Nelson Mandela
Metropolitan
University

for tomorrow

The Impact of an In-Depth Code Comprehension Tool in an Introductory Programming Module

Ronald George Leppan

Supervisors: **Dr Charmain Cilliers**
Mrs Marinda Taljaard

Submitted in partial fulfilment of the requirements for the degree of Magister Scientiae in Computer Science and Information Systems in the Faculty of Science at the Nelson Mandela Metropolitan University

January 2008

Acknowledgements

I would like to thank my supervisors, Dr. Charmain Cilliers and Mrs Marinda Taljaard, for their unwavering support, advice, enthusiasm and encouragement for the duration of this dissertation. Their valuable suggestions for improvements to this document are strongly appreciated.

Thanks must go to the Department of Computer Science and Information Systems at the Nelson Mandela Metropolitan University for enabling me to conduct this research. A word of appreciation must specifically go to the Telkom Centre of Excellence for providing the necessary technology used in this research.

I would also like to thank the students who participated in the study and Dr Jean Greyling for allowing me time during his lectures and practical assignments to conduct the experiments, as well as for his valuable feedback on the practical exercises.

Thank you to my colleagues in the School of ICT for their interest shown in my progress and the support given when needed.

A special word of gratitude to my family and friends for their patience, support and understanding during the many hours it took to complete this dissertation.

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 authors and therefore the NRF does not accept any liability in regard thereto.

Summary

Reading and understanding algorithms is not an easy task and often neglected by educators in an introductory programming course. One proposed solution to this problem is the incorporation of a technological support tool to aid program comprehension in introductory programming.

Many researchers advocate the identification of beacons and the use of chunking as support for code comprehension. Beacon recognition and chunking can also be used as support in the teaching model of introductory programming. Educators use a variety of different support tools to facilitate program comprehension in introductory programming. Review of a variety of support tools fails to deliver an existing tool to support a teaching model that incorporates chunking and the identification of beacons.

The experimental support tool in this dissertation (BeReT) is primarily designed to encourage a student to correctly identify beacons within provided program extracts. BeReT can also be used to allow students to group together related statements and to learn about plans implemented in any semantically and syntactically correct algorithm uploaded by an instructor. While these requirements are evident in the design and implementation of BeReT, data is required to measure the effect BeReT has on the in-depth comprehension of introductory programming algorithms.

A between-groups experiment is described which compares the program comprehension of students that used BeReT to study various introductory algorithms, with students that relied solely on traditional lecturing materials. The use of an eye tracker was incorporated into the empirical study to visualise the results of controlled experiments. The results indicate that a technological support tool like BeReT can have a significantly positive effect on student comprehension of algorithms traditionally taught in introductory programming. This research provides educators with an alternative way for the incorporation of in-depth code comprehension skills in introductory programming.

Keywords: Program Plans, Program Beacons, Chunking, Code Comprehension, Introductory Programming, Eye-tracking

Table of Contents

Chapter 3 Educational Technological Support for In-Depth Comprehension of Introductory Algorithms 42

Chapter 4 Investigative Research Methodology 58

List of Figures

List of Tables

Chapter 1 Research Context

1.1 Introduction

The reading and understanding of algorithms is not an easy task (Deimel and Naveda 1990; Jenkins 2002; Warren 2003) and is often neglected by educators in an introductory module. Studies of the fragile knowledge of students in introductory programming[1] indicate the need for these courses to explicitly include not only code generation strategies, but also code comprehension strategies when studying algorithms.

Students are often required to read worked examples of algorithms and use a wide range of strategies to trace the code for understanding (Lister, Adams *et al.* 2004; Fitzgerald, Simon and Thomas 2005). Some strategies prove to be more successful than others, and students tend to employ a mixture of strategies for different situations (Chapter 2). At times, good strategies are used poorly by the students. Some of the strategies reported by Lister, Adams *et al.* (2004) and Fitzgerald, Simon *et al.* (2005) are a direct result of multiple-choice questions, while others are more related to a lack of code reading and comprehension skills in general. The strategies related to multiple-choice questions may prove to be useful for passing formal assessments, but may be insufficient for general program comprehension. Expert programmers also

[1] (Deimel and Naveda 1990; McCracken, Almstrum, Diaz *et al.* 2001; Ala-Mutka 2003; Lister, Adams, Fitzgerald *et al.* 2004; Garner, Haden and Robins 2005; Lahtinen, Ala-Mutka and Järvinen 2005)

use a wide variety of program comprehension strategies when maintaining programs (Von Mayrhauser and Vans 1994).

The aim of this investigation is to determine the effect of adopting a particular strategy for code comprehension that is based on the techniques used by experts to read code for in-depth comprehension (Chapter 2). In an attempt to bridge the gap between novice programmers (referred to in this document as "students") and expert programmers, an emphasis is placed on technological tools to aid student competence in using these strategies to comprehend code.

To focus the study, the rest of this chapter presents the background to the investigation by defining in-depth code comprehension and by addressing the need for strategies for reading and understanding algorithms. The importance of plan discovery when reading algorithms and the manner in which beacon recognition and chunking can be used as techniques to encourage reading of code with in-depth comprehension is discussed (Section 1.2). The relevance (Section 1.3) and focus of the research (Section 1.4), as well as an overview of the methodology (Section 1.5) used in the study, are also presented. The chapter concludes with an overview of the structure of the remainder of this dissertation (Section 1.6).

