The Application of the Cognitive Dimension Framework for Notations as an Instrument for the Usability Analysis of an Introductory Programming Tool

Charmain Cilliers
André Calitz
Jéan Greyling

1. Introduction

In recent years, South African tertiary education institutions have experienced increasing pressure from national and provincial government to improve student throughput rates. The consequent expectations of higher throughput rates in introductory programming courses have resulted in the identification and investigation of effective methods and strategies that assist students in overcoming difficulties experienced with computer programming.

A successful learning environment for introductory programming students has been described as satisfying the following constraints (Brusilovsky *et al.* 1994):

- the learning environment should support a notation that consists of a small, simple subset of the programming constructs generally available in a programming notation;
- the visual appearance of the program structure should enable an introductory programming student to comprehend the semantics of the programming constructs supported; and

- the environment should shield introductory programming students from misinterpretations and misunderstandings.

The programming environment most commonly used by students of introductory programming courses is categorised as being a commercial programming environment, examples being Delphi™ Enterprise and Visual Studio (De Raadt *et al.* 2002). Commercial programming environments typically support textual programming notations and not the alternative visual programming notations.

Further, commercial programming environments have been partly blamed for the fact that introductory programming courses are often perceived by students as being difficult (Hilburn 1993; Calloni & Bagert 1997; Warren 2000). Commercial programming environments have also typically been designed for use by experienced programmers who are developing large programs (Ziegler & Crews 1999). The debugging tools supported by the traditional programming environments are complex to initiate and use and are at times by choice avoided by more advanced programmers. These kinds of tools are thus inappropriate for use by students of introductory programming courses.

Iconic programming notations, a subset of visual programming languages, have been proposed as an alternative programming notation specifically aimed at introductory programming students (Burnett & Baker 1994; Calloni & Bagert 1994, 1995, 1997). An iconic programming notation is one in which each visual sentence is a spatial arrangement of icons, with each icon having a distinct meaning (Chang *et al.* 1994). Iconic programming notations attempt to simplify the programming task by reducing the level of precision and the incidence of manual typing typical of textual programming notations (Blackwell 1996).

In response to the challenge of increasing throughput in introductory programming courses, the Department of Computer Science and Information Systems (CS&IS) at the Nelson Mandela Metropolitan University (the former University of Port Elizabeth (UPE)) identified the need for the development of an experimental iconic programming notation, B# (Brown 2001; Thomas 2002; Cilliers *et al.* 2003; Yeh 2003; Greyling *et al.* 2004). B# was deliberately designed to be a short term visual programming notation providing initial technological support in the learning environment of an introductory programming course.

One factor that has a bearing on the success of B# as technological support in the learning environment of an introductory programming course is the level of usability supported. The usability of computer software is typically measured in terms of the way that users interact with the software package. A well known technique that is often used in the measurement of usability is Nielsen's ten usability heuristics (Nielsen 1994b). An alternative technique is that of the cognitive dimensions framework for notations (Green & Petre 1996). The latter technique consists of fourteen individual cognitive dimensions and is primarily aimed at measuring the usability of programming tools.

This paper reports on an investigation into the usability of B# within the context of B# being classified as a successful learning environment, as attributed earlier to Brusilovsky (1994). The criteria used in the usability analysis of B# are a set of usability criteria for programming tools known as the cognitive dimensions framework for notations. The cognitive dimensions framework is used to assess the usability of B# at two levels, namely at the software design and student programmer levels. The criteria used in the usability assessment deviates slightly from Nielsen's well documented and familiar usability principles as defined by the Heuristic Evaluation Usability Engineering method.

The paper proposes a mapping that illustrates the correspondence of the fourteen cognitive dimensions to Nielsen's ten heuristics. Thereafter, each cognitive dimension is individually discussed in terms of an assessment from a design perspective of the way in which B# supports it. A quantitative and qualitative data analysis of a cognitive dimension questionnaire administered to students of an introductory programming course using B# follows. The investigation concludes that B# provides an integrated visual environment that attempts to enhance the learning experience of the introductory programming course student by supporting the cognitive dimensions of notation framework for programming languages, with a view to ultimately increase the throughput in introductory programming courses.

2. Background

Transformations in the South African political and educational scenario over the past few years have resulted in increasing pressure from national and provincial government to improve student throughput rates at national tertiary institutions (Department of Education 2001). The problem of sustaining recommended satisfactory throughput rates in tertiary level courses is further compounded by the fact that currently larger numbers of under-prepared students are entering South African tertiary education institutions (Warren 2001; Monare 2004). The resulting 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. The prevalence 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 students to overcome difficulties associated with computer programming (Carbone *et al.* 2001).

Typical difficulties experienced by students in introductory programming courses include deficiencies in problem-solving strategies, misconceptions related to programming notation constructs and the use of traditional programming environments (Studer *et al.* 1995; Proulx *et al.* 1996; Deek 1999; AC Nielsen Research Services 2000; McCracken *et al.* 2001; Satratzemi *et al.* 2001). The resulting increased demands on lecturing and computing resources as a consequence of attempts to address these difficulties creates an urgent need for 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).

One approach to this problem is the modification of the introductory programming course teaching model (Wilson & Braun 1985; Austin 1987). This strategy incorporates the modification of course presentation techniques to support students at a technological level. One such type of technological support is a class of programming languages known as visual programming languages, of which iconic programming notations is a category.

International quantitative research in the use of an iconic programming language to encourage satisfactory performance achievement in introductory programming students has prompted related research at UPE (Calloni & Bagert 1994, 1995, 1997). An iconic programming notation, B#, has consequently been developed in the Department of CS&IS at UPE for use as the technological support in the learning environment of the university's introductory programming course.

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 supporting software and learning about the content contained therein (Rader *et al.* 1998). It has been observed by numerous researchers that implementation issues evident in the software provided by traditional commercial textual programming environments can distract students of introductory programming courses so that they do not comprehend the programming abstractions required for the correct implementation thereof (Reek 1995; Lidtke & Zhou 1998; Ziegler & Crews 1999; Proulx 2000; Warren 2000, 2001).

Although conventional textual programming environments concurrently display many programming constructs on the screen, they tend to under-determine the student by providing no guidance as to the textual symbols required to be entered, resulting in a large gap between the plan of the desired program solution and the supported programming notation (LaLiberte 1994; Wright & Cockburn 2000). The student is thus forced to provide precisely correct notation syntax before receiving any response to the solution plan and implementation thereof (Crews & Ziegler 1998). Further, the lack of sufficient visual feedback in the use of such programming tools makes the comprehension of notation semantics more difficult for a student (Satratzemi *et al.* 2001). Features of conventional programming development environments include complex hierarchical menu structures and intricate user interfaces. These properties are often experienced by students as distractions from the task of programming (Reek 1995; Lidtke & Zhou 1998; Ziegler & Crews 1999; Proulx 2000; Warren 2000, 2001).

