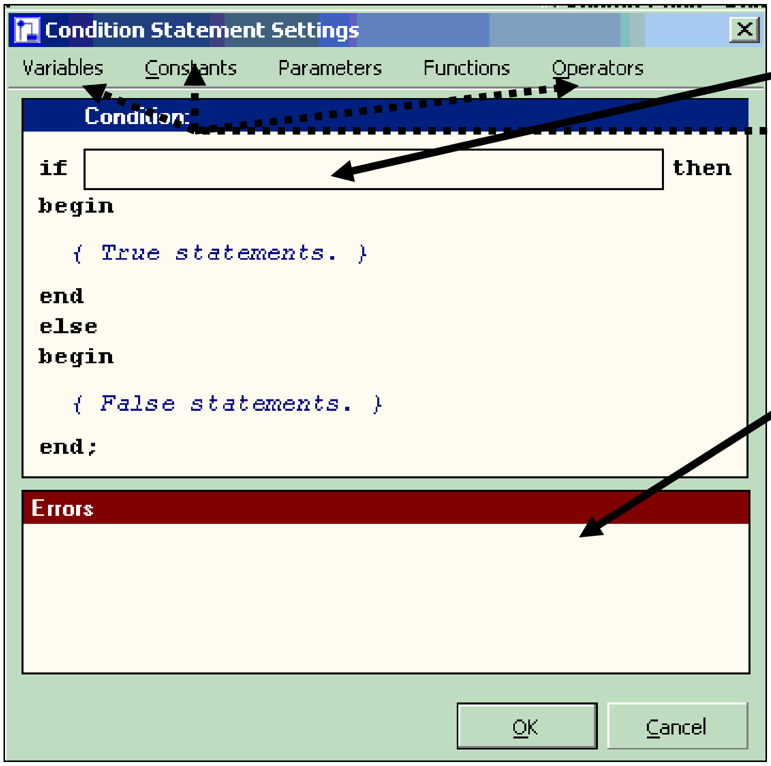
Write a program that accepts a temperature, and then displays a message according to the following table:

| | |
|---|---|
| 0 – $10^{0}$C | Very Cold |
| 11 – $16^{0}$C | Cold |
| 17 – $28^{0}$C | Warm |
| 29 – $45^{0}$C | Very Hot |

# The IF and CASE programming constructs in B#

### IF programming construct

The **IF** programming construct is represented by the *condition* (?) icon found on B#'s icon panel. Once this icon has been dragged and dropped onto the flowchart, you will be prompted to complete the ***Condition Statement Settings*** dialogue:



```
if                                    then
begin
   { True statements. }
end
else
begin
   { False statements. }
end;
```

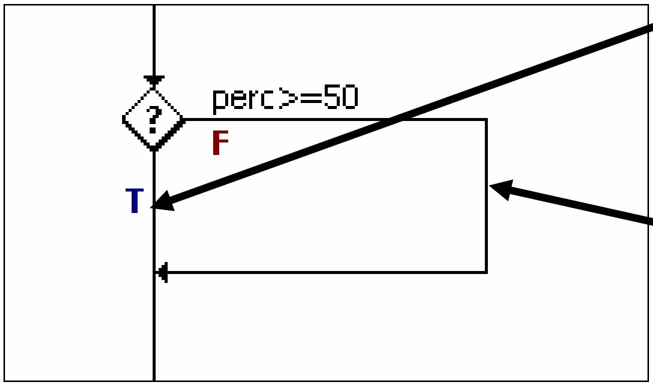
**Complete the condition by making your choice from the listed variables, constants, and relational and logical operators. When in doubt as to precedence or operations, make use of ( and ).**

***__You may also type the condition in directly__*. B# will check that it is correct. In the case of any errors, messages to this affect will appear here.**

**The condition may be simple or complex.**

**Do not concern yourself with the statements to be contained in the TRUE and FALSE branches of the condition at this stage.**

Once the flowchart has been updated to show the **IF** programming construct, it will resemble the following:

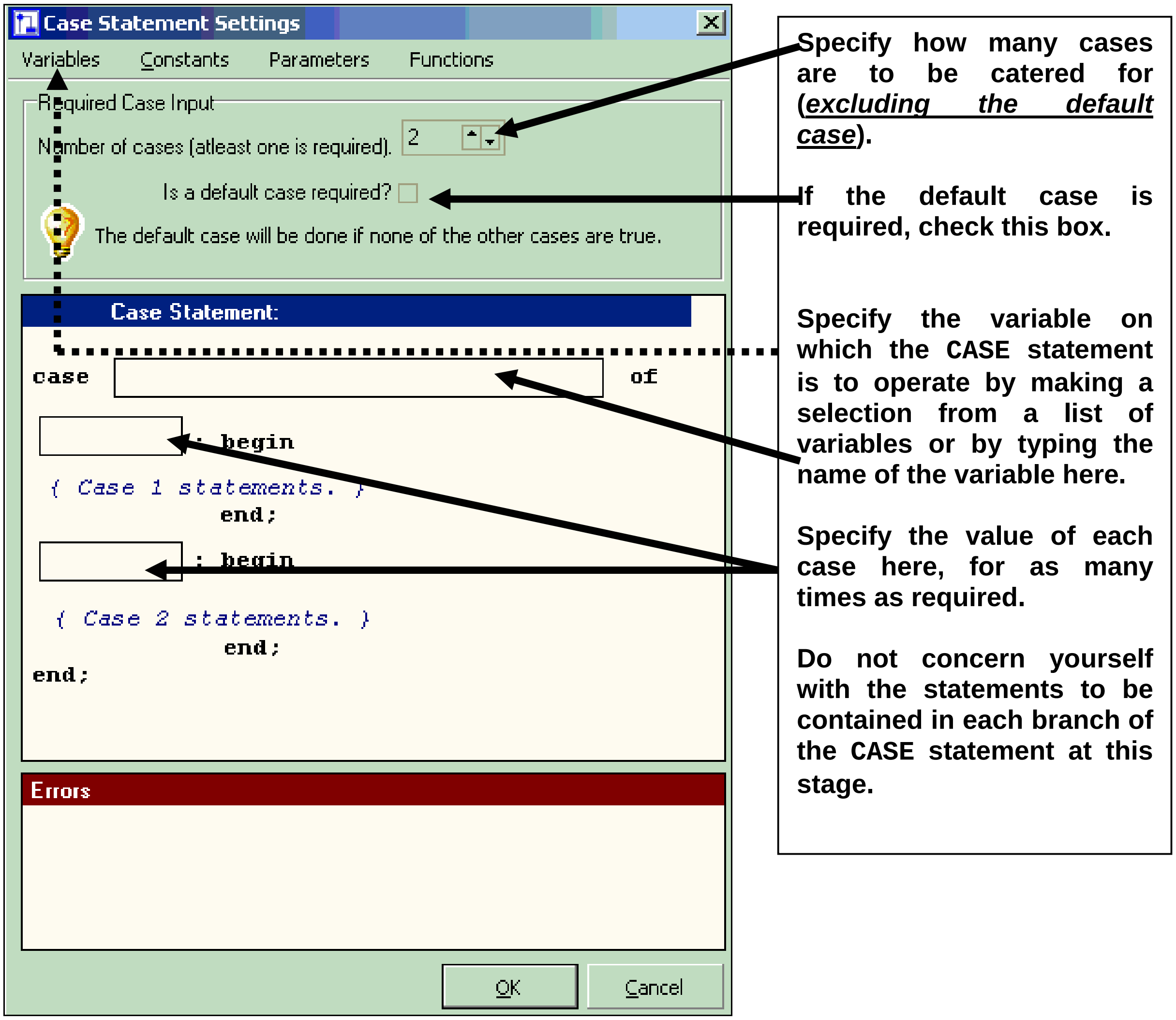


**Drag and drop *__any programming construct__* here for it to be performed whenever the condition of the IF statement evaluates to TRUE.**

**Drag and drop *__any programming construct__* here for it to be performed whenever the condition of the IF statement evaluates to FALSE.**

### CASE programming construct

The **CASE** programming construct is represented by the *case statement* ( ) icon found on B#'s icon panel. Once this icon has been dragged and dropped onto the flowchart, you will be prompted to complete the ***Case Statement Settings*** dialogue:



Number of cases (atleast one is required). 2

Is a default case required? □

The default case will be done if none of the other cases are true.

```
case                                  of
          : begin
 { Case 1 statements. }
                  end;
          : begin
 { Case 2 statements. }
                  end;
end;
```

**Specify how many cases are to be catered for (*excluding the default case*).**

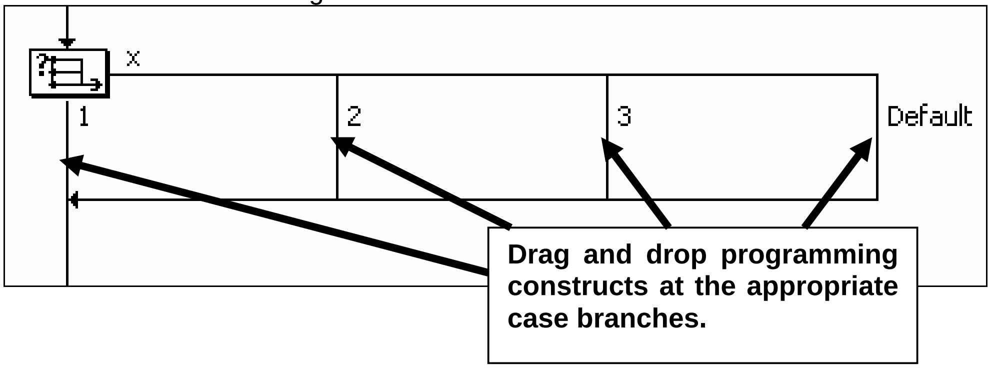
**If the default case is required, check this box.**

**Specify the variable on which the CASE statement is to operate by making a selection from a list of variables or by typing the name of the variable here.**

**Specify the value of each case here, for as many times as required.**

**Do not concern yourself with the statements to be contained in each branch of the CASE statement at this stage.**

Once the flowchart has been updated to show the **CASE** programming construct, it will resemble the following:



**Drag and drop programming constructs at the appropriate case branches.**

# WRA101 : Practical 5
## Week 6 : 18 - 20 March 2003

*Sections Covered*
**FOR** loops (pp 150 – 153)

*Objectives*
- To understand the use of the **FOR** statement.

*Practical Preparation*
- Using either a flowchart or pseudocode, plan the solution for each of the compulsory practical tasks.

***YOU WILL NOT RECEIVE ASSISTANCE DURING THE PRACTICAL SESSION IF YOU DO NOT HAVE YOUR PREPARATION WITH YOU. YOU WILL BE REQUIRED TO PRESENT IT TO THE STUDENT ASSISTANT BEFORE RECEIVING ASSISTANCE.***

### Compulsory Practical Tasks

### ***Task 1 : Using B# to solve a problem***

***In B#***, write a program that asks the user for 10 integers, one at a time. After all of the numbers have been entered, the sum and average of the numbers must be output on the screen.

### ***Task 2 : Using Delphi to solve a problem***

Write a ***Delphi program*** that asks the user for 10 integers, one at a time. After all of the numbers have been entered, display the quantity of **1**s, **2**s, **3**s, …, **5**s that were entered, as well as the quantity of the numbers that were not in the range **1..5**.

For example, if the user entered the numbers **1**, **4**, **5**, **10**, **3**, **-5**, **100**, **3**, **9**, **10** the program would display:

```
Quantity of 1s: 1
Quantity of 2s: 0
Quantity of 3s: 2
Quantity of 4s: 1
Quantity of 5s: 1
Quantity of other numbers: 5
```

### ***Task 3***

Write a ***NEW Delphi program*** that does the following:

> Ask the user for a number and determine and display whether the number is a prime number or not.

## Task 4

Using ***either B# or Delphi***, write a program that does the following:

> Ask the user for the values of *a*, *b* and *c* for a polynomial equation in the form
>
> $f(x) = ax^2 + bx + c$
>
> The user is then asked for a starting x-value (*x1*) and an ending x-value (*x2*) and displays the values for *f(x)* for all the x-values in the range *x1..x2* at intervals of 1. Assume that integer values for *a*, *b*, *c*, *x1* and *x2* are used.

For example, if the user gives the values *a*=1, *b*=-2, *c*=0 (thus $f(x) = x^2 -2x$) and *x1*=-1, *x2*=3 (thus display the value for *f(x)* for –1, 0, 1, 2, 3), the program will output:

```
f(-1) = 3
f(0) = 0
f(1) = -1
f(2) = 0
f(3) = 3
```

## Optional Practical Tasks

## Task 5

Write a program that does the following:

> Ask the user the number of students registered for WRA101. For each student in WRA101, the final mark (assumed to always be in the range 0..100), must be entered. Once all of the marks have been entered, the program must determine and display the average mark, as well as the quantity of students who passed, failed and obtained distinctions. A student obtains a distinction if his/her mark is at least 75. A student fails if his/her mark is less than 50 otherwise he/she passes.

## Task 6

Write a program that does the following:

> Ask the user for a positive integer and then display all of its factors. An appropriate message should be displayed if the user enters a non-positive number.

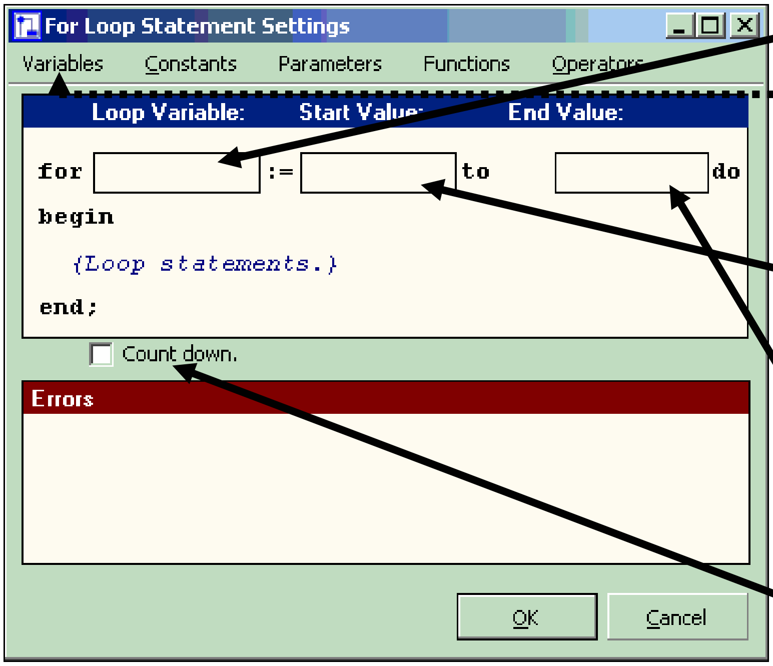
For example, if the user enters **12**, the program would display:

```
The factors of 12 are: 1 2 3 4 6 12
```

# The FOR programming construct in B#

## FOR programming construct

The **FOR** programming construct is represented by the *for loop* () icon found on B#'s icon panel. Once this icon has been dragged and dropped onto the flowchart, you will be prompted to complete the ***For Loop Statement Settings*** dialogue:



```
for [          ] := [          ] to [          ] do
begin
   {Loop statements.}
end;
```

□ Count down.

**Enter the name of the *loop control variable* here, either directly or by making a selection from a list of variables.**

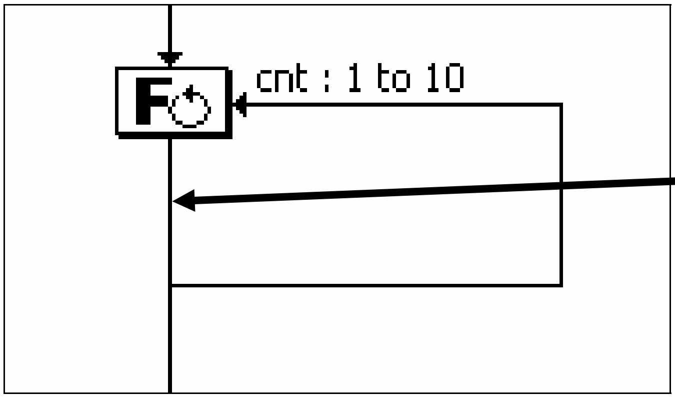
**Enter the *initial value* of the loop control variable here (this may also be selected from a list of variables, if appropriate).**

**Enter the *final value* of the loop control variable here (this may also be selected from a list of variables, if appropriate).**

**If the loop control variable is to be *decremented* with each iteration of the loop, check this box.**

**Do not concern yourself with the statements to be contained in body of the FOR loop at this stage.**

Once the flowchart has been updated to show the **FOR** loop programming construct, it will resemble the following:



**Drag and drop any programming construct here for it to be performed at part of the FOR loop body.**

# WRA101 : Practical 6
## Week 7 : 25 - 27 March 2003

*Sections Covered*
**`WHILE`** and **`REPEAT`** loops (pp 155 – 161)

*Objectives*
- To understand the use of the **`WHILE`** statement.
- To understand the use of the **`REPEAT`** statement.

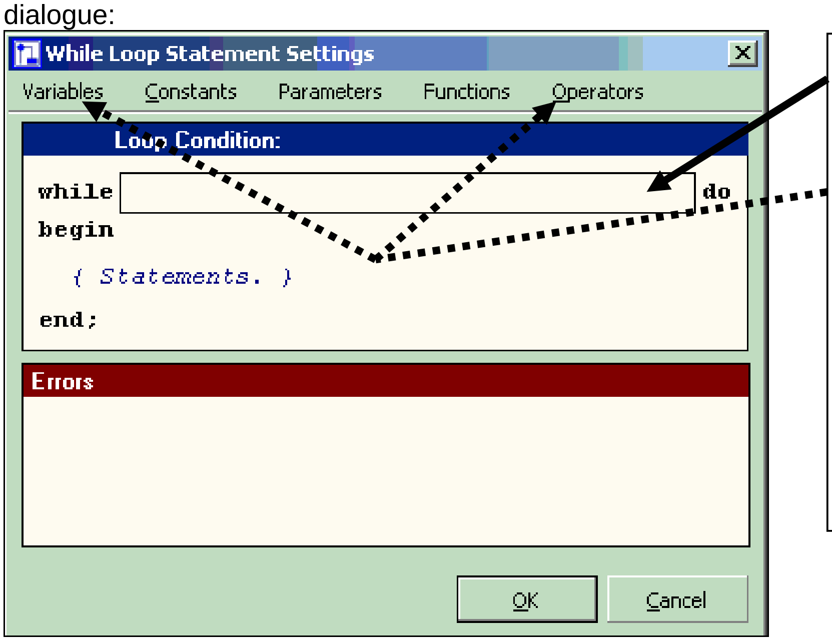
*Practical Preparation*
- Using either a flowchart or pseudocode, plan the solution for each of the compulsory practical tasks.

***YOU WILL NOT RECEIVE ASSISTANCE DURING THE PRACTICAL SESSION IF YOU DO NOT HAVE YOUR PREPARATION WITH YOU. YOU WILL BE REQUIRED TO PRESENT IT TO THE STUDENT ASSISTANT BEFORE RECEIVING ASSISTANCE.***

***Remember to make backup copies on your personal diskette of all your `.dpr` and `.bpf` files only before you leave the laboratory. Make a copy of these files on your `Homes` folder as well.***

## The `WHILE` programming construct in B#

The **`WHILE`** programming construct is represented by the *while loop* (W) icon found on B#'s icon panel. Once this icon has been dragged and dropped onto the flowchart, you will be prompted to complete the ***While Loop Statement Settings*** dialogue:

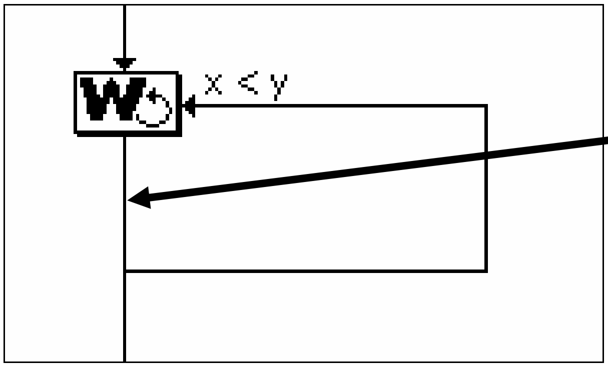


```
while                                  do
begin
  { Statements. }
end;
```

**Enter the condition applicable to the *loop control variable* here, either directly or by making a selection from a list of variables and operators. *Hint*: When in doubt about the condition, always use `NOT(`*stopping condition*`)`.**

**Do not concern yourself with the statements to be contained in the body of the `WHILE` loop at this stage.**

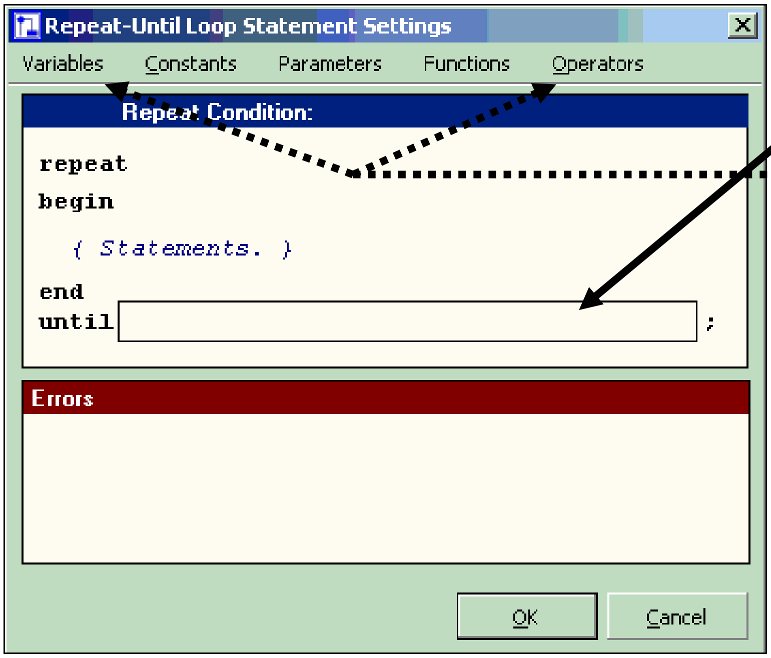
Once the flowchart has been updated to show the **WHILE** loop programming construct, it will resemble the following:



**Drag and drop any programming construct here for it to be performed at part of the WHILE loop body.**

## The REPEAT programming construct in B#

The **REPEAT** programming construct is represented by the *repeat-until loop* (R) icon found on B#'s icon panel. Once this icon has been dragged and dropped onto the flowchart, you will be prompted to complete the ***Repeat-Until Loop Statement Settings*** dialogue:

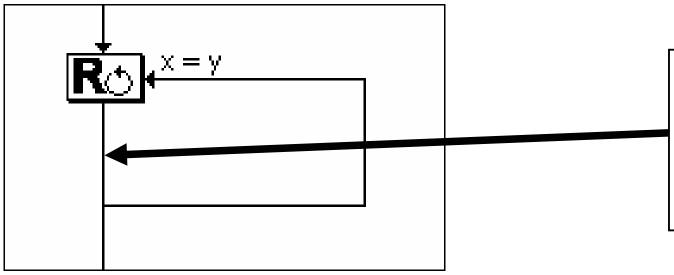


```
repeat
begin
  { Statements. }
end
until                                   ;
```

**Enter the condition applicable to the *loop control variable* here, either directly or by making a selection from a list of variables and operators. *Hint*: When in doubt about the condition, always use (*stopping condition*).**

**Do not concern yourself with the statements to be contained in the body of the REPEAT loop at this stage.**

Once the flowchart has been updated to show the **REPEAT** loop programming construct, it will resemble the following:



**Drag and drop any programming construct here for it to be performed at part of the REPEAT loop body.**

## Compulsory Practical Tasks

### ***Task 1 : Using B# to solve a problem***

***In B#***, write a program that continuously asks the user for positive integers until a non-positive integer (0 included) is entered. After all of the positive integers have been entered, the sum, maximum, minimum and average of the integers must be output on the screen.

### ***Task 2 : Using Delphi to solve a problem***

Write a ***Delphi program*** that continuously asks the user for characters until a **#** is entered. After all of the characters have been entered, the program must display the quantity of spaces (  ) and vowels (**a**'s, **e**'s, **i**'s, **o**'s and **u**'s as a whole), as well as the quantity of the other characters that were entered (excluding the final **#**).

For example, if the user entered the characters **A**, **e**, **F**, **g**, **a**,  , **!**, **z**, **5**, **f**, **#**, the program would display:

```
Quantity of spaces: 1
Quantity of vowels: 3
Quantity of other characters: 6
```

### ***Task 3 : Using B# to solve a problem***

***In B#***, write a program that continuously asks the user for an integer (say *n*) until *n* is in the range 1 – 100. The program must then display the first *n* multiples of 3 ***in descending order*** (from largest to smallest).

### ***Task 4 : Using Delphi to solve a problem***

Write a ***Delphi program*** that continuously asks the user for an integer (say *n*) until *n* is in the range 1 – 10, and a character until either a **#**, ***** or **%** is entered. The program must then display a sequence of *n* occurrences (each separated by a space) of the specific character (**#**, ***** or **%**) entered.

For example, if the user entered the integer 7 and the character **#**, the program would display:

```
# # # # # # #
```

### ***Task 5***

Using ***either B# or Delphi***, write a program that does the following:

> Ask the user for a positive value representing the population (in terms of millions of individuals) of South Africa in 2003, for example 1.8. If the growth of the population per annum is estimated at 16.37%, determine and display the first year when the population will exceed 4.3 million individuals.

## Optional Practical Tasks

### ***Task 6***

Write a program that continuously asks the user to enter a positive integer until an integer in the range 1 – 2500 is entered. The program must then determine and display the first positive integer that results in a sum of cubes ***exceeding*** the user-entered integer. The sum of the cubes exceeding the user-entered integer must also be displayed.

For example, if the user enters 17, the program will display that the sum of the cubes is 36 ($1^3$+$2^3$+$3^3$), and that the integer 3 was the first positive integer to result in a sum of cubes exceeding 17.

### ***Task 7***

Write a program that will evaluate the function $y = 4x^2 – 16x + 15$, with *x* going from 1 to 2 in steps of 0.1. For each *x* displayed, display the value of *y* and a message (**`POSITIVE`** if *y* is positive, otherwise **`NOT POSITIVE`**, if *y* is not).

# WRA101 : Practical 7

## Week 8 : 8 - 10 April 2003

*Sections Covered*
**`for`**, **`while`**, **`repeat`** (pp 150 - 159)

*Objectives*

- Tracing a nested loop.
- To practice the use of the **`FOR`**, **`WHILE`** and **`REPEAT`** loops.
- To practice the use of nested loops.
- To practice the elimination of syntactical errors.

*Practical Preparation*

- Using either a flowchart or pseudocode, plan the solution for each of the compulsory practical tasks.

***YOU WILL NOT RECEIVE ASSISTANCE DURING THE PRACTICAL SESSION IF YOU DO NOT HAVE YOUR PREPARATION WITH YOU. YOU WILL BE REQUIRED TO PRESENT IT TO THE STUDENT ASSISTANT BEFORE RECEIVING ASSISTANCE.***

- Show the output for the following program.

```
Program Preparation;
(*APPTYPE CONSOLE*)
var
   a,
   b : integer;
Begin
   for a := 4 downto 1 do
   begin
      for b:=1 to a*2 do
         write(‘#’);
      writeln;
   end;
   readln;
End.
```

***Remember to make backup copies on your personal diskette of all your `.dpr` and `.bpf` files only before you leave the laboratory. Make a copy of these files on your*** **`Homes`** ***folder as well.***

## Compulsory Practical Tasks

### ***Task 1 : Using B# to solve a problem***

***In B#***, write a program that asks the user for 10 integers in the range 0 to 100, one at a time. The user must continuously be prompted for an integer until it is in the correct range before processing it further. After all 10 of the integers in the range 0 to 100 have been entered, the sum, maximum, minimum and average of the integers must be output on the screen.

### ***Task 2 : Using Delphi to solve a problem***

Write a ***Delphi program*** that continuously asks the user for an integer *n* until an integer greater then zero is entered. The program must then determine and display the first *n* prime numbers. You are required to use a **while** loop in order to determine whether a specific integer is a prime number or not. The list of prime numbers may ***NOT*** be typed in as part of the program – the program must determine them!

### ***Task 3***

Make a copy of the file

**F:\Courses\WRA101\Practical 7\Prac7Task3.dpr**

to the directory where you compile your programs, namely your directory on **C:\Temp**, or your home directory. In Delphi, remove all the syntax errors from the file **Prac7Task3.dpr** so that it compiles without any errors.

### ***Task 4***

Using ***either B# or Delphi***, write a program that does the following:

A **menu-driven program** displays a number of choices to the user, from which they choose one. For example, the following could be displayed:

```
MENU:
Find the factors of a number
Determine if a number is prime
Display a times table
Exit
Your choice (enter the appropriate number):
```

Depending on the choice made, different actions are performed. This continues until the user chooses the **Exit** (choice 4) option.

**NOTE**: The program must cater for the case of the first choice being entered being the **Exit** choice.

The basic algorithm is one of the following (both will function equally well):

Using a **WHILE** loop

```
do any initialisation necessary
display the menu and ask the user for a choice
     until a valid one is entered (1, 2 3 or 4)
while the choice is not the exit option
  depending on the choice, do something
  display the menu and ask the user for a choice
      until a valid one is entered (1, 2 3 or 4)
end of while
do any finalizations necessary
```

Using a **REPEAT** loop

```
do any initialisation necessary
repeat
  display the menu and ask the user for a choice
      until a valid one is entered (1, 2 3 or 4)
  depending on the choice, do something
until exit option is chosen (i.e. option 4)
do any finalizations necessary
```

Make your choice of basic algorithm from the two given above, and write a menu-driven program that displays the menu shown above and allows the user a number of different choices.

- Choice 1 must continuously request the user for an integer until a number greater than zero is entered. All of this number's factors should then be displayed.
- Choice 2 must continuously request the user for an integer until a number greater than zero is entered. The program must then display whether or not this integer is prime.
- Choice 3 must continuously request the user for an integer n until a number in the range 2 to 10 is entered. The *n*-times table must then be displayed (from 1x*n* to 12x*n*).

*Hint: You might find it easier if you set up the menu loop first without writing the code for the various operations. Instead, simply have* **writeln** *statements to indicate when something would have happened (e.g. instead of finding the factors of a number, simply display that that is what would have happened). That way you can test the menu and see if everything is working correctly. Once it is, then you can add the code for the operations.*

## Optional Practical Tasks

### ***Task 5***

Write a program (using nested loops) to display the pattern below.

```
1
12
123
1234
```

Change the program you have written to first ask the user how many rows the pattern should have. The program must then display a similar pattern with the specified number of rows.

### Task 6

Write a program that does the following until the user chooses to stop processing the students in a class:

- For each student in a class, determine the total of 3 marks, each out of 10. Any mark for a student must be continuously entered until such time as it falls within the range 0 – 10.
- Display the percentage obtained by each student (total mark out of a maximum of 30).
- Determine and display the average percentage obtained for the class.

### Task 7

Write a program that makes use of nested loops to display the following pattern in terms of the single characters X and 0:

```
X0X0X0
0X0X0X
X0X0X0
0X0X0X
```

# WRA101 : Practical 8

## Week 9 : 15 - 17 April 2003

*Sections Covered*

Introduction to Structured Programming (pp 192)
Mathematical Functions (pp 85 – top 86)
The **`Inc`** procedure

*Objectives*

- To understand the modular structure of programs
- To practice the use of pre-defined mathematical functions
- To practice the use of the pre-defined **`Inc`** procedure

*Practical Preparation*

- Using either a flowchart or pseudocode, plan the solution for each of the compulsory practical tasks.
- ***Prior to your practical session***, carefully read through the section (bottom page 1 – top page 5) that describes how to use predefined functions and procedures in B#

***YOU WILL NOT RECEIVE ASSISTANCE DURING THE PRACTICAL SESSION IF YOU DO NOT HAVE YOUR PREPARATION WITH YOU. YOU WILL BE REQUIRED TO PRESENT IT TO THE STUDENT ASSISTANT BEFORE RECEIVING ASSISTANCE.***

***Remember to make backup copies on your personal diskette of all your `.dpr` and `.bpf` files only before you leave the laboratory. Make a copy of these files on your `Homes` folder as well.***

***Some useful facilities supported by B#***

You can ***delete a programming construct*** (and all of its nested icons) from the flowchart in one of 2 ways:

- Click on the icon that you wish to delete, and then on the ☒ (**`Delete currently selected icon`**) icon on the menu palette at the top of the B# window, ***OR***
- Right click on the icon that you wish to delete, and select the **`Delete`** option

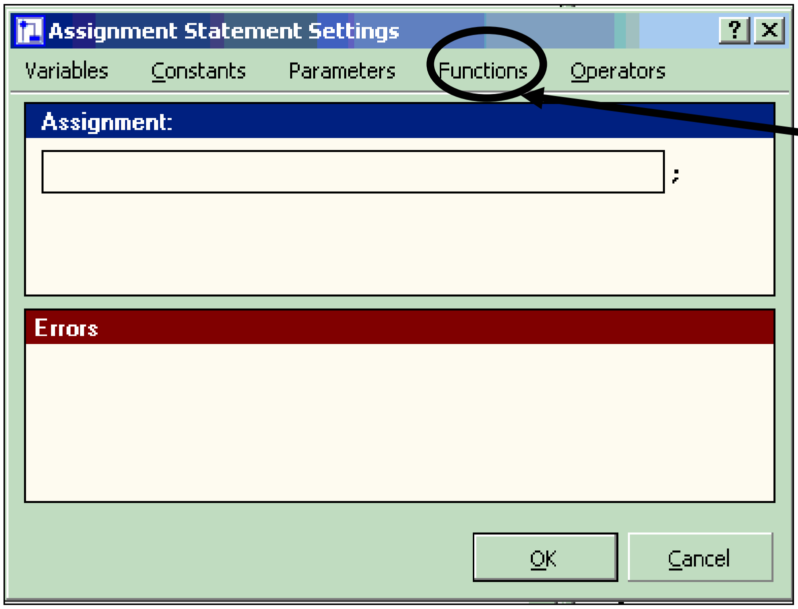
You can ***move a programming construct*** (and all of its nested icons) around in the flowchart in a single operation:

- Click on the icon that you wish to move, and holding the left mouse button in, move the icon to its new position. Release the mouse button – the entire programming constructs with all nested constructs will be relocated.