1.2 Background

One of the main purposes for reading an algorithm is to discover the plan(s) employed by the original programmer (Rist 1986; Garner 2002; Thomas, Ratcliffe and Thomasson 2004). To read an algorithm with in-depth comprehension, students have to understand which code fragments are required to perform a higher-level task or identify the task that a particular code fragment performs (Deimel and Naveda 1990; Upchurch 1997). Numerous advantages exist for students who have mastered the skill of code reading and comprehension. Code reading skills help students learn new programming language constructs and influence their style in future code generation activities. Code reading skills also aid in program maintenance activities. Consequently, an effort must be made in an introductory programming module to cultivate in-depth code reading skills.

One of the key components of an introductory programming module is the tracing, evaluation and comprehension of various algorithms, for example Binary Search, Bubble and Insertion Sort (CS&IS 2006a, 2006b). Proper algorithm reading skills are necessary prior to developing skills of composing algorithms (Kimura 1979; Deimel and Naveda 1990). A survey done by Lahtinen, Ala-Mutka *et al.* (2005) showed that students and teachers rate worked examples as the most helpful learning material to study programming. Algorithm reading skills are thus of vital importance for improved programming (Ala-Mutka 2003). In order to develop such reading skills, it is necessary to investigate models used by experts to comprehend programs.

1.2.1. Models of code cognition

Table 1-1 summarises several cognition models derived from expert programmers that have been developed over the years[2].

Author	**Components**	**Technique to build mental model**
(Letovsky 1986)	• Knowledge Base • External Representations • Assimilation Process • Internal Representations	Programmers use annotations to match goals with implementation
(Shneiderman and Mayer 1979)	• Internal Semantic Representation (program goals and algorithms) • Long-Term Memory (semantic and syntactic knowledge)	Programmers use a chunking (Figure 1–2) process to assimilate a program from short-term memory into an internal semantic representation of goals and algorithms to achieve goals
(Brooks 1983)	• Problem Domain • Intermediate Domains • Program Domain • External Representation • Internal Representation	Programmers use hypothesis generation and search for beacons (Figure 1–1) in source code to verify hypotheses
(Soloway and Ehrlich 1984)	• Knowledge (plans and conventions) • Understanding Processes • Internal & External Representations	Programmers use a chunking technique and knowledge of programming conventions to decompose goals into plans and plans into lower-level plans
(Pennington 1987)	• Program Model (text structure and programming plan knowledge) • Situation Model (functional and data flow abstraction)	Programmers build a program model using beacons to identify chunked plans in the code and then a situation model in terms of real-world objects using hypotheses

Table 1-1 Cognition Models for Code Comprehension

[2] (Shneiderman and Mayer 1979; Brooks 1983; Soloway and Ehrlich 1984; Letovsky 1986; Pennington 1987)

Letovsky's (1986) cognition model consists of a knowledge base, internal and external representations and an assimilation process to build the mental model (Letovsky 1986). The model suggests that programmers work from a knowledge base of expertise, domain goals, system plans and knowledge of programming conventions. The documentation, code and manuals are all part of the external representations and need to be assimilated by the programmer. The internal representation (mental model) consists of the system specifications, and the implementation and annotations that match specification goals with the implementation. The assimilation process can be either bottom up (from code to specifications) or top down (from the high-level system specifications to plans in the code).

Schneiderman and Mayer's (1979) model consists of a working memory containing the semantics of the program being studied and a long-term memory, which represents the knowledge base consisting of semantic and syntactic knowledge (Shneiderman and Mayer 1979). Syntactic and semantic knowledge from the knowledge base guides a chunking process to assimilate the program and problem statement via the working memory to grow the knowledge base further. Chunking refers to the grouping of related statements into a single functional unit (Section 1.1.2).

Brooks' (1983) model suggests programmers constantly match elements from the problem domain with the program domain. The problem domain is the area where the problem is defined (Pennington 1987). The program domain is the area containing the programmed solution. Assimilation is driven by the generation of hypotheses while studying the requirements and other documents from the problem domain, and the code and user manuals from the program domain. Hypotheses of the program goals are verified by searching for beacons in the external representations and matching them with hypotheses and sub goals in the mental model (internal representation). Beacons are surface features in a program that serve as indicators of a particular structure or operation (Section 1.1.2).

The cognition model described by Soloway and Ehrlich (1984) consists of knowledge of programming plans and best programming practices. During code comprehension, the external sources studied include requirements documents, code, design documents, user, reference and maintenance manuals and other relevant documentation. The assimilation process uses the procedure of chunking statements together that form part of a higher-level plan. The plans in the programmers' internal representation match the goals of the system.

Pennington's (1987) model divides a programmer's knowledge and attention between the program domain and the problem domain. When studying a new system, an internal representation of the program model is constructed first. The mental model of the program domain is built using beacon recognition and chunking while studying the external representation of the system (program code and other documentation). Text structure and syntactic knowledge assist the beacon recognition and chunking processes. An internal representation of the situation model (problem domain) is built after the program model. The mental model of the problem domain is built using beacons and the generation of hypotheses. The use of beacons and hypotheses generation can be done if the programmer has a general understanding of the problem domain.

Numerous studies have also shown that expert programmers tend to follow a combination of these models more closely than novices[3]. Two common assimilation techniques emerge from the models scrutinised in Table 1-1, namely that of beacon recognition and chunking.