Important factors to consider when making a choice of programming notation for use by students of introductory programming courses is how easily they will learn the chosen notation, the existence of any notation features that might interfere with the understanding of the fundamental programming concepts, and any notation features that ease the transformation of the beginner programmer to one who is competent (Dingle & Zander 2001).

Against the background described, the criteria applied in the analysis of the usability of B# as technological support in the learning environment of an introductory programming course are presented in the following section.

3. Usability Analysis Criteria

The usability analysis of software typically deals with the analysis of the way in which users interact with computer software. One familiar and experimentally verified technique is that of the application of the ten usability heuristics presented by Nielsen (1993; 1994a; 1994b). The application of Nielsen's usability heuristics are typically restricted to deal with user interfaces.

An evaluation technique proposed specifically for visual programming language notations and their associated development environments is that of the cognitive dimensions framework for notations (Green & Petre 1996). The cognitive dimensions framework focuses on the actions and procedures being performed by a programmer while using a programming notation and its associated environment.

Each of the two sets of usability criteria is individually overviewed in this section. The section concludes with a discussion on how the two techniques presented compare with one another.

3.1. Nielsen's Heuristics

Nielsen's usability heuristics, listed and defined in Table 1, deal with the user interfaces of software systems. The heuristics have, however, recently been applied to conventional textual programming environments in the context of the interaction of student programmers with a computer (Warren 2003).

The application of the technique to conventional textual programming environments provides insight into the problems that student programmers experience with traditional commercial textual programming notations and development environments.

Heuristic	**Description**
N1: Visibility of system status	The system should always keep users informed about what is going on through appropriate feedback within reasonable time.
N2: Match between system and the real world	The system should speak the users' language rather than system-oriented terms.
N3: User control and freedom	The system should clearly assist users in exiting from an undesired state.
N4: Consistency and standards	Users should not have to wonder whether different words, situations or actions mean the same thing.
N5: Error prevention	Even better than good error messages is a design which prevents a problem from occurring in the first place.
N6: Recognition rather than recall	The user should not have to remember information from one part of the dialogue to another.
N7: Flexibility and efficiency of use	The system should cater to both inexperienced and experienced users.
N8: Aesthetic and minimalist design	Every extra unit of information competes with the relevant units of information and diminishes their relative visibility.
N9: Help users recognise, diagnose and recover from errors	Error messages should be expressed in plain language (no codes), precisely indicate the problem, and constructively suggest a solution.
N10: Help and documentation	Even though it is better if the system can be used without documentation, it may be necessary to provide help and documentation. Any such information should be easy to search, focussed on the user's task, list concrete steps to be carried out, and not be too large.

Table 1: Nielsen's Ten Usability Heuristics

Warren (2003) concludes that conventional textual programming notations like C++, C#, Java, Delphi and Visual Basic set in their respective development environments are large intricate systems that fail to satisfy the majority of Nielsen's usability heuristics to some level. According to Warren, the specific usability heuristics not satisfied by conventional textual programming notations and their associated development environments are those listed in Table 1, with the exception of the heuristic of consistency and standards (N4) that is only partially satisfied. In contrast, Warren's recommendation in terms of Nielsen's heuristics is that the use of spreadsheet software followed by a scripting language such as JavaScript, together with an HTML editor with integrated browser capabilities as the technological support in the learning environment of an introductory programming course, adheres to the usability heuristics listed in Table 1 more closely.

An alternative technique for assessing the usability of a programming tool, namely the cognitive dimensions framework for notations, is discussed in the next section.

3.2. Cognitive Dimensions Framework for Notations

The cognitive dimensions framework for notations is an evaluation technique for interactive devices and non-interactive notations that has evolved over the past 15 years (Green 1989; Green & Petre 1996; Green & Blackwell 1998). This technique is task-specific and concentrates on the processes and activities being performed by programmers while using the software system rather than on the software deliverable itself.

In the case of the iconic programming notation B#, cognitive dimensions are the descriptions of the system-student relationship and are intended as a measurement instrument at a high level of abstraction. The cognitive dimensions framework for notations was initially intended for use during the early stages of the design process of a programming tool due to its structural assessment characteristic.

The definitions of the individual components of the cognitive dimensions framework appear in Table 2 (Green & Petre 1996; Blackwell & Green 2000). The framework has also been used in the design of questionnaires aimed at programmers to assess the usability of the programming tools used (Kadoda *et al.* 1999; Blackwell & Green 2000). The cognitive dimensions framework emphasises that programming tools include both a notation and a development environment, and that usability is a function of the two.

Cognitive Dimension	**Description**
CD1: Abstraction management	The system provides facilities for the definition of new concepts or constructs within the notation.
CD2: Closeness of mapping	The notation supported by the system closely resemblances the program solution being described.
CD3: Consistency	The system supports similarity in different parts of the notation that have like meanings.
CD4: Diffuseness	The system supports brevity in the description of solutions within the provided notation.
CD5: Error-proneness	The system permits the making of unnecessary mistakes which are a hindrance to the programming task.
CD6: Hard mental operations	The system has features that require a large amount of mental effort to use effectively.
CD7: Hidden dependencies	The system enforces consistency and a high level of visibility between closely related components of the notation.
CD8: Premature commitment	The system places a restriction on the ordering of subtasks within the programming task.
CD9: Progressive evaluation	The system supports the execution of partially completed versions of the solution.
CD10: Role-expressiveness	The system supports the easy identification of constructs within a program solution.
CD11: Secondary notation	The system supports annotations that convey meaning to the student.
CD12: Viscosity	The system supports the simplification of modifications to existing programs.
CD13: Visibility and juxtaposibility	The system supports the easy location of the various parts of the notation, and if corresponding representations are required to be compared, then the student is able to view them at the same time, preferably alongside one another.
CD14: Provisionality	The system supports the interactive modification of a solution and permits the determination of the effect of programming decisions.

Table 2: Cognitive Dimensions Framework Components

Since both Nielsen's heuristics and the cognitive dimensions for notations framework are usability analysis techniques used in the assessment of programming tools, a mapping of the 14 cognitive dimensions to Nielsen's 10 usability heuristics is discussed next.