# Using functions in B#

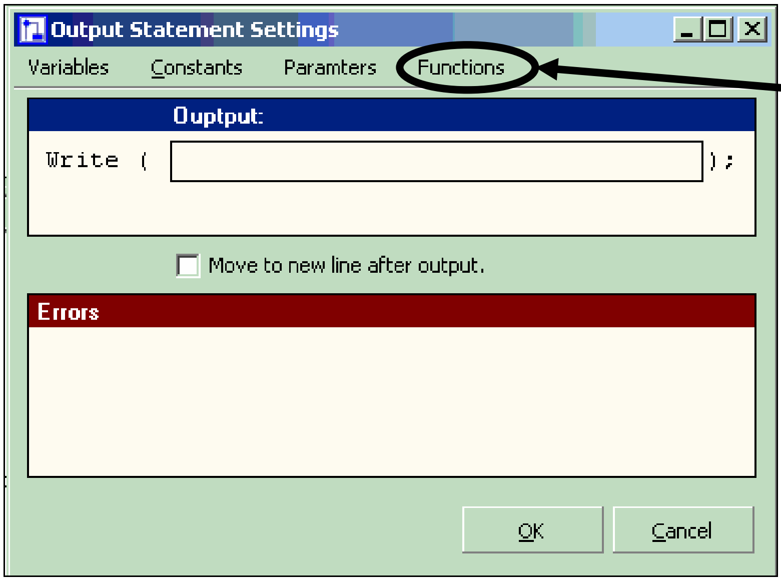
B# supports the use of predefined functions (like **`abs`**, **`ceil`**, **`floor`**, **`power`**, etc) when using the *assignment* (:=) or *screen output* (🖥) constructs.

Once the *assignment* construct has been dragged and dropped onto the flowchart, you will be prompted to complete the ***Assignment Statement Settings*** dialogue:



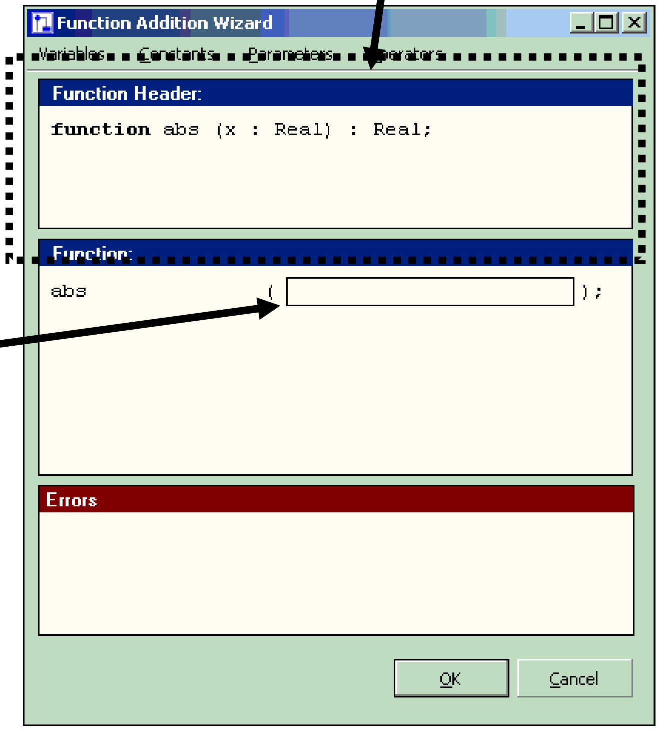
**Select the appropriate function from the list displayed when activating this option. Both predefined (that is, provided by B#) and user-defined functions will be listed.**

Once the *screen output* construct has been dragged and dropped onto the flowchart, you will be prompted to complete the ***Output Statement Settings*** dialogue:



**Select the appropriate function from the list displayed when activating this option. Both predefined (that is, provided by B#) and user-defined functions will be listed.**

Once the appropriate function has been selected in either of the above two cases, the ***Function Addition Wizard*** will display the header of the function and prompt you to enter the arguments for the function. The following is the prompt for an argument for the **abs** function.



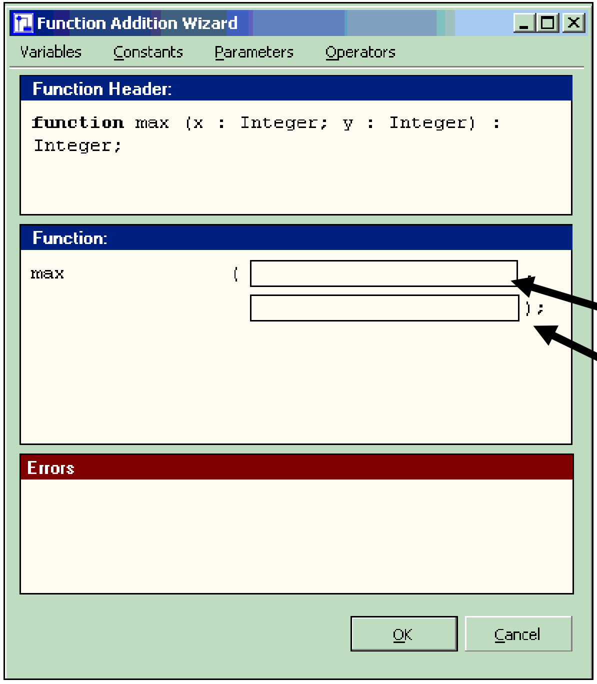
```
function abs (x : Real) : Real;
```

***Note***: Arguments may be ***literal values***, ***constants*** or ***variables***.

The data type(s) of the argument(s) must match the data type(s) of the parameter(s) given in the function header above.

The predefined functions **ceil**, **floor**, **frac**, **int**, **round**, **sqr**, **sqrt** and **trunc** display similar dialogues.

The following dialogue is displayed for the **max** function.



```
function max (x : Integer; y : Integer) :
Integer;
```

***Note***: Two arguments are required for this function. A ***single*** argument (matching the ordered parameters in the function header) must be placed in each independent field.

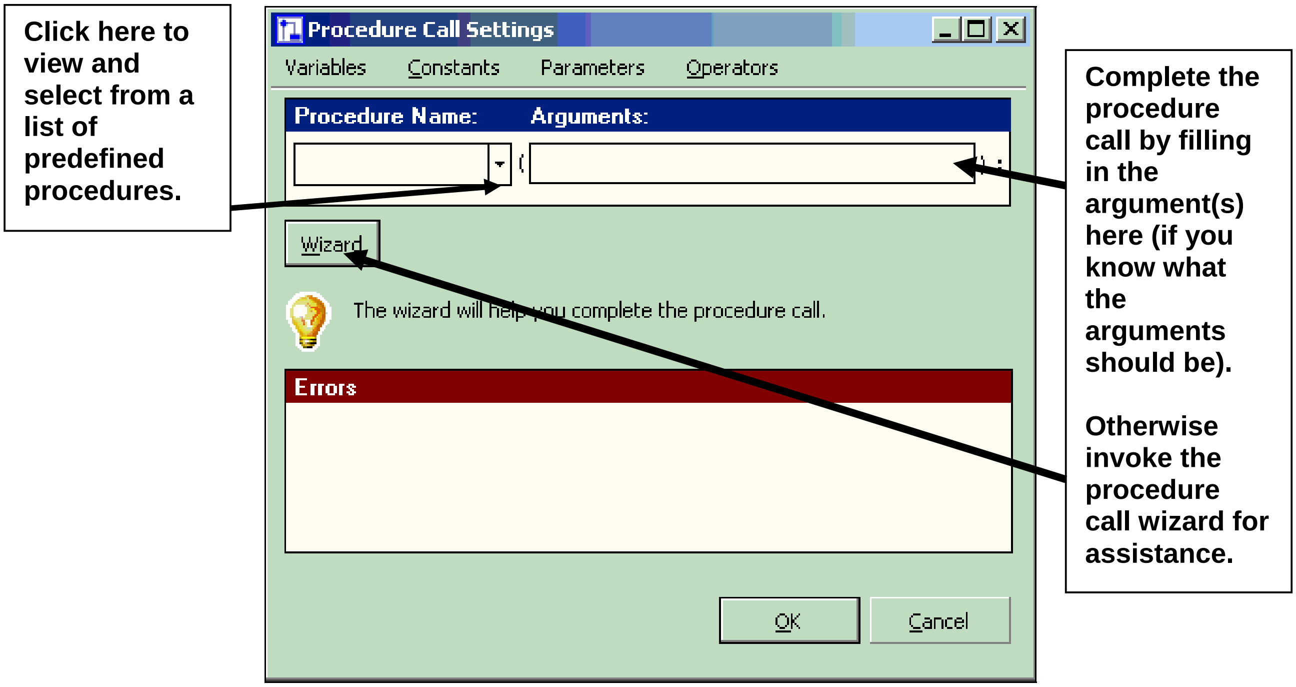
The predefined functions **`min`** and **`power`** display similar dialogues.

The assignment and screen output constructs are then completed as usual.

## Using procedures in B#

The *procedure call* programming construct is represented by the *procedure call* (P(x)) icon found on B#'s icon panel. B# supports the use of predefined procedures (like **`inc`**) when using this construct.

Once this icon has been dragged and dropped onto the flowchart, you will be prompted to complete the ***Procedure Call Settings*** dialogue:



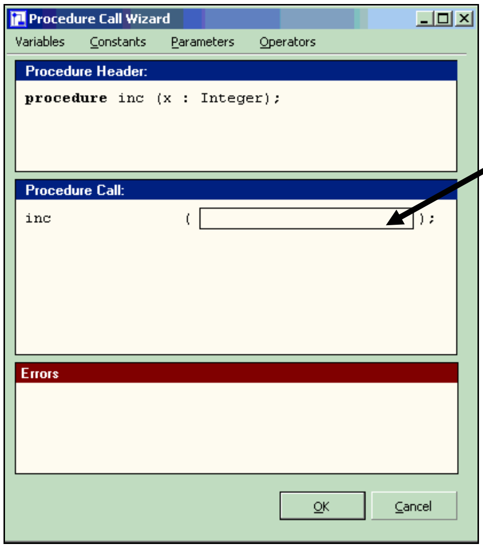
The wizard will help you complete the procedure call.

**Click here to view and select from a list of predefined procedures.**

**Complete the procedure call by filling in the argument(s) here (if you know what the arguments should be).**

**Otherwise invoke the procedure call wizard for assistance.**

Invoking the procedure call wizard results in the ***Procedure Call Wizard*** dialogue being displayed:

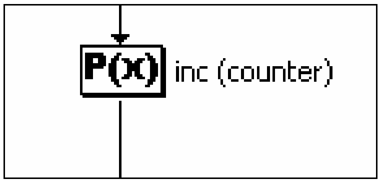


```
procedure inc (x : Integer);
```

```
inc ( );
```

Complete the procedure call in a fashion similar to that used when completing a function call (illustrated in the previous section).

Once the flowchart has been updated to show the *procedure call* programming construct, it will resemble the following:



## Compulsory Practical Tasks

### *Task 1 : Using B# to solve a problem*

The speed of light, *c*, is approximately 3 × $10^8$ metres per second. ***In B#***, write a program that continuously prompts the user to enter a time in ***minutes*** (which must ***always*** be positive) and then displays the distance (in ***kilometers***) that light travels during that time period. The program stops processing when the user requests it to (for example, by entering a **Q** for quit). The number of times that the user enters valid input must be accumulated (using a predefined procedure) and displayed. You can assume that the program will display the distance light travels ***at least once***.

### *Task 2 : Using Delphi to solve a problem*

Write a ***Delphi program*** that asks the user to enter an integer. If the integer is positive, the square root of the integer is displayed to show 2 decimal places. Should a negative integer be entered, the square root of the absolute value of the entered integer is displayed to show 2 decimal places. The program stops processing when the user enters a zero. The number of times that the user enters valid integers must be accumulated (using a predefined procedure) and displayed. Your program should cater for the scenario where the first integer entered by the user may be a zero.

## Task 3

Using ***either B# or Delphi***, write a program that does the following:

> The quadratic formula can solve for the roots of a quadratic equation of the form $ax^2 + bx + c = 0$. The quadratic formula is shown below:
>
> $$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$
>
> Write a program that allows the user to input the values for *a*, *b* and *c*, and then displays the two roots of *x*. Appropriate error messages should be displayed whenever the roots cannot be determined.

## Task 4

Using ***either B# or Delphi***, write a program that does the following:

> Write a program that uses the `max` and `min` mathematical functions to determine and display the maximum and minimum of a sequence of 10 integers entered by the user.

## Task 5

Using ***either B# or Delphi***, write a program that allows two players to play a game of high-low as follows:

> The first player (P1) enters an integer between 0 and 100 (both 0 and 100 excluded). The second player (P2) then tries to guess the first player's integer. If P1's integer is lower than that of P2, the message "`Try a lower integer`" is displayed to P2, and P2 enters another integer. If P1's integer is higher than that of P2, the message "`Try a higher integer`" is displayed to P2 and P2 enters another integer. The game only ends when P2 correctly guesses P1's integer. The program must keep track of the number of guesses made by P2 (using a predefined procedure) and display this information before it terminates.

## Optional Practical Tasks

## Task 6

Write a program **`Quiz`** that allows a user to enter any three integer values *x*, *y* and *z*, each in the range 1 – 10. The program must then allow the user to guess the answer to the expression $\boldsymbol{(x+5)^z/y}$. The program displays the expression to the user and gives the user 3 chances to guess the answer. If the guesses are incorrect on all 3 attempts, the user may decide to continue or not. If the user decides to continue, the user may have another 3 guesses. The program stops processing whenever the user guesses the answer correctly, or the user chooses to stop guessing. At this time the program must display how many guesses ***in total*** the user made use of up to that point.

## ***Task 7***

Write a program that converts a user-entered binary number consisting of exactly 8 digits (entered digit-by-digit) to its decimal equivalent, and displays the decimal equivalent.  Your program should ensure that only valid binary digits are entered.

For example, the binary number **`00110101`** will be entered and the decimal equivalent **`53`** will be displayed.

# WRA101 : Practical 9
## Week 10 : 22 - 24 April 2003

*Sections Covered*
User defined functions and procedures (pp 194 – 202)

*Objectives*
- To understand the modular structure of programs
- To practice the use of user-defined functions and procedures
- To practice the writing of simple functions and procedures

*Practical Preparation*
- Using either a flowchart or pseudocode, plan the solution for each of the compulsory practical tasks.
- ***Prior to your practical session***, carefully read through the section (pages 2 – 10) that describes how to use predefined functions and procedures in B#. This is the last of B#'s skills that you need to master.
- Make use of the function/procedure forms for deciding on the use of functions or procedures and their parameters

***YOU WILL NOT RECEIVE ASSISTANCE DURING THE PRACTICAL SESSION IF YOU DO NOT HAVE YOUR PREPARATION WITH YOU. YOU WILL BE REQUIRED TO PRESENT IT TO THE STUDENT ASSISTANT BEFORE RECEIVING ASSISTANCE.***

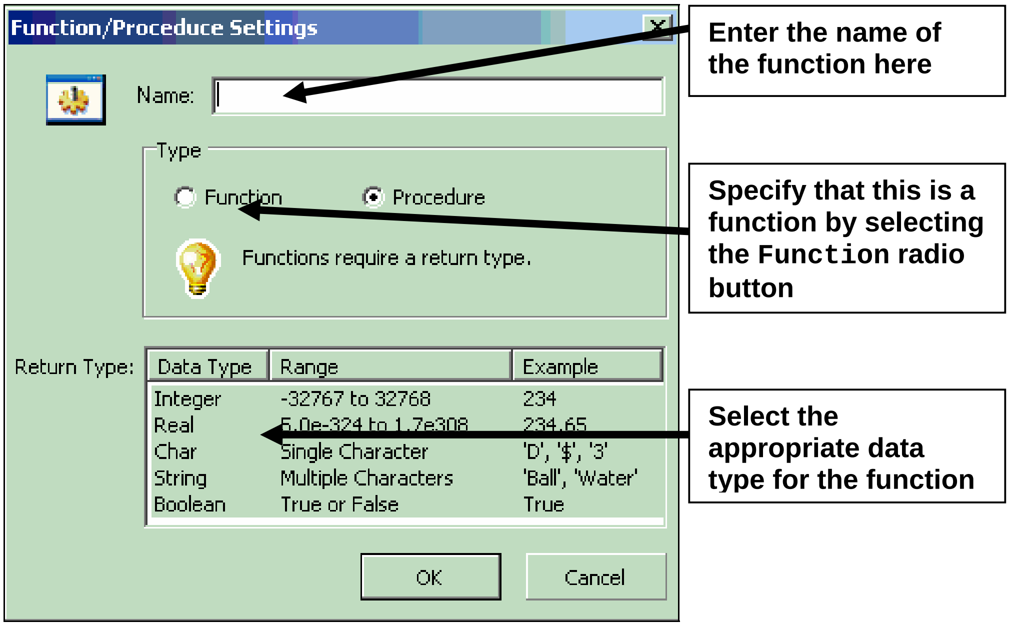
***Remember to make backup copies on your personal diskette of all your `.dpr` and `.bpf` files only before you leave the laboratory. Make a copy of these files on your `Homes` folder as well.***

# Creating a function in B#

B# supports the development and use of user-defined functions. These functions can then be used when using the *assignment* ( ) or *screen output* ( ) constructs (as illustrated with the use of predefined mathematical functions in last week's practical).

> As an example, consider that a function, **`sum`**, is required that accepts two real numbers and calculates and returns the sum of these numbers.

Click on the *New Routine* icon ( ) found in the menu of B#. You will be required to complete the ***Function/Procedure Settings*** dialogue:



Function/Procedure Settings

Name:

Type

○ Function ● Procedure

Functions require a return type.

Return Type:

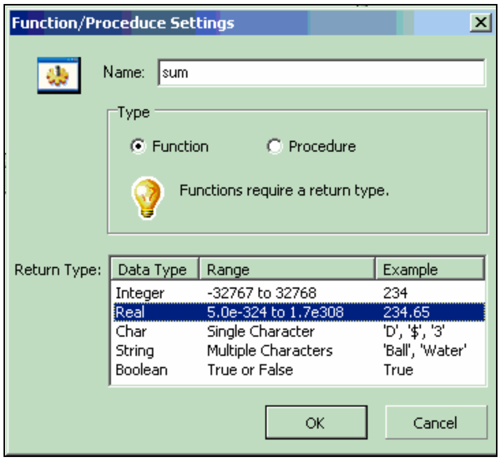
| Data Type | Range | Example |
|---|---|---|
| Integer | -32767 to 32768 | 234 |
| Real | 5.0e-324 to 1.7e308 | 234.65 |
| Char | Single Character | 'D', '$', '3' |
| String | Multiple Characters | 'Ball', 'Water' |
| Boolean | True or False | True |

**Enter the name of the function here**

**Specify that this is a function by selecting the `Function` radio button**

**Select the appropriate data type for the function**

The completed ***Function/Procedure Settings*** dialogue for the **`sum`** function will be as follows:



Function/Procedure Settings

Name: sum

Type

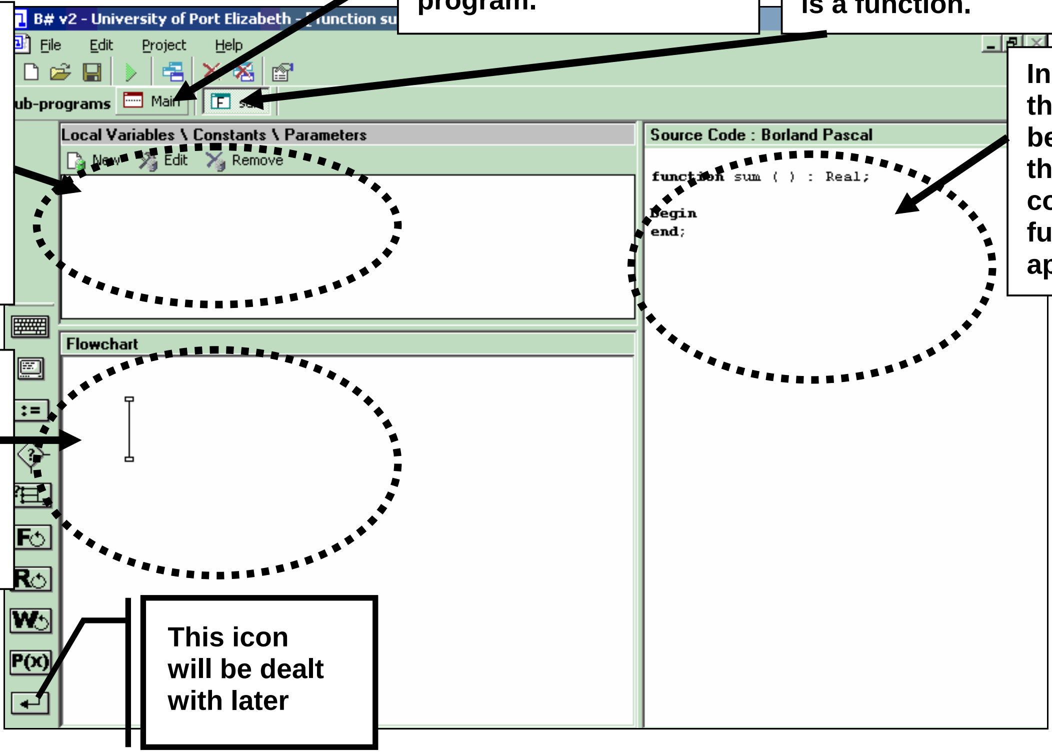
● Function ○ Procedure

Functions require a return type.

Return Type:

| Data Type | Range | Example |
|---|---|---|
| Integer | -32767 to 32768 | 234 |
| Real | 5.0e-324 to 1.7e308 | 234.65 |
| Char | Single Character | 'D', '$', '3' |
| String | Multiple Characters | 'Ball', 'Water' |
| Boolean | True or False | True |

The following is now displayed:


B# v2 - University of Port Elizabeth - [Function su
File Edit Project Help
Sub-programs
Main
sum
Local Variables \ Constants \ Parameters
New Edit Remove
Flowchart
Source Code : Borland Pascal

```
function sum ( ) : Real;

begin
end;
```

Click on this tab to view and make changes to the Main section of the program.

Click on this tab to view and make changes to the sum function. Note the F to indicate that this is a function.

In the case of the function tab being selected, the function's header, variables and constants appear here

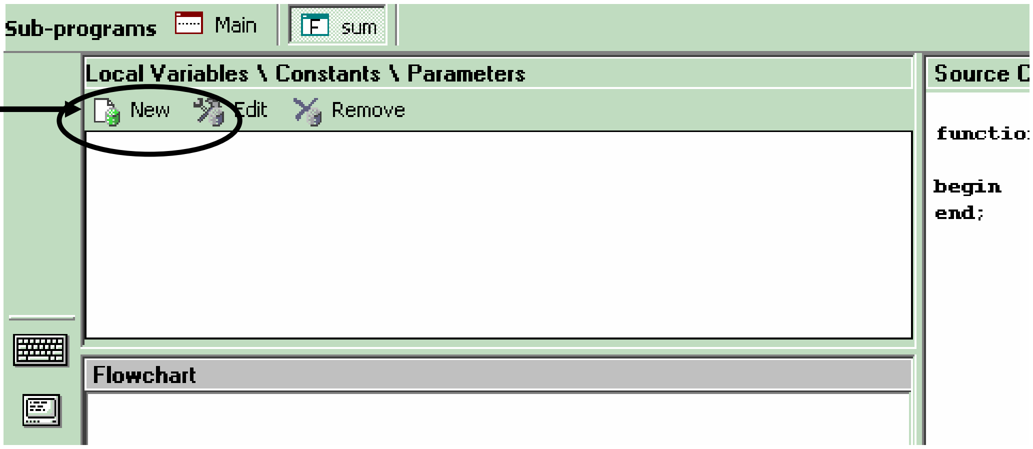
In the case of the function tab being selected, the source code for the function appears here

In the case of the function tab being selected, the function's flowchart appears here

This icon will be dealt with later

To complete the header of the function, the parameters for the function **sum** need to be declared.

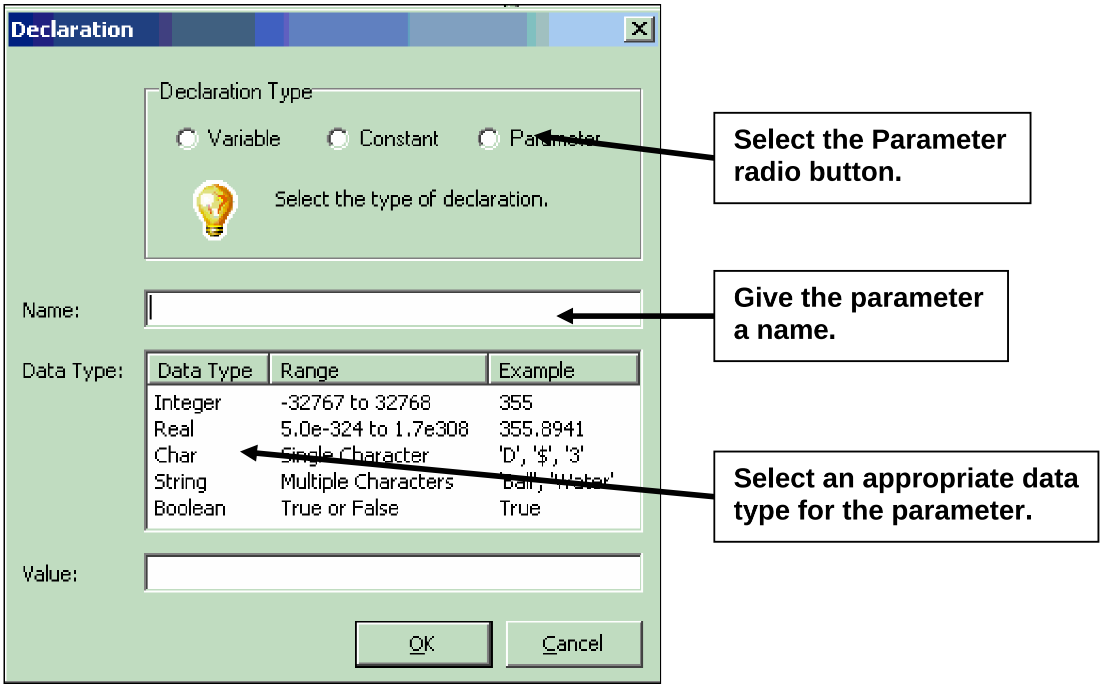
In the ***Local Variables \ Constants \ Parameters*** pane,



```
functio

begin
end;
```

select the new variable/constant/parameter option ( New ).

Compete the ***Declaration*** dialogue for a parameter, ensuring that the ***parameter*** radio button is selected.



**Declaration**

Declaration Type

○ Variable ○ Constant ○ Parameter

Select the type of declaration.

Name:

Data Type:

| Data Type | Range | Example |
|---|---|---|
| Integer | -32767 to 32768 | 355 |
| Real | 5.0e-324 to 1.7e308 | 355.8941 |
| Char | Single Character | 'D', '$', '3' |
| String | Multiple Characters | 'Ball', 'Water' |
| Boolean | True or False | True |

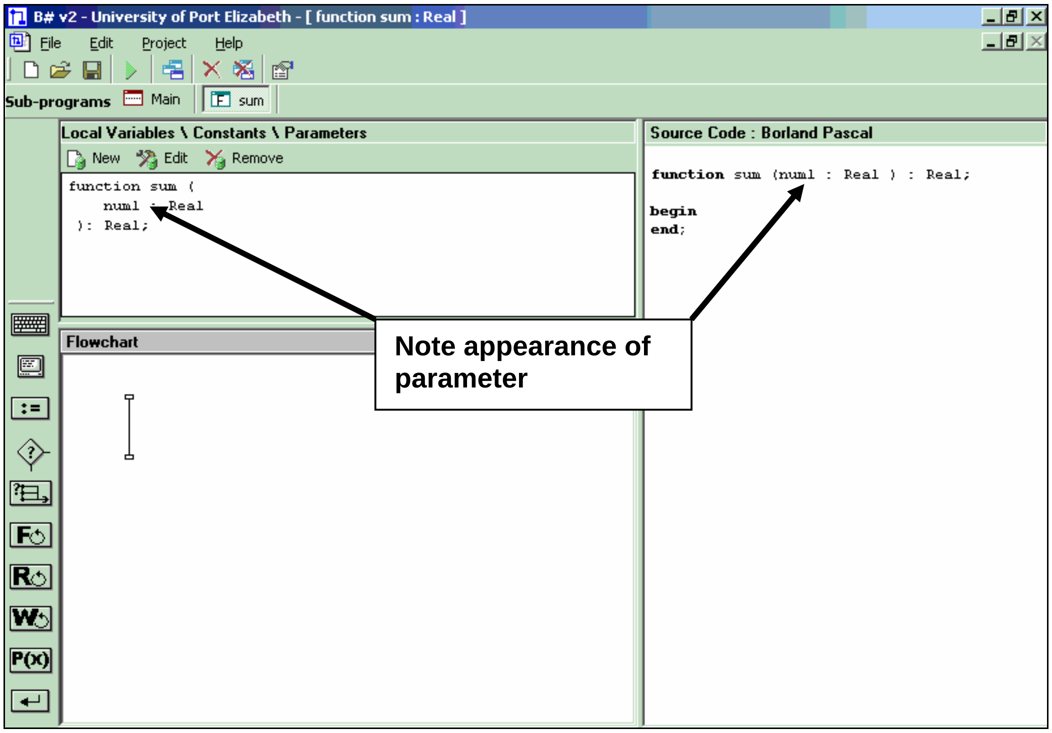
Value:

**Select the Parameter radio button.**

**Give the parameter a name.**

**Select an appropriate data type for the parameter.**

After declaration of **`sum`**'s first parameter **`num1`** (data type **`real`**), the following is displayed:



B# v2 - University of Port Elizabeth - [ function sum : Real ]

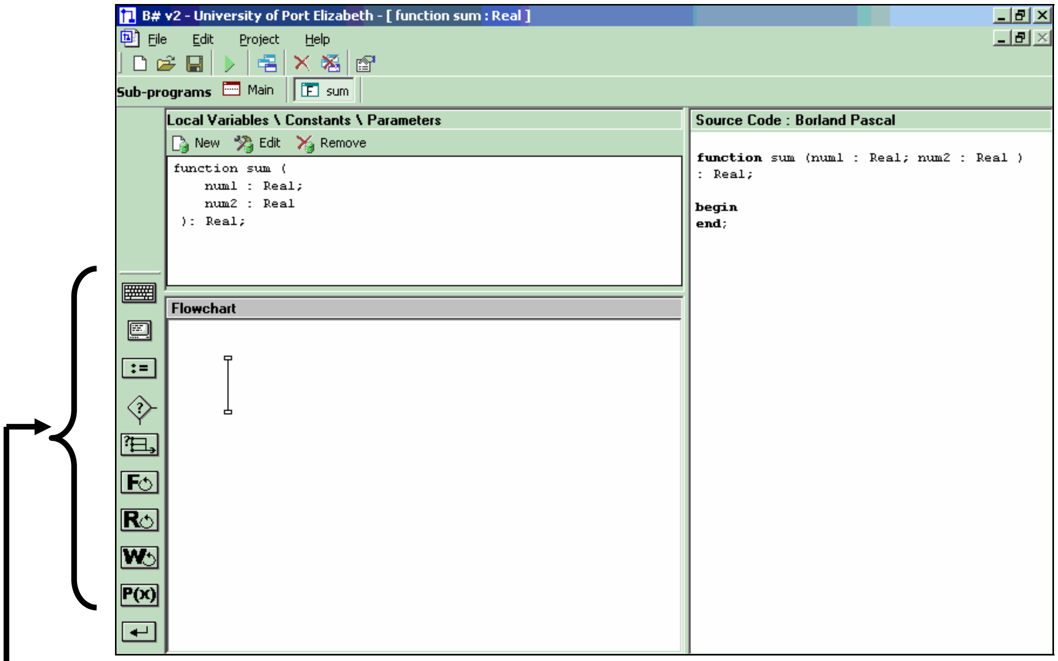
File
Edit
Project
Help
Sub-programs
Main
sum
Local Variables \ Constants \ Parameters
New
Edit
Remove
Flowchart
Source Code : Borland Pascal

```
function sum (
    num1 : Real
 ): Real;
```

```
function sum (num1 : Real ) : Real;

begin
end;
```

**Note appearance of parameter**

After declaration of **`sum`**'s second parameter **`num2`** (data type **`real`**), the following is displayed:

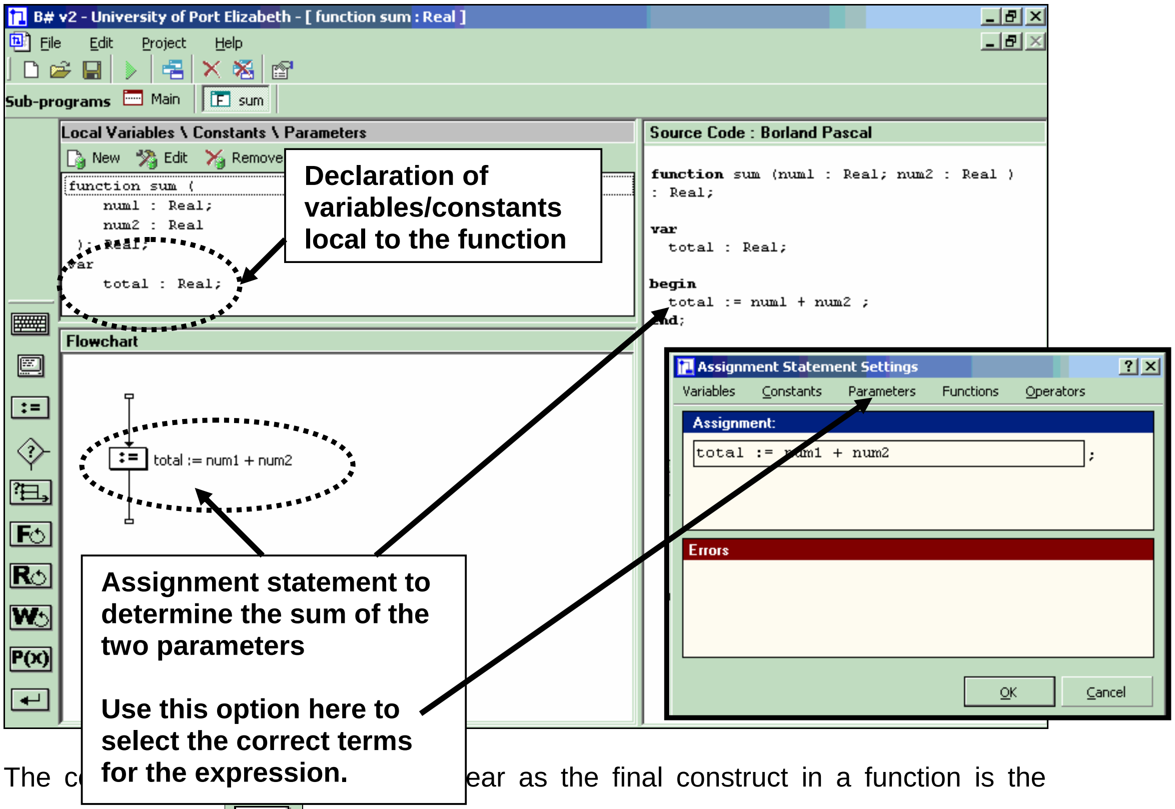

B# v2 - University of Port Elizabeth - [ function sum : Real ]
File
Edit
Project
Help
Sub-programs
Main
sum
Local Variables \ Constants \ Parameters
New
Edit
Remove
Source Code : Borland Pascal
Flowchart
:=
?
F
R
W
P(x)

```
function sum (
    num1 : Real;
    num2 : Real
 ): Real;
```

```
function sum (num1 : Real; num2 : Real )
: Real;

begin
end;
```

The flowchart for the function is created using the skills that you have already acquired in all previous practicals – variables and programming constructs are used and declared as required. Any of the programming constructs learnt about may be used in the function's flowchart.