1.2.2. Beacon recognition and chunking

Brooks (1983) defines beacons as features in a program that serve as indicators of a particular structure or operation. Beacons can be fragments of code or variable/function names that point to the existence of some high-level goal[4]. Figure 1-1 shows three stereotypical lines of code that can be used to swap the values of two variables. The cross-switching of the variables is indicative of the fact that the variables are swapped. The fragment can, therefore, be classified as a beacon that indicates the possibility of a sorting algorithm. The appearance of these three lines of assignment statements are so unique that it can also be classified as a beacon that indicates a higher-level task, for example, *Swap(a,b)*, which appears in the mental

[3] (Wiedenbeck and Evans 1986; Wiedenbeck and Scholtz 1989; Fix, Wiedenbeck and Scholtz 1993; Crosby, Scholtz and Wiedenbeck 2002)

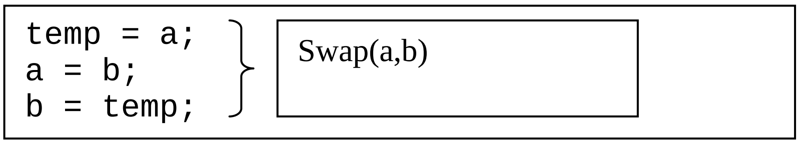
[4] (Wiedenbeck and Evans 1986; Pane and Myers 1996; Wiedenbeck, Ramalingam, Sarasamma *et al.* 1999)

model of the programmer reading the code. Experienced programmers familiar with this plan of swapping the values of two variables may spot this combination of statements in an unseen algorithm and hypothesise the goal of the algorithm. Further investigation by the experienced programmer would reveal more beacon-like statements to verify or nullify the hypothesis.

```
temp = a;
a = b;
b = temp;
```

Figure 1-1 Example of a beacon

A programmer with less experience will typically resort to a bottom-up approach to reading the algorithm. In this instance, the programmer might realize that the three statements in Figure 1-2 can be grouped together and assigned a semantic description (*Swap(a,b)*). The semantic description indicates a higher-level goal, namely the swapping of the values of two variables in this example.



```
temp = a;
a = b;
b = temp;
```

Figure 1-2 Example of chunking

The chunking of the three lines of code into a semantic description can aid the comprehension process by reducing the cognitive overload. The semantic description replaces the lower-level detail and can be used to memorise an algorithm. The combination of the three lines may now become a beacon in future code reading activities.

From the examples of beacon recognition and chunking, it can be seen that there are primarily two ways of using beacons and chunking. The bottom-up approach requires a scanning of the code and assigning meaning to chunks of code using syntactic and semantic knowledge of the programmer. The top-down approach of code comprehension starts with the programmer generating hypotheses about the higher-level functions of the program. The code is scanned for the existence of beacons in order to confirm these hypotheses; however, it was noted by Wiedenbeck and Evans

(1986) that, unlike expert programmers, novices do not naturally focus on beacons during code comprehension activities.

1.3 Relevance of research

The primary relevance of this study originates from the ongoing research in the Department of Computer Science and Information Systems (CS&IS) at the Nelson Mandela Metropolitan University (NMMU) aimed at increasing the throughput rate of students in the introductory programming modules[5]. This research can be divided into two main categories; namely identifying potentially successful students and changing/enhancing the methods of presenting the module (Figure 1-3). The proposed study forms part of this endeavour by modifying the teaching model.