3.3. Equivalence of Usability Analysis Criteria

Both Nielsen's usability heuristics and the cognitive dimensions framework have been used by various researchers as measurement instruments of the usability of programming tools (Kadoda *et al.* 1999; Blackwell & Green 2000; Warren 2003). Table 3 illustrates the correspondence and overlap of the two sets of usability analysis criteria.

Nielsen's Usability Heuristic	**Cognitive Dimension**
N1: Visibility of system status	CD7: Hidden dependencies CD9: Progressive evaluation CD11: Secondary notation CD13: Visibility and juxtaposibility CD14: Provisionality
N2: Match between system and the real world	CD2: Closeness of mapping
N3: User control and freedom	CD1: Abstraction management CD6: Hard mental operations CD8: Premature commitment CD9: Progressive evaluation CD11: Secondary notation CD12: Viscosity
N4: Consistency and standards	CD3: Consistency
N5: Error prevention	CD5: Error-proneness
N6: Recognition rather than recall	CD10: Role-expressiveness
N7: Flexibility and efficiency of use	CD1: Abstraction management CD6: Hard mental operations CD8: Premature commitment CD12: Viscosity
N8: Aesthetic and minimalist design	CD4: Diffuseness
N9: Help users recognise, diagnose and recover from errors	CD5: Error-proneness
N10: Help and documentation	CD5: Error-proneness

Table 3: Correspondence between Nielsen's Usability Heuristics and the Cognitive Dimensions Framework

All 14 cognitive dimensions (CD1 – CD14) can be equated to Nielsen's 10 usability heuristics (N1 – N10) based on the definitions of each technique's components as discussed previously. In many instances, the same cognitive dimension is mapped to multiple heuristics, and the same heuristic is mapped to multiple cognitive dimensions.

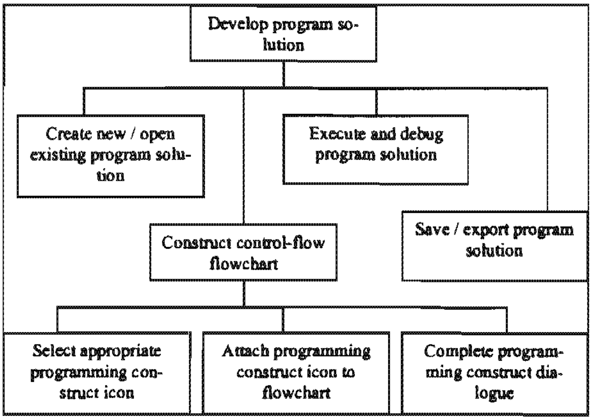
As mentioned previously, for the purposes of the study reported on in this paper, the usability of B# is evaluated according to the cognitive dimensions framework. Since this measurement instrument is task-specific, the tasks relevant to creating a program solution in B# are thus the focus of the following section.

4. Development of a Program Solution in B#

In order for a program solution to be developed using B#, the student is first required to either locate and identify an existing program solution, or provide identification for a new program solution. Thereafter, the B# program is constructed in the form of a control-flow solution that closely resembles that of a flowchart. Once this has been completed, the student may execute and debug the program solution and save it for future use. An overview of the task model for developing a program solution in B# is shown in Figure 1.

Each B# program solution is constructed in the form of a flowchart of icons, with each icon representing a distinct programming construct. The flowchart is a top-down single-sequence structure of icons connected by lines, forming a box-and-line graph which is typical of visual programming languages (Green & Petre 1996; Materson & Meyer 2001). The student selects an appropriate icon from the icon palette and drags-and-drops it in the correct position on the flowchart. An example of a program solution in the B# programming notation and associated development environment is illustrated by Figure 2. The flowchart representation of the program solution appears in the left hand pane of the window.

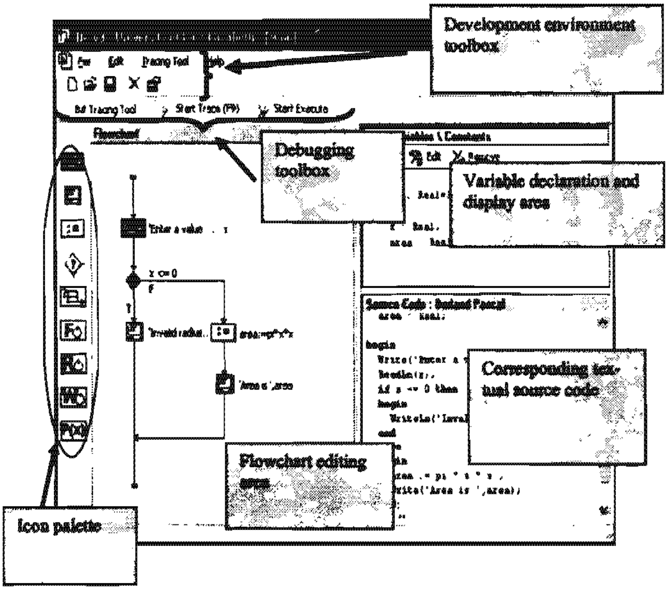
On attachment of the icon to the flowchart, a dialogue box is opened to guide the student in the specification of the properties required by the particular programming construct being manipulated. Figure 3 illustrates an example of a programming construct dialogue. The dialogue box in Figure 3 is that applicable to the counter iteration programming construct, which typically corresponds to a **FOR** textual programming statement. The student is required and guided to correctly complete the dialogue before an icon can be successfully attached to the flowchart representation of the programming construct.



Develop program solution
Create new / open existing program solution
Execute and debug program solution
Construct control-flow flowchart
Save / export program solution
Select appropriate programming construct icon
Attach programming construct icon to flowchart
Complete programming construct dialogue

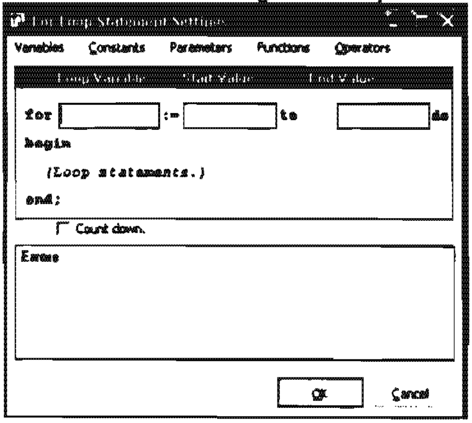
Figure 1: Task Model for the Development of a Program Solution in B#

Icons can be edited, repositioned and removed from the flowchart representation of the program solution. During the construction of a flowchart program solution, B# automatically and immediately displays the correct textual counterpart for the program solution. An example of this display is evident in the bottom right hand pane of the window illustrated in Figure 2. Figure 1 illustrates that once a B# program solution has been constructed, the student may test and debug it. Execution of program solutions is supported in two ways. The first technique is one whereby only the output from the program solution is displayed. The second technique permits the student to control the speed of the execution of the program solution. In this way, the student can trace the execution of the program solution, programming construct by programming construct.