In the case of the **`sum`** function, the two parameters must be added together. Once this construct has been added to **`sum`**'s flowchart, the following will be displayed:


B# v2 - University of Port Elizabeth - [ function sum : Real ]
File Edit Project Help
Sub-programs Main sum
Local Variables \ Constants \ Parameters
New Edit Remove
Source Code : Borland Pascal
Flowchart
total := num1 + num2
Assignment Statement Settings
Variables Constants Parameters Functions Operators
Assignment:
Errors
OK Cancel

```
function sum (
    num1 : Real;
    num2 : Real
  ): Real;
var
    total : Real;
```

```
function sum (num1 : Real; num2 : Real )
: Real;

var
  total : Real;

begin
  total := num1 + num2 ;
end;
```

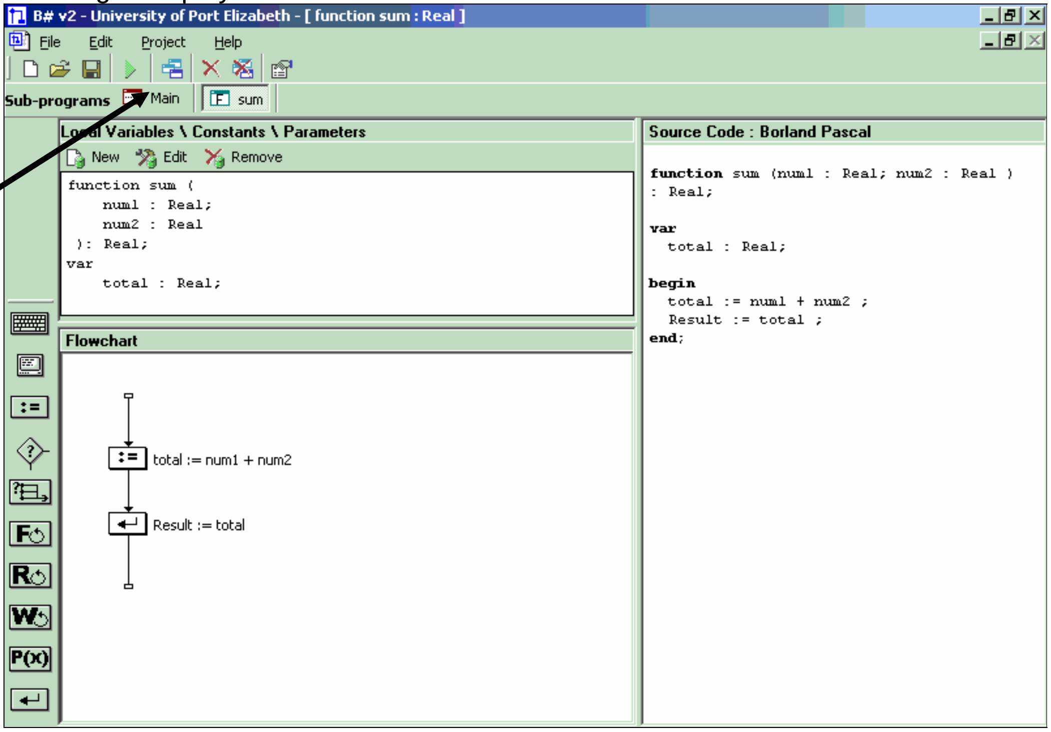
Assignment: `total := num1 + num2 ;`

**Declaration of variables/constants local to the function**

**Assignment statement to determine the sum of the two parameters**

**Use this option here to select the correct terms for the expression.**

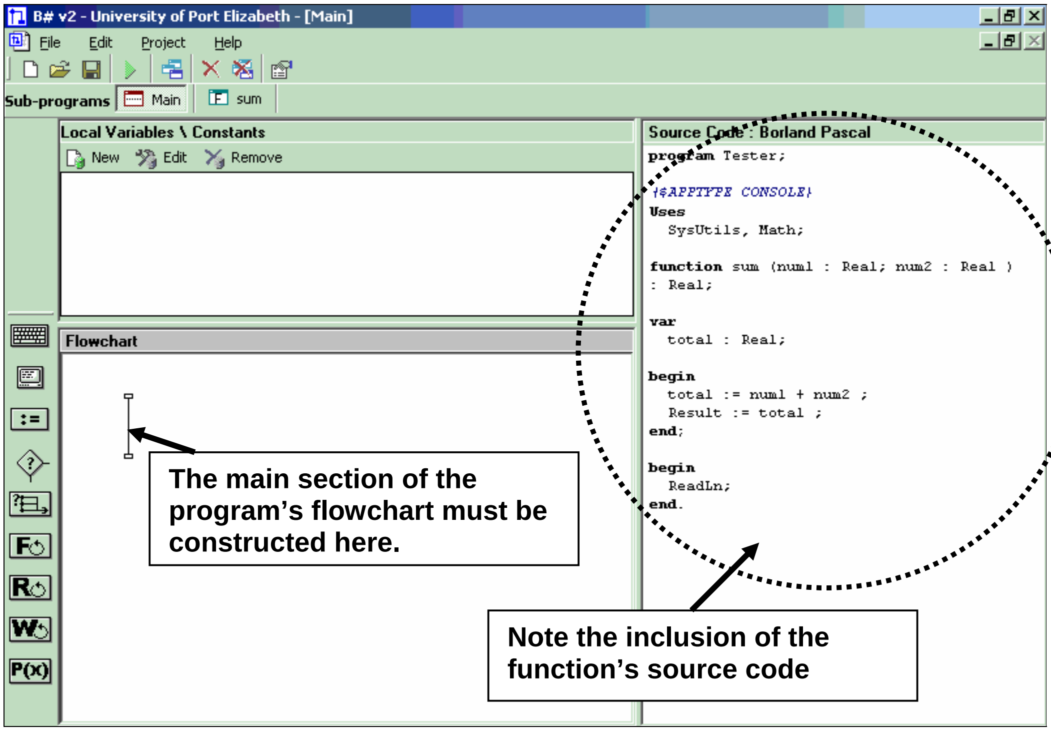
The c ear as the final construct in a function is the *return statement* (↵). Once this has been added to the function's flowchart, the following is displayed:


B# v2 - University of Port Elizabeth - [ function sum : Real ]
File Edit Project Help
Sub-programs Main sum
Local Variables \ Constants \ Parameters
New Edit Remove
Source Code : Borland Pascal
Flowchart
total := num1 + num2
Result := total

```
function sum (
    num1 : Real;
    num2 : Real
  ): Real;
var
    total : Real;
```

```
function sum (num1 : Real; num2 : Real )
: Real;

var
  total : Real;

begin
  total := num1 + num2 ;
  Result := total ;
end;
```

In order to view the main section of the program (with all of the source code for the program), click on the ***Main*** tab at the top left hand side of the screen. The following will then be displayed:


B# v2 - University of Port Elizabeth - [Main]
File
Edit
Project
Help
Sub-programs
Main
sum
Local Variables \ Constants
New
Edit
Remove
Flowchart
Source Code : Borland Pascal
F
R
W
P(X)

```
program Tester;

{$APPTYPE CONSOLE}
Uses
  SysUtils, Math;

function sum (num1 : Real; num2 : Real )
: Real;

var
  total : Real;

begin
  total := num1 + num2 ;
  Result := total ;
end;

begin
  ReadLn;
end.
```

The main section of the program's flowchart must be constructed here.

Note the inclusion of the function's source code

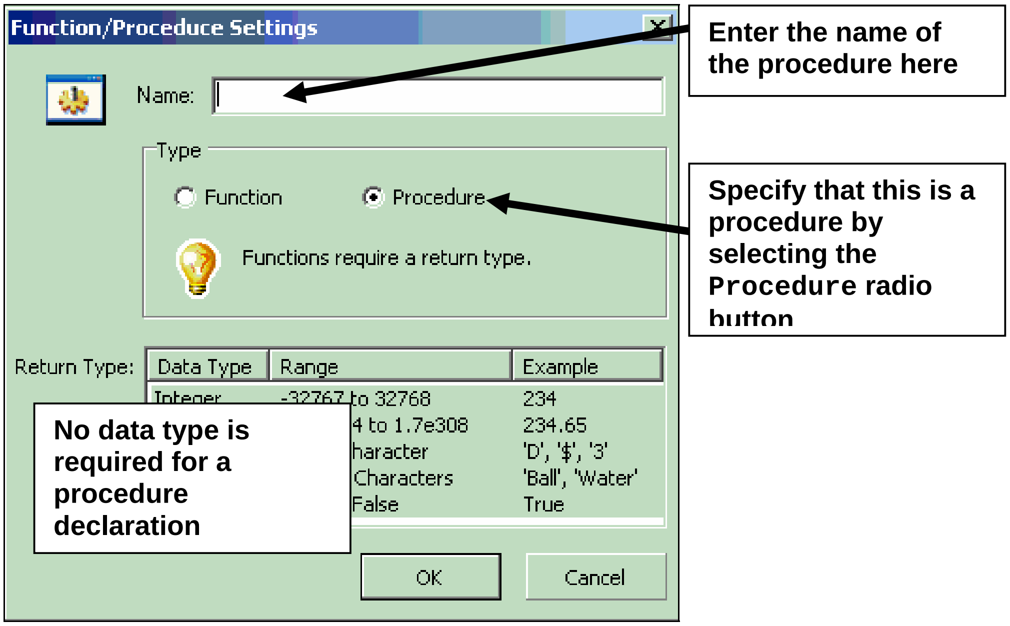
The function **sum** is now available when using the *assignment* (:=) or *screen output* (🖥) constructs – this function will be listed under the ***Functions*** available for use, together with the list of predefined mathematical functions.

# Creating a procedure in B#

B# supports the development and use of user-defined procedures. These procedures can then be used when using the *procedure call* (P(X)) construct (as illustrated with the use of the predefined procedure in last week's practical).

> As an example, consider that a procedure, **`total`**, is required that accepts one real number, asks the user for another and calculates and displays the sum of these numbers.

Click on the *New Routine* icon ( ) found in the menu of B#. You will be required to complete the ***Function/Procedure Settings*** dialogue in much the same way as for a function:



Function/Proceduce Settings

Name:

Type

○ Function ● Procedure

Functions require a return type.

Return Type:

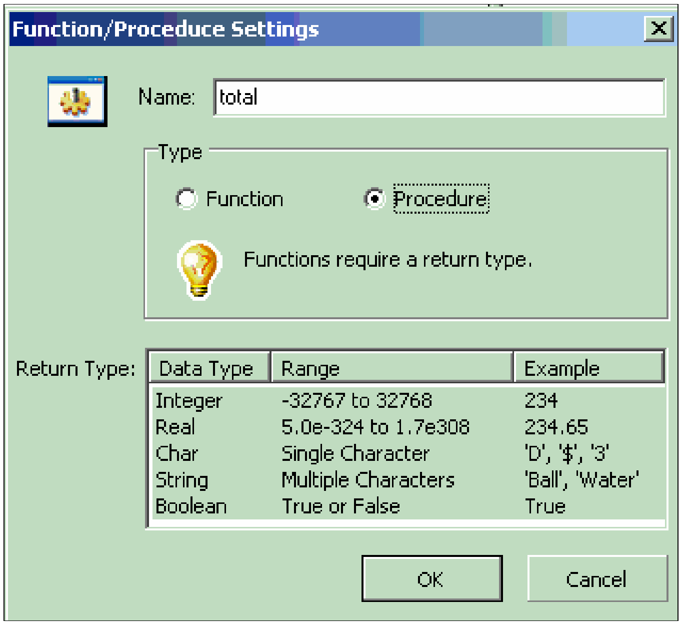
| Data Type | Range | Example |
|---|---|---|
| Integer | -32767 to 32768 | 234 |
|  | 4 to 1.7e308 | 234.65 |
|  | haracter | 'D', '$', '3' |
|  | Characters | 'Ball', 'Water' |
|  | False | True |

**Enter the name of the procedure here**

**Specify that this is a procedure by selecting the `Procedure` radio button**

**No data type is required for a procedure declaration**

The completed ***Function/Procedure Settings*** dialogue for the **`total`** function will be as follows:



Function/Procedure Settings

Name: total

Type

○ Function ● Procedure

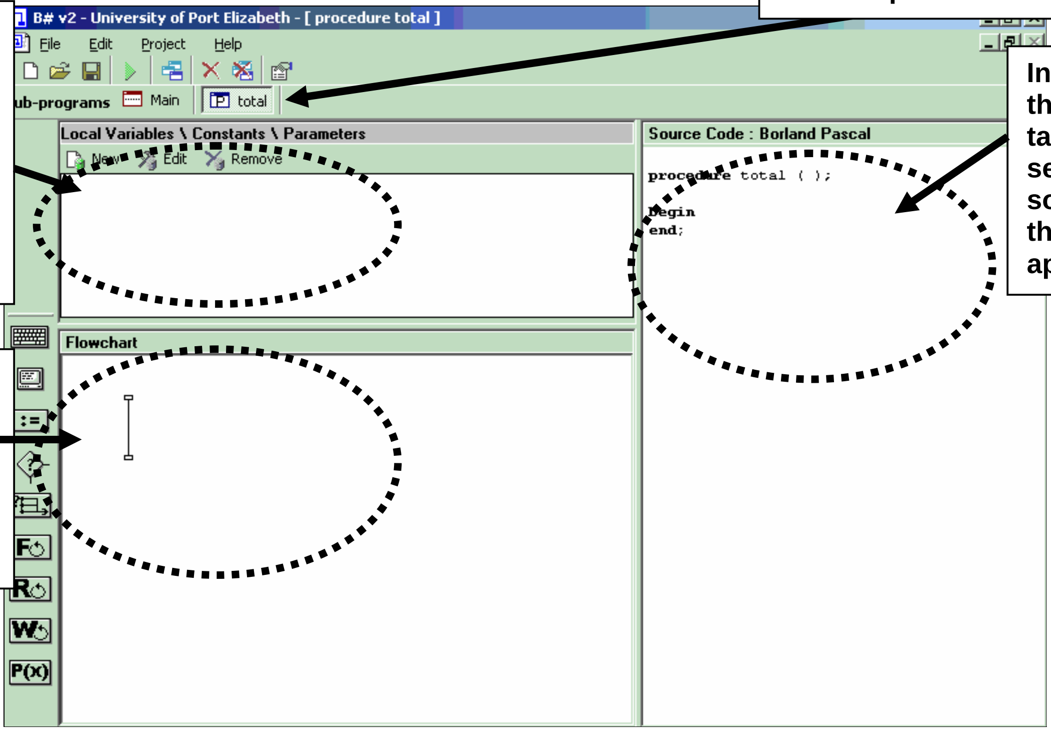
Functions require a return type.

Return Type:

| Data Type | Range | Example |
|---|---|---|
| Integer | -32767 to 32768 | 234 |
| Real | 5.0e-324 to 1.7e308 | 234.65 |
| Char | Single Character | 'D', '$', '3' |
| String | Multiple Characters | 'Ball', 'Water' |
| Boolean | True or False | True |

The following is now displayed:

**Click on this tab to view and make changes to the `total` procedure. Note the** P **to indicate that this is a procedure.**



B# v2 - University of Port Elizabeth - [ procedure total ]

File
Edit
Project
Help
ub-programs
Main
total
Local Variables \ Constants \ Parameters
New
Edit
Remove
Flowchart
Source Code : Borland Pascal

```
procedure total ( );
begin
end;
```

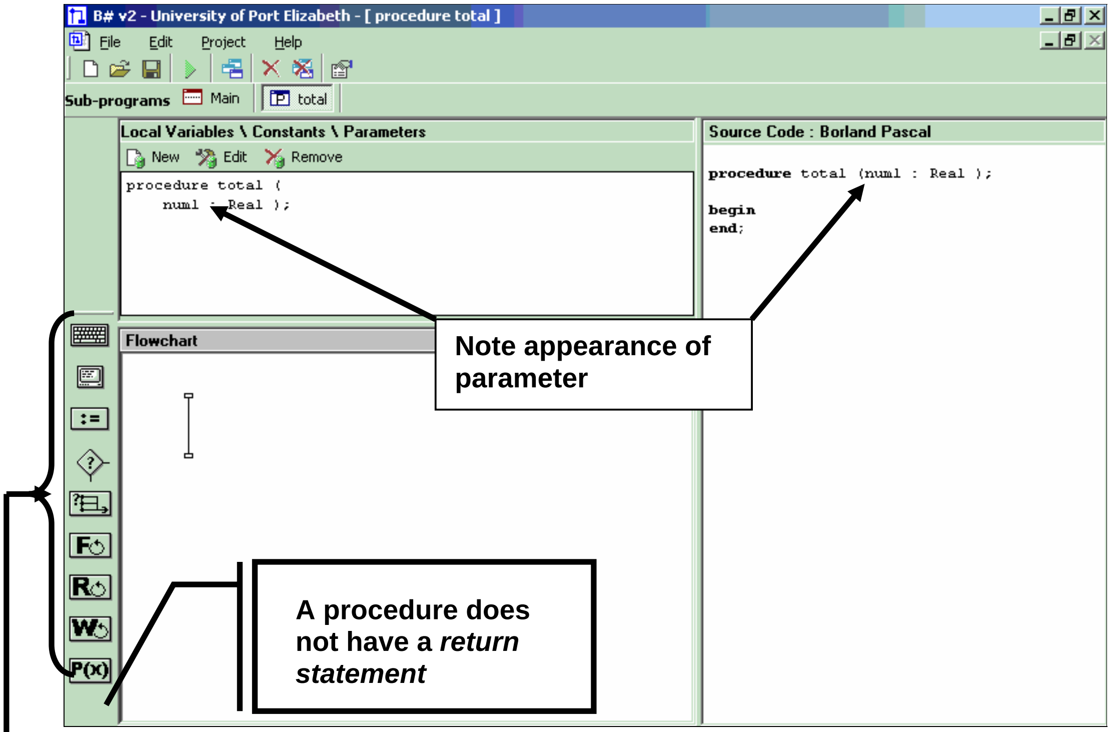
**In the case of the procedure tab being selected, the procedure's header, variables and constants appear here**

**In the case of the procedure tab being selected, the source code for the procedure appears here**

**In the case of the procedure tab being selected, the procedure's flowchart appears here**

To complete the header of the procedure, any parameters for the procedure **`total`** need to be declared. This is done in exactly the same way as for a function (see page 3 (bottom) and 4).

After declaration of **`total`**'s parameter **`num1`** (data type **`real`**), the following is displayed:



B# v2 - University of Port Elizabeth - [ procedure total ]
File
Edit
Project
Help
Sub-programs
Main
total
Local Variables \ Constants \ Parameters
New
Edit
Remove
Flowchart
Source Code : Borland Pascal

```
procedure total (
     num1 : Real );
```

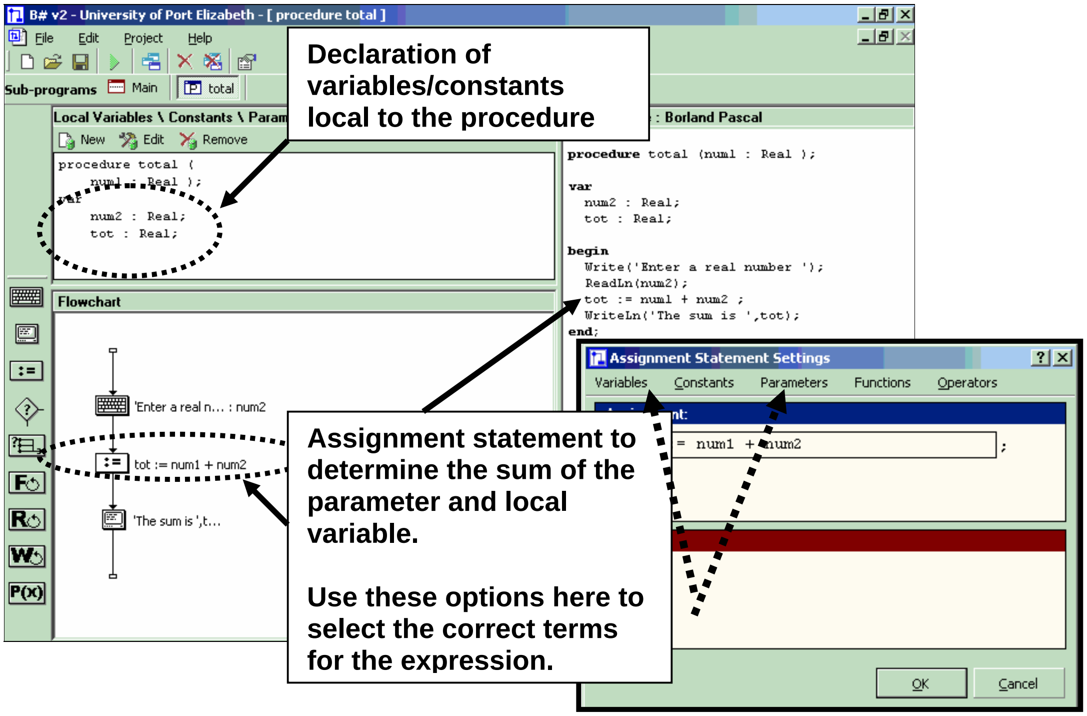
```
procedure total (num1 : Real );

begin
end;
```

Note appearance of parameter

A procedure does not have a *return statement*

The flowchart for the procedure is created using the skills that you have already acquired in all previous practicals – variables and programming constructs are used and declared as required. Any of the programming constructs learnt about may be used in the procedure's flowchart.

In the case of the **`total`** procedure, the user is required to enter another real number that is added to the parameter to produce a result that is displayed. Once these constructs has been added to **`total`**'s flowchart, the following will be displayed:



```
procedure total (
    num1 : Real );
var
    num2 : Real;
    tot : Real;
```

'Enter a real n… : num2

tot := num1 + num2

'The sum is ',t…

```
procedure total (num1 : Real );

var
  num2 : Real;
  tot : Real;

begin
  Write('Enter a real number ');
  ReadLn(num2);
  tot := num1 + num2 ;
  WriteLn('The sum is ',tot);
end;
```

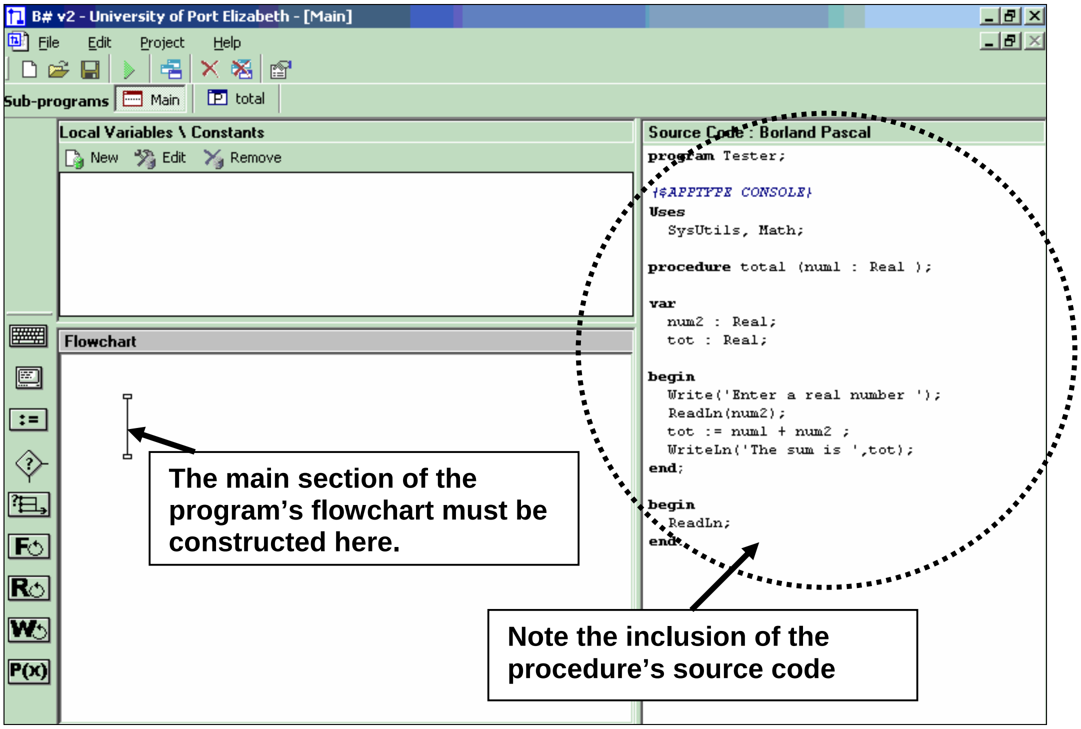
```
= num1 + num2 ;
```

Declaration of variables/constants local to the procedure

Assignment statement to determine the sum of the parameter and local variable.

Use these options here to select the correct terms for the expression.

In order to view the main section of the program (with all of the source code for the program), click on the ***Main*** tab at the top left hand side of the screen.  The following will then be displayed:


B# v2 - University of Port Elizabeth - [Main]
File
Edit
Project
Help
Sub-programs
Main
total
Local Variables \ Constants
New
Edit
Remove
Flowchart
Source Code : Borland Pascal

```
program Tester;

{$APPTYPE CONSOLE}
Uses
  SysUtils, Math;

procedure total (num1 : Real );

var
  num2 : Real;
  tot : Real;

begin
  Write('Enter a real number ');
  ReadLn(num2);
  tot := num1 + num2 ;
  WriteLn('The sum is ',tot);
end;

begin
  ReadLn;
end.
```

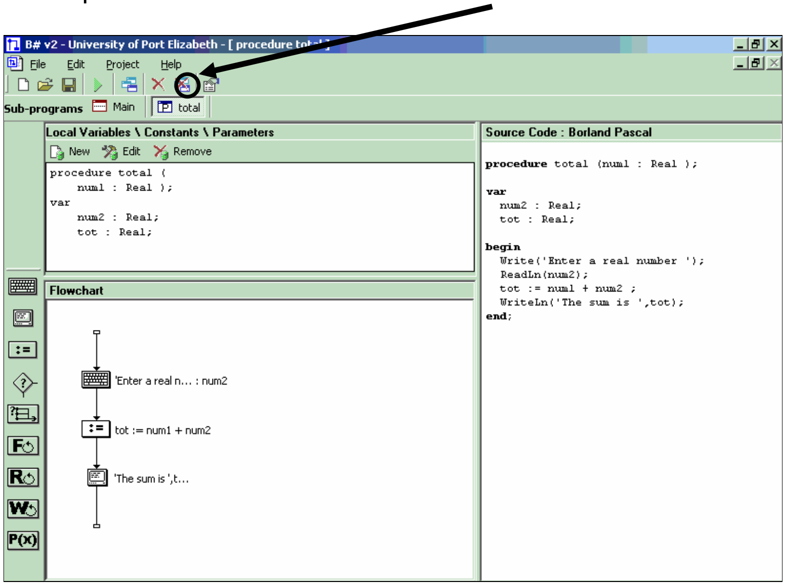
The main section of the program's flowchart must be constructed here.

Note the inclusion of the procedure's source code

The procedure **`total`** is now available when using the *procedure call* (P(X)) construct – this procedure will be listed under the ***Procedure Names*** available for use, together with the list of predefined procedures.

# Deleting a procedure/function in B#

In order to delete a function or procedure from a B# program, click on the tab for the relevant function/procedure and then select the ***Delete active routine*** menu icon.


B# v2 - University of Port Elizabeth - [ procedure total ]
File
Edit
Project
Help
Sub-programs
Main
total
Local Variables \ Constants \ Parameters
New
Edit
Remove
Source Code : Borland Pascal
Flowchart
'Enter a real n... : num2
tot := num1 + num2
'The sum is ',t...

```
procedure total (
    num1 : Real );
var
    num2 : Real;
    tot : Real;
```

```
procedure total (num1 : Real );

var
  num2 : Real;
  tot : Real;

begin
  Write('Enter a real number ');
  ReadLn(num2);
  tot := num1 + num2 ;
  WriteLn('The sum is ',tot);
end;
```

## Compulsory Practical Tasks

### ***Task 1 : Using B# to solve a problem***

***In B#***, write a function called **`isPrime`** that accepts a positive integer and returns `true` if the number is prime, otherwise returns `false`.

Write a ***B#*** program that continuously requests a user for an integer until a non-zero positive integer is entered and then uses the **`isPrime`** function and displays whether the number entered is prime or not.

### ***Task 2 : Using Delphi to solve a problem***

***In Delphi*** write a function called **`f`** that accepts any integer, **`x`**, and returns the value of $\mathbf{x^2-4x+10}$.

Write a ***Delphi*** program that displays a table of **`x`** values from **–5** to **+5** and the corresponding value of **`f(x)`**. The table must look something like:

```
 X    f(x)
-5    55
-4    42
-3    31
 :     :
```

## ***Task 3 : Using B# to solve a problem***

***In B#***, write procedure called **`displayFactors`** that accepts a positive integer and displays all the factors of the integer.

Write a ***B#*** program that continuously requests a user for an integer until a non-zero positive integer is entered and then displays its factors using the **`displayFactors`** procedure.

## ***Task 4 : Using Delphi to solve a problem***

***In Delphi***, write procedure called **`isVowel`** that accepts a character and displays whether or not it is a vowel (Hint*: The vowels in the alphabet are **a**, **e**, **i**, **o** and **u***).

Write a ***Delphi*** program that continuously requests a user for a character until the user enters an **`X`**. For each character entered, using the procedure **`isVowel`**, the program must display whether the character is a vowel or not.

## ***Task 5***

Using ***either B# or Delphi***, write a program that does the following:

Write a function called **`determinant`** that accepts real numbers **a**, **b** and **c** that are co-efficients for a polynomial equation $ax^2+bx+c$ and returns the determinant of the equation. In other words, it returns
$b^2-4ac$.

Write a program that asks the user for the values of **a**, **b** and **c** that are co-efficients for a polynomial equation $ax^2+bx+c$ and, using the function **`determinant`**, displays the roots of the polynomial equation. The roots are determined as follows:

- If the **`determinant`** is negative, then there are no real roots
- If **a** is 0, then there are no real roots
- If the **`determinant`** is zero, then there is one root with the value of ${}^{-b}/_{2a}$
- If the **`determinant`** is positive, then there are two roots, with values of ${}^{-b+\sqrt{}\text{determinant}}/_{2a}$ and ${}^{-b-\sqrt{}\text{determinant}}/_{2a}$

### Optional Practical Tasks

## ***Task 6***

Write a program that asks the user for a positive integer number, **n**, and uses the **`isPrime`** function written earlier to determine and display the first **n** prime numbers.

# WRA101 : Practical 10

## Week 11 : 29 April - 1 May 2003

*Sections Covered*

User defined functions and procedures (pp 194 – 202)

*Objectives*

- To understand the modular structure of programs
- To practice the use of user-defined functions and procedures
- To practice the writing of functions and procedures

*Practical Preparation*

- Using either a flowchart or pseudocode, plan the solution for each of the compulsory practical tasks.
- Make use of the function/procedure forms for deciding on the use of functions or procedures and their parameters

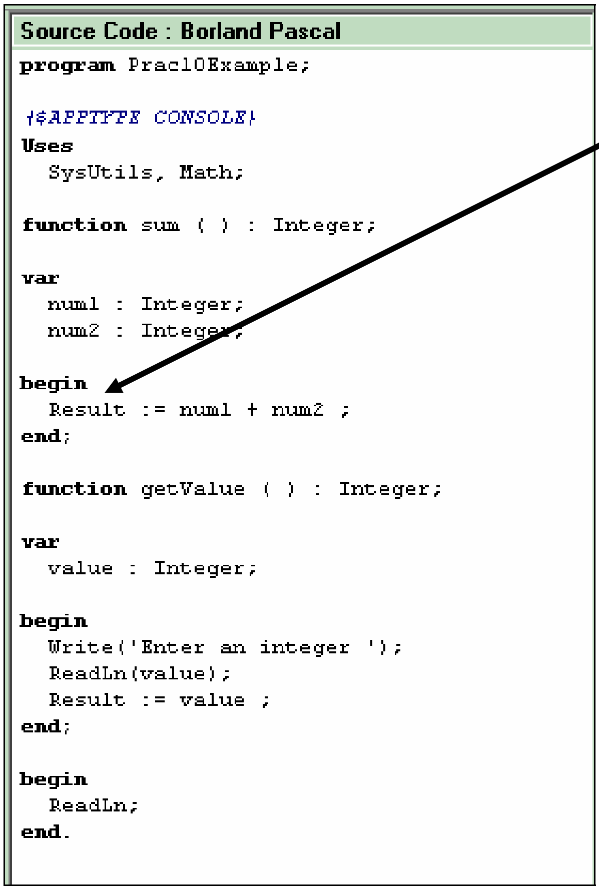
***YOU WILL NOT RECEIVE ASSISTANCE DURING THE PRACTICAL SESSION IF YOU DO NOT HAVE YOUR PREPARATION WITH YOU. YOU WILL BE REQUIRED TO PRESENT IT TO THE STUDENT ASSISTANT BEFORE RECEIVING ASSISTANCE.***

***Remember to make backup copies on your personal diskette of all your* `.dpr` *and* `.bpf` *files only before you leave the laboratory. Make a copy of these files on your* `Homes` *folder as well.***

*Something to remember when defining your own functions and procedures in B#*

The current version of B# does not support the changing of data type of a function nor the reordering of user defined function/procedure declarations within a program – these will be supported in the new version due at the end of 2003.  B# also currently does not support reference parameters.

Consider the following example of source code generated by B#.  If you wish to change the data type of function sum to real, you would have to delete function real and re-enter it.  The reordering of functions/procedures in B# is explained below.



**Source Code : Borland Pascal**

```
program Prac10Example;

 {$APPTYPE CONSOLE}
Uses
  SysUtils, Math;

function sum ( ) : Integer;

var
  num1 : Integer;
  num2 : Integer;

begin
  Result := num1 + num2 ;
end;

function getValue ( ) : Integer;

var
  value : Integer;

begin
  Write('Enter an integer ');
  ReadLn(value);
  Result := value ;
end;

begin
  ReadLn;
end.
```

The following statements are required here to gather the data for the function **`sum`**:

**`num1:=getValue;`**
**`num2:=getValue;`**

Adding these statements would result in errors due to rules governing the scope of variables.

The only way to fix this in B# is

- leave function **`sum`** as it is
- delete function **`getValue`**
- add function **`getValue`** again (B# will then place it above function **`sum`**)
- make the necessary changes to function **`sum`** to incorporate the required function calls

The rule of thumb is then:  first declare the functions/procedures that will call others (eg function **`sum`** above), ***<u>but leave the calling statements out</u>***.  Then enter the functions/procedures to be called (eg function **`getValue`** above), and then go back and edit the functions/procedures doing the calling (eg function **`sum`** above).