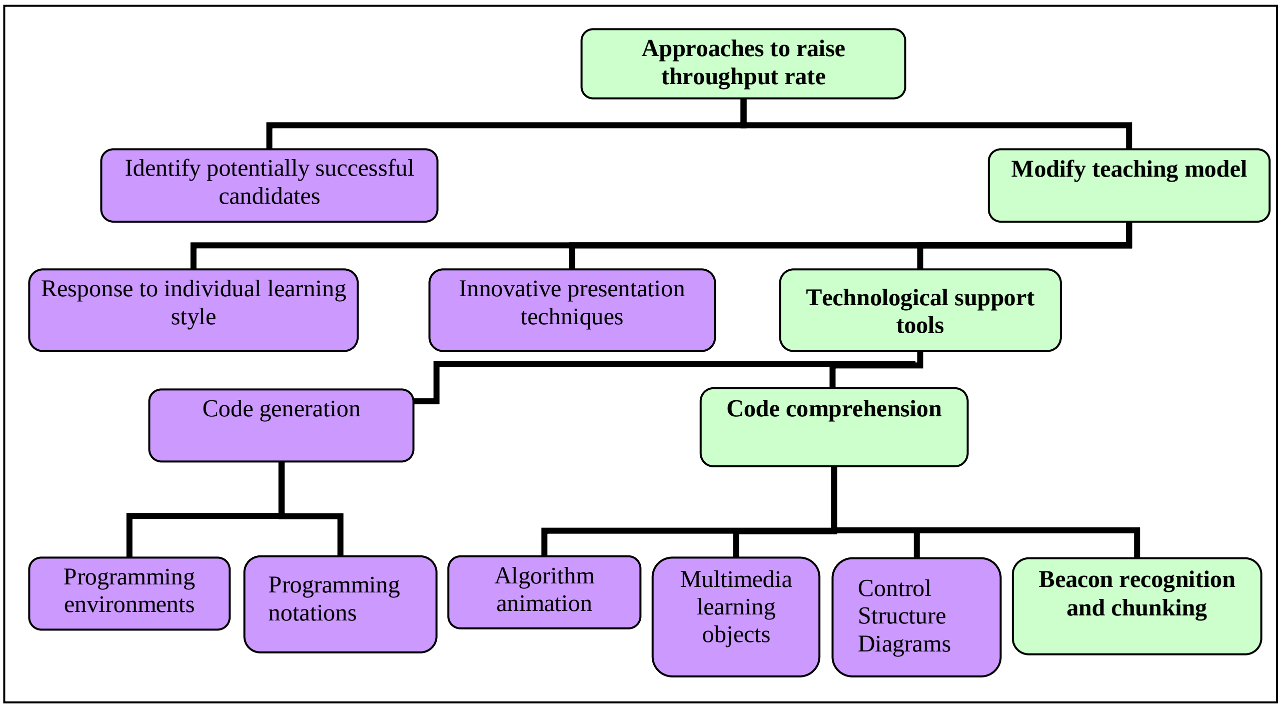

Approaches to raise throughput rate
Identify potentially successful candidates
Modify teaching model
Response to individual learning style
Innovative presentation techniques
Technological support tools
Code generation
Code comprehension
Programming environments
Programming notations
Algorithm animation
Multimedia learning objects
Control Structure Diagrams
Beacon recognition and chunking

Figure 1-3 Overview of research to increase throughput in introductory programming

A secondary relevance is the lack of empirical data in current research reports to validate technological support that aid in-depth learning of students in introductory programming modules. The Department of CS&IS has recently developed such an experimental tool that aids plan discovery by allowing students to chunk related lines and coaches students to identify beacons in source code (Harris 2005; Harris and Cilliers 2006). The tool was developed to aid the introduction of program

[5] (Calitz 1997; Greyling and Calitz 2003; Cilliers 2004; Vogts 2006; Yeh, Greyling and Cilliers 2006)

comprehension activities in an introductory algorithms module. There is a need to validate the usefulness of this tool by means of a between groups' experimental study that determines its impact on the ability of introductory programming students to comprehend worked examples after exposure to the tool.

1.4 Focus of research

Figure 1-3 gives an overview of some of the research done in the field of introductory programming to increase the throughput rate of students in an introductory programming module. The purple theme indicates research done in the area of the modification of the teaching model for introductory programming modules, but falling outside the scope of this study.

As indicated by the green theme of Figure 1-3, this study takes as its departure point the modification of the teaching model. Emphasis is placed on technological support tools to aid code comprehension. This study concentrates on a support tool (Beacon Recognition Tool (BeReT)) to allow students to methodically investigate algorithms (Harris 2005; Harris and Cilliers 2006). BeReT (Chapter 3) incorporates theories of program comprehension and provides some level of support while students discover plans in the source code. By being guided to discover plans in source code, the student is able to study the algorithms at the in-depth level of learning, instead of focusing on superficial syntactical issues. Underlying skills presented to the student are beacon recognition and chunking (Section 1.2.2). Mastering these skills will allow students to construct novel solutions in future code generation activities and investigate unseen algorithms with confidence. The experimental design using BeReT and subsequent studies are restricted to the situation at the Department of CS&IS at NMMU (Chapter 4).

The primary objective of this project is, therefore, to investigate the impact on in-depth code comprehension of an experimental technological educational support tool (BeReT) in an introductory programming module by means of an empirical investigation. The primary research question is:

> *What is the effect of a technological learning tool that supports plan discovery on the in-depth comprehension of introductory programming algorithms typically studied by novice programmers?*

The primary research question suggests a number of preliminary component research questions (Table 1-2):

Research Question		**Research Technique**	**Chapter**
1.	**What are the requirements for in-depth comprehension of Introductory Programming Algorithms?**		
	1.1. What is in-depth learning within the context of introductory programming?	Literature Review	Chapter 2
	1.2. How do experts build internal representations of source code?		
	1.3. What techniques exist to aid the comprehension of introductory algorithms?		
	1.4. What methods can be used to test algorithm comprehension?		
2.	**How can Educational Technological Support be used to aid in-depth learning of Introductory Programming Algorithms?**		
	2.1. What are the requirements for technological support that encourages in-depth learning in novice programmers?	Literature Review	Chapter 3
	2.2. What technological support has been developed/is used to encourage in-depth learning in novice programmers?		
	2.3. How does the Beacon Recognition Tool meet the requirements for such technological support?	Heuristic Evaluation	
3.	**What experimental design is appropriate for the empirical investigation?**		
	3.1. In what learning environment will the study take place?	Literature Review	Chapter 4
	3.2. What material, tools and procedures will be used to collect data?	Experimental Design	
	3.3. How will the data be analyzed?	Literature Review, Experimental Design	
	3.4. What can be done to ensure validity of data collected?		
4.	**Did the Educational Technological Support tool encourage in-depth learning of introductory programming algorithms?**		
	4.1. What are the results of the empirical study?	Empirical Evaluation & Questionnaires Analysis of Empirical Data. Questionnaires	Chapter 5
	4.2. What conclusions can be drawn from the results of the empirical study?		
5.	**How can the results be applied in an introductory programming module?**		
	5.1. How can the results of the empirical investigation be applied in the selection of appropriate technological support to encourage in-depth learning of introductory programming algorithms?	Deliberation on findings	Chapter 6

Table 1-2 Research Questions

1.5 Research methodology

The primary research question stated in Section 1.4 gives rise to the following hypotheses:

H_0: BeReT does not have a positive effect on the in-depth comprehension of introductory programming algorithms

H_1: BeReT has a positive effect on the in-depth comprehension of introductory programming algorithms

The research methodology is aimed at finding sufficient evidence to reject the null hypothesis. A literature review produces a list of requirements for technological support that encourages in-depth learning of introductory programming algorithms (Chapter 2). These requirements are used to evaluate existing technological support tools (CS&IS experimental tool included). This study contributes an investigative methodology for an empirical investigation in code comprehension, which incorporates eye-tracker data (Chapter 4). Empirical evidence gathered from class test and controlled eye-tracking experiments to validate the use of technological support for in-depth learning, results from the phases of the investigative methodology. A proposal for the incorporation of technological support tools to aid in-depth learning of algorithms in an introductory programming module follows as a direct result of this study.