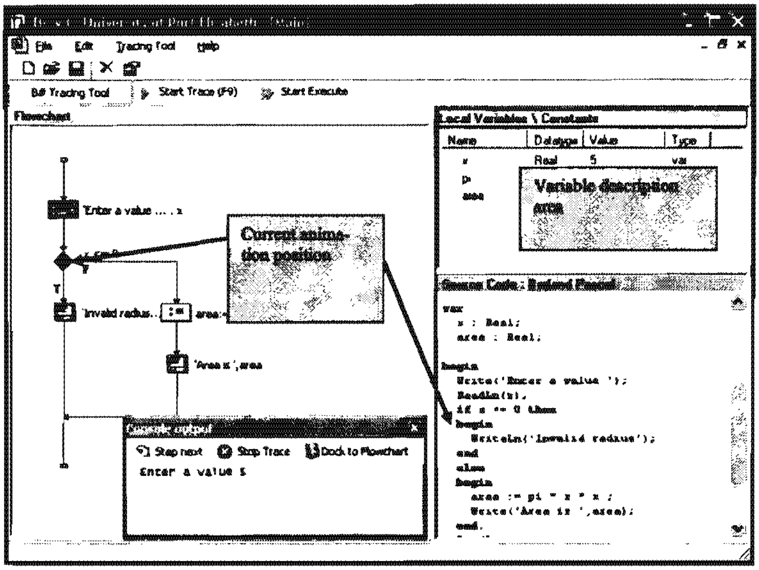
```
Source Code : Borland Pascal
  area : Real;

begin
  Write('Enter a v
  Readln(x);
  if x <= 0 then
  begin
    Writeln('Inval
  end
    area := pi * x * x;
    Write('Area is ',area);
```

Figure 2: Sample B# Program Solution



```
for [          ] := [          ] to [          ] do
begin
  (Loop statements.)
end;
```