## Compulsory Practical Tasks

### ***Task 1 : Using B# to solve a problem***

In B#, write a simple calculator program that continuously displays the following menu to the user until the user chooses to terminate processing:

```
Calculator Menu
---------------
   Addition
   Subtraction
   Multiplication
   Division
   Exponential
   Stop processing
```

The following functions/procedures must be written and used by the program:

| ***Function/Procedure name*** | ***Pre-condition*** | ***Post-condition*** |
|---|---|---|
| **getChoice** | None | Continuously requests the user for a menu selection until a valid choice is made (i.e. 1, 2, 3, 4, 5 or 6). This valid choice is then returned. |
| **getNumber** | None | Requests the user to enter a real number and returns this value |
| **Add** | Accepts two real numbers | Returns the sum of the two real numbers |
| **Subtract** | Accepts two real numbers, the first of which is smaller than or equal to the second | Subtracts the first number from the second and returns the result of this operation |
| **Multiply** | Accepts two real numbers | Returns the product of the two real numbers |
| **Divide** | Accepts two real numbers, the second of which is non-zero. | Returns the quotient (result of first divided by second) |

The program processes each choice as follows:

| *Option* | *Description* |
|---|---|
| `Addition` | The user is required to enter two numbers that are added together and the sum is displayed. |
| `Subtraction` | The user is required to enter two numbers, the smaller of which is subtracted from the larger, and the result of the operation is displayed. |
| `Multiplication` | The user is required to enter two numbers that are multiplied together and the product is displayed. |
| `Division` | The user is required to enter two numbers. If the second is non-zero, the result of the first divided by the second is displayed; otherwise an appropriate message is displayed. |
| `Exponential` | The user is required to enter two numbers. The result of the first number raised to the power of the second is displayed. |

## *Task 2 : Using Delphi to solve a problem*

You have been asked by a friend of yours, who owns a small cafeteria, to write a simple menu-driven computer program to help keep track of sales at a till. When the program runs, it must display a menu that looks something like:

```
MAIN MENU:
----------
1. New sale
2. New purchase
3. Display sub-total
4. Tender amount
5. Display daily information
6. Exit
```

- When the program starts, it must initialize the daily sales to zero, (every time a sale is made, the daily sales is increased)
- When a new customer arrives at the till, option 1 is selected to start a new sale. The sub-total for that customer’s purchases is initialized to zero.
- For each type of item the customer wishes to purchase, the cashier will select option 2. It will ask for the quantity and unit cost of the items bought. Quantity and unit cost must both be positive. Once these values have been obtained, a cost is calculated as **quantity x unit cost** and added to the sub-total. The daily sales value is also updated by this cost.
- To view the current sub-total for a customer, the cashier can select option 3, which simply displays the sub-total on the screen.
- Once all the items have been rung up and the customer wishes to pay, the cashier chooses option 4. The amount tendered is asked (it must be positive) and subtracted from the sub-total. If the sub-total is now negative, then the customer is owed change. The change is displayed on the screen and the sub-total is set to zero. If the sub-total is still positive, then the customer did not tender enough money.
- If a display of the total daily sales total is required, then option 5 can be selected and the total will be displayed on the screen.
- Option 6 exits the program.

Copy the file **`F:\Courses\WRA101\Practical 10\Prac10Task2.dpr`** to your usercode on either the **`Homes`** or **`C:\Temp`** folder. This file contains only the code for the main body of the program and must ***NOT*** be changed. Only other functions and/or procedures may be added.

***In Delphi***, modify this program to include the individual functions and/or procedures to implement options 1 – 5 as described above. The program must also include the **`mainMenu`** procedure that continuously does the following until the user requests to terminate the process:

- Display the menu
- Allow the user to make a valid choice (i.e 1 – 6)
- Perform the correct action depending on the choice made. Each of these actions was described above.

## *Task 3*

The square root of a number **`n`** can be approximated by repeated calculation of the formula:

**NextGuess = (LastGuess + n ÷ LastGuess) ÷ 2**

The initial guess will be set to **`n`** and the next guess repeatedly calculated until the next guess and the last guess differ by a very small amount (in this case, use 0.001).

Write a ***Delphi*** procedure/function called **`squareRoot`** that implements the approximation method above. (Hint: You will need to store both **`nextGuess`** and **`lastGuess`**. Before each recalculation of **`nextGuess`**, the value that was in **`nextGuess`** must be assigned to **`lastGuess`** since the previous next guess is the current last guess.)

Write a ***Delphi*** program that will compare your **`squareRoot`** function/procedure with the built-in **`sqrt`** function. Test for all the values of **`n`** in the range **`1..20`**, displaying the values returned by **`squareRoot`** and **`sqrt`**, as well as the difference between the values returned (a positive amount). An example of what the output should look like is given below:

```
  n        sqrt(n)  squareRoot(n)     difference
------------------------------------------------
  1        1.00000        1.00000   0.0000000000
  2        1.41421        1.41421   0.0000000000
  3        1.73205        1.73205   0.0000000024
  4        2.00000        2.00000   0.0000000929
  5        2.23607        2.23607   0.0000000000
  6        2.44949        2.44949   0.0000000000
  7        2.64575        2.64575   0.0000000000
  :            :             :            :
```

## Optional Practical Tasks

### ***Task 4***

Write a ***Delphi*** procedure/function that will accept a real number representing an amount of money (e.g. 112.75 represents R112.75) and returns the **minimum** number of R200, R100, R50, R20, R10, R2, R1, 50c, 20c, 10c, 5c, 2c, 1c notes and coins needed to cover that amount. For example, for R112.75, the minimum number of notes and coins needed is 1xR100, 1xR10, 1xR2, 1x50c, 1x20c and 1x5c.

Write a ***Delphi*** program to test your procedure/function above by repeatedly asking the user of an amount of money and displaying the notes and coins needed (only those with non-zero amounts) until the user enters a non-positive amount of money.

### ***Task 5***

Suppose that the athletic department at UPE wants a program for its secretarial staff that can be used to determine the academic eligibility of its athletes for the next academic year. This eligibility check is made at the end of each of the first three years of the student's academic career. Eligibility is determined by two criteria: the number of hours that the student has successfully completed and the student's annual average (AVG). To maintain eligibility, a student must have completed at least 25 hours with a minimum AVG of 60% by the end of the first year. At the end of the second year, 50 hours must have been completed and a minimum AVG of 65% and at the end of the third year, 85 hours and a minimum of 70% AVG.

For each student, the program must ask for the various hours and AVG marks and display a report if the student meets the criteria at each year. If a student has not completed a year, then the hours and AVG must be entered as 0.

The program must allow for multiple students to be tested for eligibility and at the end must display the number of eligible students at each year level and the percentage they make up of all the students entered.

Implement a ***Delphi*** program, using procedures and functions to suit the above case study.

# WRA101 : Practical 11
## Week 12 : 6 - 8 May 2003

*Sections Covered*
User defined functions and procedures (pp 194 – 202)
Value and reference parameters

*Objectives*

- To understand the modular structure of programs
- To practice the use of user-defined functions and procedures
- To practice the writing of functions and procedures with value and reference parameters

*Practical Preparation*

- Using either a flowchart or pseudocode, plan the solution for each of the compulsory practical tasks.
- Make use of the function/procedure forms for deciding on the use of functions or procedures and their parameters

***YOU WILL NOT RECEIVE ASSISTANCE DURING THE PRACTICAL SESSION IF YOU DO NOT HAVE YOUR PREPARATION WITH YOU. YOU WILL BE REQUIRED TO PRESENT IT TO THE STUDENT ASSISTANT BEFORE RECEIVING ASSISTANCE.***

***Remember to make backup copies on your personal diskette of all your `.dpr` and `.bpf` files only before you leave the laboratory. Make a copy of these files on your `Homes` folder as well.***

## Compulsory Practical Tasks

### ***Task 1 : Using Delphi to solve a problem***

You are required to write a ***Delphi*** program that determines if a first year Computer Science and Information Systems student may continue with second semester modules.

a) Write a function/procedure **`GetOneStudent`** that asks a student for both of his/her WRA101 and WRU101 marks (each as a real number). The function/procedure must return both of these marks. The function/procedure must ensure that each of the marks is in the range 0 to 100.

b) Write a function/procedure **`CanContinue`** that takes as input two marks, one for WRA101 and one for WRU101. Each of these marks is in the range 0 to 100.

The function/procedure must return the value **`true`** if both of these marks are at least 50, otherwise the function/procedure must return the value **`false`**.

c) Write a function/procedure **`DisplayDecision`** that takes as input two marks, one for WRA101 and one for WRU101. Each of these marks is in the range 0 to 100. If the student can continue with the second semester modules (use **`CanContinue`** above), an appropriate message and the average mark for the two modules is displayed, otherwise the student is informed that he may not continue.

d) Now, write the main program that continuously does the following for any number of students until the lecturer requests to terminate the process:
   - Ask for the WRA101 and WRU101 marks and display a message to indicate whether the student can continue with second semester modules or not.

## ***<u>Task 2 : Using B# to solve a problem (using only value parameters)</u>***

You are required to write a ***<u>B#</u>*** program that determines in what year a country's total population exceeded a specified value.

a) Write a ***<u>B#</u>*** function/procedure **`PopulationTotal`** that accepts 2 positive values, namely the current population (in millions, eg a value of 1.5 means 1.5 million people) and the growth rate (eg 0.14 means 14%). The function/procedure determines and returns the total population based on the current population and growth rate. For example, if the current population for a country is 1.165 million people, and the annual population growth rate is 10%, then the total population is 1.2815 million people after 1 year.

b) Write a ***<u>B#</u>*** function/procedure **`Over180Million`** that accepts a positive value representing the population (in millions). The function/procedure determines whether the population is over 180 million people, and returns a value of true if this is so, otherwise returns a value of false.

c) Write a ***<u>B#</u>*** program that requests the user for a year (always at least 2000), the population of Brazil in that year (in terms of millions of people) and the annual population growth rate (always as a real number in the range 0 – 1), and then uses the two functions/procedures written above to determine and display the year in which Brazil's population first exceeded/exceeds 180 million people. The program must also display Brazil's total population in this year.

d) In 2001 the population of Brazil was 97.6 million people and the rate in which the population grows is 18.4% per year. Use your <u>B#</u> program to answer the following two questions:

   i) Using this test data, what is the year displayed by the program written in c)?

   ii) What is the total population displayed by the program written in c)?

## Optional Practical Tasks

### ***Task 3***

You are required to write a ***Delphi*** program that will calculate and display the total balance on your bank accounts and the name of the account with the highest balance.

a) Write a function/procedure **`GetAccountInfo`** that asks a user for an account's identifying name (as a single character) and the balance on that account. The function/procedure must return these values.

b) Write a function/ procedure **`ProcessAccount`** that takes as input the following:

- The account name and value of its balance for the account that has the maximum balance of all the accounts processed so far
- The total balance of all the accounts processed so far
- The account name and value of its balance for the account that is currently being processed

The function/procedure must update the account name and value of its balance so that they reflect the account that has the maximum balance of all the accounts processed so far, as well as the total balance of all the accounts processed so far.

c) Now, write a main program that will allow the user to process as many accounts as he/she wishes (he/she does not necessarily know beforehand how many accounts he/she wishes to process). Once the details for all the accounts have been entered, the program must display the total balance on the accounts and the name of the account with the highest balance.

# Appendix D

## Investigation Assessment Material

The assessment material presented to all participants in the investigation consisted of theoretical and practical assessments.

Theoretical assessments required the subjects to answer programming related questions using pen and paper. There were three of these assessments; one occurring 2 weeks into the treatment period, the second occurring 7 weeks into the treatment period and the final administered 5 weeks after the completion of the treatment.

Practical assessments required the subjects to answer programming related questions in a controlled laboratory environment using the programming languages identified for use in the investigation. There were two of these assessments,; the first occurring 4 weeks after commencement of the treatment period and the second a week after completion of the treatment period.

This appendix includes the mapping of each question/task contents to the relevant independent variable used in the quantitative analysis.

## D.1 Pen-and-Paper Assessment Materials (Material M6)

### *D.1.1 Problem-solving Theoretical Assessment*

| Question | Independent Variable |
|---|---|
| Question 1 | Th1Q1 |
| Question 2 | Th1Q2 |
| Question 3 | Th1Q3 |

**QUESTION 1: FLOWCHART & PSEUDO-CODE** **(20)**

$x^y$ (x to the power y) is defined for any integers as:

$$x^y = x * x * x * \ldots * x \text{ (y times)}$$
$$x^{-y} = 1 / x^y$$
$$x^0 = 1$$

1.1 Draw a flowchart that computes $x^y$ for any integer inputs x and y. (8)

1.2 Give the pseudo-code for computing $x^y$ for any integer inputs x and y (8)

1.3 Test the solution for the following values of x and y

a) x = 3 and y = 2 (2)

b) x = 4 and y = -2 (2)

**QUESTION 2: FLOWCHART → PSEUDO-CODE** **(10)**

Consider the following flowchart

2.1 What are the inputs and outputs of this program? (2)

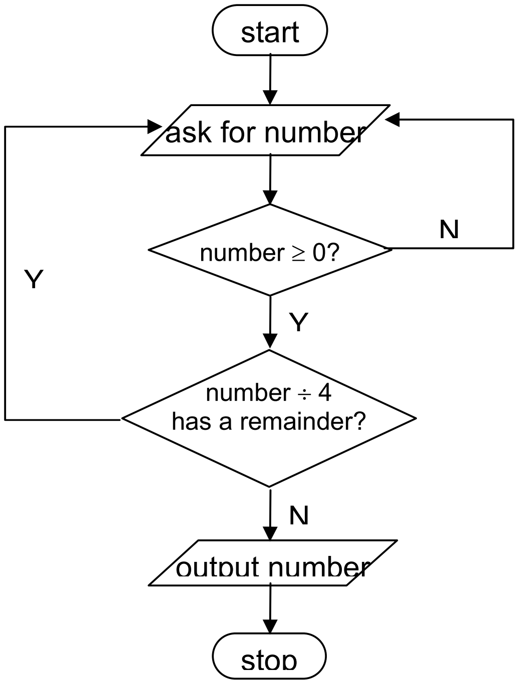
2.2 Write the pseudo-code for that flowchart (5)

2.3 Explain what the program does with the following inputs: (3)

a) 2

b) -5

c) 8



## QUESTION 3: PSEUDO-CODE → FLOWCHART **(20)**

3.1 A number m is a factor of another number n, if n÷m has no remainder. E.g. 1,2,3,6 are all factors of 6 (each one divides completely into 6).

Write the pseudo-code that will ask the user for a number and will then display all the factors of the number to the user. (6)

3.2 A number m is a prime number if it has only two factors, namely 1 and m (the number itself).

E.g. the factors of 8 are 1,2,4,8. Therefore 8 is not prime.
e.g. the factors of 7 are 1,7. Therefore 7 is prime.

Examine the pseudo-code below:

```
Inputs : N

Outputs: prime or not prime

Process:
 1.ask for N
 2.count = 1, #factors = 0
 3.if (N mod count) = 0 then
     3.1.#factors = #factors + 1
 4.count = count +1
 5.if count <= N
     5.1.then goto 3
 6.if #factors > 2
    then
     6.1.display prime
    else
     6.2.display not prime
```

3.2.1 Does this algorithm actually compute whether a number is prime?
Explain by using N = 13. If it does not compute whether the number is prime, what must be changed to correct it? (3)

3.2.2 Draw the flowchart of the algorithm above. (11)

### *D.1.2 Syntax-based Pen-and-Paper Assessment*

| Question | Independent Variable |
|---|---|
| Section A | Th2MC |
| Section B: Question 1 | Th2Q1 |
| Section B: Question 2 | Th2Q2 |

## Section A: 24 marks

1 How many of the following are **valid** variable names in Object Pascal? 1

```
isFactor, factorial, root 1, beginCount, end, _fx,
1stAnswer, #Students, population2003
```

a) 3 c) 5
b) 4 d) 6

2 What will be displayed by the following program extract? 1

```
var
  value:integer;
begin
  value:=3;
  write(value);
  writeln('value');
  writeln(value, value);
end.
```

a)
```
3value
33
```

c)
```
value3
valuevalue
```

b)
```
3
value
3
3
```

d)
```
3 value
value value
```

3 What will be displayed by the following program extract? 1

```
var
  answer:boolean;
begin
  answer:=10>5;
  writeln('The value of answer is ', answer);
end.
```

a) `The value of answer is answer`
b) `The value of answer is 10>5`
c) `The value of answer is true`
d) The program will not compile

4 What will be displayed by the following program extract? 1

```
var
  x, y, z:integer;
Begin
  x:=20;
  y:=2*x - 10;
  z:=x*x – 10*y + 7;
  x:=10;
  writeln(z);
End.
```

a) 7 c) -193
b) 107 d) 93

5 What will be displayed by the following program extract? 1

```
var
   x, y, z:integer;
Begin
  x:=7;
  y:=9;
  z:=x+3 * y-10;
  writeln(z);
End.
```

a) 80 c) 4
b) -10 d) 24

6 What will be displayed by the following program extract? 1

```
var
  x, y:integer;
Begin
  x:=22;
  y:=7;
  writeln(x div y,’,’,x mod y);
End.
```

a) 3,1 c) 3.0,0.142857
b) 1,3 d) 0.142857,3.0

Consider the following scenario when answering questions 7 and 8. When buying media from the department, the cost per item depends on what was bought, as well as how many of that item are bought at one time:

| Code | Type | Quantity | Cost per item |
|---|---|---|---|
| D | Disk | 1 | R3.00 |
| D | Disk | 2-4 | R2.75 |
| D | Disk | more than 4 | R2.50 |
| C | CD | 1 | R6.00 |
| C | CD | 2-4 | R5.50 |
| C | CD | more than 4 | R5.00 |

*A program is required that asks the user for the type of media bought (e.g. D for disk or C for CD) and the number bought. The cost is then calculated and this amount is displayed to the user.*

7 What variable(s) is/are required as **input** to the solution to the problem? 1

a) `Cost`
b) `NumDisks, NumCDs, Cost`
c) `MediaCode, Quantity`
d) `C, D, Quantity`

8 What variable(s) is/are required as **output** from the solution to the problem? 1

a) `Cost`
b) `NumDisks, NumCDs, Cost`
c) `MediaCode, Quantity`
d) `C, D, Quantity`

9 What will be displayed by the following program extract? 1

```
var
  a, b:integer;
begin
  a:=25;
  b:=a div 7 – 2 - 3;
  writeln(b);
end.
```

a) 12
b) 2
c) -1
d) -2

10 What will be displayed by the following program extract? 1

```
var
  number:integer;
begin
  number:=10;
  if number>10
  then
    writeln(’ABC’)
  else
    writeln(’DEF’);
end.
```

a) ABC

b) DEF

c) ABC
DEF

d) Nothing will be displayed

11 What will be displayed by the following program extract if the user enters a ’A’? 1

```
var
  c:char;
begin
  write(’Enter a character :’);
  readln(c);

  case c of
    ’a’, ’e’, ’i’, ’o’, ’u’: writeln(’vowel’);
    ’b’.. ’d’,
    ’f’.. ’h’,
    ’j’.. ’n’,
    ’p’.. ’t’,
    ’v’.. ’z’ : writeln(’consonant’);
    ’ ’       : writeln(’space’)
    else writeln(’non-letter’);
  end;
end.
```

a) vowel

b) consonant

c) space

d) non-letter

12 What will be displayed by the following program extract if the user enters a 6? 1

```
var
  x:integer;
begin
  write(’Enter a number :’);
  readln(x);
  if x=2 then
     write(’2’);
  if x=4 then
     write(’4’);
  if x=6 then
     write(’6’)
  else
     write(’???’);
end.
```

a) ???

b) 2

c) 4???

d) 24???

13 Which condition(s) would be tested by the following program extract if x had a value of 6? 2

```
if x>4                //line 1
then write(’A’)
else
  if x>=2             //line 2
  then
    write(’B’)
  else
    write(’C’);
```

a) line 1 and line 2

b) line 1 only

c) line 2 only

d) impossible to say

14 Which condition(s) would be tested in the following program extract if x were assigned the value 2? 2

```
if x>7                //line 1
  then write(’A’);
if x>=25              //line 2
  then write(’B’);
if x<5                //line 3
  then write(’C’);
```

a) line 1 only

b) line 1, line 2 and line 3

c) line 1 and line 2 only

d) line 1, line 3 and line 4

15 What will be displayed by the following program extract? 2

```
for j := 1 to 4 do
begin
   for k := 1 to j do
   begin
      write(k);
   end;
   writeln;
end;
```

a)
```
4444
333
22
1
```

b)
```
1
22
333
4444
```

c)
```
1
12
123
1234
```

d)
```
1234
123
12
1
```

16 Given the function definition below: 3

```
function fancy(x:real; y:integer):boolean;
```

Which of the calls to function `fancy` are **valid** in the following program extract?

```
var
  x, y:integer;
  a   :boolean;
begin
  x:=3; y:=5;
  y:=fancy(10,5);                        //line 1
  writeln(fancy(x,y));                   //line 2

  if fancy(x/y,y)                        //line 3
  then writeln('very fancy!!');

  a:=fancy(cos(2.36),trunc(sin(15.7)));  //line 4
end.
```

a) Line 1, 2, 3 and 4
b) Line 2,3 and 4
c) Line 2 and line 4 only
d) Line 1, 3 and 4

17 What is displayed by the following program extract? 3

```
function DoIt(x, y:integer):integer;
var
  a:integer;
begin
  a:=x+y;
  y:=5;
  result:=a;
end;

var
  a, b, x, y:integer;
begin
  a:=5;
  x:=7;
  y:=3;
  b:=15;
  b:=DoIt(y, x);
  writeln(a, ’,’, b);
end.
```

a) 5,5          c) 10,10
b) 5,15         d) 5,10

## Section B: 26 marks

1.1 Write a function that accepts two integer values, x and y, and calculates the highest common factor between x and y. 9

E.g.
The factors of 18 are 1,2,3,6,9,18
The factors of 24 are 1,2,3,4,6,8,12,24
Common factors between 18 and 24 are 1,2,3,6
The highest common factor between 18 and 24 is 6

1.2 Using the function written in (1.1), write a program that *asks* for two positive integers, x and y and displays each of them as the product of the highest common factor and another number. *You must not rewrite the function written in (1.1), only use it!* 5

E.g. if x=18 and y=24, then the program displays

18 = 6 x 3
24 = 6 x 4

2 The game of *cattle* (a variation of master mind) is played as follows:

*A random 4 digit number (H) is generated and the user attempts to guess it as follows:*

- *for every digit in the user's guess which is in H* ***and*** *in the correct position, the user scores 1 bull*
- *for every digit in the user's guess which is in H* ***but not*** *in the correct position, the user scores 1 cow*

*The user keeps guessing until he/she guesses H correctly.*

For example:

| H | Guess | Bulls | Cows | Comment |
|---|---|---|---|---|
| 1234 | 6284 | 2 | 0 | 2 & 4 in correct positions (1 bull each) |
| 1234 | *2453* | 0 | 3 | 2,4,3 in H, but wrong position (1 cow each) |
| 1234 | 1*122* | 1 | 3 | 1st 1 in correct position (1 bull)<br>2nd 1, 2, 2 in H, but wrong position (3 cows) |
| 1234 | *4*23*1* | 2 | 2 | 2,3 in correct position (2 bulls)<br>1,4 in H, but wrong position (2 cows) |
| 1234 | 1*111* | 1 | 3 | 1st 1 in correct position (1 bull)<br>other 1's in H, but wrong position (3 cows) |

Assume that the following functions have been written for you:

```
function getDigit(number, which: integer) :integer;
//PRE-CONDITIONS:
//  number is between 0000 and 9999
//  which is between 1 and 4
//POST-CONDITIONS:
//  getDigit returns one of the digits in number
//  specified by which. If which=1 then the
//  leftmost digit in number is returned, if
//  which=2 then the 2nd leftmost digit in number is
//  returned, etc.
//  e.g. getDigit(1024,3) returns 2
```

```
function getH :integer;
//POST-CONDITIONS:
//  returns a random number between 0000 and 9999
```

```
function getCows(number, H:integer) : integer;
//PRE-CONDITIONS:
//  number and H are in the range 0000..9999
//POST-CONDITIONS:
//  returns the number of cows scored
```

2.1 Write a procedure or function that accepts H and the user's guess and returns the number of bulls scored. 7

2.2 Write the main program that allows the computer to play *cattle* with a user as described above. (You **must not** rewrite any functions or procedures. Only write the code for the main section of the program!) 5

## D.2 Practical Assessment Materials (Material M7)

### *D.2.1  Introductory Level Practical Assessment*

| Question in each paper | Independent Variable |
|---|---|
| Task 1 | Pr1Q1 |
| Task 2 | Pr1Q2 |

To discourage and eliminate potential plagiarism, each subject was randomly assigned one of each of the following question papers.  A total time of 20 minutes was allotted in which the two tasks were to be completed.

### PAPER 1

***Task 1 : 8 marks***

***In B#,*** write a program that requests the user for the total number of cyclists in a race.  For each cyclist, the time taken (in hours, for example, 4.7 hours) to complete the race is entered.  The program must then display the average time taken by a cyclist to complete the race.

***Task 2 : 12 marks***

***In Delphi***, write a program that requests the user for an integer.  For an integer entered that is not in the range 1 – 500, an appropriate error message should be displayed; otherwise the program must display ***a count*** all the factors of the entered integer.

For example, if the user enters **10**, the program displays

```
10 has 4 factors
```

### PAPER 2

***Task 1 : 8 marks***

Write a program that requests the user for the total number of students in a class. For each student, the mark obtained in a test (for example, 55) is entered.  The program must then display the maximum mark obtained in the test.

***Task 2 : 12 marks***

Write a program that requests the user for an integer (say *n*).  For an integer entered that is not in the range 1 – 500, an appropriate error message should be displayed; otherwise the program must display all the positive integers in the range 1 – *n* with each factor of *n* being replaced by a *****.

For example, if the user enters **10**, the program displays

```
* * 3 4 * 6 7 8 9 *
```

## PAPER 3

***Task 1 : 8 marks***

Write a program that requests the user for the total number of passengers on an aeroplane. For each passenger, the mass of the luggage (in kilograms, for example, 12) is entered. The program must then display the mass of the lightest luggage (that is, the minimum mass).

***Task 2 : 12 marks***

Write a program that requests the user for an integer (say *n*). For an integer entered that is not in the range 1 – 500, an appropriate error message should be displayed; otherwise the program must display all the positive integers in the range 1 – *n* in reverse order with each non-factor of *n* being replaced by a **`*`**.

For example, if the user enters **`10`**, the program displays

```
10 * * * * 5 * * 2 1
```

## PAPER 4

***Task 1 : 8 marks***

Write a program that requests the user for the total number of printers in the computer laboratories. For each printer, the total number of pages printed is entered. The program must then display the average number of pages printed by a printer.

***Task 2 : 12 marks***

Write a program that requests the user for an integer (say *n*). For an integer entered that is not in the range 1 – 50, an appropriate error message should be displayed; otherwise the program must display ***a count*** of the first *n* multiples of 13 that are also multiples of 3.

For example, if the user enters **`7`**, the program displays

```
No of multiples of 13 that are also multiples of 3 : 2
```

## PAPER 5

***Task 1 : 8 marks***

***In B#,*** write a program that requests the user for the total number of modules offered at UPE. For each module, the number of students registered is entered. The program must then display the number of students registered for the module with the most number of students registered.

***Task 2 : 12 marks***

***In Delphi***, write a program that requests the user for an integer. For any positive integer (zero excluded), the program must display all of its factors; otherwise an appropriate error message must be displayed.

For example, if the user enters **`20`**, the program displays

```
1 2 4 5 10 20
```

## PAPER 6

***Task 1 : 8 marks***

Write a program that requests the user for the total number of modules offered at UPE. For each module, the number of students registered is entered. The program must then display the number of students registered for the module with the most number of students registered.

***Task 2 : 12 marks***

Write a program that requests the user for an integer. For any positive integer (zero excluded), the program must display all of its factors; otherwise an appropriate error message must be displayed.

For example, if the user enters **`20`**, the program displays

```
1 2 4 5 10 20
```

## PAPER 7

***Task 1 : 8 marks***

Write a program that requests the user for the total number of cyclists in a race. For each cyclist, the time taken (in hours, for example, 4.7 hours) to complete the race is entered. The program must then display the average time taken by a cyclist to complete the race.

***Task 2 : 12 marks***

Write a program that requests the user for an integer. For an integer entered that is not in the range 1 – 500, an appropriate error message should be displayed; otherwise the program must display ***a count*** all the factors of the entered integer.

For example, if the user enters **10**, the program displays

```
          10 has 4 factors
```

## PAPER 8

***Task 1 : 8 marks***

Write a program that requests the user for the total number of passengers on an aeroplane. For each passenger, the mass of the luggage (in kilograms, for example, 12) is entered. The program must then display the average mass of the luggage on the aeroplane.

***Task 2 : 12 marks***

Write a program that requests the user for an integer (say *n*). For an integer entered that is in the range 1 – 50, the program must display all the integers in the range 1 – *n* in reverse order with each multiple of 5 being replaced by a *****; otherwise an appropriate error message should be displayed.

For example, if the user enters **12**, the program displays

```
          12 11 * 9 8 7 6 * 4 3 2 1
```

### *D.2.2 Intermediary Level Practical Assessment*

| Question | Independent Variable |
|---|---|
| Question 1 | Pr2Q1 |
| Question 2 | Pr2Q2 |
| Question 3 | Pr2Q3 |
| Question 4 | Pr2Q4 |

**Question 1 : 10 marks** (*15 minutes*)

Copy the file **`F:\Courses\WRA101\Prac Test\StudentQ1.dpr`** to your directory on **`C:\Temp`**. ***Should you require any assistance in copying this file, you will be penalised by 5 marks***.

Remove all the syntax errors from the file **`StudentQ1.dpr`** so that it compiles without any errors and works correctly. (10)

***Question 2 : 10 marks*** *(*15 minutes*)*

Copy the file **`F:\Courses\WRA101\Prac Test\StudentQ2.dpr`** to your directory on **`C:\Temp`**. ***Should you require any assistance in copying this file, you will be penalised by 5 marks***.

Consider the following function:

```
function Is1FactorOf2(number1, number2 : integer) : boolean;
begin
    result := (number2 mod number1 = 0);
end;
```

This function returns **True** if the first parameter passed (number1) is a factor of the second parameter (number2), and **False** if number1 is not a factor of number2. Thus:

`Is1FactorOf2(4, 20)` will return **True**, but
`Is1FactorOf2(3, 17)` will return **False**.

Making use of the skeleton program provided; write a program that requests a number from the user, and then displays all the factors of the number, as well as a count of the factors. **Make use of the function given above**.

(10)

**Question 3 : 25 marks** (*45 minutes*)

2.1 Write the following procedure:

```
procedure DisplayInts(a, b : integer);
{ Preconditions:  a and b are integers with a <= b.
Post Conditions:   All the integers between a and b
(including a and b) are displayed on one line and the cursor
is moved to the next line.  E.g.  the procedure call
DisplayInts(4, 9) will display 456789 on the screen.}
```

(6)

2.2 Making use of the procedure written above, write a program that continuously prompts the user for a number *(n)* until the user enters an number in the range 0 ≤ *n* ≤ 9. Depending on the user's input, a triangle is then displayed on the screen. E.g. if the user inputs 7, the following triangle should be displayed:

```
01234567
0123456
012345
01234
0123
012
01
0
```

If the user inputs 3, the following triangle should be displayed:

```
0123
012
01
0
```

After a triangle is displayed, the user should be asked the following question:
**Would you like to continue? (y/n)**
The entire process should be repeated until the user enters 'N' or 'n' as a response to the above question.

(19)
[25]

**Question 4 : 25 marks** (*45 minutes*)

Copy the file **`F:\Courses\WRA101\Prac Test\StudentQ4.dpr`** to your directory on **`C:\Temp`**. ***Should you require any assistance in copying this file, you will be penalised by 5 marks***. **Read the entire question before writing any code.**

**`StudentQ4.dpr`** contains the partial implementation of the following menu driven program:

```
The current student is: , 0
The current practical average mark is: 0.00
The current test average mark is: 0.00

MENU:
1. Enter New Student
2. Enter Practical Marks
3. Enter Test Marks
4. Display Class Mark and Outcome
5. Exit

Please enter a selection.
```

If option 1 is selected, the user will be prompted for the Name and Student Number of a student. When the menu is displayed again, the details at the top will be updated (e.g. `The current student is: Bob Smart, 203837282`).

If option 2 is selected, the user will be prompted for the number of practicals completed by the student. The user will then be prompted for the mark (out of 100) the student received for each practical. When the menu is displayed again, the details at the top will be updated (e.g. `The current practical average mark is: 83.65`).

If option 3 is selected, the user will be prompted for the marks of two tests (out of 100). When the menu is displayed again, the details at the top will be updated (e.g. `The current test average mark is: 79.5`).

If option 4 is selected, the class mark of the student will be displayed (assume that the practical mark counts 40% and the test mark counts 60% of the class mark) as well as whether the student may write the exam or not (e.g. `Bob Smart, 203837282 has a class mark of 81.23 and may write the exam`). A student needs a class mark of at least 40% to be allowed to write the exam.