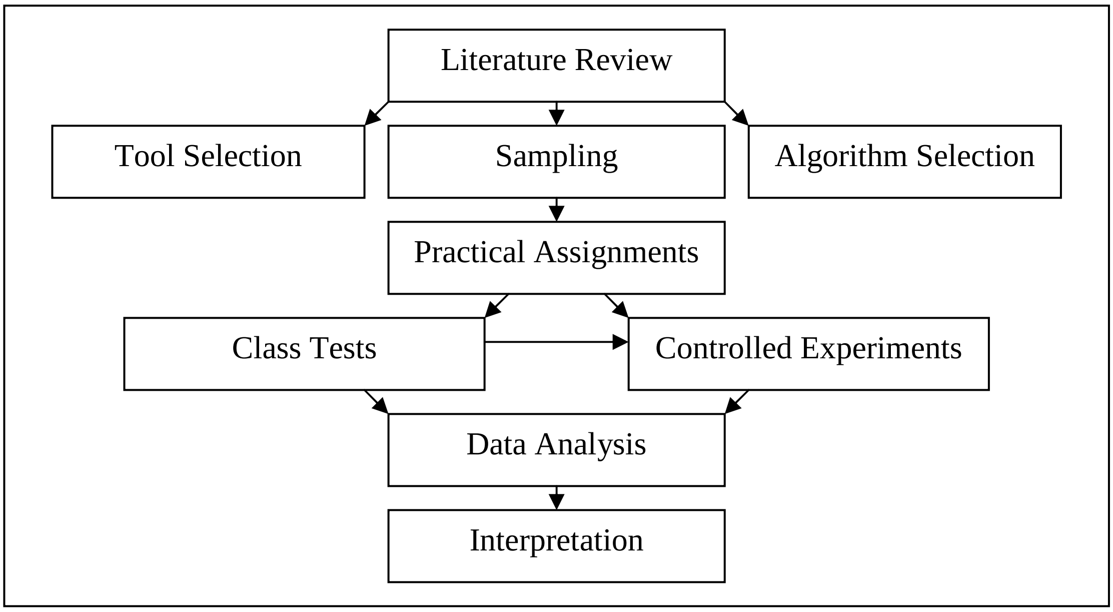


Figure 1-4 Phases of this study

The main phases of this study are presented in Figure 1-4. This study includes an extensive literature review of past and current developments in educational technological support that encourage the development of introductory programming students at the in-depth level of learning (Chapter 2). The literature review provides an overview of the cognitive factors influencing students studying introductory programming. The review of literature also summarises techniques used by experts to aid algorithm comprehension and investigates techniques used by educators to aid the comprehension of introductory algorithms. The literature review further focuses on the relevance of beacon recognition and chunking as techniques to build the mental model. A study of available support tools follows the literature review and a support tool (BeReT), designed to incorporate the underlying principles of beacon recognition and chunking, is introduced. The literature review also guides the selection of algorithms.

The introductory programming students are randomly sampled into two groups (control and treatment groups) and data is collected from these groups over a period of a semester. Practical assignments are used to train the treatment group of students using the selected support tool. The control group perform the same practical assignments without the experimental support tool. An empirical study follows in which class tests and controlled experiments are used to determine the effectiveness of BeReT in aiding the comprehension of introductory algorithms. The controlled experiments involve a subset of participants from each group and are performed on an individual basis in a usability laboratory. Participants in the controlled experiments are monitored while completing program comprehension exercises and an eye-tracker is used to record their actions resulting from their cognitive processes during these code comprehension exercises. The data collected from the class tests and controlled experiments are analysed and interpreted and conclusions are drawn.

All resources, activities and support tools are tested in a pilot study. Similar to other studies, data and feedback from the pilot study are used to refine the experimental design in preparation for the proper investigation (Van Greunen 2002; Pretorius 2005).

1.6 Structure of dissertation

The remaining chapters discuss the effect of a technological support tool on the in-depth comprehension of introductory algorithms in an introductory programming module. Figure 1–5 shows an overview of the structure of the dissertation. The first two chapters provide the theoretical framework supporting the decisions made in the investigation. Chapter 2 focuses on the requirements for the in-depth learning of an introductory programming module. Chapter 2 provides an overview of the techniques used by educators in the module and looks at the cognitive dimensions of introductory programming students. Chapter 2 focuses on techniques used to reduce the cognitive overhead when reading worked examples. Bloom's taxonomy is discussed in Chapter 2 to provide a tool for the assessment of in-depth comprehension in introductory programming. Chapter 3 provides an overview of technological support tools used to support in-depth learning of introductory algorithms and uses a list of requirements derived in Chapter 2 to evaluate current tools used for comprehension in introductory programming. BeReT is discussed in terms of the requirements.