Figure 3: Dialogue Box for Counter Iteration Programming Construct



```
Enter a value 5
```

```
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 4: B# Tracing and Debugging Feature

As the student controls the tracing process, both the flowchart and textual representations of the program solution are simultaneously animated. An example of the animation appears in Figure 4 in the form of blue highlighting in each of the program solution representations. Any changes in variable values are highlighted in the variable description area to focus the student's attention on them. This area appears in the top right hand pane of the window illustrated in Figure 4.

The following section analyses the usability of B# as an introductory programming tool in terms of the cognitive dimensions overviewed in Section 3, and the task model described in this section.

5. Usability Analysis of B#

The cognitive dimensions framework, as described previously, is an evaluation technique that is task-specific and intended for use with respect to the design process of programming tools. The framework can also be used to determine the usability of a programming tool from the student programmers' point of view. The following sections report on the usability analysis of B# as a programming tool for student programmers in an introductory programming course at UPE in terms of each of these approaches.

5.1. Design Perspective

In terms of its design, B# supports the cognitive dimension of abstraction management (CD1) by allowing a student to define new programming operations using the notation provided, specifically permitting the definition and use of subroutines. The cognitive dimension of closeness of mapping (CD2) is enforced by a notation that closely resembles the solution being described. B# adopts the use of a visual flowchart which closely mirrors the control-flow or procedural paradigm initially required by the students in the context of UPE's introductory programming course. An example of the visual flowchart notation is illustrated in Figure 2 (shown in section 4).

B# maximises support for the cognitive dimension of diffuseness (CD4) by providing a small number of powerful, non-overlapping programming constructs. The number of programming constructs supported is minimised since more constructs imply more notational syntax and unnecessary complexity which could result in a student programmer experiencing confusion. The programming constructs supported by the notation of B# are shown in Table 4.

Construct	Icon	Construct	Icon
Assignment	:=	Input	⌨
Simple conditional	◇	Output	▭
Multiple conditional	▤	Post-test conditional iteration	R
Counter iteration	F	Return to calling function	↵
Pre-test conditional iteration	W	Procedure call	P()

Table 4: Programming Constructs supported by B#'s notation

Also shown in Table 4 is the fact that the cognitive dimension of consistency (CD3) is supported by the notation of B#. All looping or iteration constructs (counter iteration, pre-test conditional iteration and post-test conditional iteration) have similar metaphorical images, yet remain visually distinct.

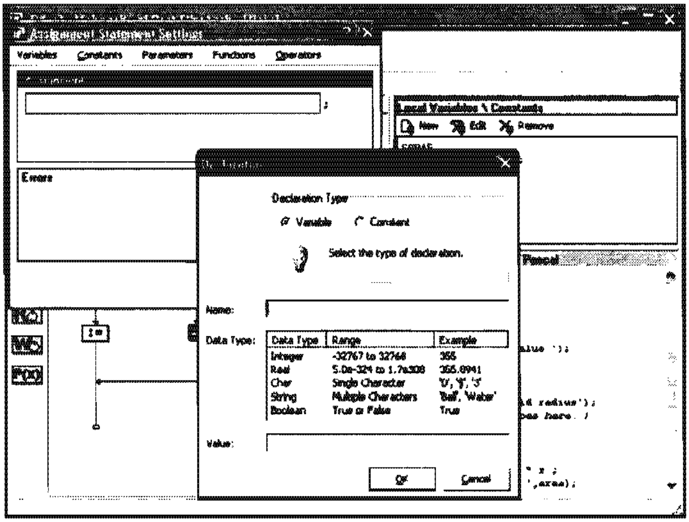
B# minimises the incidence of unnecessary errors which hinder the programming task by means of its context sensitive views. This property is evident specifically in the case of the customised dialogues implemented for programming constructs, an example of which is illustrated by Figure 3 in the previous section. Further evidence of the prevention of unnecessary errors is the fact that B# hides from the student all mundane syntactical issues (for example, the correct positioning and inclusion of semi-colons within the corresponding textual language). In these ways the cognitive dimension of error proneness (CD5) is minimised.

During the prototype development of B#, it was observed that the feature of providing the facility for the construction of student defined programming operations required the most mental effort to comprehend and successfully implement. Consequently, it was determined that B# exhibited the cognitive dimension of hard mental operations (CD6). Support for the identified offending feature was thus modified in the most recent version of B# (Yeh 2003) in order to minimise support for the cognitive dimension of hard mental operations.

B# maximises the visibility of the inter-dependence of components by the concurrent display of the flowchart program solution and the corresponding syntactically correct textual version of the program solution. This feature is illustrated in Figure 2 in the previous section. In this way, B# minimises incidences of the cognitive dimension of hidden dependencies (CD7).

An example of the manner in which B# minimises the incidence of the cognitive dimension of premature commitment (CD8) is the way in which variables required by the program solution only need to be defined as they are required. This enhances the student control and freedom of the programming environment. Figure 5 illustrates that the declaration of a variable is required only after the selection and attachment of the assignment programming construct has been initiated by the student. The student is thus not required to declare variables prior to deciding that the assignment operation is the programming construct required to be implemented. B# does, however, provide for a facility whereby more experienced students may pre-declare variables should they so wish. In this way, B# refrains from enforcing a particular order of performing programming tasks during the development of a program solution.

Any B# program solution representation (both flowchart and corresponding textual) is always syntactically correct and the student may consequently execute and debug a program solution using the tracing facility at any point during program solution construction. This feature promotes the cognitive dimension of progressive evaluation (CD9) and is illustrated in Figure 4 of the previous section.



Select the type of declaration.

Name:

Data Type:

Data Type	Range	Example
Integer	-32767 to 32768	356
Real	5.0e-324 to 1.7e308	356.0941
Char	Single Character	'0', '$', '5'
String	Multiple Characters	'Ball', 'Water'
Boolean	True or False	True

Value:

Figure 5: Declaration of variable required only when necessary

The cognitive dimension of role expressiveness (CD10) is supported in B# by means of the use of distinct metaphorical icons for each programming construct found in the flowchart representation for a program solution. Support for the cognitive dimension of role expressiveness is illustrated by the easy visual identification of distinct programming construct images displayed in the flowchart representation of the program solution in Figure 2 (shown in section 4).

Examples of the manner in which B# exhibits the cognitive dimension of secondary notation (CD11) is by means of the vertical and horizontal arrangement of programming construct icons in relation to one another in the flowchart representation of a program solution. A sample of this feature is illustrated in Figure 2.

Vertical arrangement of icons in the flowchart representation of a program solution is an indication of the flow of control, whereas horizontal arrangement is an indication of mutual exclusive selection. B# supports the cognitive dimension of secondary notation in the textual representation of a program solution by means of signalling (or code highlighting), also illustrated in Figure 2.

B# further treats nested programming constructs as a group and the student, by means of a single action, can, for example, successfully and easily reposition a group of programming constructs as a single unit within the flowchart representation of a program solution. In this way B# provides for the support of the simplification of modifications to existing program solutions thereby supporting the cognitive dimension of viscosity (CD12).