The menu part of the program and the function `GetTestAverage` has already been written. You have to implement the functions/procedures below (see **`StudentQ4.dpr`** for their use). ***You should only add code to the specified area of* `StudentQ4.dpr`.**

| | | |
|---|---|---|
| `GetStudent` | Prompts the user and returns the name and student number of a student. | (4) |
| `GetPracAverage` | Prompts the user for the number of practicals and the mark of each practical, and returns the average. | (10) |
| `DisplayClassMark` | Displays the name, student number and class mark of the current student and whether the student may write the exam. | (11) |

## D.3 Final Examination (Material M9)

| Question | Independent Variable |
|---|---|
| Section A | ExMC |
| Section B: Question 1 | ExQ1 |
| Section B: Question 2 | ExQ2 |
| Section B: Question 3 | ExQ3 |
| Section B: Question 4 | ExQ4 |
| Section B: Question 5 | ExQ5 |

## Section A : 36 marks

1. What will be displayed by the following program extract? (1)

```
var
  c,d :integer;
Begin
  c:=4*4;
  d:=3;
  c:=6;
  writeln(c,d);
End.
```

a) 163 c) 63

b) 316 d) 36

2. What will be displayed by the following program extract, if the user enters a 7 when asked? (1)

```
var
  c :integer;
  d :char;

begin
  write('Please enter a number');
  readln(c);

  if (c=d) then
    writeln('Yippee');
  else
    writeln('NO');
end.
```

a) Yippee c) Program will not compile

b) NO d) Yippee
NO

3. Consider the following function prototype. (1)

```
function IsPositive(x:integer):bool;
```
//Postcondition: returns true if x > 0, otherwise it returns false

Which of the following is an invalid call to the function defined above?

a)
```
if IsPositive(7) then
    writeln('Its Positive');
```

b)
```
if (IsPositive(x)>0) then
    writeln('Its Positive');
```

c)
```
if (IsPositive(x)=true) then
    writeln('Thats right');
```

d)
```
if IsPositive(x) = IsPositive(y) then
   write('Same sign')
```

4. What will be displayed by the following program extract? (1)

```
var
  x :integer;

begin
  x:=32;
  while (x>=8)
  begin
    write(x,' ');
    x:=x div 2;
  end;
end;
```

a) `32 16 8`

c) `32 16 8 4`

b) `16 8`

d) `16 8 4`

5. Which of the following is the most appropriate definition for a function or procedure that accepts the length, width and height of a cube and returns the surface **and** volume of the cube? (1)

a) `function cube(length, width, height : integer; var surface, volume:real):real;`

b) `procedure cube(length, width, height : integer; var surface, volume:real);`

c) `function cube(length, width, height : integer; var surface :real): real;`

d) `procedure cube(length, width, height : integer; surface, volume : real);`

6. What will be displayed by the following program extract? (1)

```
var
  i :integer;

begin
  for i:=8 downto 3 do
    write(i,’ ‘);
end.
```

a) 8 7 6 5 4 3

b) 7 6 5 4 3

c) 3 4 5 6 7 8

d) 7 6 5 4

7. What will be displayed by the following program extract? (1)

```
var
  a,b :integer;
begin
  a:=4;
  b:=0;
  writeln(a mod b);
end.
```

a) 0

b) 4

c) Impossible to predict

d) Error: division by 0

8. What will be displayed by the following program extract? (1)

```
function h(x, y:integer):integer;
var
  count:integer;
begin
  result:=1;

  for count:=1 to x do
    if (x mod count=0) and (y mod count=0)
    then
      result:=count;
end;

Begin
  writeln(h(6,21));
  readln;
End.
```

a) 1
b) 2
c) 3
d) 4

9. What will be displayed by the following program extract? (2)

```
var
  i,j :integer;
begin
  i:=4;
  for j:=1 to i do
    inc(i)
  write(i)
end.
```

a) 1234

b) 123

c) Impossible to say

d) The loop will go into an infinite loop

10. Under what condition will the loop in the following program extract stop? (2)

```
var
  a :integer;
begin
  repeat
    readln(a);
  until (a <> 7);
  write(a);
end.
```

a) When the user enters any positive integer

b) When the users enters any integer except a 7

c) The loop will never stop

d) When the user enters a 7

11. Consider the following program extract. (2)

```
var
  input :integer;
begin
  if (input > 12) then
    write(‘monkey’);
  else
    if (input =< 4) then
      write(‘cheese’);
    else
      write(‘ice cream’);
end.
```

for what values of input would Ice Cream be displayed?

a) Whenever input is greater than 12 and less than or equal to 4

b) Whenever input is less than or equal to 4

c) Whenever input is greater than 4 and less than or equal to 12

d) Whenever input is less than or equal to 12

12. What will be displayed by the following program extract? (2)

```
var
  total,j :integer;
begin
  total:=0;
  j:=1;
  repeat
    total:=total + j;
    inc(j);
  until (total>17);
  writeln(total);
end.
```

a) `7`

b) `8`

c) `28`

d) `21`

13. What will be displayed by the following program extract: (3)

```
procedure modify(t1:integer; var t2:real;var t3:integer);
var
  m:integer;
begin
  m:=t1;
  t1:=t3;
  t2:=m*0.5;
  t3:=9;
end;
var
  time1,time3:integer;
  time2:real;
begin
  time1:=3;
  time2:=6.0;
  time3:=12;
  modify(time1,time2,time3);
  writeln(time1,’ ‘,time2:0:2,’ ‘,time3);
end.
```

a) `12 1.50 12`

b) `3 6.00 12`

c) `3 6.00 9`

d) `3 1.50 9`

14. Consider the following program extract: (3)

```
if a>=12 then
  case a of
    1..5:writeln('top 5');
    6..10:writeln('middle 5');
    11..20:writeln('bottom 5');
  else
    writeln('not ranked');
  end;
```

For what values of a will `top 5` be displayed?

a) whenever a is less than 5 and greater than 1

b) whenever a is less than or equal to 5 and greater than or equal to 1

c) top 5 will never be displayed

d) whenever a is less than 1 or greater than 5

15. What would be displayed by the following program extract? (3)

```
var
  a:array[1..5] of integer;
  b:integer;
begin
  for b:=1 to 5 do
    a[b]:=sqr(b)-6;
  b:=5;
  while a[b]>0 do
  begin
    write(a[b],' ');
    dec(b);
  end;
  readln;
end.
```

a) `19 10 3 -2`

b) `19 10 3`

c) `6 5 4 3`

d) `6 5 4`

16. Given a function with the following definition: (3)

```
function f(x:integer; y:real):boolean;
```

Which of the following lines are **invalid** ways of calling it?

```
Var
  a,b:integer;
  x,y:real;
  t:boolean;
Begin
  if f(a,x) then writeln(a);  //line 1
  a:=f(b,y);                  //line 2
  f(y,x);                     //line 3
  x:=f(10,y);                 //line 4
  t:=f(a,x);                  //line 5
  writeln(f(a,10.2));         //line 6
End.
```

a) All of them

b) Lines 3, 4 and 6

c) Lines 1, 2, 4 and 6

d) Lines 2, 3 and 4

17. What will be displayed by the following program extract? (4)

```
 var
   i,j:integer;
 begin
   for i:=0 to 3 do
   begin
     for j:=5-i downto i do
       write(i);
     writeln;
   end;
end.
```

a)
```
543210
4321
32
```

c)
```
000000
1111
22
```

b)
```
000000
11111
2222
333
44
5
```

d)
```
012345
1234
23
```

18. What would be displayed by the following program extract? (4)

```
Var
  a:array[1..3] of string;
  i,j:integer;
Begin
  a[1]:='the';
  a[2]:='fat';
  a[3]:='cat';

  for i:=1 to 3 do
    for j:=1 to 3 do
      write(a[j][i]);

  readln;
End.
```

a) `the fat cat`

c) `tfchaaett`

b) `thefatcat`

d) The program will not compile

## Section B : 36 marks

1 Write a function called `getDigit` that accepts two integers, `n` and `index`, which are both greater than zero and returns the `index`th digit from the right in `n`. (6)

For example, if `n` = 5246, then the digit at index 1 is 6, the digit at index 2 is 4, the digit at index 3 is 2, etc.

| Index | *6* | *5* | *4* | *3* | *2* | *1* |
|---|---|---|---|---|---|---|
| n | **0** | **0** | **5** | **2** | **4** | **6** |

Hint:

```
getDigit(5246, 1) = (5246 div 1) mod 10 = 6
getDigit(5246, 2) = (5246 div 10) mod 10 = 4
getDigit(5246, 3) = (5246 div 100) mod 10 = 2
getDigit(5246, 4) = (5426 div 1000) mod 10 = 5
getDigit(5246, 5) = (5426 div 10000) mod 10 = 0
```

etc.

2 Write a function\procedure that prompts the user for 10 integers and then returns the sum and average of those numbers to the main program. (6)

3. Given below is an implementation of a procedure, including its pre- and post-conditions. It contains a number of *logical* errors, i.e. the code will compile, but will not work as expected.

```
 1: procedure CalcMinMax(n:integer; min, max:integer);
 2: //PRE-CONDITIONS:
 3: // n>=0
 4: //POST-CONDITIONS:
 5: // asks for n numbers, returning the minimum and
 6: maximum
 7: // value entered
 8: var
 9:   count, number : integer;
10: begin
11:   count:=1;
12:   repeat
13:     write('number ', count, '='); readln(number);
14:     if count=1
15:     then
16:       begin
17:         min:=number;
18:         max:=number;
19:       end;
20:     if number<min then min:=number;
21:     if number>max then max:=number;
22:   until count=n;
    end;
```

3.1 Name the logical errors you find in the code. (Use the line numbers on the left to help show where the error occurs, or what causes an error) (4)

3.2 Give a corrected version of the code. You do not need to include the pre- and post-conditions in your answer. (4)

4. Write a program the will prompt the user for 10 real numbers and will then display the numbers in reverse order. (5)

5. Assume you have a text file named "Data.txt" that contains comma-delimited lines each containing a student's name and four test marks, e.g. (15)

```
Bob Smith,87,75,92,89
```

Write a program that will open the text file and display for each line the student's name and average, e.g.

```
Bob Smith has an average of 85.75
```

Note: It is not specified how many students the text file will contain.

# Appendix E

# B# Questionnaires

## E.1 B# ver. 1.0 Informal Usability Survey Questionnaire

| | | | | | |
|---|---|---|---|---|---|
| 1 | B# is easy to use. | | | | |
| | Strongly disagree | Disagree | Undecided | Agree | Strongly agree |
| 2 | Could you complete your practicals in B# without assistance? | | | | |
| | Can't remember | No | Hardly | Almost | Yes |
| 3 | Answer if you did not say yes above, what held you back? | | | | |
| 4 | What level of assistance did you need throughout your usage of B#? | | | | |
| | Very low | Low | Average | High | Very high |
| 5 | How often did you look at the code associated with your program? | | | | |
| | Seldom | Below average | Often | Above average | Always |
| 6 | What would you say your experience level with Delphi (the language) was before using B#? | | | | |
| | None | Very low | Low | High | Very high |
| 7 | What would you say your experience level with Delphi (the language) is after using B# and lectures? | | | | |
| | None | Very low | Low | High | Very high |
| 8 | Do you think B# and its flowchart method helped you in solving algorithmic problems? To what level: 1=(very low) to 5=(very high) | | | | |
| | Very low | Low | Average | High | Very high |
| 9 | Assuming B# was problem free and complete (more functionality). Do you think you will make use B# to solve future algorithmic problems? | | | | |
| | Yes | | | No | |
| 10 | Did you ever run into a problem in B# which you thought was not due to your doing? | | | | |
| | Yes | | | No | |
| 11 | If so, briefly describe the task (if possible) and B#'s reaction. | | | | |
| 12 | What would you like to see changed in B#? | | | | |

## E.2 B# ver. 1.0 Icon Evaluation Survey Form: First impressions

Link an icon (left) with a concept (right) by drawing a line.

| Icons | Programming Concept |
|---|---|
| ◇? | Assignment |
| ?↳ | Condition |
| ⌨ | For Loop |
| 🖥 | Return |
| ← | Repeat-Until Loop |
| R↺ | Input |
| P(x) | While Loop |
| F↺ | Output |
| W↺ | Case Statement |
| ↵ | Procedure call |

1. Tick the icons that were the easiest to identify.
2. Cross the icons that were the most difficult to identify.
3. Draw any suggestion of an icon next to the original one.

## E.3 B# Ver. 1.0 Icon Evaluation Survey: Recollection

| Icons | Programming Concept? |
|---|---|
| ◇? | ____________________ |
| ?↳ | ____________________ |
| ⌨ | ____________________ |
| 🖥 | ____________________ |
| ← | ____________________ |
| R↺ | ____________________ |
| P(x) | ____________________ |
| F↺ | ____________________ |
| W↺ | ____________________ |
| ↵ | ____________________ |

# Appendix F

# Survey Results of B# Questionnaires

## F.1 B# Ver. 1.0 Informal Usability Survey Results

Qualitative analysis of items 3, 11 and 12 in the questionnaire in Section E.1 assisted in identifying problems that required attention to better satisfy the novice programmer user and thereby increase the system's usability.

Quantitative analysis of items 1, 2 and 4 through to 10 were used to measure the success and shortcomings of the B# ver. 1.0.

The quantitative analysis of the responses to the questionnaire is graphically presented in Figure F.1. The responses to each question posed in the questionnaire was linked to the scale [0, 1, 2, 3, 4], with 0 corresponding to the leftmost rating and 4 to the rightmost rating for each item.

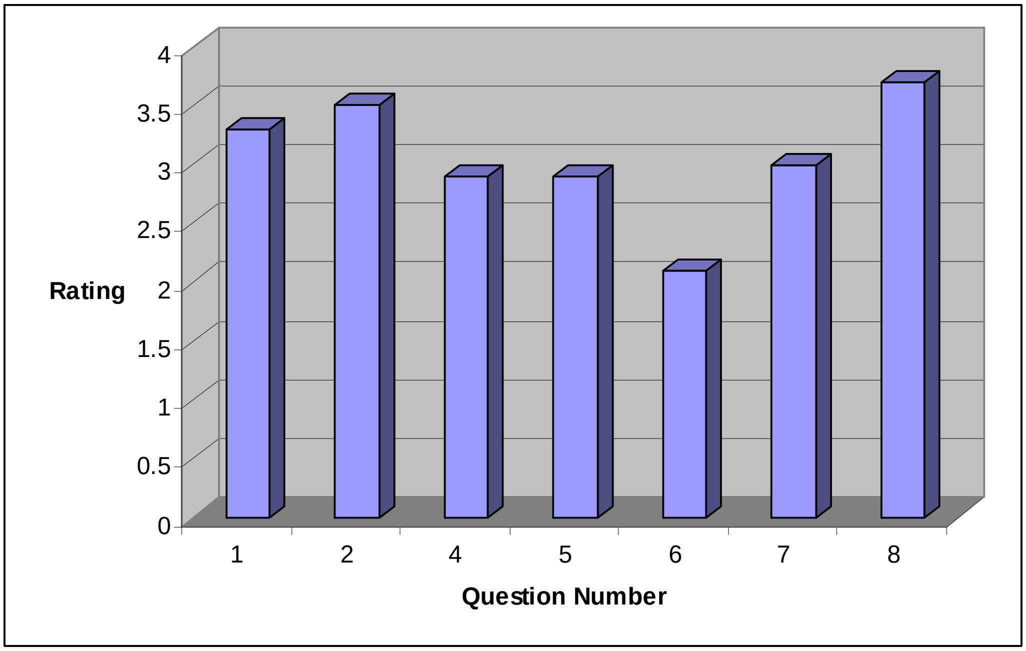


*Figure F.1: Quantitative analysis of informal usability survey of B# ver. 1.0 (Thomas 2002b)*

Qualitative analysis of the survey responses revealed that novice programmers required a great deal assistance while completing practical tasks in B# ver. 1.0.

## F.2 B# Icon Evaluation Results

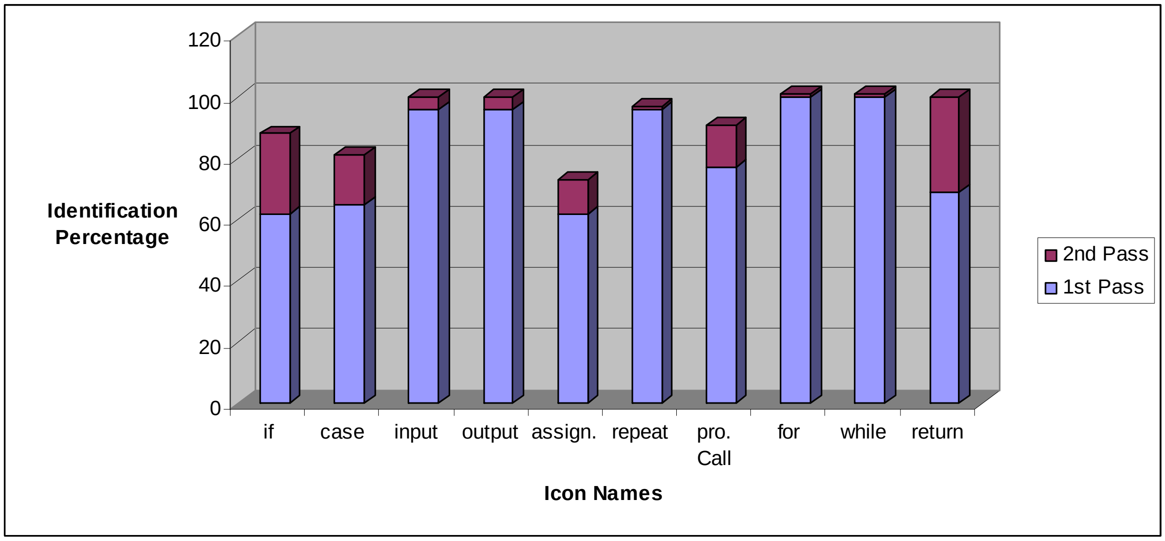


*Figure F.2: Quantitative analysis of icon evaluation results (Thomas 2002b)*

Figure F.2 shows the increase in correct association of icons with programming concepts by survey participants in red. The icons identified as being most problematic are the single and multiple branching conditional, and the assignment programming constructs.

# Appendix G

# Ethics Committee Application

Proposal for Experimental Study using First Year Introductory Programming Students as Test Subjects

**Study Proposal**

At the commencement of the first semester of 2003, divide first year introductory programming students (module codes WRA101[1] and WRA131[1]) into three (3) stratified samples according to success risk level in an introductory programming module, degree programme and computer literacy classification. Each sample will consist of at least 80 first year students.

For the period February – May 2003 for WRA101 participants, and February – October 2003 for WRA131 participants, expose each group of students to ***one*** of the following alternative practical delivery methods on a weekly basis as part of their required module commitments:

- Commercial programming environment, namely Delphi™ Enterprise (which is the current prescribed programming environment);
- Experimental simplified programming environment, namely D-Lite; and
- Hybrid model comprising of an experimental iconic programming environment, namely B#, and an experimental simplified programming environment, namely D-Lite.

By means of regular academic assessments during the period of the study, determine which of the three alternative delivery methods is the most suitable delivery method to teach an introductory programming module, and in so doing increase the pass rate in the module.

**Study Investigators**

Ms C B Cilliers (senior lecturer and PhD candidate)
Prof A P Calitz (PhD promoter)
Prof M B Watson (PhD promoter)
Dr J H Greyling (co-investigator)
All WRA101 and WRA131 lecturers, namely Mr D Vogts, Mr M C du Plessis (monitors)

***Motivation***

*Summary of main points have been detailed below for brief perusal*:

- Primary goal of introductory programming module is to teach problem-solving using a programming language, and not teach the programming language itself
- Unsatisfactory individual and group performance rates are apparent in introductory programming modules
- First year students find it difficult to memorise and understand syntax and semantic constraints of a programming language

- Commercial programming environments are designed for experienced programmers, tend to have complex interfaces and do not successfully support novice programmers
- Details in commercial programming environment tend to distract the novice programmer
- A large proportion of the Algorithmics 1.1 annual intake are service module students who could benefit from being shielded from the effect of complex commercial programming environments
- A single successful study in a similar research area has been reported on internationally. No empirical data exists for a South African study.
- Locally developed experimental programming environments addressing the issues highlighted above have been successfully developed in the Department over the past 2 years.
- Study forms part of an ongoing registered Departmental research project
- Academic staff of the Department supports the conducting of the study
- Participants will be closely monitored on a weekly basis
- No extra time or effort will be expected from participants in the study

***The primary purpose of this study is therefore to provide empirical evidence to support the hypotheses found in literature that novice programmers benefit from the use of simplified programming environments and/or iconic programming tools. Empirical evidence could also assist in the planning of the content of a problem-solving service module.***

Tertiary institutions are continuously addressing the issue of satisfactory group and individual performance rates in introductory programming modules, where the primary goal is to use a programming language to implement solutions to problems rather than to teach a programming language [Kus1994, Loc1986].

All first year programming students at UPE are, since 2001, required to be streamed into one (or none) of the introductory programming modules (WRA101 or WRA131) presented by the Department of Computer Science and Information Systems [Gre2000]. Streaming is based on results obtained in the University's prescribed and mandatory placement test. Repeating students are, since 2002, streamed according to past performance in their attempts at these modules[2]. The main reasons for these measures are the Department's limited technological resources and continuous attempts to increase the pass rate in the introductory programming modules while at all times maintaining the expected level of instruction.

Despite these innovative attempts to assist individual students and improve the overall throughput in the introductory programming modules, numerous students who possess a potential to be successful are in fact unsuccessful in the introductory programming module. The pass rate for WRA101 and WRA131 (Algorithmics 1.1) has for the last few years been recorded at values far beneath the expected rate. In 2002, the pass rate for WRA101 was 38% (including those who had either cancelled the module or been refused admission to write the examination in June 2002). Historical first years, namely those students registering for the first time in 2002, recorded a final pass rate of 58%, with repeating students recording a pass rate of 23%.

One reason for this could be that one of the more difficult aspects for any first year student when learning to program is the memorising and understanding of the syntactic and semantic constraints of a specific programming language [Stu1995]. Another reason could be that the interactive development environment of commercially available programming languages (for example Visual C++ and Borland® Delphi™ Enterprise) adds unnecessary complexity to practical implementation sessions in an introductory programming module since they have been designed for more experienced users [Zie1999]. During practical implementation sessions, introductory programming students now have to contend with the issues of the scenario of the problem to be solved, the syntax of the language being used as well as the environment in which the actual programming takes place.

It has been recommended that first time students should commence programming at a level where the technicalities of the programming language being used are not more important than the concepts being taught [Pro1996]. When introducing students to initial program design and development, an environment that is conducive to the requirements of teaching and learning how to program is required. This type of environment tends to differ from commercially available programming environments that have been designed for experienced users who are developing large programs [Zie1999].

Research has indicated that commercial programming environments, which are those environments that are traditionally used in the practical instruction of programming constructs to novice programmers, do not meet their goals as support tools for novice programmers and are not suited to the types of problems experienced by novice programmers [Dee1999].

Furthermore, of the first years registered for WRA101 in 2002, 52% were non-professional programming majors, that is, for these students, WRA101 is a necessary and required service module. Debates are continuously entered into regarding whether the ability to code solutions for problems in a given programming language is a necessary requirement for a service problem-solving module.

International research results have shown a significant improvement in the results obtained by students in an introductory programming module using BACCII$^{©}$, an iconic programming tool [Cal1994, Cal1997]. There is also a belief among some researchers that students should master a few broad concepts at a time, leaving the remaining tasks to be implemented by the components provided [Pro1996]. Further, some researchers have experienced that low-level implementation issues distract the students so that they do not understand the abstraction of the problem-solving process [Ree1995]. Some researchers also feel that it is important for first time students to be exposed to high quality code that solve real problems. In this way students are encouraged to imitate correct code [Pro1996].

For all of these reasons, an experimental iconic programming tool called B#, with a limited scope of programming functionality was developed by postgraduate students in the Department during 2001 and 2002 [Bro2001, Tho2002]. An experimental simplified programming environment, D-Lite, providing all of the functionality of the equivalent commercial programming environment was also developed in 2001. During 2002, field-testing to assess the usability of the two experimental programming environments was successfully conducted. A positive attitude on the part of students assessing the experimental software was also observed.

There is, however, insufficient methodological research and little empirical evidence, both internationally and especially within the Southern African context, to justify the use of simplified programming environments and/or iconic programming tools in the training of novice programmers, and as a problem-solving tool for non-professional programming majors.

The primary purpose of this study is therefore to provide empirical evidence to support the hypotheses found in literature that novice programmers benefit from the use of simplified programming environments and/or iconic programming tools. Empirical evidence could also assist in the planning of the content of a problem-solving service module.

This study proposal, which is an integral part of a Departmental registered research project and forms the core for a registered PhD, was discussed on 18 October 2002 with the majority of the members of the academic staff of the Department of Computer Science and Information Systems. The Department supports the conducting of this study for the period February – October 2003.

Due to the fact that each participant in the study will be monitored weekly by at least one investigator as a part of his/her regular module commitments, should any irregularity of any nature be observed that is judged to be detrimental to any individual participant or group of participants, the primary investigators will have no hesitation in terminating the study immediately. All participants in all groups will then revert to using the prescribed commercial programming environment, namely Delphi™ Enterprise.

No participant will be expected to invest any extra time or effort on this study over that time and effort which is considered as part of the normal workload for the credit bearing commitments for the introductory programming (Algorithmics 1.1) modules. The experimental programming environments will serve as alternatives for the prescribed commercial programming environment.

1. WRA101 and WRA131 are equivalent introductory programming modules. WRA101 is presented in the first semester of the academic year, and WRA131 is a slower paced module presented over the entire academic year.
2. Details on the regulations regarding the streaming of students are available in the Rules and Regulations handout for the Faculty of Science.

# APPLICATION TO UPE HUMAN ETHICS COMMITTEE

| | | | | |
|---|---|---|---|---|
| **Title of proposed project:** An Investigation into Alternative Delivery Methods in an Introductory Programming Module | | | | |
| **Details of the investigator(s)** | | | | |
| Name(s) | Ms C B Cilliers | Prof A P Calitz | Prof M B Watson | Dr J H Greyling |
| Qualification(s) | MSc | PhD | DPhil, NHED | PhD |
| Position(s) | Senior Lecturer | Professor | Professor | Senior Lecturer |
| Departmental addresses | Computer Science and Information Systems | | Psychology | Computer Science and Information Systems |
| Functions in the proposed research | Principal investigator | Promoter | Promoter | Co-investigator |
| Name of principal investigator | Ms C B Cilliers | | | |
| Experience of principal investigator in the field of research concerned | Ms Cilliers has 14+ years experience as an academic in the Department, 5 of which are as an introductory programming module lecturer. She has a clear understanding of the education goals of an introductory programming module, specifically with respect to the intended learning outcomes, and she has experience of what is required as prior learning for follow up modules. She has been actively involved in the development of the experimental programming environments and the assessment of these environments. She has undertaken an intensive literature survey to support the study. | | | |

**Place where research is to be undertaken:**
Computer laboratories in the Department of Computer Science and Information Systems, UPE during scheduled introductory programming module practical sessions

Objective of the research (***ie* the hypothesis which is to be tested)**
**A delivery method comprising of an iconic programming environment together with a simplified programming language editing environment is the most suitable to teach an introductory programming module, and serves to increase the pass rate in the module.**

**Scientific background** (If similar work has been done previously, state why it needs to be repeated. If it has not been done before, has the problem been worked out as fully as possible using animals or other alternative research methods?)

There is insufficient methodological research and little empirical evidence, both internationally and especially within the Southern African context, to justify the use of simplified programming environments and/or iconic programming tools in the training of novice programmers, and as a problem-solving tool for non-professional programming majors.

*For further details on the scientific background, please refer to the **Motivation** section in the attached study proposal.*

**Design of the study** (Give an outline of the proposed project, including procedures to be used, the measurements to be made and how the results will be analysed. State the likely duration of the project)
Participants will be selected during the first week of February 2003 according to their computer literacy proficiency, success risk level and degree programme into 3 stratified samples per introductory programming module, thus a total of 6 samples. The computer literacy proficiency will be assessed by means of a ***Background Questionnaire*** (see attached example). The success risk level per participant is the score obtained in the prescribed and mandatory placement test administered by the University on application as a student. The degree programme will be determined upon successful registration.
**For the period February – May 2003 for WRA101 participants, and February – October 2003 for WRA131 participants, expose each group of students to *one* of the following alternative practical delivery methods on a weekly basis as part of their required module commitments:**
**Commercial programming environment, namely Delphi™ Enterprise (which is the current prescribed programming environment);**

**Experimental simplified programming environment, namely D-Lite; and**
**Hybrid model comprising of an experimental iconic programming environment, namely B#, and an experimental simplified programming environment, namely D-Lite.**
**A number of academic assessments, which form part of the normal module commitment, will be administered. Participants will be assessed twice in a practical environment by means of the delivery method to which they have been exposed. Each participant will be assessed twice in a theoretical environment. The results of these assessments will be for credit bearing purposes. All assessments will be identical across the stratified groups.**
**All academic results will be statistically analysed per stratified sample to determine whether the hybrid model indeed increases the pass rate and increases an individual participant's performance. Statistical analysis will also be performed on the results of assessments for non-professional IT majors, and a proposal as to the preferred delivery method for a service introductory programming module will be drawn up.**
**The control group will consist of participants who are using the prescribed commercial programming environment as the delivery method.**
**On completion of the study, participants will be required to complete a feedback questionnaire, the contents of which still need to be finalized.**

**Type of subjects** (Give details of method of recruitment for each category [patients, controls, healthy volunteers]). Specify whether subjects are in a dependent relationship with the investigators, *eg* students or whether specially vulnerable *eg* children, mentally handicapped).
Participants will be selected during the first week of February 2003 according to their computer literacy proficiency, success risk level and degree programme into 3 stratified samples per introductory programming module, thus a total of 6 samples. These measurement tools have been detailed in the previous section (*Design of the study*).
Participants (students) are in a dependent relationship with the Department, but not directly with the investigators since the principal investigator is currently on sabbatical and will only return to academic duty in July 2003.

**Substances to be given** (Eg drugs, special diets, isotopes, vaccines, etc. State route, dose, frequency, precautions)
NONE

**Samples to be obtained (**Eg blood, urine, cerebrospinal fluid, biopsy specimens, etc. Give method and frequency of sampling, amount of each sample)
NONE

**Other procedures** (Give details eg of X-rays, endoscopy, anaesthesia)
NONE

**Potential risks and inconvenience to the subjects** (Estimated probability (if possible) and precautions to be taken to minimise risks and inconvenience)
A possibility exists where the participant might not feel comfortable with the experimental delivery method assigned to him/her. He/she might therefore feel at risk of failing the module. In this case, the participant may at any time revert to the prescribed commercial programming environment.
Due to the fact that each participant in the study will be monitored weekly (as a part of the normal module commitment) by at least one investigator, should any irregularity of any nature be observed that is judged to be detrimental to any individual participant or group of participants, the primary investigators will have no hesitation in terminating the study immediately. All participants in all groups will then revert to using the prescribed commercial programming environment.

Each experimental delivery method will encourage a participant to code programs in a fashion that would normally be expected of introductory programming students, and therefore, should a participant revert back to the prescribed commercial environment, no inconvenience should be experienced. The only inconvenience would be adapting to the programming environment interface. Allowance for adaptation time during practical sessions will be made for should the need arise.

At various times during the development of the experimental delivery methods in 2001/2, field-testing to assess the usability of the two experimental programming environments (and not of the performance of the users of the software) was successfully conducted. The investigators also observed a positive and enthusiastic attitude on the part of students assessing the experimental software. Because of this observation, the estimated probability of risk and inconvenience to the participants is low.

One possible risk is that participants in the control group, that is those using the prescribed commercial programming environment, might perceive the experimental programming environments as being “simpler” (which they are intended to be in order to test the hypothesis) and may wish to be included as participants using the experimental programming environments. Due to the rigid sampling procedure that will be applied, no participants will be permitted to change to an experimental programming environment from the commercial programming environment, nor will they be permitted to change between experimental programming environments once the sampling has been done. For this reason, all participants in the study will be required to complete an informed consent form so that they are aware of and acknowledge their rights.

Benefits of research to research subject and/or community
The empirical data resulting from this study will benefit the presentation of an introductory programming module at all levels, namely the performance of an individual student, UPE’s Department of Computer Science and Information Systems by increasing the group pass rates in these modules, as well as other tertiary institutions nationally and internationally. Educational benefits of the study on alternative delivery methods can be incorporated for use at secondary schools in South Africa offering programming subjects.
Two research papers reporting on the development of the experimental delivery methods have already resulted – one of the papers was presented at a conference in June 2002, and the second will be presented at an international conference in January 2003.
Research reports on this study will be submitted to acknowledged international journals as well as recognized international conferences.

**Manner in which the subject’s consent will be obtained** (if written, include a copy)
See attached informed consent forms, one for each stratified group of participants.

State whether the subject’s personal doctor is to be informed of recruitment of the subject before the study begins, and whether the subject’s consent to such information being passed on is a condition of participation

NO

Regulatory status under the relevant legislation of any drug or appliance to be used or tested

NONE

**Investigators’ interests** (Profit, personal or departmental, financial or otherwise, relating to the study)
Results of this study will form an integral portion of both a currently registered PhD and a Departmental registered research project. The output from the study can assist the Department in determining an appropriate model for a delivery method for introductory programming modules that are for information technology majors and non-majors (namely service modules).

RNcwadi/dg/Genforms
31 May 2001

**Background Questionnaire**

| | | |
|---|---|---|
| **Questionnaire No.:** | **Date:** | |
| Name and Surname: | Select Gender:<br>__ Female<br>__ Male | Indicate your Home Language:<br>Eng / Afr / isiXhosa / seSotho<br>Other: |
| Contact Telephone No.: | Student No.: | |

**PART 1: Previous Experience**
**Please circle the number which is most appropriate.**
**1=never, 2=rarely, 3=once to three times per month, 4=once a week, 5=daily.**

1. How long have you worked on a computer? Never Daily
**1 2 3 4 5** □

**PART 2: Self-assessment of ability**

**Please circle the numbers which most appropriately reflect your impressions.**
**1=poor, 2=not so good, 3=reasonable, 4=good, 5=excellent, NA=not applicable**

2.1 How would you rate your ability to use a computer in general? Poor Excellent
**1 2 3 4 5 NA** □

2.2 How would you rate your ability to use the typing and the cursor sections of the keyboard? Poor Excellent
**1 2 3 4 5 NA** □

2.3 How would you rate your ability to use the mouse? Poor Excellent
**1 2 3 4 5 NA** □

2.4 How would you rate your ability to use Windows objects such as buttons, checkboxes and radio buttons? Poor Excellent
**1 2 3 4 5 NA** □

**PART 3: Attitude**
**1=difficult, 2=not too difficult, 3=fairly easy, 4=easy, 5= very easy, NA=not applicable**

1.1 How do you find working on a computer in general? Difficult Very easy
**1 2 3 4 5 NA** □

1.2 How do you find it working with the keyboard? Difficult Very easy
**1 2 3 4 5 NA** □

3.3 How do you find it working with the mouse? Difficult Very easy
**1 2 3 4 5 NA** □

3.4 How do you find reading and understanding information on the computer screen? Difficult Very easy
**1 2 3 4 5 NA** □

**PART 4: Previous Experience**
**1=never, 2=about once a month, 3=several times a month, 4=once a week, 5=daily, NA=not applicable**

4.1 How often is there a computer available for you to use? Never Daily
**1 2 3 4 5 NA** □

4.2 How often do you play games on a computer? Never Daily
**1 2 3 4 5 NA** □

4.3 How often do you use a Word Processing package? Never Daily
**1 2 3 4 5 NA** □

4.4 How often do you use a Spreadsheet package? Never Daily

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| | **1** | **2** | **3** | **4** | **5** | **NA** | □ |
| 4.5 How often do you use the Internet? | Never | | | Daily | | | |
| | **1** | **2** | **3** | **4** | **5** | **NA** | □ |
| 4.6 How often do you use Email facilities? | Never | | | Daily | | | |
| | **1** | **2** | **3** | **4** | **5** | **NA** | □ |
| 4.7 How often do you write computer programs? | Never | | | Daily | | | |
| | **1** | **2** | **3** | **4** | **5** | **NA** | □ |

**PART 5: Contact with other Technology**
**1=never, 2=about once a month, 3=several times a month, 4=once a week, 5=daily, NA=not applicable**

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 5.1 How often to you program a Video Cassette | Never | | | Daily | | | |
| Recorder (VCR)? | **1** | **2** | **3** | **4** | **5** | **NA** | □ |
| 5.2 How often to you use an Automatic Banking | Never | | | Daily | | | |
| Teller Machine (ATM)? | **1** | **2** | **3** | **4** | **5** | **NA** | □ |
| 5.3 How often do you use a Tape Recorder? | Never | | | Daily | | | |
| | **1** | **2** | **3** | **4** | **5** | **NA** | □ |
| 5.4 How often do you use a CD Player? | Never | | | Daily | | | |
| | **1** | **2** | **3** | **4** | **5** | **NA** | □ |
| 4.2 How often do you play TV games (not Computer games)? | Never | | | Daily | | | |
| | **1** | **2** | **3** | **4** | **5** | **NA** | □ |
| 5.6 How often do you use a cellular (cell) phone? | Never | | | Daily | | | |
| | **1** | **2** | **3** | **4** | **5** | **NA** | □ |
| 5.7 How often do you select programs on Digital | Never | | | Daily | | | |
| Satellite Television (DSTV)? | **1** | **2** | **3** | **4** | **5** | **NA** | □ |

Thank you for your time and effort.