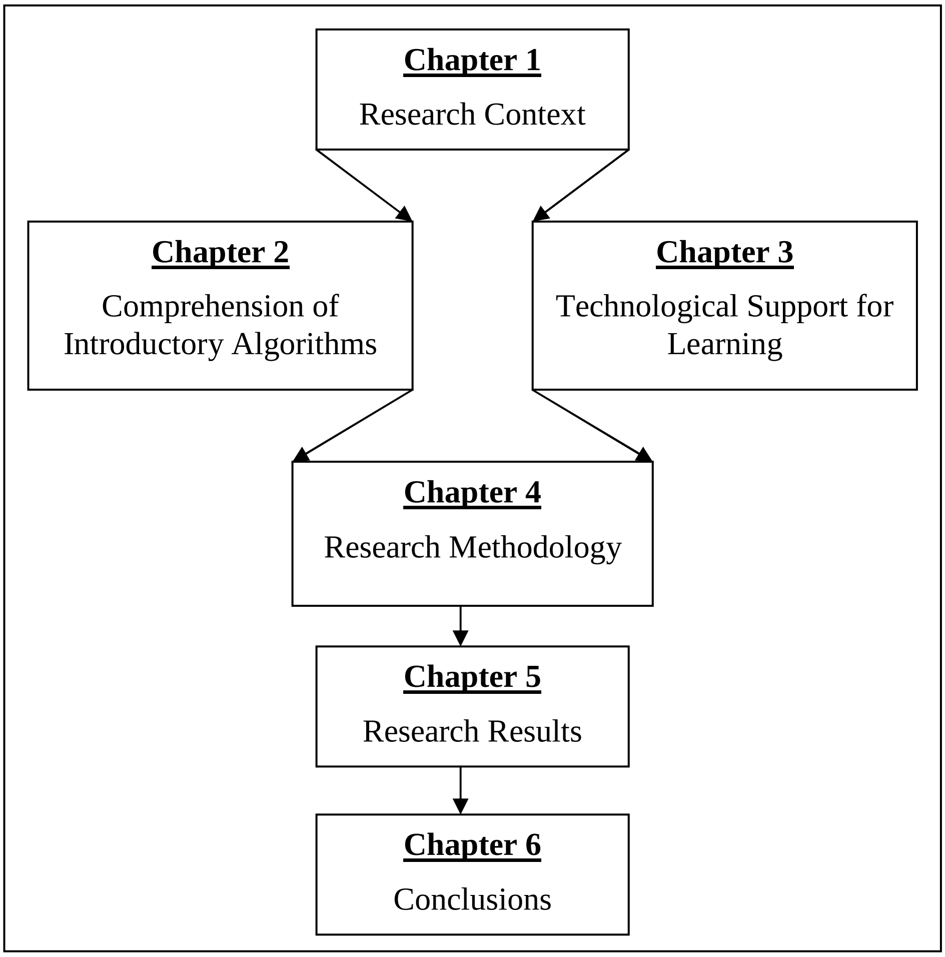


Figure 1-5 Structure of Dissertation

Chapter 4 presents the methodology used in the empirical investigation, which includes the use of an eye tracker to determine students' gaze patterns when answering code comprehension questions. Chapter 5 presents the results of the empirical study and draws some conclusions based on the results. Chapter 6 concludes the dissertation by evaluating the final outcomes against the initial objectives and provides some recommendations for the incorporation of a technological support tool that aids plan discovery in the introductory programming curriculum. Limitations of the current study and recommendations for future research are also given.

Chapter 2 Requirements for In-Depth Comprehension of Introductory Programming Algorithms

2.1 Introduction

The question has been raised as to whether technological support tools can be incorporated into introductory programming to aid the comprehension of algorithms presented in the module (Chapter 1). In order to determine the requirements of technological support tools in such an environment, it is necessary to first focus on the needs of the students with regards to the comprehension of introductory algorithms. This chapter does not only present the requirements to develop students' comprehension of introductory programming algorithms, it also determines the criteria necessary to test whether students have achieved an acceptable level of program comprehension.

In order to compile a comprehensive list of the requirements for the in-depth learning of algorithms in an introductory programming module, the cognitive model of expert programmers studying an algorithm is investigated (Section 2.2). This analysis is used as a model to guide students in algorithm reading skills. The chapter also provides a synopsis of methods used by educators to aid learning of introductory programming algorithms (Section 2.3). Beacon recognition and chunking are presented as methods that educators can use to aid comprehension of introductory programming algorithms. Bloom's taxonomy offers a thoroughly tested tool to determine the criteria for testing algorithm comprehension. This taxonomy and the way it is applied to computer science in general and introductory programming in particular are discussed (Section 2.4).

2.2 Cognitive model of Programmers during Code Comprehension

A programmer's cognitive model is the internal mental model of the program while studying the code (Engebretson and Wiedenbeck 2002). Apart from the actual code, the mental model incorporates other documentation (where available), knowledge of programming and knowledge of the problem domain modelled by the program.

During code comprehension activities, programmers build their mental model of the program using various strategies. Experts have demonstrated a tendency to employ a top-down strategy of code comprehension whereby extensive use is made of hypothesis verification through the recognition of beacons (Koenemann and Robertson 1991). Where hypotheses failed or are missing, experts employ a bottom-up strategy by integrating individual code statements into meaningful frames. This integration is referred to as chunking. In order to coach students in the use of an appropriate strategy for code comprehension, it is necessary to first determine the requirements for understanding a program (Section 2.2.1) and investigate the elements of programmers' mental models during code comprehension (Section 2.2.2). The way a program is navigated during code reading also impacts on their mental model (Section 2.2.3).