Support for the cognitive dimension of visibility and juxtaposibility (CD13) is supported by B# in that the development environment provided by B# minimises the effort required on the part of the student to search for related information. This is evident in the way in which B# displays corresponding representations of a program solution alongside one other (Figure 2). B# also ensures that the student is able to manage the screen display with the minimal amount of windows. Further, minimal effort on the part of the student is required to determine the current status of the system.

B# supports the displaying of the current status of the system by means of context sensitive views, one example being where the user is confronted with a blank display and is thereby encouraged to either create a new program solution or open an existing one. Figure 6 illustrates this system status.

Other examples exhibited by B# in support of the cognitive dimension of visibility and juxtaposibility is the display of an appropriate dialogue for any specific programming construct being added to the flowchart (Figure 3 in section 4) as well as the tracing facility which provides simultaneous animation of corresponding flowchart and textual program solutions (Figure 4 in section 4).

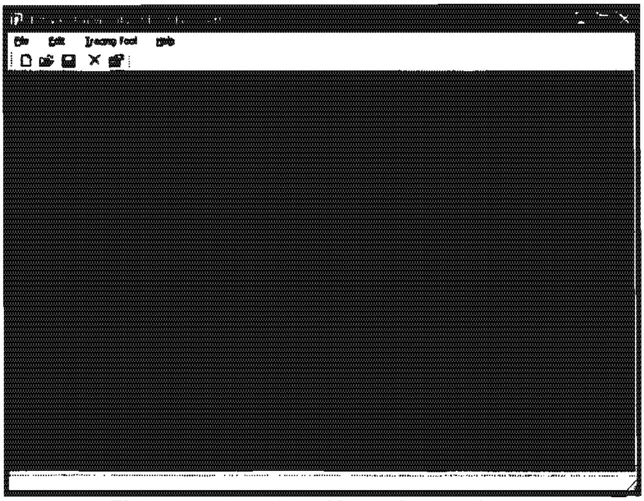


Figure 6: Illustration of Initial System Status

B# provides support for the final cognitive dimension of provisionality (CD14) by facilitating the easy repositioning of nested groups of programming constructs represented by icons as well as by means of the tracing facility (Figure 4).

In addition to evidence of B# support for the 14 cognitive dimensions at a design level, an evaluation of the system by means of a student questionnaire derived from the set of cognitive dimensions was conducted, the results of the analysis thereof being discussed in the following section.

5.2. Student Programmer Perspective

A questionnaire (Appendix A) customised and adapted from the generic questionnaire proposed by Blackwell & Green (2000) was administered to students using B# as technological support in an introductory programming learning environment. The aim of the survey was to collect data for the examination and testing of the following hypothesis for significance at the 95% percentile ($\alpha = 0.05$) (Berenson & Levine 1999):

H_0: *An equal number of positive and negative responses for each cognitive dimension are received from the student assessment of the usability of B#.*

H_1: *An unequal number of positive and negative responses for each cognitive dimension are received from the student assessment of the usability of B#.*

The test statistic applicable to the quantitative data analysis is a computed proportion based on the number of positive responses observed for each cognitive dimension on administering the questionnaire. The statistical technique thus appropriate to the analysis of the data collected is the χ^2-test for the homogeneity of proportions using a contingency table to test the equality of the number of positive and negative responses for each of the 14 cognitive dimensions as defined in section 3. STATISTICA (StatSoft Inc. 2001) is the data analysis tool used in the computations required for the χ^2-test.

The cognitive dimensions questionnaire was administered to a group of 18 (of a possible 25) introductory programming students at UPE during 2004. The course was one of the smaller introductory programming courses and was the only course where B# was extensively and exclusively used. The subjects of the study had been using B# as technological support in the learning environment of the introductory programming course for a period of 10 weeks. Each weekly exposure to B# consisted of a single session of at least 75 minutes.

The portion of the questionnaire that is dedicated to the main notation of B# is designed to provide at least one numbered item relevant to each of the 14 cognitive dimensions. The purpose of each numbered item is to provide the opportunity for the respondents to recognise the existence of features in B# that are relevant to each cognitive dimension in terms of the programming task performed by the students. Table 5 maps each of the items appearing as numbered questions in the questionnaire in Appendix A to the appropriate cognitive dimension.

The cognitive dimension of abstraction management (CD1) is not included due to the fact that at the time of the administering of the questionnaire, the curriculum being followed by the introductory programming students had not yet progressed sufficiently to the point where the students were able to effectively assess the particular features supported by B# that are relevant to this particular cognitive dimension.

Cognitive Dimension	Question number	Positive response values	Negative response values
CD2: Closeness of mapping	7	3, 4, 5	1, 2
CD3: Consistency	13	3, 4, 5	1, 2
CD4: Diffuseness	4	3, 4, 5	1, 2
CD5: Error-proneness	6	1, 2, 3	4, 5
CD6: Hard mental operations	5	1, 2, 3	4, 5

CD7: Hidden dependencies	9	3, 4, 5	1, 2
CD8: Premature commitment	12	1, 2, 3	4, 5
CD9: Progressive evaluation	10	3, 4, 5	1, 2
CD10: Role-expressiveness	8	3, 4, 5	1, 2
CD11: Secondary notation	14	3, 4, 5	1, 2
CD12: Viscosity	3	3, 4, 5	1, 2
CD13: Visibility and juxtaposibility	1 and 2	3, 4, 5	1, 2
CD14: Provisionality	11	3, 4, 5	1, 2

Table 5: Correspondence of Questionnaire Questions to Cognitive Dimensions

Results of the statistical analysis of the responses to each of the numbered items in the questionnaire are presented in Table 6. The null hypothesis is clearly rejected for 10 of the 13 cognitive dimensions at the 99% level of confidence, and rejected for the remaining 3 cognitive dimensions at the 95% level of confidence. By inspection, it can be interpreted that in the case of the majority of the cognitive dimensions, a significantly greater proportion of positive responses were observed. Only in the cases of the cognitive dimensions of error-proneness (CD5), hard mental operations (CD6) and premature commitment (CD8) were a significantly greater proportion of negative responses observed.

The questionnaire administered to the students also attempts to solicit the respondents' characterisation of the type of activity for which B# is used in an introductory programming course. Qualitative analysis through the technique of thematic analysis was applied to the data collected for the purpose of this characterisation (Ely *et al.* 1995; Ely *et al.* 1999; Dee Medley 2001).

Cognitive Dimension	Number of Positive observations	χ^2-test statistic	p-value
CD2: Closeness of mapping	13	20.350	0.000**
CD3: Consistency	11	9.750	0.002**
CD4: Diffuseness	12	4.000	0.046*
CD5: Error-proneness	5	5.460	0.019*
CD6: Hard mental operations	6	4.000	0.046*
CD7: Hidden dependencies	13	13.830	0.000**
CD8: Premature commitment	5	7.110	0.008**
CD9: Progressive evaluation	16	28.800	0.000**

CD10: Role-expressiveness	17	28.440	0.000**
CD11: Secondary notation	14	16.200	0.000**
CD12: Viscosity	16	21.780	0.000**
CD13: Visibility and juxtaposibility	17 and 12	28.440 and 11.690	0.000** and 0.001**
CD14: Provisionality	16	25.080	0.000**

*α = 0.05 **α = 0.01

Table 6: Results of Usability Evaluation of Main Notation of B# with respect to Cognitive Dimensions of Notations

Thematic analysis of the responses to this question determined that students interpreted the main task/activity for which B# is used to be the creation of program solutions for problems in the form of a flowchart. Typical responses to this question are:

"Solving problems similar to flowcharts"
"Creation of flowcharts"
"Creating programs"

The questionnaire further attempts to determine the existence of any problems related to the usability of B# that are not specifically addressed by the cognitive dimensions. Analysis of the responses determined that the task that took the most time in B# was the reorganisation and restructuring of a B# program solution. The tasks that occupied the least amount of time were searching for the correct programming icon to use and the unproductive experimentation of programming construct icons within program solutions.