# Informed Consent Form

You have been selected as a research participant for the evaluation of the

B#

experimental delivery method in the Algorithmics 1.1 module. This evaluation is being conducted by Ms Charmain Cilliers (csacbc@upe.ac.za, ☎ 5042235), Prof André Calitz (csaapc@upe.ac.za) and Dr Jean Greyling (csajhg@upe.ac.za). We will be happy to answer any questions you may have about the evaluation.

During the entire semester, you will be required to perform various practical tasks with this experimental software on a weekly basis. These tasks will form part of the required commitment for the Algorithmics 1.1 module. You will not be required to invest any additional time or effort in the experimental delivery method other than that time and effort that would normally be required from you as an acknowledged portion of the Algorithmics 1.1 module commitment.

You will be assessed on your performance in the experimental delivery method and these assessments will count towards both your class and final mark for Algorithmics 1.1. All assessments will be identical to those administered to students using the prescribed commercial programming environment, namely Delphi™ Enterprise. The results of your assessments will be used for statistical analysis to determine whether a specific delivery method benefits individual and/or group performance in Algorithmics 1.1.

Your name will not be associated with any data that are collected during this research study. There are no known risks associated with this evaluation. At the end of the research study period, you will be requested to complete a feedback questionnaire, containing questions relevant to this evaluation.

***Your rights as a participant are as follows*:**

You have the right to withdraw from using the experimental delivery method B# and D-Lite at any time for any reason. In this case, you will then for the remainder of the duration of the registered module have to complete all practical tasks for Algorithmics 1.1 in the prescribed commercial programming environment, namely Delphi™ Enterprise. You will also then be assessed in all further practical assessments in the commercial programming environment.

You will not be permitted to complete your practical tasks for Algorithmics 1.1 credit bearing purposes in any other programming environment other than B# and D-Lite combination, or Delphi™ Enterprise (see point above).

We greatly appreciate your time and effort for participating in this evaluation. Your signature below indicates that you have read this consent form in its entirety and that you voluntarily agree to participate.

Name: ______________________________________________________

Signature: ___________________________________________________

Telephone no.: ________________________ Date: _____________________

## Informed Consent Form

You have been selected as a research participant for the evaluation of the

Commercial programming environment Delphi™ Enterprise

as the delivery method in the Algorithmics 1.1 module. This evaluation is being conducted by Ms Charmain Cilliers (csacbc@upe.ac.za, ☎ 5042235), Prof André Calitz (csaapc@upe.ac.za) and Dr Jean Greyling (csajhg@upe.ac.za). We will be happy to answer any questions you may have about the evaluation.

During the entire semester, you will be required to perform various practical tasks with this software on a weekly basis. These tasks will form part of the required commitment for the Algorithmics 1.1 module. You will not be required to invest any additional time or effort in the delivery method other than that time and effort that would normally be required from you as an acknowledged portion of the Algorithmics 1.1 module commitment.

You will be assessed on your performance in the delivery method and these assessments will count towards both your class and final mark for Algorithmics 1.1. All assessments will be identical to those administered to students using the experimental programming environments. The results of your assessments will be used for statistical analysis to determine whether a specific delivery method benefits individual and/or group performance in Algorithmics 1.1.

Your name will not be associated with any data that are collected during this research study. There are no known risks associated with this evaluation. At the end of the research study period, you will be requested to complete a feedback questionnaire, containing questions relevant to this evaluation.

***Your rights as a participant are as follows*:**

You will not be permitted to complete your practical tasks for Algorithmics 1.1 credit bearing purposes in any other programming environment other than Delphi™ Enterprise, which is the prescribed programming environment.

We greatly appreciate your time and effort for participating in this evaluation. Your signature below indicates that you have read this consent form in its entirety and that you voluntarily agree to participate.

Name: ____________________________________________________

Signature: _________________________________________________

Telephone no.: ________________________ Date: _____________________

# APPLICATION TO UPE HUMAN ETHICS COMMITTEE

**Title of proposed project:** An Investigation into Alternative Delivery Methods in an Introductory Programming Module

| | | | | |
|---|---|---|---|---|
| **Details of the investigator(s)** | | | | |
| Name(s) | Ms C B Cilliers | Prof A P Calitz | Prof M B Watson | Dr J H Greyling |
| Qualification(s) | MSc | PhD | DPhil, NHED | PhD |
| Position(s) | Senior Lecturer | Professor | Professor | Senior Lecturer |
| Departmental addresses | Computer Science and Information Systems | | Psychology | Computer Science and Information Systems |
| Functions in the proposed research | Principal investigator | Promoter | Promoter | Co-investigator |
| Name of principal investigator | Ms C B Cilliers | | | |
| Experience of principal investigator in the field of research concerned | Ms Cilliers has 14+ years experience as an academic in the Department, 5 of which are as an introductory programming module lecturer. She has a clear understanding of the education goals of an introductory programming module, specifically with respect to the intended learning outcomes, and she has experience of what is required as prior learning for follow up modules. She has been actively involved in the development of the experimental programming environments and the assessment of these environments. She has undertaken an intensive literature survey to support the study. | | | |

**Place where research is to be undertaken:**

Computer laboratories in the Department of Computer Science and Information Systems, UPE during scheduled introductory programming module practical sessions

**Objective of the research** (*ie* the hypothesis which is to be tested)

A delivery method comprising of an iconic programming environment together with a simplified programming language editing environment is the most suitable to teach an introductory programming module, and serves to increase the pass rate in the module.

**Scientific background** (If similar work has been done previously, state why it needs to be repeated. If it has not been done before, has the problem been worked out as fully as possible using animals or other alternative research methods?)

There is insufficient methodological research and little empirical evidence, both internationally and especially within the Southern African context, to justify the use of simplified programming environments and/or iconic programming tools in the training of novice programmers, and as a problem-solving tool for non-professional programming majors.

*For further details on the scientific background, please refer to the* ***Motivation*** *section in the attached study proposal.*

**Design of the study** (Give an outline of the proposed project, including procedures to be used, the measurements to be made and how the results will be analysed. State the likely duration of the project)

Participants will be selected during the first week of February 2003 according to their computer literacy proficiency, success risk level and degree programme into 3 stratified samples per introductory programming module, thus a total of 6 samples. The computer literacy proficiency will be assessed by means of a ***Background Questionnaire*** (see attached example). The success risk level per participant is the score obtained in the prescribed and mandatory placement test administered by the University on application as a student. The degree programme will be determined upon successful registration.

For the period February – May 2003 for WRA101 participants, and February – October 2003 for WRA131 participants, expose each group of students to *one* of the following alternative practical delivery methods on a weekly basis as part of their required module commitments:

- Commercial programming environment, namely Delphi™ Enterprise (which is the current prescribed programming environment);
- Experimental simplified programming environment, namely D-Lite; and
- Hybrid model comprising of an experimental iconic programming environment, namely B#, and an experimental simplified programming environment, namely D-Lite.

A number of academic assessments, which form part of the normal module commitment, will be administered. Participants will be assessed twice in a practical environment by means of the delivery method to which they have been exposed. Each participant will be assessed twice in a theoretical environment. The results of these assessments will be for credit bearing purposes. All assessments will be identical across the stratified groups.

All academic results will be statistically analysed per stratified sample to determine whether the hybrid model indeed increases the pass rate and increases an individual participant's performance. Statistical analysis will also be performed on the results of assessments for non-professional IT majors, and a proposal as to the preferred delivery method for a service introductory programming module will be drawn up.

The control group will consist of participants who are using the prescribed commercial programming environment as the delivery method.

On completion of the study, participants will be required to complete a feedback questionnaire, the contents of which still need to be finalized.

**Type of subjects** (Give details of method of recruitment for each category [patients, controls, healthy volunteers]). Specify whether subjects are in a dependent relationship with the investigators, *eg* students or whether specially vulnerable *eg* children, mentally handicapped).

Participants will be selected during the first week of February 2003 according to their computer literacy proficiency, success risk level and degree programme into 3 stratified samples per introductory programming module, thus a total of 6 samples. These measurement tools have been detailed in the previous section (*Design of the study*).

Participants (students) are in a dependent relationship with the Department, but not directly with the investigators since the principal investigator is currently on sabbatical and will only return to academic duty in July 2003.

**Substances to be given** (Eg drugs, special diets, isotopes, vaccines, etc. State route, dose, frequency, precautions)

NONE

**Samples to be obtained (**Eg blood, urine, cerebrospinal fluid, biopsy specimens, etc. Give method and frequency of sampling, amount of each sample)

NONE

**Other procedures** (Give details eg of X-rays, endoscopy, anaesthesia)

NONE

**Potential risks and inconvenience to the subjects** (Estimated probability (if possible) and precautions to be taken to minimise risks and inconvenience)

A possibility exists where the participant might not feel comfortable with the experimental delivery method assigned to him/her. He/she might therefore feel at risk of failing the module. In this case, the participant may at any time revert to the prescribed commercial programming environment.

Due to the fact that each participant in the study will be monitored weekly (as a part of the normal module commitment) by at least one investigator, should any irregularity of any nature be observed that is judged to be detrimental to any individual participant or group of participants, the primary investigators will have no hesitation in terminating the study immediately. All participants in all groups will then revert to using the prescribed commercial programming environment.

Each experimental delivery method will encourage a participant to code programs in a fashion that would normally be expected of introductory programming students, and therefore, should a participant revert back to the prescribed commercial environment, no inconvenience should be experienced. The only inconvenience would be adapting to the programming environment interface. Allowance for adaptation time during practical sessions will be made for should the need arise.

At various times during the development of the experimental delivery methods in 2001/2, field-testing to assess the usability of the two experimental programming environments (and not of the performance of the users of the software) was successfully conducted. The investigators also observed a positive and enthusiastic attitude on the part of students assessing the experimental software. Because of this observation, the estimated probability of risk and inconvenience to the participants is low.

One possible risk is that participants in the control group, that is those using the prescribed commercial programming environment, might perceive the experimental programming environments as being “simpler” (which they are intended to be in order to test the hypothesis) and may wish to be included as participants using the experimental programming environments. Due to the rigid sampling procedure that will be applied, no participants will be permitted to change to an experimental programming environment from the commercial programming environment, nor will they be permitted to change between experimental programming environments once the sampling has been done. For this reason, all participants in the study will be required to complete an informed consent form so that they are aware of and acknowledge their rights.

**Benefits of research to research subject and/or community**

The empirical data resulting from this study will benefit the presentation of an introductory programming module at all levels, namely the performance of an individual student, UPE’s Department of Computer Science and Information Systems by increasing the group pass rates in these modules, as well as other tertiary institutions nationally and internationally. Educational benefits of the study on alternative delivery methods can be incorporated for use at secondary schools in South Africa offering programming subjects.

Two research papers reporting on the development of the experimental delivery methods have already resulted – one of the papers was presented at a conference in June 2002, and the second will be presented at an international conference in January 2003.

Research reports on this study will be submitted to acknowledged international journals as well as recognized international conferences.

**Manner in which the subject’s consent will be obtained** (if written, include a copy)

See attached informed consent forms, one for each stratified group of participants.

| |
|---|
| **State whether the subject's personal doctor is to be informed of recruitment of the subject before the study begins, and whether the subject's consent to such information being passed on is a condition of participation**<br>NO |
| **Regulatory status under the relevant legislation of any drug or appliance to be used or tested**<br>NONE |
| **Investigators' interests** (Profit, personal or departmental, financial or otherwise, relating to the study)<br>Results of this study will form an integral portion of both a currently registered PhD and a Departmental registered research project. The output from the study can assist the Department in determining an appropriate model for a delivery method for introductory programming modules that are for information technology majors and non-majors (namely service modules). |

RNcwadi/dg/Genforms
31 May 2001

**Background Questionnaire**

| | | |
|---|---|---|
| **Questionnaire No.:** | **Date:** | |
| Name and Surname: | Select Gender:<br>__ Female<br>__ Male | Indicate your Home Language:<br>Eng / Afr / isiXhosa / seSotho<br>Other: |
| Contact Telephone No.: | Student No.: | |

**PART 1: Previous Experience**
**Please circle the number which is most appropriate.**
**1=never, 2=rarely, 3=once to three times per month, 4=once a week, 5=daily.**

1. How long have you worked on a computer? Never Daily
**1 2 3 4 5** □

**PART 2: Self-assessment of ability**
**Please circle the numbers which most appropriately reflect your impressions.**
**1=poor, 2=not so good, 3=reasonable, 4=good, 5=excellent, NA=not applicable**

2.1 How would you rate your ability to use a computer in general? Poor Excellent
**1 2 3 4 5 NA** □

2.2 How would you rate your ability to use the typing and the cursor sections of the keyboard? Poor Excellent
**1 2 3 4 5 NA** □

2.3 How would you rate your ability to use the mouse? Poor Excellent
**1 2 3 4 5 NA** □

2.4 How would you rate your ability to use Windows objects such as buttons, checkboxes and radio buttons? Poor Excellent
**1 2 3 4 5 NA** □

**PART 3: Attitude**
**1=difficult, 2=not too difficult, 3=fairly easy, 4=easy, 5= very easy, NA=not applicable**

1.1 How do you find working on a computer in general? Difficult Very easy
**1 2 3 4 5 NA** □

1.2 How do you find it working with the keyboard? Difficult Very easy
**1 2 3 4 5 NA** □

3.3 How do you find it working with the mouse? Difficult Very easy
**1 2 3 4 5 NA** □

3.4 How do you find reading and understanding information on the computer screen? Difficult Very easy
**1 2 3 4 5 NA** □

**PART 4: Previous Experience**
**1=never, 2=about once a month, 3=several times a month, 4=once a week, 5=daily, NA=not applicable**

4.1 How often is there a computer available for you to use? Never Daily
**1 2 3 4 5 NA** □

4.2 How often do you play games on a computer? Never Daily
**1 2 3 4 5 NA** □

4.3 How often do you use a Word Processing package? Never Daily
**1 2 3 4 5 NA** □

4.4 How often do you use a Spreadsheet package? Never Daily
**1 2 3 4 5 NA** □

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 4.5 How often do you use the Internet? | Never | | | | Daily | | |
| | **1** | **2** | **3** | **4** | **5** | **NA** | □ |
| 4.6 How often do you use Email facilities? | Never | | | | Daily | | |
| | **1** | **2** | **3** | **4** | **5** | **NA** | □ |
| 4.7 How often do you write computer programs? | Never | | | | Daily | | |
| | **1** | **2** | **3** | **4** | **5** | **NA** | □ |

**PART 5: Contact with other Technology**
**1=never, 2=about once a month, 3=several times a month, 4=once a week, 5=daily, NA=not applicable**

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 5.1 How often to you program a Video Cassette | Never | | | | Daily | | |
| Recorder (VCR)? | **1** | **2** | **3** | **4** | **5** | **NA** | □ |
| 5.2 How often to you use an Automatic Banking | Never | | | | Daily | | |
| Teller Machine (ATM)? | **1** | **2** | **3** | **4** | **5** | **NA** | □ |
| 5.3 How often do you use a Tape Recorder? | Never | | | | Daily | | |
| | **1** | **2** | **3** | **4** | **5** | **NA** | □ |
| 5.4 How often do you use a CD Player? | Never | | | | Daily | | |
| | **1** | **2** | **3** | **4** | **5** | **NA** | □ |
| 4.2 How often do you play TV games (not Computer games)? | Never | | | | Daily | | |
| | **1** | **2** | **3** | **4** | **5** | **NA** | □ |
| 5.6 How often do you use a cellular (cell) phone? | Never | | | | Daily | | |
| | **1** | **2** | **3** | **4** | **5** | **NA** | □ |
| 5.7 How often do you select programs on Digital | Never | | | | Daily | | |
| Satellite Television (DSTV)? | **1** | **2** | **3** | **4** | **5** | **NA** | □ |

Thank you for your time and effort.

# Informed Consent Form

You have been selected as a research participant for the evaluation of the

**B#**

experimental delivery method in the Algorithmics 1.1 module. This evaluation is being conducted by Ms Charmain Cilliers (csacbc@upe.ac.za, ☎ 5042235), Prof André Calitz (csaapc@upe.ac.za) and Dr Jean Greyling (csajhg@upe.ac.za). We will be happy to answer any questions you may have about the evaluation.

During the entire semester, you will be required to perform various practical tasks with this experimental software on a weekly basis. These tasks will form part of the required commitment for the Algorithmics 1.1 module. You will not be required to invest any additional time or effort in the experimental delivery method other than that time and effort that would normally be required from you as an acknowledged portion of the Algorithmics 1.1 module commitment.

You will be assessed on your performance in the experimental delivery method and these assessments will count towards both your class and final mark for Algorithmics 1.1. All assessments will be identical to those administered to students using the prescribed commercial programming environment, namely Delphi™ Enterprise. The results of your assessments will be used for statistical analysis to determine whether a specific delivery method benefits individual and/or group performance in Algorithmics 1.1.

Your name will not be associated with any data that are collected during this research study. There are no known risks associated with this evaluation. At the end of the research study period, you will be requested to complete a feedback questionnaire, containing questions relevant to this evaluation.

***Your rights as a participant are as follows*:**

You have the right to withdraw from using the experimental delivery method B# and D-Lite at any time for any reason. In this case, you will then for the remainder of the duration of the registered module have to complete all practical tasks for Algorithmics 1.1 in the prescribed commercial programming environment, namely Delphi™ Enterprise. You will also then be assessed in all further practical assessments in the commercial programming environment.

You will not be permitted to complete your practical tasks for Algorithmics 1.1 credit bearing purposes in any other programming environment other than B# and D-Lite combination, or Delphi™ Enterprise (see point above).

We greatly appreciate your time and effort for participating in this evaluation. Your signature below indicates that you have read this consent form in its entirety and that you voluntarily agree to participate.

Name: ______________________________________________

Signature: ___________________________________________

Telephone no.: ____________________ Date: _________________

# Informed Consent Form

You have been selected as a research participant for the evaluation of the

**Commercial programming environment Delphi™ Enterprise**

as the delivery method in the Algorithmics 1.1 module. This evaluation is being conducted by Ms Charmain Cilliers (csacbc@upe.ac.za, ☎ 5042235), Prof André Calitz (csaapc@upe.ac.za) and Dr Jean Greyling (csajhg@upe.ac.za). We will be happy to answer any questions you may have about the evaluation.

During the entire semester, you will be required to perform various practical tasks with this software on a weekly basis. These tasks will form part of the required commitment for the Algorithmics 1.1 module. You will not be required to invest any additional time or effort in the delivery method other than that time and effort that would normally be required from you as an acknowledged portion of the Algorithmics 1.1 module commitment.

You will be assessed on your performance in the delivery method and these assessments will count towards both your class and final mark for Algorithmics 1.1. All assessments will be identical to those administered to students using the experimental programming environments. The results of your assessments will be used for statistical analysis to determine whether a specific delivery method benefits individual and/or group performance in Algorithmics 1.1.

Your name will not be associated with any data that are collected during this research study. There are no known risks associated with this evaluation. At the end of the research study period, you will be requested to complete a feedback questionnaire, containing questions relevant to this evaluation.

***Your rights as a participant are as follows*:**

You will not be permitted to complete your practical tasks for Algorithmics 1.1 credit bearing purposes in any other programming environment other than Delphi™ Enterprise, which is the prescribed programming environment.

We greatly appreciate your time and effort for participating in this evaluation. Your signature below indicates that you have read this consent form in its entirety and that you voluntarily agree to participate.

Name: ______________________________________________

Signature: ___________________________________________

Telephone no.: ____________________ Date: _________________

# Appendix H

# Hypotheses

## H.1 Investigation Hypothesis

$H_0$: *Academic performance in an introductory programming course is independent of programming notation and development environment.*

$H_1$: *Academic performance in an introductory programming course is dependent on programming notation and development environment.*

## H.2 Refinement of $H_0$

$H_{0.1}$: *The average mark achieved in an introductory programming course is independent of programming notation and development environment.*

$H_{1.1}$: *The average mark achieved in an introductory programming course is dependent on programming notation and development environment.*

and

$H_{0.2}$: *The observed throughput in an introductory programming course is independent of programming notation and development environment.*

$H_{1.2}$: *The observed throughput in an introductory programming course is dependent on programming notation and development environment.*

## H.3 Refinement of $H_{0.1}$

$H_{0.1.1}$: *The average mark achieved in pen-and-paper assessments of program solution comprehension and composition is independent of programming notation and development environment*.

$H_{1.1.1}$: *The average mark achieved in pen-and-paper assessments of program solution comprehension and composition is dependent on programming notation and development environment.*

and

$H_{0.1.2}$: *The average mark achieved in practical assessments of program solution comprehension and composition is independent of programming notation and development environment*.

$H_{1.1.2}$: *The average mark achieved in practical assessments of program solution comprehension and composition is dependent on programming notation and development environment.*

and

$H_{0.1.3}$: *The average class mark achieved is independent of programming notation and development environment.*

$H_{1.1.3}$: *The average class mark achieved is dependent on programming notation and development environment.*

and

$H_{0.1.4}$: *The average final mark achieved is independent of programming notation and development environment.*

$H_{1.1.4}$: *The average final mark achieved is dependent on programming notation and development environment.*

## H.4 Refinement of $H_{0.1.2}$

$H_{0.1.2.1}$: *The average mark achieved for the same problem is independent of programming notation and development environment.*

$H_{1.1.2.1}$: *The average mark achieved for the same problem is dependent on programming notation and development environment.*

and

$H_{0.1.2.2}$: *The average mark achieved for the same problem solved in Delphi™ Enterprise is independent of programming notation and development environment.*

$H_{1.1.2.2}$: *The average mark achieved for the same problem solved in Delphi™ Enterprise is dependent on programming notation and development environment.*

and

$H_{0.1.2.3}$: *The average mark achieved for the solving of syntactical errors using Delphi™ Enterprise is independent of programming notation and development environment.*

$H_{1.1.2.3}$: *The average mark achieved for the solving of syntactical errors using Delphi™ Enterprise is dependent on programming notation and development environment.*

and

$H_{0.1.2.4}$: *The average mark achieved for the same problem using individual choice of development environment is independent of programming notation and development environment.*

$H_{1.1.2.4}$: *The average mark achieved for the same problem using individual choice of development environment is dependent on programming notation and development environment.*

## H.5 Refinement of $H_{0.2}$

$H_{0.2.1}$: *The observed throughput in pen-and-paper assessments of program solution comprehension and composition is independent of programming notation and development environment*.

$H_{1.2.1}$: *The observed throughput achieved in pen-and-paper assessments of program solution comprehension and composition is dependent on programming notation and development environment.*

and

$H_{0.2.2}$: *The observed throughput in practical assessments of program solution comprehension and composition is independent of programming notation and development environment.*

$H_{1.2.2}$: *The observed throughput in practical assessments of program solution comprehension and composition is dependent on programming notation and development environment.*

and

$H_{0.2.3}$: *The observed throughput for the class mark is independent of programming notation and development environment.*

$H_{1.2.3}$: *The observed throughput for the class mark is dependent on programming notation and development environment.*

and

$H_{0.2.4}$: *The observed throughput for the final mark is independent of programming notation and development environment.*

$H_{1.2.4}$: *The observed throughput for the final mark is dependent on programming notation and development environment.*

## H.6 Refinement of $H_{0.2.2}$

$H_{0.2.2.1}$: *The observed throughput for the same problem is independent of programming notation and development environment.*

$H_{1.2.2.1}$: *The observed throughput for the same problem is dependent on programming notation and development environment.*

and

$H_{0.2.2.2}$: *The observed throughput for the same problem solved in Delphi™ Enterprise is independent of programming notation and development environment.*

$H_{12.2.2}$: *The observed throughput for the same problem solved in Delphi™ Enterprise is dependent on programming notation and development environment.*

and

$H_{0.2.2.3}$: *The observed throughput for the solving of syntactical errors using Delphi™ Enterprise is independent of programming notation and development environment.*

$H_{1.2.2.3}$: *The observed throughput for the solving of syntactical errors using Delphi™ Enterprise is dependent on programming notation and development environment.*

and

$H_{0.2.2.4}$: *The observed throughput for the same problem using individual choice of development environment is independent of programming notation and development environment.*

$H_{1.2.2.4}$: *The observed throughput for the same problem using individual choice of development environment is dependent on programming notation and development environment.*

# Appendix I

# Demographic Questionnaire Analysis

## I.1 Background Questionnaire

| | | |
|---|---|---|
| **Questionnaire No.:** | **Date:** | |
| Name and Surname: | Select Gender:<br>__ Female<br>__ Male | Indicate your Home Language:<br>Eng / Afr / isiXhosa / seSotho<br>Other: |
| Contact Telephone No.: | Student No.: | |

**PART 1: Previous Experience**
**Please circle the number which is most appropriate.**
**1=never, 2=rarely, 3=once to three times per month, 4=once a week, 5=daily.**

1. How long have you worked on a computer? Never Daily
**1 2 3 4 5** □

**PART 2: Self-assessment of ability**

**Please circle the numbers which most appropriately reflect your impressions.**
**1=poor, 2=not so good, 3=reasonable, 4=good, 5=excellent, NA=not applicable**

2.1 How would you rate your ability to use a computer in general? Poor Excellent
**1 2 3 4 5 NA** □

2.2 How would you rate your ability to use the typing and the cursor sections of the keyboard? Poor Excellent
**1 2 3 4 5 NA** □

2.3 How would you rate your ability to use the mouse? Poor Excellent
**1 2 3 4 5 NA** □

2.4 How would you rate your ability to use Windows objects such as buttons, checkboxes and radio buttons? Poor Excellent
**1 2 3 4 5 NA** □

**PART 3: Attitude**
**1=difficult, 2=not too difficult, 3=fairly easy, 4=easy, 5= very easy, NA=not applicable**

1.1 How do you find working on a computer in general? Difficult Very easy
**1 2 3 4 5 NA** □

1.2 How do you find it working with the keyboard? Difficult Very easy
**1 2 3 4 5 NA** □

3.3 How do you find it working with the mouse? Difficult Very easy
**1 2 3 4 5 NA** □

3.4 How do you find reading and understanding information on the computer screen? Difficult Very easy
**1 2 3 4 5 NA** □

**PART 4: Previous Experience**
**1=never, 2=about once a month, 3=several times a month, 4=once a week, 5=daily, NA=not applicable**

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 4.1 How often is there a computer available for you to use? | Never | | | | Daily | | |
| | **1** | **2** | **3** | **4** | **5** | **NA** | □ |
| 4.2 How often do you play games on a computer? | Never | | | | Daily | | |
| | **1** | **2** | **3** | **4** | **5** | **NA** | □ |
| 4.3 How often do you use a Word Processing | Never | | | | Daily | | |
| package? | **1** | **2** | **3** | **4** | **5** | **NA** | □ |
| 4.4 How often do you use a Spreadsheet package? | Never | | | | Daily | | |
| | **1** | **2** | **3** | **4** | **5** | **NA** | □ |
| 4.5 How often do you use the Internet? | Never | | | | Daily | | |
| | **1** | **2** | **3** | **4** | **5** | **NA** | □ |
| 4.6 How often do you use Email facilities? | Never | | | | Daily | | |
| | **1** | **2** | **3** | **4** | **5** | **NA** | □ |
| 4.7 How often do you write computer programs? | Never | | | | Daily | | |
| | **1** | **2** | **3** | **4** | **5** | **NA** | □ |

**PART 5: Contact with other Technology**
**1=never, 2=about once a month, 3=several times a month, 4=once a week, 5=daily, NA=not applicable**

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 5.1 How often to you program a Video Cassette | Never | | | | Daily | | |
| Recorder (VCR)? | **1** | **2** | **3** | **4** | **5** | **NA** | □ |
| 5.2 How often to you use an Automatic Banking | Never | | | | Daily | | |
| Teller Machine (ATM)? | **1** | **2** | **3** | **4** | **5** | **NA** | □ |
| 5.3 How often do you use a Tape Recorder? | Never | | | | Daily | | |
| | **1** | **2** | **3** | **4** | **5** | **NA** | □ |
| 5.4 How often do you use a CD Player? | Never | | | | Daily | | |
| | **1** | **2** | **3** | **4** | **5** | **NA** | □ |
| 4.2 How often do you play TV games (not Computer games)? | Never | | | | Daily | | |
| | **1** | **2** | **3** | **4** | **5** | **NA** | □ |
| 5.6 How often do you use a cellular (cell) phone? | Never | | | | Daily | | |
| | **1** | **2** | **3** | **4** | **5** | **NA** | □ |
| 5.7 How often do you select programs on Digital | Never | | | | Daily | | |
| Satellite Television (DSTV)? | **1** | **2** | **3** | **4** | **5** | **NA** | □ |

## I.2 Survey Item Coding

### *I.2.1 Part 1 : Previous Experience*

| Survey Item | Positive response values | Negative response values |
|---|---|---|
| 1: How long have you worked on a computer? | 4, 5 | 1, 2, 3 |

### *I.2.2 Part 2 : Self Assessment of Ability*

| Survey Item | Positive response values | Negative response values |
|---|---|---|
| 2.1: How would you rate your ability to use a computer in general? | 3, 4, 5 | 1, 2 |
| 2.2: How would you rate your ability to use the typing and the cursor sections of the keyboard? | 3, 4, 5 | 1, 2 |
| 2.3: How would you rate your ability to use the mouse? | 3, 4, 5 | 1, 2 |
| 2.4: How would you rate your ability to use Windows objects such as buttons, checkboxes and radio buttons? | 3, 4, 5 | 1, 2 |

### *I.2.3 Part 3 : Attitude*

| Survey Item | Positive response values | Negative response values |
|---|---|---|
| 3.1: How do you find working on a computer in general? | 2, 3, 4, 5 | 1 |
| 3.2: How do you find it working with the keyboard? | 2, 3, 4, 5 | 1 |
| 3.3: How do you find it working with the mouse? | 2, 3, 4, 5 | 1 |
| 3.4: How do you find reading and understanding information on the computer screen? | 2, 3, 4, 5 | 1 |

### *I.2.4 Part 4 : Previous Experience*

| Survey Item | Positive response values | Negative response values |
|---|---|---|
| 4.1: How often is there a computer available for you to use? | 4, 5 | 1, 2, 3 |
| 4.2: How often do you play games on a computer? | 4, 5 | 1, 2, 3 |
| 4.3: How often do you use a Word Processing package? | 4, 5 | 1, 2, 3 |
| 4.4: How often do you use a Spreadsheet package? | 4, 5 | 1, 2, 3 |
| 4.5: How often do you use the Internet? | 4, 5 | 1, 2, 3 |
| 4.6: How often do you use Email facilities? | 4, 5 | 1, 2, 3 |
| 4.7: How often do you write computer programs? | 4, 5 | 1, 2, 3 |

### *I.2.5 Part 5 : Contact with Other Technology*

| Survey Item | Positive response values | Negative response values |
|---|---|---|
| 5.1: How often to you program a Video Cassette Recorder (VCR)? | 4, 5 | 1, 2, 3 |
| 5.2: How often to you use an Automatic Banking Teller Machine (ATM)? | 4, 5 | 1, 2, 3 |
| 5.3: How often do you use a Tape Recorder? | 4, 5 | 1, 2, 3 |
| 5.4: How often do you use a CD Player? | 4, 5 | 1, 2, 3 |
| 5.5: How often do you play TV games (not Computer games)? | 4, 5 | 1, 2, 3 |
| 5.6: How often do you use a cellular (cell) phone? | 4, 5 | 1, 2, 3 |
| 5.7: How often do you select programs on Digital Satellite Television (DSTV)? | 4, 5 | 1, 2, 3 |

## I.3 Observed values

### *I.3.1 Part 1 : Previous Experience*

| Survey Item | Proportion of Positive Responses | | $\chi^2$-test statistic | p-value |
|---|---|---|---|---|
| | Treatment Group (*n* = 59) | Control Group (*n* = 59) | | |
| 1 | 70% (*n* = 41) | 71% (*n* = 42) | 0.04 | 0.840 |

### *I.3.2 Part 2 : Self Assessment of Ability*

| **Survey Item** | **Proportion of Positive Responses** | | **$\chi^2$-test statistic** | **p-value** |
|---|---|---|---|---|
| | **Treatment Group (*n* = 59)** | **Control Group (*n* = 59)** | | |
| 2.1 | 95% (*n* = 56) | 85% (*n* = 50) | 3.34 | 0.068 |
| 2.2 | 90% (*n* = 53) | 81% (*n* = 48) | 1.72 | 0.190 |
| 2.3 | 97% (*n* = 57) | 100% (*n* = 59) | 2.03 | 0.154 |
| 2.4 | 93% (*n* = 55) | 95% (*n* = 56) | 0.15 | 0.697 |

### *I.3.3 Part 3 : Attitude*

| **Survey Item** | **Proportion of Positive Responses** | | **$\chi^2$-test statistic** | **p-value** |
|---|---|---|---|---|
| | **Treatment Group (*n* = 59)** | **Control Group (*n* = 59)** | | |
| 3.1 | 98% (*n* = 58) | 98% (*n* = 58) | 0.00 | 1.000 |
| 3.2 | 100% (*n* = 59) | 100% (*n* = 59) | 0.00 | 1.000 |
| 3.3 | 100% (*n* = 59) | 100% (*n* = 59) | 0.00 | 1.000 |
| 3.4 | 98% (*n* = 58) | 100% (*n* = 59) | 1.01 | 0.315 |

### *I.3.4 Part 4 : Previous Experience*

| **Survey Item** | **Proportion of Positive Responses** | | **$\chi^2$-test statistic** | **p-value** |
|---|---|---|---|---|
| | **Treatment Group (*n* = 59)** | **Control Group (*n* = 59)** | | |
| 4.1 | 81% (*n* = 48) | 86% (*n* = 51) | 0.56 | 0.452 |
| 4.2 | 35% (*n* = 21) | 63% (*n* = 37) | 8.68 | 0.003 ** |
| 4.3 | 31% (*n* = 18) | 35% (*n* = 21) | 0.34 | 0.557 |
| 4.4 | 26% (*n* = 15) | 24% (*n* = 14) | 0.05 | 0.831 |
| 4.5 | 50% (*n* = 29) | 45% (*n* = 27) | 0.14 | 0.712 |
| 4.6 | 43% (*n* = 25) | 34% (*n* = 20) | 0.90 | 0.343 |
| 4.7 | 0% (*n* = 0) | 4% (*n* = 2) | 2.03 | 0.154 |

** significant at $p < 0.01$

### *I.3.5 Part 5 : Contact with Other Technology*

| **Survey Item** | **Proportion of Positive Responses** | | **$\chi^2$-test statistic** | **p-value** |
|---|---|---|---|---|
| | **Treatment Group (*n* = 59)** | **Control Group (*n* = 59)** | | |
| 5.1 | 57% (*n* = 34) | 42% (*n* = 25) | 2.75 | 0.098 |
| 5.2 | 61% (*n* = 36) | 51% (*n* = 30) | 1.24 | 0.266 |
| 5.3 | 56% (*n* = 33) | 51% (*n* = 30) | 0.31 | 0.580 |
| 5.4 | 85% (*n* = 50) | 92% (*n* = 54) | 1.30 | 0.255 |
| 5.5 | 24% (*n* = 14) | 27% (*n* = 16) | 0.18 | 0.672 |
| 5.6 | 83% (*n* = 49) | 94% (*n* = 55) | 2.92 | 0.088 |
| 5.7 | 62% (*n* = 37) | 42% (*n* = 25) | 4.89 | 0.027 |

# Appendix J

# Practical Learning Activity Survey Analysis

## J.1 Survey Forms

**WRA101 : Practical 3 Reflections**
**Week 4 : 4 - 6 March 2003**

Please enter your student number below – this will only be used in the comparison of today's reflections with future reflections for research purposes.