2.2.1 Program understanding

One definition of program understanding is that it is a process of using existing knowledge to acquire new knowledge (Mayer 1981). Mayer (1981) proposed a framework of steps to process information for in-depth learning of technical information (Figure 2-1). The first step in the process is to transfer new information into the student's short-term memory (**a**). This would typically be done when the student reads a worked example. The new information should be matched with appropriate existing knowledge in the long-term memory (**b**). When a suitable match is made between new information and knowledge from long-term memory, the new information is stored in long-term memory for future use (**c**).

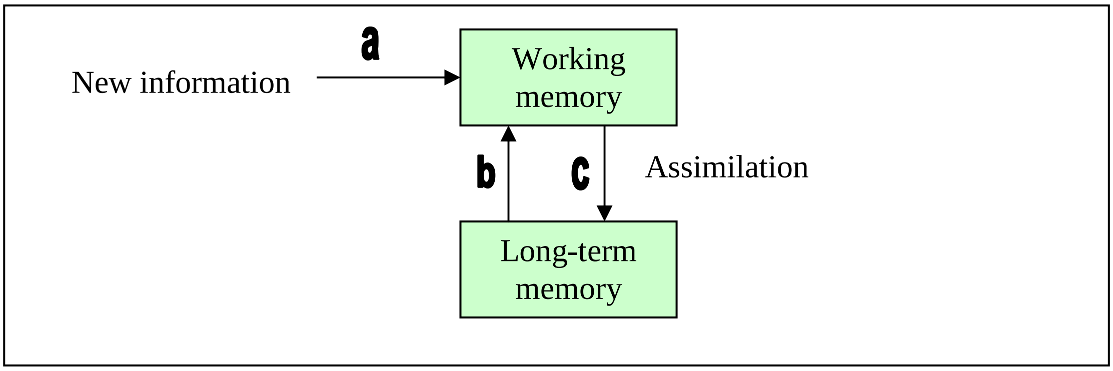


Figure 2-1 Mayer's model for learning new technical information (adapted from *Mayer (1981)*)

It is necessary for students to build into their long-term memory a knowledge base of plans that can be used in the construction of larger programs (Garner 2002, 2003). Students also need to build a collection of beacons to aid future code comprehension activities. A student reading a worked example who notices the three lines illustrated in Figure 2-2 for the first time would only recognise that the code extract consists of three assignment statements. On deeper inspection, the student might realise that these three assignment statements perform a swapping of the values in variables *a* and *b*. Chunking is used to assimilate the combination of assignment statements into long-term memory. This new information then becomes a beacon stored in long-term memory that can be used when studying an unseen algorithm with the same plan.

```
temp = a;
a = b;
b = temp;
```

Figure 2-2 Illustration of the Swap plan

Programmers are called upon to read code for a variety of different reasons. In some instances, they read it with a maintenance task in mind, to adapt an algorithm, to optimise it, to correct it or to reuse it (Von Mayrhauser and Vans 1994). In these cases, partial understanding of certain elements of the code would be sufficient. An opportunistic reading approach that focuses on the area of the algorithm that needs to be understood is suited to partial understanding. Hypothesis verification through the recognition of beacons would be appropriate when employing an opportunistic reading strategy.

There are also times when it is required to read code for general understanding. This implies a comprehensive understanding of the entire algorithm. A systematic reading strategy is typically used for complete understanding (Koenemann and Robertson 1991; Von Mayrhauser and Vans 1994; Chan and Munro 1997). A systematic reading approach implies a thorough scan of the code with emphasis on how the individual lines work together to meet the overall goal of the algorithm. Chunking is, therefore, a suitable technique when employing a systematic reading strategy.

2.2.2 Elements of the Mental Model

While programmers study an algorithm (including associated documentation) they form a working representation of the code in their working memory. This is referred to as their mental model. Experienced programmers have built up a sufficient repertoire of plans in their long-term memory to quickly determine the functionality of the code they are studying (Wiedenbeck, Ramalingam *et al.* 1999). Plans can be assimilated by a technique described as chunking when using a bottom-up strategy of code comprehension. Programmers studying an algorithm, using a top-down strategy, generate hypotheses about the existence of certain plans and use beacons to verify these hypotheses. Knowledge of programming constructs and programming conventions are required in programmers' long-term memory to aid plan discovery when studying an algorithm.

Plans

Plans are useful for understanding program functionality (Rist 1986; Garner 2003). Soloway and Ehrlich (1984) describe programming as the construction of plans to solve a specific problem. Wiedenbeck and Evans (1986) argue that during program comprehension, programmers become aware of these plans by searching for beacons. Figure 2-3 illustrates the way in which beacon recognition facilitates plan discovery during a top-down program comprehension strategy.

Programmers scan the code looking for beacons. A discovered beacon hints at a possible plan in the source code. The presence of the plan is then validated by searching for additional elements of the plan.