Thematic analysis of the responses also indicated that additional assistance was required to be provided by B#. Typical responses indicative of this are:

"By putting a Help function"
"Have back going arrows like in flowcharts. Not only forward arrows"
"Being able to indicate location of the error within the program"

As a result of the quantitative and qualitative analysis of the student programmer assessment of the usability of B#, it is noticeable that additional system support for on-line help is required.

6. Discussion and Conclusions

The paper reported on an investigation into the usability of B#, an iconic programming notation and development environment developed by the Department of CS&IS at UPE, as an appropriate technological support tool in the learning environment of an introductory programming course. The results of the analysis are presented on two levels, namely in terms of the design of B# and with respect to the experience of students using B# to construct program solutions.

At the design level, B# is shown to positively provide support for all 14 cognitive dimensions. Application of a mapping between the technique of the cognitive dimensions framework for notations and Nielsen's heuristics implies that in satisfying the 14 cognitive dimensions, B# satisfies the latter technique of Nielsen's 10 heuristics in their entirety.

In order to confirm the level of support for the cognitive dimensions, a survey was administered to introductory programming students using B# and the responses quantitatively and qualitatively analysed.

The analysis of the responses observed by means of the survey determined that 77% of the cognitive dimensions are positively supported by B#. The analysis, however, provides evidence of usability problems in B# that are not specifically addressed by the cognitive dimensions. These identified usability problems have a relationship with the remaining 23% of cognitive dimensions not directly positively supported by B# in terms of the student assessments.

It is interesting to note that all of the cognitive dimensions assessed by student programmers as not being positively supported by B# are without exception the only questionnaire items that were negatively phrased.

In terms of the cognitive dimensions for notations assessment of the usability of B#, the experimental programming notation and environment can be classified as a successful learning environment for introductory programming students for the following reasons:

- B# supports a notation that consists of a small, simple subset of the programming constructs generally available in a programming notation. This property is evident in B# maximising support for the cognitive dimension of diffuseness (CD4).
- The visual appearance of B#'s program structure enables an introductory programming student to comprehend the semantics of the programming constructs supported. This property is evident in B# maximising support for the cognitive dimension of closeness of mapping (CD2).
- B# attempts to shield introductory programming students from misinterpretations and misunderstandings. This property is supported in B# by maximising support for the cognitive dimension of consistency (CD3). In spite of this observed support, there seems to be evidence that the property is insufficiently supported in that support for the cognitive dimension of error-proneness (CD5) requires further consideration.

Although the paper argues that in terms of support for the cognitive dimensions framework for programming languages, B# provides an integrated visual environment that attempts to enhance the learning experience of the introductory programming course student, it is clear that further research in the usability of B# is necessary.

7. Acknowledgements

The material contained in this paper 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 authors and therefore the NRF does not accept any liability in regard thereto.

References

AC Nielsen Research Services 2000. *Employer Satisfaction with Graduate Skills*.

Austin, HS 1987. Predictors of Pascal Programming Achievement for Community College Students. *ACM SIGCSE Bulletin* 19,1:161 - 164.

Berenson, ML & DM Levine 1999: *Basic Business Statistics: Concepts and Applications*. 7th Edition, Prentice-Hall International, Inc.

Blackwell, AF 1996. Metacognitive Theories of Visual Programming: What do we think we are doing? *Proceedings of IEEE Symposium on Visual Languages*: 240 - 246.

Blackwell, AF & TRG Green 2000. A Cognitive Dimensions questionnaire optimised for users. AF Blackwell & E Bilotta (eds.) *Proceedings of 12th Annual Workshop of the Psychology of Programming Interest Group*, Corigliano Calaboro, Cosenza, Italy: 137 - 154.

Boyle, R, J Carter & M Clark 2002. What Makes Them Succeed? Entry, Progression and Graduation in Computer Science. *Journal of Further and Higher Education* 26,1:3 - 18.

Brown, D 2001. B#: *A Visual Programming Tool*. Honours Treatise. Department of Computer Science and Information Systems, University of Port Elizabeth. Port Elizabeth, South Africa.

Brusilovsky, P, A Kouchnirenko, P Miller & I Tomek 1994. Teaching Programming to Novices: A Review of Approaches and Tools. *Proceedings of ED-MEDIA 94-World Conference on Educational Multimedia and Hypermedia*, Vancouver, British Columbia, Canada.

Burnett, MM & MJ Baker 1994. A Classification System for Visual Programming Languages. *Technical Report 93-60-14*. Department of Computer Science, Oregon State University, Corvallis.

Calloni, B A & D J Bagert 1994. Iconic Programming in BACCII© vs. Textual Programming: Which is a Better Learning Environment? *ACM SIGCSE Bulletin* 26,1:188 - 192.

Calloni, B A & D J Bagert 1995. Iconic Programming for Teaching the First Year Programming Sequence http://fie.engrng.pitt.edu/fie95/2a5/2a53/2a53.htm. Accessed on 26 September 2002.

Calloni, B A & D J Bagert 1997. Iconic Programming Proves Effective for Teaching First Year Programming Sequence. *ACM SIGCSE Bulletin* 28,1:262 - 266.

Carbone, A, J Hurst, I Mitchell & D Gunctone 2001. Characteristics of Programming Exercises that lead to Poor Learning Tendencies: Part II. *ACM SIGCSE Bulletin* 33,3:93 - 96.

Chang, S K, G Polese, S Orefice & M Tucci 1994. A Methodology and Interactive Environment for Iconic Language Design. *International Journal of Human-Computer Studies* 41,5:683 - 716.

Cilliers, CB, JH Greyling & AP Calitz 2003. The Development and Evaluation of Introductory Programming Tools. *Proceedings of 3rd International Conference on Science Maths Technology Education*, East London, South Africa.

Crews, T & U Ziegler 1998. The Flowchart Interpreter for Introductory Programming Courses. *Proceedings of 28th Annual Frontiers in Education Conference* 1: 307 - 312.

De Raadt, M, R Watson & M Toleman 2002. Language Trends in Introductory Programming Courses. *Proceedings of Informing Science + IT Education Joint Conference (IⁿSITE)*: 329 - 337.

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.

Department of Education 2001. National Plan for Higher Education. South African Department of Education http://education.pwv.gov.za/DoE_Sites/Higher_Education/HE_Plan/section_2.htm . Accessed on 8 March 2004,

Dingle, A & C Zander 2001. Assessing the Ripple Effect of CS1 Language Choice. *Journal of Computing in Small Colleges* 16,2:85 - 93.

Ely, M, M Anzul, T Friedman, D Garner & A MacCormack- Steinmetz 1995: *Doing Qualitative Research: Circles within Circles*. Falmer Press.

Ely, M, R Vinz, M Downing & M Anzul 1999: *On Writing Qualitative Research: Living by Words*. Falmer Press.

Green, T R G 1989. The Cognitive Dimensions of Notations. *People and Computers V*: 443 - 460. A Sutcliffe & L Macaulay (eds.). Cambridge: Cambridge University Press.

Green, TRG & AF Blackwell 1998. Cognitive Dimensions of Information Artefacts: A Tutorial http://www.ndirect.co.uk/~thomas.green/workStuff/Papers/ Accessed on 2 September 2002.

Green, TRG & M Petre 1996. Usability Analysis of Visual Progamming Environments: A 'Cognitive Dimensions' Framework. *Journal of Visual Languages and Computing* 7:131 - 174.