*Task 5 (Children visiting Bayworld) Reflections*

Choice of Environment (*please tick ☑ the package which you used to develop the final solution to this task*)

□ B#　　□ Delphi　　□ Didn't do Task 5

Please state all your reasons for choosing the above mentioned environment – if you didn't do Task 5, please state your reasons for this:

________________________________________

________________________________________

________________________________________

________________________________________

If you at any stage attempted the solution in the alternative environment, please state your reasons for rather choosing the environment ticked above:

________________________________________

________________________________________

________________________________________

________________________________________

# WRA101 : Practical 4 Reflections

## Week 5 : 11 - 13 March 2003

Please enter your student number below – this will only be used in the comparison of today's reflections with future reflections for research purposes.

*Task 6 (Determine whether a point falls in a given square) Reflections*

Choice of Environment (*please tick* ☑ *the package which you used to develop the final solution to this task*)

□ B#    □ Delphi    □ Didn't do Task 5

Please state all your reasons for choosing the above mentioned environment – if you didn't do Task 6, please state your reasons for this:

If you at any stage attempted the solution in the alternative environment, please state your reasons for rather choosing the environment ticked above:

# WRA101 : Practical 6 Reflections
## Week 7 : 25 - 27 March 2003

Please enter your student number below – this will only be used in the comparison of today's reflections with future reflections for research purposes.

*Task 5 (Determine population of South Africa) Reflections*

Choice of Environment (*please tick* ☑ *the package which you used to develop the final solution to this task*)

□ B# □ Delphi □ Didn't do Task 5

Please state all your reasons for choosing the above mentioned environment – if you didn't do Task 5, please state your reasons for this:

If you at any stage attempted the solution in the alternative environment, please state your reasons for rather choosing the environment ticked above:

# WRA101 : Practical 7 Reflections

## Week 8 : 8 - 10 April 2003

Please enter your student number below – this will only be used in the comparison of today's reflections with future reflections for research purposes.

*Task 4 (Menu driven program) Reflections*

Choice of Environment (*please tick ☑ the package which you used to develop the final solution to this task*)

□ B# □ Delphi □ Didn't do Task 4

Please state all your reasons for choosing the above mentioned environment – if you didn't do Task 4, please state your reasons for this:

If you at any stage attempted the solution in the alternative environment, please state your reasons for rather choosing the environment ticked above:

# WRA101 : Practical 8 Reflections

## Week 9 : 15 - 17 April 2003

Please enter your student number below – this will only be used in the comparison of today's reflections with future reflections for research purposes.

| *Task Reflections* | *Please tick ☑ the package which you used to develop the final solution to each task* | | | *Please state all your reasons for choosing the ticked environment – if you didn't do the task, please state your reasons for this:* | *If you at any stage attempted the solution in the alternative environment, please state your reasons for rather choosing the environment ticked:* |
|---|---|---|---|---|---|
| **Task** | **B#** | **Delphi** | **Didn't do** | | |
| **3** (Quadratic formula) | □ | □ | □ | | |
| **4** (Maximum and minimum) | □ | □ | □ | | |
| **5** (High-low game) | □ | □ | □ | | |

# WRA101 : Practical 9 Reflections

## Week 10 : 22 - 24 April 2003

Please enter your student number below – this will only be used in the comparison of today's reflections with future reflections for research purposes.

*Task 5 (Determinant function) Reflections*

Choice of Environment (*please tick ☑ the package which you used to develop the final solution to this task*)

☐ B#  ☐ Delphi  ☐ Didn't do Task 5

Please state all your reasons for choosing the above mentioned environment – if you didn't do Task 5, please state your reasons for this:

If you at any stage attempted the solution in the alternative environment, please state your reasons for rather choosing the environment ticked above:

## J.2 Survey Analysis

### *J.2.1 Week 4*

| | | | **Preferred Environment Response Frequency** | | **Examples of Actual Responses** |
|---|---|---|---|---|---|
| **Theme** | **Sub-theme** | **Category** | **Delphi** | **B#** | |
| **Motivation** | Extrinsic Motivation | Experienced in Delphi | 3 | 0 | *I understand Delphi a bit more and it's faster* |
| | | Delphi is examinable | 4 | 0 | *I am going to be examined in Delphi* |
| | | Could just copy and adapt a previous program | 1 | 0 | *I was too lazy to close everything and open another again* |
| | | **Total Sub-theme** | **8** | **0** | |
| | Inaccessibility | Did not have B# at home | 7 | 0 | *Did the task at home and I only have Delphi there. I find B# to be quite helpful as well.* |
| | | **Total Sub-theme** | **7** | **0** | |
| | Intrinsic Motivation | Delphi is challenging | 2 | 0 | *B# feels like cheating and I need practice in Delphi* |
| | | Just wanted to use B# | 2 | 3 | *I felt like it at the moment* |
| | | **Total Sub-theme** | **4** | **3** | |
| | ***Total Theme*** | | ***19*** | ***3*** | |
| **Usability** | Difficult to use | B# complicated | 1 | 0 | *B# is too complicated* |
| | | Got stuck with Delphi | 0 | 1 | *Its easier to make mistakes like typing errors in Delphi* |
| | | **Total Sub-theme** | **1** | **1** | |
| | Easy to use | B# user friendly | 0 | 4 | *It was much easier to see what I was doing in B#* |
| | | B# prevents errors | 0 | 4 | *Quicker and less likely to make a mistake with* |
| | | Easier than alternative | 2 | 28 | *B# is more visual, easier to use!* |
| | | Less syntactical issues | 0 | 6 | *Less chance of inputting incorrect information* |
| | | Quicker | 2 | 4 | *I found it much quicker and easier to use* |
| | | **Total Sub-theme** | **4** | **46** | |
| | Enhances comprehension | Visual | 0 | 6 | *Easier because of iconic format* |
| | | More understandable | 3 | 1 | *Easy 2 understand* |
| | | Related to flowcharting | 0 | 5 | *I am able to visualise the flowchart* |
| | | Showed correct code | 0 | 1 | *You don't have to write the code yourself* |
| | | **Total Sub-theme** | **3** | **13** | |
| | ***Total Theme*** | | ***8*** | ***60*** | |

### *J.2.2 Week 5*

| | | | **Preferred Environment Response Frequency** | | **Examples of Actual Responses** |
|---|---|---|---|---|---|
| **Theme** | **Sub-theme** | **Category** | **Delphi** | **B#** | |
| **Motivation** | Extrinsic Motivation | Experienced in Delphi | 4 | 0 | *Have become accustomed to Delphi code* |
| | | Delphi is examinable | 6 | 0 | *Writing the code in Delphi yourself is more beneficial as you’ll have to write all future programmes in Delphi anyway* |
| | | **Total Sub-theme** | **10** | **0** | |
| | Inaccessibility | Did not have B# at home | 16 | 0 | *I don’t have B# a home yet* |
| | | **Total Sub-theme** | **16** | **0** | |
| | Intrinsic Motivation | Delphi is challenging | 2 | 0 | *Prefer writing code from start* |
| | | Just wanted to use B# | 2 | 0 | *Dislike typing; like working with mouse* |
| | | **Total Sub-theme** | **4** | **0** | |
| | ***Total Theme*** | | ***30*** | ***0*** | |
| **Usability** | Difficult to use | B# complicated | 1 | 0 | *Wasn’t sure how the B# program did if and else statements* |
| | | B# time consuming | 1 | 0 | *B# takes to long and sometimes doesn’t run* |
| | | **Total Sub-theme** | **1** | **0** | |
| | | B# prevents errors | 0 | 4 | *It corrects you and help you to re-check your programm* |
| | | Easier than alternative | 6 | 23 | *B# responds the first time round when I create a program* |
| | | Less syntactical issues | 0 | 1 | *Not as many things to remember* |
| | | Quicker | 3 | 7 | *Its easier and less time consuming* |
| | | **Total Sub-theme** | **9** | **35** | |
| | Enhances comprehension | Visual | 0 | 3 | *Better setted out than Delphi* |
| | | More understandable | 3 | 3 | *I wasn’t sure of how to write the program so B# helped to develop it* |
| | | Related to flowcharting | 0 | 1 | *I find B# easier than Delphi bcoz flowcharts are easily understandable* |
| | | **Total Sub-theme** | **3** | **7** | |
| | ***Total Theme*** | | ***43*** | ***42*** | |

### *J.2.3 Week 7*

| | | | **Preferred Environment Response Frequency** | | **Examples of Actual Responses** |
|---|---|---|---|---|---|
| **Theme** | **Sub-theme** | **Category** | **Delphi** | **B#** | |
| **Motivation** | Extrinsic Motivation | Experienced in Delphi | 2 | 0 | *We are using Delphi in lectures so I am finding it easier* |
| | | Delphi is examinable | 7 | 0 | *Wanted to get more experience in Delphi. Did not want B# assistance* |
| | | **Total Sub-theme** | **9** | **0** | |
| | Inaccessibility | Did not have B# at home | 21 | 3 | *Don't have B# at home* |
| | | **Total Sub-theme** | **21** | **3** | |
| | Intrinsic Motivation | Delphi is challenging | 1 | 1 | *Delphi is easier to use as it flows instead of having to stop every five seconds to add a chart* |
| | | Just wanted to use the environment | 2 | 0 | *I prefer working with Delphi* |
| | | **Total Sub-theme** | **3** | **1** | |
| | ***Total Theme*** | | ***33*** | ***4*** | |
| **Usability** | Difficult to use | B# complicated | 1 | 0 | *Much easier to visualise the output from Delphi code* |
| | | **Total Sub-theme** | **1** | **0** | |
| | Easy to use | B# user friendly | 0 | 1 | *B# is much less complicated. I definitely prefer using B#* |
| | | B# prevents errors | 0 | 1 | *The chance of making mistakes is reduced* |
| | | Easier than alternative | 15 | 18 | *Easier to understand the progam* |
| | | Quicker | 3 | 7 | *It is not as complicated as Delphi. It is also quicker to do than working with Delphi* |
| | | **Total Sub-theme** | **18** | **27** | |
| | | More understandable | 3 | 5 | *Easier to understand* |
| | | Related to flowcharting | 0 | 1 | *It is easier to understand the flowchart* |
| | | **Total Sub-theme** | **3** | **6** | |
| | ***Total Theme*** | | ***24*** | ***31*** | |

### *J.2.4 Week 8*

| | | | **Preferred Environment Response Frequency** | | **Examples of Actual Responses** |
|---|---|---|---|---|---|
| **Theme** | **Sub-theme** | **Category** | **Delphi** | **B#** | |
| **Motivation** | Extrinsic Motivation | Experienced in Delphi | 1 | 0 | *Taught Delphi in class* |
| | | Delphi is examinable | 10 | 0 | *Because it is the program we write exams on* |
| | | **Total Sub-theme** | **11** | **0** | |
| | Inaccessibility | Did not have B# at home | 9 | 1 | *I have Delphi at home which is more convenient* |
| | | **Total Sub-theme** | **9** | **1** | |
| | Intrinsic Motivation | Just wanted to use the environment | 3 | 0 | *I prefer using Delphi* |
| | | **Total Sub-theme** | **3** | **0** | |
| | ***Total Theme*** | | ***20*** | ***1*** | |
| **Usability** | Difficult to use | B# complicated | 0 | 1 | *B# to tedious and more difficult to interpret* |
| | | B# time consuming | 0 | 1 | |
| | | **Total Sub-theme** | **0** | **2** | |
| | Easy to use | B# user friendly | 1 | 1 | *It is more user friendly and less difficult* |
| | | B# prevents errors | 0 | 1 | *Case statement easier in B#* |
| | | Easier than alternative | 10 | 5 | *It was much easier and less complicated using Delphi* |
| | | Quicker | 0 | 1 | *Time got a bit short so I had to take the fastest option* |
| | | **Total Sub-theme** | **11** | **8** | |
| | Enhances comprehension | More understandable | 2 | 3 | *Task 4 was very long and complicated and B# seemed to make it easier to understand* |
| | | **Total Sub-theme** | **2** | **3** | |
| | ***Total Theme*** | | ***13*** | ***13*** | |

### *J.2.5 Week 9 (Task 3)*

| | | | **Preferred Environment Response Frequency** | | **Examples of Actual Responses** |
|---|---|---|---|---|---|
| **Theme** | **Sub-theme** | **Category** | **Delphi** | **B#** | |
| **Motivation** | Extrinsic Motivation | Experienced in Delphi | 1 | 0 | *More comfortable working in Delphi* |
| | | Delphi is examinable | 2 | 0 | *Delphi is what I am examined on* |
| | | **Total Sub-theme** | **3** | **0** | |
| | Inaccessibility | Did not have B# at home | 15 | 0 | *I don't have the B# disk at home* |
| | | **Total Sub-theme** | **15** | **0** | |
| | Intrinsic Motivation | Just wanted to use the environment | 3 | 0 | *Choose to alternate* |
| | | **Total Sub-theme** | **3** | **0** | |
| | ***Total Theme*** | | ***21*** | ***0*** | |
| **Usability** | Difficult to use | B# complicated | 3 | 0 | *I was confused with B#* |
| | | B# time consuming | 2 | 0 | *Delphi a lot quicker to use* |
| | | **Total Sub-theme** | **5** | **0** | |
| | Easy to use | User friendly | 1 | 0 | *User friendly* |
| | | Easier than alternative | 9 | 10 | *Much easier for me* |
| | | **Total Sub-theme** | **10** | **10** | |
| | Enhances comprehension | More understandable | 2 | 0 | *Delphi got a bit tricky so I went to B# to get a few tips* |
| | | **Total Sub-theme** | **2** | **0** | |
| | ***Total Theme*** | | ***17*** | ***10*** | |

### *J.2.6 Week 9 (Task 4)*

| | | | **Preferred Environment Response Frequency** | | **Examples of Actual Responses** |
|---|---|---|---|---|---|
| **Theme** | **Sub-theme** | **Category** | **Delphi** | **B#** | |
| **Motivation** | Extrinsic Motivation | Experienced in Delphi | 1 | 0 | *More comfortable working in Delphi* |
| | | Delphi is examinable | 2 | 0 | *Delphi is what I am examined on* |
| | | **Total Sub-theme** | **3** | **0** | |
| | Inaccessibility | Did not have B# at home | 15 | 0 | *Don’t have B#* |
| | | **Total Sub-theme** | **15** | **0** | |
| | Intrinsic Motivation | Just wanted to use the environment | 2 | 1 | *Less frustrating than B#* |
| | | **Total Sub-theme** | **2** | **1** | |
| | | ***Total Theme*** | ***20*** | ***1*** | |
| **Usability** | Difficult to use | B# complicated | 1 | 0 | *B# is difficult to use and does not always work* |
| | | B# didn’t work | 1 | 0 | |
| | | B# time consuming | 2 | 0 | *B# to tedious* |
| | | **Total Sub-theme** | **4** | **0** | |
| | Easy to use | Easier than alternative | 9 | 11 | *Much easier for me* |
| | | Quicker | 1 | 0 | *Delphi a lot quicker to use* |
| | | **Total Sub-theme** | **10** | **11** | |
| | Enhances comprehension | More understandable | 3 | 0 | *Did it in B# and exported code to Delphi* |
| | | **Total Sub-theme** | **3** | **0** | |
| | | ***Total Theme*** | ***17*** | ***11*** | |

### *J.2.7 Week 9 (Task 5)*

| | | | **Preferred Environment Response Frequency** | | **Examples of Actual Responses** |
|---|---|---|---|---|---|
| **Theme** | **Sub-theme** | **Category** | **Delphi** | **B#** | |
| **Motivation** | Extrinsic Motivation | Experienced in Delphi | 1 | 0 | *More comfortable working in Delphi* |
| | | Delphi is examinable | 2 | 0 | *Delphi is what I am examined on* |
| | | **Total Sub-theme** | **3** | **0** | |
| | Inaccessibility | Did not have B# at home | 16 | 0 | *I did it at home where I have use of Delphi* |
| | | **Total Sub-theme** | **16** | **0** | |
| | Intrinsic Motivation | Just wanted to use the environment | 2 | 0 | *Just did Delphi for a change* |
| | | **Total Sub-theme** | **2** | **0** | |
| | | ***Total Theme*** | ***19*** | ***0*** | |
| **Usability** | Difficult to use | B# complicated | 1 | 0 | *B# is difficult to use* |
| | | B# time consuming | 2 | 0 | *Delphi a lot quicker to use* |
| | | **Total Sub-theme** | **3** | **0** | |
| | Easy to use | B# user friendly | 1 | 0 | *B# is a lovely program for people starting programming!* |
| | | Easier than alternative | 8 | 6 | *Easier* |
| | | Quicker | 1 | 0 | *Delphi a lot quicker to use* |
| | | **Total Sub-theme** | **10** | **6** | |
| | Enhances comprehension | Visual | 0 | 1 | *Simpler to see program in picture format* |
| | | More understandable | 2 | 0 | *Its better* |
| | | **Total Sub-theme** | **2** | **1** | |
| | | ***Total Theme*** | ***15*** | ***7*** | |

### *J.2.8 Week 10*

| | | | **Preferred Environment Response Frequency** | | **Examples of Actual Responses** |
|---|---|---|---|---|---|
| **Theme** | **Sub-theme** | **Category** | **Delphi** | **B#** | |
| **Motivation** | Extrinsic Motivation | Experienced in Delphi | 2 | 0 | *At first I always used B#, but now I've gotten quite comfortable with Delphi* |
| | | Delphi is examinable | 14 | 0 | *Need practice for exams* |
| | | **Total Sub-theme** | **16** | **0** | |
| | Inaccessibility | Did not have B# at home | 6 | 0 | *No B# at home* |
| | | **Total Sub-theme** | **6** | **0** | |
| | Intrinsic Motivation | Withdrew from investigation | 1 | 0 | *I dropped B#* |
| | | Just wanted to use the environment | 2 | 0 | *Prefer to do in Delphi* |
| | | **Total Sub-theme** | **3** | **0** | |
| | ***Total Theme*** | | ***25*** | ***0*** | |
| **Usability** | Difficult to use | B# complicated | 6 | 0 | *Functions and procedures – B# confusing* |
| | | B# time consuming | 1 | 0 | *B# is a bit slower than Delphi. What I mean is that it takes longer to write a program in B# than it does in Delphi* |
| | | **Total Sub-theme** | **7** | **0** | |
| | Easy to use | B# user friendly | 2 | 2 | *I used B# as it is much more user-friendly and copied the code over to Delphi* |
| | | Easier than alternative | 16 | 9 | *It's easier for me to work in Delphi* |
| | | Less syntactical issues | 0 | 1 | *Was much clearer* |
| | | Flexible | 1 | 0 | *Allows me to apply new function and ways of developing a program and also ways of displaying information. Own corrections also easy to implement* |
| | | Quicker | 5 | 4 | *It was quicker and easier* |
| | | **Total Sub-theme** | **24** | **16** | |
| | Enhances comprehension | Visual | 0 | 1 | *Sets out the different "ifs" clearly* |
| | | More understandable | 1 | 0 | *Easier to correct a mistake* |
| | | Related to flowcharting | 0 | 1 | *It is easier with pictures (flowchart)* |
| | | Showed correct code | 0 | 1 | *The program sets it out for you* |
| | | **Total Sub-theme** | **1** | **3** | |
| | ***Total Theme*** | | ***32*** | ***19*** | |

# Appendix K

# Programming Development Environment Survey Analysis

## K.1 Survey Questions

1. If you had the option to do the entire course, lectures and exams included, using only one of B# or Delphi, which would you choose? Give reasons for your answer.

2. How did you experience solving practicals in B#?

3. How did you experience solving practicals in Delphi?

4. What did you specifically like and dislike about solving practicals in B#?

5. What did you specifically like and dislike about solving practicals in Delphi?

6. In what ways do you feel that using B# benefited and/or hindered you while doing your practicals?

7. In what ways do you feel that using Delphi benefited and/or hindered you while doing your practicals?

## K.2 Survey Analysis

### *K.2.1 Treatment Group: Question 1*

| | | | **Preferred Environment Response Frequency** | | **Examples of Actual Responses** |
|---|---|---|---|---|---|
| **Theme** | **Sub-theme** | **Category** | **Delphi (*n* = 49)** | **B# (*n* = 28)** | |
| **Motivation** | Extrinsic Motivation | Delphi challenging | 3 | 0 | *Prefer a challenge* |
| | | Delphi used in industry | 2 | 0 | *It gets use in practice more often* |
| | | Don't like B# | 1 | 0 | *I don' like B#* |
| | | Experienced in Delphi | 2 | 0 | *Get taught in Delphi, all the notes are in Delphi* |
| | | Prefer to concentrate on a single environment | 2 | 1 | *It is easier to concentrate on one program* |
| | | **Total Sub-theme** | **10** | **1** | |
| | Inaccessibility | Did not have B# at home | 1 | 1 | *I don't have B# at home* |
| | | **Total Sub-theme** | **1** | **1** | |
| | Intrinsic Motivation | B# tedious | 3 | 0 | *B# too tedious* |
| | | Preferred environment is fun | 1 | 1 | *Delhi is much more fun to work with* |
| | | Preferred environment is quicker | 4 | 4 | *Quicker to do program* |
| | | **Total Sub-theme** | **8** | **5** | |
| | ***Total Theme*** | | ***19*** | ***7*** | |

| | | | **Preferred Environment Response Frequency** | | **Examples of Actual Responses** |
|---|---|---|---|---|---|
| **Theme** | **Sub-theme** | **Category** | **Delphi (*n* = 49)** | **B# (*n* = 28)** | |
| **Usability** | Enhanced feedback | Delphi indicates mistakes clearly | 4 | 0 | *Delphi shows where the mistakes are* |
| | | **Total Sub-theme** | **4** | **0** | |
| | Easy to use | B# good starting environment | 6 | 0 | *B# would have been wonderful to have when starting programming* |
| | | B# is user friendly | 0 | 3 | *Provides more help and is user friendly* |
| | | Preferred environment is easier | 15 | 22 | *Easier to see what you are doing* |
| | | Easier to edit a Delphi program | 5 | 0 | *Easier to edit if you made errors* |
| | | **Total Sub-theme** | **26** | **25** | |
| | Reduces detail | B# requires less textual programming | 0 | 3 | *Does not require typing a lot of text* |
| | | Less to remember in B# | 0 | 1 | *Less terms to remember* |
| | | **Total Sub-theme** | **0** | **4** | |
| | Restricted functionality | B# code not sufficiently functional | 13 | 1 | *I don't like the fact that you have to delete procedures/functions in they are in the wrong place* |
| | | B# code not directly accessible | 2 | 0 | *Maybe if you could get into the code of B# also* |
| | | Delphi functionality works | 1 | 0 | *All functions work in Delphi* |
| | | Delphi less rigid | 5 | 0 | *B# makes me feel restricted* |
| | | Low level visibility in Delphi | 1 | 0 | *Able to see exactly what is happening* |
| | | Tutor assistance lacking for B# | 2 | 0 | *Tutors couldn't help properly* |
| | | **Total Sub-theme** | **24** | **1** | |
| | Enhances comprehension | B# assists with planning solution | 5 | 4 | *Making students used to flowcharts* |
| | | B# ensures correctness step-by-step | 0 | 3 | *Difficult to make errors when using B#* |
| | | B# is visual program | 0 | 7 | *Having a visually based program helps me to see what is happening* |
| | | B# provides correct code | 0 | 2 | *You get to see the code on the side* |
| | | Preferred environment more understandable | 3 | 7 | *B# is much more clear* |
| | | **Total Sub-theme** | **8** | **23** | |
| | | ***Total Theme*** | ***62*** | ***53*** | |

### *K.2.2 Control Group: Question 1*

| | | | **Preferred Environment Response Frequency** | | **Examples of Actual Responses** |
|---|---|---|---|---|---|
| **Theme** | **Sub-theme** | **Category** | **Delphi<br>(*n* = 110)** | **B#<br>(*n* = 6)** | |
| **B# unknown** | Do not know B# | **Total Sub-theme** | **73** | **0** | *I don't know what B# is* |
| | ***Total Theme*** | | ***73*** | ***0*** | |
| **B# known** | B# confusing/difficult | **Total Sub-theme** | **3** | **0** | *Because less confusing* |
| | No direct access to code | **Total Sub-theme** | **4** | **1** | *In B# you can't change code* |
| | Time consuming | **Total Sub-theme** | **3** | **0** | *Delphi consumes less time* |
| | Restrictive | **Total Sub-theme** | **3** | **1** | *B# does not have all capabilities of Delphi* |
| | Industry related | **Total Sub-theme** | **7** | **0** | *More popular program* |
| | Heard about B# | **Total Sub-theme** | **14** | **2** | *I don't know in detail what B# is about* |
| | ***Total Theme*** | | ***34*** | ***4*** | |

### *K.2.3 Treatment Group: Question 2*

| | | | **Preferred Environment Response Frequency** | | **Examples of Actual Responses** |
|---|---|---|---|---|---|
| **Theme** | **Sub-theme** | **Category** | **Delphi<br>(*n* = 49)** | **B#<br>(*n* = 28)** | |
| **Motivation** | Extrinsic Motivation | B# an unnecessary environment for the course | 0 | 2 | *Don't need B# for the future* |
| | | B# less fun | 3 | 0 | *I did not enjoy B#* |
| | | B# stressful | 0 | 1 | *B# difficult and confusing* |
| | | B# waste of time | 2 | 0 | *Waste of time; could spend more time on Delphi* |
| | | B# only at UPE | 1 | 0 | *B# works only at UPE* |
| | | B# not examinable | 1 | 4 | *It was nice at first until we were told that we use Delphi for exams* |
| | | Didn't attempt B# | 0 | 1 | *Dropped B#* |
| | | More assistance required in B# | 3 | 1 | *I needed frequent help* |
| | | Delphi experience | 1 | 0 | *Programmed in Delphi before* |
| | | **Total Sub-theme** | **11** | **9** | |
| | Time consuming | B# tedious | 4 | 1 | *Way too inconvenient* |
| | | Too long in B# | 2 | 0 | *It took too long* |
| | | **Total Sub-theme** | **6** | **1** | |
| | Intrinsic Motivation | B# challenging | 1 | 1 | *Enjoyed the challenge* |
| | | B# enjoyable | 1 | 7 | *Found B# enjoyable* |
| | | B# fun | 1 | 0 | *Fun to use B#* |
| | | Better than Delphi | 0 | 1 | *Preferred B#* |
| | | Don't like flowcharts | 1 | 0 | *Don't like flowcharts* |
| | | Feels like cheating | 0 | 1 | *I felt that it made me lazy* |
| | | Greater chance of program running in B# than Delphi | 0 | 2 | *My programs worked in B#* |
| | | Less creative than Delphi | 1 | 0 | *I liked that I could use my own method in Delphi* |
| | | Less of a sense of achievement in B# | 0 | 1 | *I felt better when my programs ran in Delphi* |
| | | More of a sense of achievement in B# | 0 | 1 | *I enjoyed that my programs worked in B#* |
| | | Prefer to solve directly in code | 1 | 0 | *Prefer correcting code directly* |
| | | **Total Sub-theme** | **6** | **14** | |
| | | ***Total Theme*** | ***23*** | ***24*** | |

| | | | **Preferred Environment Response Frequency** | | **Examples of Actual Responses** |
|---|---|---|---|---|---|
| **Theme** | **Sub-theme** | **Category** | **Delphi**<br>**(*n* = 49)** | **B#**<br>**(*n* = 28)** | |
| **Usability** | Visual | Graphical approach stimulating | 1 | 0 | *Enjoyed the graphical approach* |
| | | **Total Sub-theme** | **1** | **0** | |
| | Easy to use | B# good starting environment | 13 | 3 | *OK in beginning* |
| | | B# is user friendly | 0 | 1 | *Find B# user-friendly* |
| | | B# restricted number of errors made | 1 | 1 | *B# prevented me putting in ; at the wrong place* |
| | | B# easier than Delphi | 11 | 13 | *Easier to write programs in B#* |
| | | B# for novice programmers | 1 | 0 | *Good for beginning programming* |
| | | Grasp programming concepts in B# | 1 | 0 | *Enables me to grasp the concepts of programming* |
| | | Quicker and more efficient than Delphi | 2 | 2 | *Quicker to get a program written and working in B#* |
| | | **Total Sub-theme** | **29** | **20** | |
| | Restricted functionality | B# difficult to use when solving problems | 2 | 0 | *I found it difficult to solve problems using B#* |
| | | B# forces unnecessary implementation | 1 | 0 | *Sometimes B# lets me do unnecessary things* |
| | | B# gave technical problems | 4 | 0 | *Ran old programs* |
| | | B# has limited functionality | 1 | 0 | *Cannot round or use decimals* |
| | | B# is complicated in advanced features | 15 | 2 | *Towards procedures/functions it became too complicated* |
| | | B# is frustrating | 5 | 0 | *It was a little frustrating to use B#* |
| | | B# is tough | 1 | 0 | *I found it tough* |
| | | Did not retain knowledge of code | 1 | 0 | *I did not remember the code from B#* |
| | | Hated completing dialogue boxes in B# | 1 | 0 | *I hated B#'s dialogue boxes* |
| | | No copy-and-paste facility in B# | 1 | 0 | *Cannot duplicate things in B#* |
| | | Wanted to change code directly in B# | 1 | 0 | *It is better to write code* |
| | | **Total Sub-theme** | **33** | **2** | |
| | Enhances comprehension | Code provided by B# assisted with Delphi code | 1 | 0 | *I learnt from B#'s code* |
| | | Exported B# code for use in Delphi | 1 | 1 | *I exported programs from B# to Delphi* |
| | | Preferred environment more understandable | 0 | 4 | *It was more understandable* |
| | | **Total Sub-theme** | **2** | **5** | |
| | | ***Total Theme*** | ***64*** | ***27*** | |

### *K.2.4 Treatment Group: Question 3*

| | | | **Preferred Environment Response Frequency** | | **Examples of Actual Responses** |
|---|---|---|---|---|---|
| **Theme** | **Sub-theme** | **Category** | **Delphi (*n* = 49)** | **B# (*n* = 28)** | |
| **Motivation** | Extrinsic Motivation | Different to other languages | 1 | 0 | *B# is not like other languages* |
| | | Experienced in Delphi | 1 | 0 | *I know Delphi better* |
| | | Mirrored lectures | 0 | 2 | *Same as lectures* |
| | | More comfortable with Delphi | 1 | 0 | *I am confident using Delphi* |
| | | Preparation for exam | 0 | 1 | *Delphi helps me practice for the exam* |
| | | Tutors more knowledgeable in Delphi | 1 | 0 | *The tutors knew more about Delphi than B#* |
| | | **Total Sub-theme** | **4** | **3** | |
| | Intrinsic Motivation | Permits creativity | 5 | 1 | *I could use my own way* |
| | | Boring | 0 | 1 | *I was bored* |
| | | Do not like programming | 1 | 1 | *I hate programming* |
| | | Enjoyable | 13 | 2 | *I enjoyed using Delphi* |
| | | Faster | 1 | 0 | *It was faster* |
| | | Fun | 3 | 0 | *It was fun in Delphi* |
| | | Not fun | 0 | 1 | *I did not enjoy using Delphi* |
| | | Tedious | 2 | 2 | *Time consuming* |
| | | **Total Sub-theme** | **25** | **8** | |
| | ***Total Theme*** | | ***29*** | ***11*** | |

| | | | **Preferred Environment Response Frequency** | | **Examples of Actual Responses** |
|---|---|---|---|---|---|
| **Theme** | **Sub-theme** | **Category** | **Delphi (*n* = 49)** | **B# (*n* = 28)** | |
| **Usability** | Lots of detail | Lots of typing | 1 | 1 | *Needed a lot of typing* |
| | | Remember tiny things/detail | 1 | 4 | *Too many begins/ends/semi-colons* |
| | | **Total Sub-theme** | **2** | **5** | |
| | Easy to use | B# assisted | 1 | 1 | *B# helped in making it easier* |
| | | Better than B# | 1 | 1 | *Using Delphi was at least better* |
| | | Delphi easier | 9 | 1 | *Easier for me in Delphi* |
| | | Easy to fix errors in code directly | 2 | 0 | *Easier to correct errors in Delphi* |
| | | User friendly | 1 | 0 | *It is user-friendly* |
| | | **Total Sub-theme** | **14** | **3** | |
| | Difficult | Challenging | 5 | 2 | *I found it challenging* |
| | | Difficult | 2 | 5 | *It was difficult* |
| | | Don't know how to correct errors in Delphi | 1 | 0 | *Still don't know how to fix errors in Delphi* |
| | | Error messages not understood | 0 | 2 | *The errors written down there would be a whole lot of jargon* |
| | | Frustrating | 0 | 3 | *Frustrating when my programs did not run* |
| | | Had to think more | 1 | 0 | *Ad to concentrate more* |
| | | Hardly ever got a program to work | 1 | 1 | *Harder to grasp programming concepts* |
| | | Improved with time | 3 | 1 | *Got easier as time went on* |
| | | Initially had to refer to B# to get Delphi code right | 0 | 1 | *B# helped me learn some things* |
| | | Moderately difficult | 2 | 1 | *Found it a bit difficult* |
| | | More difficult than B# | 1 | 2 | *B# is easier* |
| | | More difficult than B# initially | 1 | 2 | *Got better* |
| | | No guidance as to correct program | 1 | 1 | *Needed assistance* |
| | | Struggled to rectify errors | 0 | 1 | *Could not fix errors* |
| | | **Total Sub-theme** | **18** | **22** | |
| | Enhances comprehension | B# protects from mistakes | 1 | 0 | *B# helps you make less errors* |
| | | Delphi shows location of mistakes | 10 | 0 | *See exactly where errors are* |
| | | Used Help facility | 1 | 0 | *There is a Help system* |
| | | **Total Sub-theme** | **12** | **0** | |
| | ***Total Theme*** | | ***46*** | ***30*** | |