Greyling, JH, CB Cilliers & AP Calitz 2004. The Development and Assessment of an Iconic Programming Tool. Submitted for Publication in the Special eLearning Issue of the Scholarly Journal *Alternation*: Knowledge Management and Technology in Education.

Hilburn, TB 1993. A Top-Down Approach to Teaching an Introductory Computer Science Course. *ACM SIGCSE Bulletin* 25,1:58 - 62.

Kadoda, G, R Stone & D Diaper 1999. Desirable Features of Educational Theorem Provers - A Cognitive Dimensions Viewpoint. TRG Green, R Abdullah & P Brna (eds.) *Proceedings of 11th Annual Workshop of the Psychology of Programming Interest Group*: 18 - 23.

LaLiberte, D 1994. Visual Languages

http://www.hypernews.org/~liberte/computing/visual.html Accessed on 8 August 2002.

Lidtke, D K & H H Zhou 1998. A Top-Down Collaborative Teaching Approach to Introductory Courses in Computer Sciences. *ACM SIGCSE Bulletin* 30,3:291.

Lister, R & J Leaney 2003. Introductory Programming, Criterion-Referencing, and Bloom. *Proceedings of 34th SIGCSE Technical Symposium on Computer Science Education*: 143 - 147.

Materson, TF & RM Meyer 2001. SIVIL: A True Visual Programming Language for Students. *Journal of Computing in Small Colleges* 16,4:74 - 86.

McCracken, M, V Almstrum, D Diaz, M Guzdial, D Hagan, YB-D Kolikant, C Laxer, L Thomas, I Utting & T Wilusz 2001. A Multi-National, Multi-Institutional Study of Assessment of Programming Skills of First-Year CS Students. *ACM SIGCSE Bulletin*, 33,4:125 - 140.

Monare, M 2004. Now Matric Exams Face Official Probe. *Sunday Times* 4 January 2004.

Nielsen, J 1993: *Usability Engineering*. Boston, MA: Academic Press.

Nielsen, J 1994a. Enhancing the Explanatory Power of Usability Heuristics. *Proceedings of ACM CHI'94 Conference*: 152 - 158.

Nielsen, J 1994b: *Heuristic Evaluation*. J Nielsen & RL Mack (eds.) Usability Inspection Methods: 25 - 62.

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, R Rasala & H Fell 1996. Foundations of Computer Science: What are they and how do we teach them? *ACM SIGCSE Bulletin*, 28,S1:42 - 48.

Rader, C, G Cherry, C Brand, A Repenning & C Lewis 1998. Designing Mixed Textual and Iconic Programming Languages for Novice Users. *Proceedings of IEEE Symposium on Visual Languages*: 187 - 194.

Reek, MM 1995. A Top-Down Approach to Teaching Programming. *ACM SIGCSE Bulletin* 27,1:6 - 9.

Satratzemi, M, V Dagdilelis & G Evagelidis 2001. A System for Program Visualization and Problem-Solving Path Assessment of Novice Programmers. *ACM SIGCSE Bulletin* 33,3:137 - 140.

StatSoft Inc. 2001. STATISTICA (Data Analysis Software System) Ver. 6

Studer, SD, J Taylor & K Macie 1995. Youngster: A Simplified Introduction to Computing: Removing the Details so that a Child May Program. *ACM SIGCSE Bulletin*, 27,1:102 - 105.

Thomas, J 2002. B#: Version 2. Honours Treatise. Department of Computer Science and Information Systems, University of Port Elizabeth. Port Elizabeth, South Africa.

UCAS 2000. Universities and Colleges Admissions Service for the UK. Technical Report http://www.ucas.ac.uk/figures/index.html. Accessed on 12 February 2004.

Warren, P R 2000. Using JavaScript to Teach an Introduction to Programming http://saturn.cs.unp.ac.za/~peterw/JavaScript/calculator.html. Accessed on 5 June 2003.

Warren, PR 2001. Teaching Programming Using Scripting Languages. *Journal of Computing in Small Colleges* 17,2:205 - 216.

Warren, PR 2003. Learning to Program: Spreadsheets, Scripting and HCI. *Proceedings of Southern African Computer Lecturers Association (SACLA).*

Wilson, JD & GF Braun 1985. Psychological Differences in University Computer Student populations. *ACM SIGCSE Bulletin* 17,1:166 - 177.

Wright, T & A Cockburn 2000. Writing, Reading, Watching: A Task-Based Analysis and Review of Learners' Programming Environments. *Proceedings of International Workshop on Advanced Learning Technologies*. IEEE Computer Society Press.

Yeh, CL 2003. Tracing Programs in B#. Honours Treatise. Department of Computer Science and Information Systems, University of Port Elizabeth. Port Elizabeth, South Africa.

Ziegler, U & T Crews 1999. An Integrated Program Development Tool for Teaching and Learning How to Program. *ACM SIGCSE Bulletin* 31,1:276 - 280.

Appendix A

This questionnaire collects your thoughts on how easy/difficult it is to use B#. The series of questions presented encourage you to think about the ways you used B# and whether B# helped you to do the things that you were required to do.

Place a 1 next to the task that took the most time in B#, a 2 next to the task that took the next most time, etc ..., with a 4 next to the task that took the least time.

Searching for the correct programming icon to use	
Translating information from pseudocode/flowcharts into a B# program	
Reorganising and restructuring a B# program	
Playing around with the different programming icons in B# without being sure of the result/purpose of each	

Please answer the following questions to the best of your ability by marking the appropriate number (1, 2, 3, 4, 5 or 6). If you are able to, please provide additional details for each point.	**Never**	**Seldom**	**Some of the time**	**Often**	**Always**	**Unsure**
1. I find it easy to locate and use any B# programming icon.	1	2	3	4	5	6
If there are any constructs that are difficult to use and/or locate, please identify them.						
2. The windows of B# that are depend-	1	2	3	4	5	6

ent upon each other are easily visible and are always consistent.						
Please identify any windows of B# that fall into this category. Why is it necessary for these windows to be dependent upon one another?						
3. I find it easy to make changes to a B# program.	1	2	3	4	5	6
If there are any things that are difficult to change in a program, please identify them.						
4. I can state a solution to a problem reasonably briefly in B#.	1	2	3	4	5	6
Is there anything specific that you think could be simpler? How could it be simplified?						
5. I find that it requires a lot of thinking to create a solution in B#.	1	2	3	4	5	6
If there are any things in B# that require a lot of thought, please identify them and describe why you find them difficult.						
6. I find that it is easy to make irritating mistakes in B#.	1	2	3	4	5	6
Describe the kind of irritating mistakes that B# allowed you to make.						
7. I find that the way in which B# programs are created and displayed closely matches the types of problems that I must solve in WRA131.	1	2	3	4	5	6
Why?						
8. B# programs are easy to follow and understand.	1	2	3	4	5	6
Why?						
9. If I make a change to a B# program, the effect is always reflected in the parts that dependent on the change.	1	2	3	4	5	6
Describe the occasion(s) when this did not occur.						
10. It is easy to stop at anytime during a B# program creation and test my work so far.	1	2	3	4	5	6
Describe the times when this was not possible.						

11. B# encourages me to experiment with a solution.	1	2	3	4	5	6
How?						
12. B# forces me to think ahead and make certain decisions about a solution first.	1	2	3	4	5	6
How?						
13. The different parts of B# that mean similar things are clearly similar from the way that they appear.	1	2	3	4	5	6
Please identify anything which you consider to be similar in B#, and describe how you identified them as being similar.						
14. The arrangement of the icons in a B# program helps me to understand the purpose of the program better.	1	2	3	4	5	6
Describe exactly how the arrangement of the icons in a B# programs affects your understanding of the program task.						

What task/activity do you use B# for? ________________________
What, in your opinion, is the end product that B# produces? _____
How do you interact with B#? ______________________________
Can you think of any obvious ways that could improve the B# system?
____

Authors' Contact Details

Charmain Cilliers, (Charmain.Cilliers@nmmu.ac.za)
André Calitz, (Andre.Calitz@nmmu.ac.za)
Jéan Greyling (Jean.Greyling@nmmu.ac.za)
Department of Computer Science and Information Systems,
Nelson Mandela Metropolitan University, Port Elizabeth, South Africa