### *K.2.5 Control Group: Question 3*

| **Theme** | **Sub-theme** | **Category** | **Frequency<br>(*n* = 116)** | **Examples of Actual Responses** |
|---|---|---|---|---|
| **Motivation** | Intrinsic Motivation | Hated it | 1 | *Hated using Delphi* |
| | | Motivating | 18 | *Found it enjoyable* |
| | | **Total Sub-theme** | **19** | |
| | ***Total Theme*** | | ***19*** | |
| **Usability** | Difficult | Difficult to use | ***38*** | *Difficult to use* |
| | | **Total Sub-theme** | **38** | |
| | Lots of detail | Detail | ***2*** | *Has to remember where to put semi-colons and stuff* |
| | | **Total Sub-theme** | **2** | |
| | Ease of use | Easy | ***29*** | *Easy to use* |
| | | Progressively easier | ***20*** | *Got better* |
| | | **Total Sub-theme** | **49** | |
| | ***Total Theme*** | | ***89*** | |

### *K.2.6 Treatment Group: Question 4 (Dislikes of B#)*

| | | | **Preferred Environment Response Frequency** | | **Examples of Actual Responses** |
|---|---|---|---|---|---|
| **Theme** | **Sub-theme** | **Category** | **Delphi (*n* = 49)** | **B# (*n* = 28)** | |
| **Motivation** | Extrinsic Motivation | No formal instruction | 1 | 3 | *Not the same as lectures* |
| | | Not for exams | 3 | 0 | *Not examinable* |
| | | **Total Sub-theme** | **4** | **3** | |
| | Inaccessibility | Could not use at home | 1 | 1 | *Did not have at home* |
| | | **Total Sub-theme** | **1** | **1** | |
| | Time Consuming | Long to solve | 5 | 0 | *Tedious to solve more difficult programs* |
| | | **Total Sub-theme** | **5** | **0** | |
| | ***Total Theme*** | | ***10*** | ***4*** | |
| **Usability** | Difficult | Confusing | 3 | 6 | *Confusing to use* |
| | | Difficult to edit | 2 | 1 | *Cannot edit code directly* |
| | | **Total Sub-theme** | **5** | **7** | |
| | Restricted functionality | Bugs | 0 | 1 | *Sometimes didn't work properly* |
| | | Cannot access generated code | 4 | 0 | *Cannot fix errors in code* |
| | | Confined space | 1 | 0 | *Had to work in confined space* |
| | | Functions | 5 | 3 | *Functions/procedures difficult to use* |
| | | Interpretive error checking | 4 | 4 | *Caught errors immediately but didn't let me carry on* |
| | | Restricted | 12 | 4 | *Cannot use my own way* |
| | | Variables | 1 | 0 | *Cannot round or use decimals* |
| | | **Total Sub-theme** | **27** | **12** | |
| | ***Total Theme*** | | ***32*** | ***19*** | |

### *K.2.7 Treatment Group: Question 4 (Likes of B#)*

| | | | **Preferred Environment Response Frequency** | | **Examples of Actual Responses** |
|---|---|---|---|---|---|
| **Theme** | **Sub-theme** | **Category** | **Delphi (*n* = 49)** | **B# (*n* = 28)** | |
| **Usability** | Visual | Icons | 6 | 5 | *Liked the pictures* |
| | | Visual good | 2 | 2 | *Flowcharts look pretty* |
| | | **Total Sub-theme** | **8** | **7** | |
| | Less detail | Elimination of detail | 7 | 10 | *Puts in semi-colons* |
| | | **Total Sub-theme** | **7** | **10** | |
| | Ease of use | Programs work | 3 | 1 | *Programs run* |
| | | Drag and drop | 2 | 1 | *Just had to drag-and-drop and organise your program* |
| | | Easy | 6 | 5 | *Solving problems is easier* |
| | | Faster | 4 | 1 | *Solving problems is quicker* |
| | | **Total Sub-theme** | **15** | **8** | |
| | Enhances comprehension | Detected errors immediately | 3 | 2 | *Detected on spot errors* |
| | | Exporting of generated code | 0 | 2 | *I could export my program to Delphi* |
| | | Flowcharts | 10 | 4 | *Liked the fact that I used flowcharts* |
| | | Generated code | 5 | 6 | *Gives me Delphi code* |
| | | Problem solving enhanced | 5 | 3 | *Better to work with components of flowchart* |
| | | Teaches | 3 | 2 | *Teaches me to program* |
| | | **Total Sub-theme** | **26** | **19** | |
| | ***Total Theme*** | | ***56*** | ***44*** | |

### *K.2.8 Treatment Group: Question 5 (Dislikes of Delphi)*

|  |  |  | **Preferred Environment Response Frequency** |  | **Examples of Actual Responses** |
|---|---|---|---|---|---|
| **Theme** | **Sub-theme** | **Category** | **Delphi (*n* = 49)** | **B# (*n* = 28)** |  |
| **Motivation** | Intrinsic Motivation | No reason | 0 | 1 | *I hate programming* |
|  |  | **Total Sub-theme** | **0** | **1** |  |
|  | Time consuming | Time consuming | 3 | 3 | *Too long to solve programs* |
|  |  | **Total Sub-theme** | **3** | **3** |  |
|  | ***Total Theme*** |  | ***3*** | ***4*** |  |
| **Usability** | Ease of use | Detail | 9 | 8 | *Too many begins and ends* |
|  |  | Difficult | 2 | 5 | *Difficult to use* |
|  |  | Errors not helpful | 8 | 4 | *I disliked the list of errors that I encountered* |
|  |  | **Total Sub-theme** | **19** | **17** |  |
|  | Visual | Textual nature | 0 | 1 | *Prefer visual* |
|  |  | **Total Sub-theme** | **0** | **1** |  |
|  | ***Total Theme*** |  | ***19*** | ***18*** |  |

### *K.2.9 Treatment Group: Question 5 (Likes of Delphi)*

| | | | **Preferred Environment Response Frequency** | | **Examples of Actual Responses** |
|---|---|---|---|---|---|
| **Theme** | **Sub-theme** | **Category** | **Delphi**<br>**(*n* = 49)** | **B#**<br>**(*n* = 28)** | |
| **Motivation** | Extrinsic Motivation | Familiar with Delphi | 2 | 0 | *I know Delphi better* |
| | | Matches lectures | 4 | 3 | *My lecturer taught in Delphi* |
| | | **Total Sub-theme** | **6** | **3** | |
| | ***Total Theme*** | | ***6*** | ***3*** | |
| **Usability** | Enhances comprehension | Help system | 1 | 0 | *I used the Help of Delphi* |
| | | Shows where errors are | 10 | 8 | *Shows me where my mistakes are* |
| | | **Total Sub-theme** | **11** | **8** | |
| | Restricted functionality of B# | B# didn't work | 1 | 0 | *B# didn't work on my computer at home* |
| | | Code accessed directly | 1 | 0 | *Could fix errors in code* |
| | | Functions easier than in B# | 0 | 1 | *Implementing functions in Delphi better* |
| | | Not restrictive on format of program | 7 | 5 | *Liked that I could use my own method* |
| | | Standard interface in environment | 1 | 0 | *Has some common features like copy, paste, etc* |
| | | **Total Sub-theme** | **10** | **6** | |
| | Ease of use | Easy to edit | 12 | 1 | *Easy and quick to edit program* |
| | | **Total Sub-theme** | **12** | **1** | |
| | ***Total Theme*** | | ***33*** | ***15*** | |

### *K.2.10 Control Group: Question 5 (Dislikes of Delphi)*

| Theme | Sub-theme | Category | Frequency<br>(*n* = 116) | Examples of Actual Responses |
|---|---|---|---|---|
| **Motivation** | Extrinsic Motivation | Demotivating | 9 | *I found Delphi demotivating* |
| | | Time consuming | 8 | *Too long to solve problems* |
| | | Pace too fast | 2 | *Lectures too fast to get to know Delphi properly* |
| | | **Total Sub-theme** | **19** | |
| | | ***Total Theme*** | ***19*** | |
| **Usability** | Enhances comprehension | Errors confusing | 10 | *Error messages not clear* |
| | | Lack of feedback | 13 | *Did not know what was wrong with my programs* |
| | | Difficult | 11 | *Difficult to use* |
| | | **Total Sub-theme** | **34** | |
| | Lots of detail | Detail | 9 | *Too many things to remember* |
| | | Textual | 1 | *Too much typing* |
| | | **Total Sub-theme** | **10** | |
| | | ***Total Theme*** | ***44*** | |

### *K.2.11 Control Group: Question 5 (Likes of Delphi)*

| Theme | Sub-theme | Category | Frequency<br>(*n* = 116) | Examples of Actual Responses |
|---|---|---|---|---|
| **Motivation** | Extrinsic Motivation | Matches lectures | 2 | *Lectures in Delphi* |
| | | Programs run | 7 | *Programs run in Delphi* |
| | | **Total Sub-theme** | **9** | |
| | | ***Total Theme*** | ***9*** | |
| **Usability** | Enhances comprehension | Shows errors | 24 | *List where errors are* |
| | | Testing program | 2 | *Can test my program in Delphi* |
| | | Easy to learn | 6 | *Found it easy to learn* |
| | | B# too rigid | 1 | *Less rigid* |
| | | **Total Sub-theme** | **33** | |
| | Ease of use | Easy to read | 1 | *Easy to read program in Delphi* |
| | | Easy to edit | 2 | *Easy to fix program* |
| | | **Total Sub-theme** | **3** | |
| | | ***Total Theme*** | ***36*** | |

### *K.2.12 Treatment Group: Question 6 (Benefits of B#)*

| | | | **Preferred Environment Response Frequency** | | **Examples of Actual Responses** |
|---|---|---|---|---|---|
| **Theme** | **Sub-theme** | **Category** | **Delphi (*n* = 49)** | **B# (*n* = 28)** | |
| **Motivation** | Intrinsic Motivation | Confidence booster | 0 | 1 | *Without B#, I would have never been able to use Delphi* |
| | | Faster | 3 | 1 | *Faster to get program working* |
| | | **Total Sub-theme** | **3** | **2** | |
| | ***Total Theme*** | | ***3*** | ***2*** | |
| **Usability** | Ease of use | Easier | 3 | 2 | *Easy to use* |
| | | **Total Sub-theme** | **3** | **2** | |
| | Enhances comprehension | Export facility | 0 | 2 | *I could export ;programs to Delphi* |
| | | Generated code | 2 | 2 | *I could see Delphi code* |
| | | Reduced detail | 3 | 0 | *Unnecessary for me to write certain things* |
| | | Transfer to Delphi assisted | 12 | 11 | *I learnt some things from B# that I did not know* |
| | | Visual | 2 | 1 | *I like flowcharts* |
| | | **Total Sub-theme** | **19** | **16** | |
| | ***Total Theme*** | | ***22*** | ***18*** | |

### *K.2.13 Treatment Group: Question 6 (Hindrances of B#)*

| | | | **Preferred Environment Response Frequency** | | **Examples of Actual Responses** |
|---|---|---|---|---|---|
| **Theme** | **Sub-theme** | **Category** | **Delphi (*n* = 49)** | **B# (*n* = 28)** | |
| **Motivation** | Extrinsic Motivation | Extra environment | 5 | 3 | *B# was extra* |
| | | Not matching lectures | 4 | 4 | *Did not match the lectures* |
| | | Unfamiliar | 1 | 0 | *I do not know B#* |
| | | **Total Sub-theme** | **10** | **7** | |
| | Inaccessibility | Not at home | 2 | 0 | *Do not have B# at home* |
| | | **Total Sub-theme** | **2** | **0** | |
| | Intrinsic motivation | Encouraged laziness | 1 | 4 | *I feel like it makes me lazy* |
| | | Time consuming | 7 | 0 | *Took up too much time* |
| | | **Total Sub-theme** | **8** | **4** | |
| | | ***Total Theme*** | ***20*** | ***11*** | |
| **Usability** | Ease of use | Difficult | 0 | 1 | *Difficult to use fro fuctions/procedures* |
| | | **Total Sub-theme** | **0** | **1** | |
| | Restricted functionality | Functions | 2 | 0 | *Confusing when you have to delete a function just because it is in the wrong place* |
| | | Large flowcharts | 1 | 0 | *Confined space for complex and large flowcharts* |
| | | Restrictive format | 1 | 0 | *B# gave me little choice* |
| | | **Total Sub-theme** | **4** | **0** | |
| | | ***Total Theme*** | ***4*** | ***1*** | |

### *K.2.14 Treatment Group: Question 7 (Benefits of Delphi)*

| | | | **Preferred Environment Response Frequency** | | **Examples of Actual Responses** |
|---|---|---|---|---|---|
| **Theme** | **Sub-theme** | **Category** | **Delphi (*n* = 49)** | **B# (*n* = 28)** | |
| **Motivation** | Extrinsic Motivation | Matched lectures | 10 | 2 | *Like lectures* |
| | | **Total Sub-theme** | **10** | **2** | |
| | Intrinsic Motivation | Comfortable/confident with Delphi | 6 | 1 | *I feel more comfortable with Delphi* |
| | | **Total Sub-theme** | **6** | **1** | |
| | Inaccessibility of B# | Have Delphi at home | 1 | 0 | *No B# at home* |
| | | **Total Sub-theme** | **1** | **0** | |
| | | ***Total Theme*** | ***17*** | ***3*** | |
| **Usability** | Ease of use | Easier | 4 | 2 | *Easy to write programs* |
| | | Quicker | 2 | 0 | *Write programs fast* |
| | | **Total Sub-theme** | **6** | **2** | |
| | Enhances comprehension | From first principles | 2 | 1 | *Better to practice writing code from the beginning* |
| | | Error messages | 4 | 1 | *Shows where errors are* |
| | | **Total Sub-theme** | **6** | **2** | |
| | Restricted functionality of B# | More flexible | 1 | 1 | *More capabilities than B#* |
| | | Textual code | 3 | 2 | *Programs written in code* |
| | | Access code directly | 1 | 1 | *Can access code directly* |
| | | **Total Sub-theme** | **5** | **4** | |
| | | ***Total Theme*** | ***17*** | ***8*** | |

### *K.2.15 Treatment Group: Question 7 (Hindrances of Delphi)*

| | | | **Preferred Environment Response Frequency** | | **Examples of Actual Responses** |
|---|---|---|---|---|---|
| **Theme** | **Sub-theme** | **Category** | **Delphi (*n* = 49)** | **B# (*n* = 28)** | |
| **Motivation** | Extrinsic Motivation | Time consuming | 1 | 6 | *It is time consuming* |
| | | **Total Sub-theme** | **1** | **6** | |
| | Intrinsic Motivation | No reason | 0 | 1 | *Don't like Delphi* |
| | | **Total Sub-theme** | **0** | **1** | |
| | ***Total Theme*** | | ***1*** | ***7*** | |
| | Ease of use | More intensive concentration | 0 | 1 | *Need to concentrate* |
| | | Detail | 2 | 3 | *Too many things to remember* |
| | | Difficult | 1 | 1 | *I can't get through the errors* |
| | | **Total Sub-theme** | **3** | **5** | |
| | ***Total Theme*** | | ***3*** | ***5*** | |

### *K.2.16 Control Group: Question 7 (Benefits of Delphi)*

| Theme | Sub-theme | Category | Frequency (*n* = 116) | Examples of Actual Responses |
|---|---|---|---|---|
| **Usability** | Enhances comprehension | Analysis skills | 14 | *Helps with my analysis and programming skills* |
| | | See errors | 5 | *Can see where my errors are* |
| | | **Total Sub-theme** | **19** | |
| | ***Total Theme*** | | ***19*** | |
| **Motivation** | Extrinsic Motivation | Matched lectures | 4 | *Like the lectures* |
| | | **Total Sub-theme** | **4** | |
| | Intrinsic Motivation | Motivating | 4 | *Enjoyed it* |
| | | Quick | 1 | *Quick to write programs* |
| | | **Total Sub-theme** | **5** | |
| | ***Total Theme*** | | ***9*** | |

### *K.2.17 Control Group: Question 7 (Hindrances of Delphi)*

| Theme | Sub-theme | Category | Frequency (*n* = 116) | Examples of Actual Responses |
|---|---|---|---|---|
| **Motivation** | Time consuming | Time consuming | 8 | *Takes a long time* |
| | | **Total Sub-theme** | **8** | |
| | Intrinsic Motivation | Demotivating | 1 | *Didn't enjoy it* |
| | | **Total Sub-theme** | **1** | |
| | ***Total Theme*** | | ***9*** | |
| **Usability** | Ease of use | Difficult | 7 | *Difficult to use* |
| | | **Total Sub-theme** | **7** | |
| | Lots of detail | Detail | 4 | *Had to remember to put the semi-colons in the correct places* |
| | | **Total Sub-theme** | **4** | |
| | Enhances comprehension | Errors confusing | 4 | *The error messages didn't help me* |
| | | Lack of feedback | 2 | *Couldn't get programs to run* |
| | | **Total Sub-theme** | **6** | |
| | ***Total Theme*** | | ***17*** | |

# Appendix L

# Quantitative Analysis

## L.1 Testing for Differences between Pairs of Means

| Variable<br>($n_{\text{treatment}} = n_{\text{control}} = 59$) | Treatment Mean | Control Mean | p-value |
|---|---|---|---|
| Th1Q1 | 48% | 47% | 0.772 |
| Th1Q2 | 81% | 79% | 0.499 |
| Th1Q3 | 73% | 74% | 0.772 |
| Th1Tot | 65% | 64% | 0.862 |
| Pr1Q1 | 50% | 54% | 0.564 |
| Pr1Q2 | 48% | 41% | 0.319 |
| Pr1Tot | 48% | 46% | 0.685 |
| Th2MC | 61% | 60% | 0.823 |
| Th2Q1 | 55% | 49% | 0.203 |
| Th2Q2 | 25% | 30% | 0.316 |
| Th2Cmp | 41% | 40% | 0.894 |
| Th2Tot | 50% | 50% | 0.856 |
| Pr2Q1 | 50% | 53% | 0.555 |
| Pr2Q2 | 64% | 59% | 0.477 |
| Pr2Q3 | 60% | 49% | 0.041 * |
| Pr2Q4 | 60% | 49% | 0.059 |
| Pr2Tot | 59% | 51% | 0.098 |
| Class | 58% | 55% | 0.332 |
| ExMC | 73% | 66% | 0.028 * |
| ExQ1 | 25% | 25% | 0.966 |
| ExQ2 | 79% | 71% | 0.046 * |
| ExQ3 | 37% | 32% | 0.421 |
| ExQ4 | 55% | 49% | 0.326 |
| ExQ5 | 50% | 51% | 0.835 |
| ExCmpr | 66% | 60% | 0.044 * |
| ExCmp | 51% | 49% | 0.626 |
| ExTot | 60% | 55% | 0.161 |
| Final | 59% | 55% | 0.192 |

* significant where $p < 0.05$

*Table L.1: T-test Computed Values for Independent Variables: Full Complement of Participants*

| High Risk Stratum : Predicted Mark Range 41% - 50% | | | |
|---|---|---|---|
| Variable<br>$(n_{\text{treatment}} = n_{\text{control}} = 12)$ | Treatment Mean | Control Mean | p-value |
| Th1Q1 | 40% | 30% | 0.100 |
| Th1Q2 | 79% | 75% | 0.513 |
| Th1Q3 | 68% | 64% | 0.636 |
| Th1Tot | 59% | 53% | 0.133 |
| Pr1Q1 | 24% | 36% | 0.379 |
| Pr1Q2 | 19% | 21% | 0.830 |
| Pr1Tot | 21% | 27% | 0.498 |
| Th2MC | 55% | 48% | 0.326 |
| Th2Q1 | 45% | 35% | 0.140 |
| Th2Q2 | 18% | 11% | 0.353 |
| Th2Cmp | 33% | 24% | 0.135 |
| Th2Tot | 43% | 36% | 0.161 |
| Pr2Q1 | 54% | 30% | 0.015 * |
| Pr2Q2 | 57% | 32% | 0.063 |
| Pr2Q3 | 60% | 20% | 0.001 ** |
| Pr2Q4 | 56% | 16% | 0.002 ** |
| Pr2Tot | 57% | 22% | 0.001 ** |
| Class | 52% | 36% | 0.005 ** |
| ExMC | 61% | 57% | 0.526 |
| ExQ1 | 20% | 14% | 0.087 |
| ExQ2 | 71% | 53% | 0.034 * |
| ExQ3 | 29% | 8% | 0.006 ** |
| ExQ4 | 44% | 13% | 0.006 ** |
| ExQ5 | 42% | 33% | 0.375 |
| ExCmpr | 55% | 48% | 0.221 |
| ExCmp | 44% | 30% | 0.059 |
| ExTot | 50% | 40% | 0.091 |
| Final | 51% | 39% | 0.017 * |

* significant where $p < 0.05$
** significant where $p < 0.01$

*Table L.2: T-test Computed Values for Independent Variables: High Risk Stratum*

| **Medium Risk Stratum : Predicted Mark Range 51% - 65%** | | | |
|---|---|---|---|
| **Variable**<br>**($n_{treatment} = n_{control} = 32$)** | **Treatment Mean** | **Control Mean** | **p-value** |
| Th1Q1 | 46% | 48% | 0.579 |
| Th1Q2 | 83% | 81% | 0.754 |
| Th1Q3 | 74% | 75% | 0.809 |
| Th1Tot | 64% | 65% | 0.660 |
| Pr1Q1 | 49% | 51% | 0.848 |
| Pr1Q2 | 51% | 37% | 0.108 |
| Pr1Tot | 50% | 42% | 0.301 |
| Th2MC | 59% | 59% | 0.894 |
| Th2Q1 | 52% | 47% | 0.411 |
| Th2Q2 | 24% | 29% | 0.489 |
| Th2Cmp | 39% | 39% | 0.945 |
| Th2Tot | 49% | 48% | 0.926 |
| Pr2Q1 | 48% | 53% | 0.466 |
| Pr2Q2 | 58% | 59% | 0.910 |
| Pr2Q3 | 53% | 47% | 0.458 |
| Pr2Q4 | 54% | 49% | 0.437 |
| Pr2Tot | 53% | 50% | 0.614 |
| Class | 55% | 54% | 0.808 |
| ExMC | 73% | 64% | 0.036 * |
| ExQ1 | 21% | 21% | 0.826 |
| ExQ2 | 77% | 71% | 0.250 |
| ExQ3 | 34% | 32% | 0.843 |
| ExQ4 | 48% | 51% | 0.680 |
| ExQ5 | 50% | 51% | 0.878 |
| ExCmpr | 66% | 58% | 0.088 |
| ExCmp | 49% | 49% | 0.977 |
| ExTot | 59% | 54% | 0.318 |
| Final | 57% | 54% | 0.446 |

* significant where $p < 0.05$

*Table L.3: T-test Computed Values for Independent Variables: Medium Risk Stratum*

| **Low Risk Stratum : Predicted Mark Range 66% - 100%** | | | |
|---|---|---|---|
| **Variable** <br>**($n_{treatment} = n_{control} = 15$)** | **Treatment Mean** | **Control Mean** | **p-value** |
| Th1Q1 | 61% | 59% | 0.790 |
| Th1Q2 | 80% | 79% | 0.802 |
| Th1Q3 | 75% | 79% | 0.508 |
| Th1Tot | 70% | 71% | 0.893 |
| Pr1Q1 | 71% | 74% | 0.840 |
| Pr1Q2 | 63% | 66% | 0.824 |
| Pr1Tot | 66% | 69% | 0.807 |
| Th2MC | 69% | 72% | 0.320 |
| Th2Q1 | 68% | 65% | 0.713 |
| Th2Q2 | 33% | 49% | 0.186 |
| Th2Cmp | 52% | 57% | 0.479 |
| Th2Tot | 60% | 64% | 0.342 |
| Pr2Q1 | 50% | 71% | 0.034 * |
| Pr2Q2 | 83% | 83% | 1.000 |
| Pr2Q3 | 77% | 75% | 0.831 |
| Pr2Q4 | 74% | 75% | 0.956 |
| Pr2Tot | 73% | 76% | 0.747 |
| Class | 69% | 71% | 0.637 |
| ExMC | 82% | 77% | 0.284 |
| ExQ1 | 35% | 42% | 0.510 |
| ExQ2 | 89% | 84% | 0.534 |
| ExQ3 | 48% | 50% | 0.881 |
| ExQ4 | 81% | 71% | 0.433 |
| ExQ5 | 56% | 65% | 0.430 |
| ExCmpr | 76% | 72% | 0.441 |
| ExCmp | 62% | 65% | 0.714 |
| ExTot | 70% | 69% | 0.828 |
| Final | 70% | 70% | 0.968 |

* significant where $p < 0.05$

*Table L.4: T-test Computed Values for Independent Variables: Low Risk Stratum*

## L.2 Testing for Homogeneity of Proportions

| Variable ($n_{\text{treatment}}$ = 60 $n_{\text{control}}$ = 88) | Number of Passes | | $\chi^2$-test statistic | p-value |
|---|---|---|---|---|
| | Treatment Group | Control Group | | |
| Th1Q1 | 27 | 36 | 0.244 | 0.621 |
| Th1Q2 | 59 | 86 | 0.066 | 0.797 |
| Th1Q3 | 52 | 78 | 0.130 | 0.719 |
| Th1Tot | 56 | 79 | 0.565 | 0.453 |
| Pr1Q1 | 35 | 58 | 0.877 | 0.349 |
| Pr1Q2 | 29 | 39 | 0.232 | 0.630 |
| Pr1Tot | 33 | 46 | 0.107 | 0.744 |
| Th2MC | 47 | 68 | 0.023 | 0.879 |
| Th2Q1 | 39 | 52 | 0.526 | 0.468 |
| Th2Q2 | 13 | 24 | 0.598 | 0.439 |
| Th2Cmp | 20 | 28 | 0.040 | 0.847 |
| Th2Tot | 35 | 44 | 1.000 | 0.318 |
| Pr2Q1 | 39 | 55 | 0.096 | 0.756 |
| Pr2Q2 | 44 | 56 | 1.531 | 0.216 |
| Pr2Q3 | 41 | 43 | 5.510 | 0.019 * |
| Pr2Q4 | 40 | 44 | 4.038 | 0.045 * |
| Pr2Tot | 43 | 48 | 4.420 | 0.036 * |
| Class | 44 | 55 | 1.890 | 0.169 |
| ExMC | 55 | 70 | 3.99 | 0.046 * |
| ExQ1 | 8 | 13 | 0.061 | 0.805 |
| ExQ2 | 58 | 63 | 15.040 | 0.000 ** |
| ExQ3 | 26 | 29 | 1.646 | 0.200 |
| ExQ4 | 29 | 42 | 0.005 | 0.942 |
| ExQ5 | 30 | 42 | 0.074 | 0.786 |
| ExCmpr | 50 | 54 | 8.240 | 0.004 ** |
| ExCmp | 31 | 46 | 0.010 | 0.942 |
| ExTot | 40 | 54 | 0.430 | 0.511 |
| Final | 45 | 54 | 3.000 | 0.084 |

* significant where $p < 0.05$

** significant where $p < 0.01$

*Table L.5: $\chi^2$-test Computed Values: Full Complement of Participants*

| **High Risk Stratum : Predicted Mark Range 41% - 50%** | | | | |
|---|---|---|---|---|
| **Variable** | **Number of Passes** | | **$\chi^2$-test statistic** | **p-value** |
| **($n_{\text{treatment}}$ = 12<br>$n_{\text{control}}$ = 13)** | **Treatment Group** | **Control Group** | | |
| Th1Q1 | 4 | 2 | 1.102 | 0.294 |
| Th1Q2 | 12 | 12 | 0.962 | 0.327 |
| Th1Q3 | 11 | 10 | 1.009 | 0.315 |
| Th1Tot | 11 | 10 | 1.009 | 0.315 |
| Pr1Q1 | 4 | 5 | 0.071 | 0.790 |
| Pr1Q2 | 2 | 2 | 0.008 | 0.930 |
| Pr1Tot | 2 | 2 | 0.008 | 0.930 |
| Th2MC | 7 | 7 | 0.051 | 0.821 |
| Th2Q1 | 6 | 4 | 0.962 | 0.327 |
| Th2Q2 | 1 | 0 | 1.128 | 0.288 |
| Th2Cmp | 2 | 1 | 0.476 | 0.490 |
| Th2Tot | 4 | 2 | 1.102 | 0.294 |
| Pr2Q1 | 7 | 4 | 1.924 | 0.165 |
| Pr2Q2 | 8 | 4 | 3.222 | 0.073 |
| Pr2Q3 | 9 | 2 | 9.000 | 0.003 ** |
| Pr2Q4 | 8 | 1 | 9.420 | 0.002 ** |
| Pr2Tot | 9 | 2 | 9.000 | 0.009 ** |
| Class | 9 | 1 | 11.779 | 0.003 ** |
| ExMC | 9 | 7 | 1.212 | 0.271 |
| ExQ1 | 0 | 0 | 0.000 | 1.000 |
| ExQ2 | 11 | 5 | 7.667 | 0.006 ** |
| ExQ3 | 3 | 1 | 1.391 | 0.238 |
| ExQ4 | 3 | 1 | 1.391 | 0.238 |
| ExQ5 | 5 | 1 | 3.949 | 0.047 * |
| ExCmpr | 8 | 4 | 3.220 | 0.073 |
| ExCmp | 5 | 2 | 2.140 | 0.144 |
| ExTot | 5 | 3 | 0.991 | 0.320 |
| Final | 8 | 3 | 4.810 | 0.028 * |

* significant where $p < 0.05$

** significant where $p < 0.01$

*Table L.6: $\chi^2$-test Computed Values: High Risk Stratum*

| **Medium Risk Stratum : Predicted Mark Range 51% - 65%** | | | | |
|---|---|---|---|---|
| **Variable**<br>**($n_{\text{treatment}}$ = 32**<br>**$n_{\text{control}}$ = 60)** | **Number of Passes** | | **$\chi^2$-test statistic** | **p-value** |
| | **Treatment Group** | **Control Group** | | |
| Th1Q1 | 11 | 25 | 0.466 | 0.495 |
| Th1Q2 | 31 | 59 | 0.209 | 0.648 |
| Th1Q3 | 27 | 54 | 0.627 | 0.428 |
| Th1Tot | 30 | 54 | 0.370 | 0.543 |
| Pr1Q1 | 18 | 41 | 1.325 | 0.250 |
| Pr1Q2 | 15 | 27 | 0.030 | 0.864 |
| Pr1Tot | 17 | 31 | 0.018 | 0.894 |
| Th2MC | 24 | 47 | 0.132 | 0.717 |
| Th2Q1 | 20 | 36 | 0.055 | 0.815 |
| Th2Q2 | 7 | 13 | 0.001 | 0.982 |
| Th2Cmp | 8 | 17 | 0.120 | 0.732 |
| Th2Tot | 18 | 30 | 0.330 | 0.568 |
| Pr2Q1 | 21 | 38 | 0.048 | 0.827 |
| Pr2Q2 | 21 | 39 | 0.004 | 0.952 |
| Pr2Q3 | 18 | 28 | 0.767 | 0.381 |
| Pr2Q4 | 17 | 32 | 0.000 | 0.985 |
| Pr2Tot | 19 | 33 | 0.160 | 0.687 |
| Class | 20 | 41 | 0.320 | 0.573 |
| ExMC | 30 | 48 | 3.06 | 0.080 |
| ExQ1 | 2 | 5 | 0.129 | 0.720 |
| ExQ2 | 31 | 45 | 6.951 | 0.008 ** |
| ExQ3 | 12 | 19 | 0.318 | 0.573 |
| ExQ4 | 13 | 31 | 1.020 | 0.313 |
| ExQ5 | 14 | 30 | 0.327 | 0.568 |
| ExCmpr | 26 | 37 | 3.710 | 0.054 |
| ExCmp | 14 | 33 | 1.057 | 0.304 |
| ExTot | 20 | 37 | 0.010 | 0.938 |
| Final | 21 | 37 | 0.140 | 0.708 |

* significant where $p < 0.05$

** significant where $p < 0.01$

*Table L.7: $\chi^2$-test Computed Values: Medium Risk Stratum*

| Low Risk Stratum : Predicted Mark Range 66% - 100% | | | | |
|---|---|---|---|---|
| Variable ($n_{treatment}$ = 16 $n_{control}$ = 15) | Number of Passes | | $\chi^2$-test statistic | p-value |
| | Treatment Group | Control Group | | |
| Th1Q1 | 12 | 9 | 0.797 | 0.372 |
| Th1Q2 | 16 | 15 | 0.000 | 1.000 |
| Th1Q3 | 14 | 14 | 0.301 | 0.583 |
| Th1Tot | 15 | 15 | 0.969 | 0.325 |
| Pr1Q1 | 13 | 12 | 0.008 | 0.930 |
| Pr1Q2 | 12 | 10 | 0.261 | 0.610 |
| Pr1Tot | 14 | 13 | 0.005 | 0.945 |
| Th2MC | 16 | 14 | 1.102 | 0.294 |
| Th2Q1 | 13 | 12 | 0.008 | 0.930 |
| Th2Q2 | 5 | 11 | 5.490 | 0.019 * |
| Th2Cmp | 10 | 10 | 0.059 | 0.809 |
| Th2Tot | 13 | 12 | 0.008 | 0.930 |
| Pr2Q1 | 11 | 13 | 1.422 | 0.233 |
| Pr2Q2 | 15 | 13 | 0.444 | 0.505 |
| Pr2Q3 | 14 | 13 | 0.005 | 0.945 |
| Pr2Q4 | 15 | 11 | 2.386 | 0.123 |
| Pr2Tot | 15 | 13 | 0.444 | 0.505 |
| Class | 15 | 13 | 0.444 | 0.505 |
| ExMC | 16 | 15 | 0.000 | 1.000 |
| ExQ1 | 6 | 8 | 0.784 | 0.376 |
| ExQ2 | 16 | 13 | 2.280 | 0.131 |
| ExQ3 | 11 | 9 | 0.259 | 0.611 |
| ExQ4 | 13 | 10 | 0.860 | 0.354 |
| ExQ5 | 11 | 11 | 0.079 | 0.779 |
| ExCmpr | 16 | 13 | 2.280 | 0.131 |
| ExCmp | 12 | 11 | 0.011 | 0.916 |
| ExTot | 15 | 14 | 0.000 | 0.962 |
| Final | 16 | 14 | 1.102 | 0.294 |

* significant where $p < 0.05$

*Table L.8: $\chi^2$-test Computed Values: Low Risk Stratum*