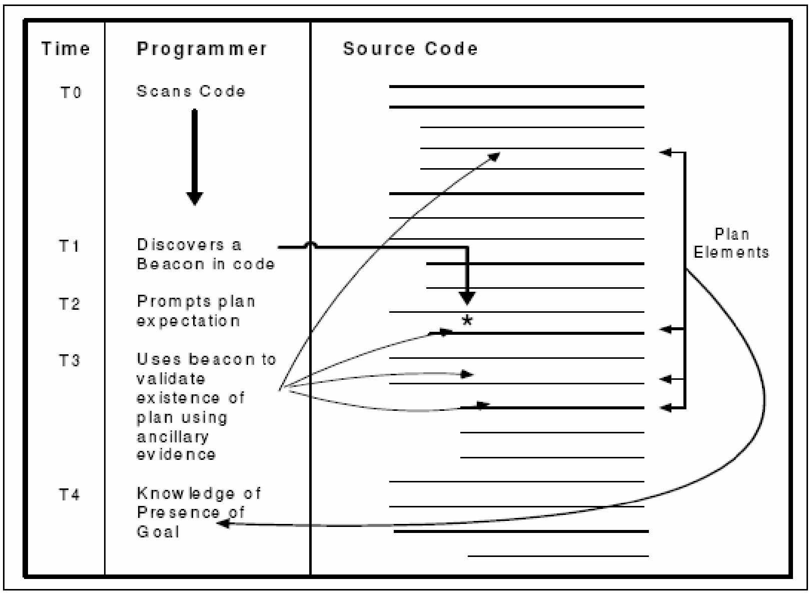
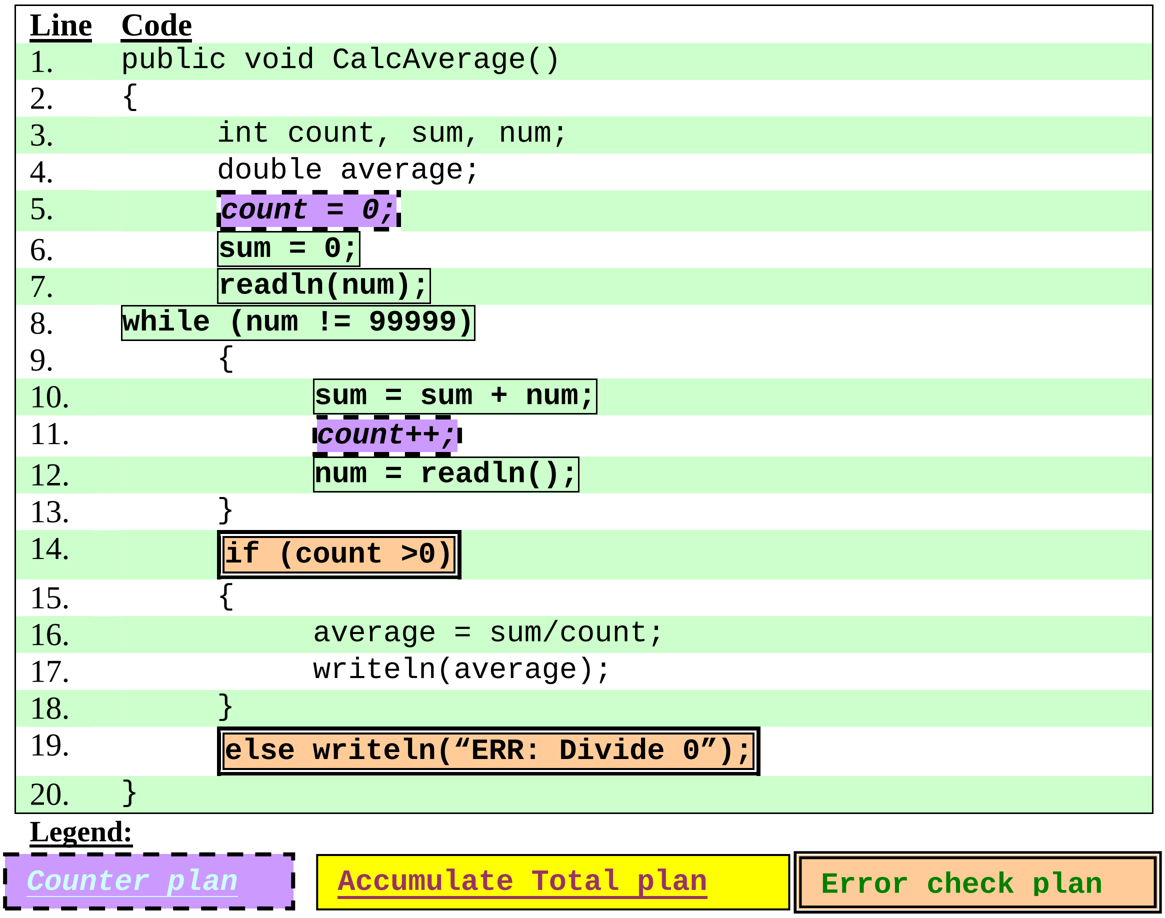

Time
Programmer
Source Code
T0
Scans Code
T1
Discovers a Beacon in code
T2
Prompts plan expectation
T3
Uses beacon to validate existence of plan using ancillary evidence
T4
Knowledge of Presence of Goal
*
Plan Elements

Figure 2-3 Using beacons to discover plans in source code (O'Brien 2003)



```
public void CalcAverage()
{
    int count, sum, num;
    double average;
    count = 0;
    sum = 0;
    readln(num);
while (num != 99999)
    {
        sum = sum + num;
        count++;
        num = readln();
    }
    if (count >0)
    {
        average = sum/count;
        writeln(average);
    }
    else writeln("ERR: Divide 0");
}
```

Figure 2-4 Using chunking to discover plans in source code

Figure 2–4 illustrates the way in which chunking facilitates plan discovery during a bottom-up program comprehension strategy. The example used in Figure 2–4 is an algorithm calculating the average of a number of inputs. Such an algorithm can be broken down into the following steps:

1. Calculate a running total of all values entered
2. Count the number of values entered
3. Ensure that legal values were input
4. Keep asking for inputs until a sentinel value is entered
5. Calculate and display the average

Steps 1, 2 and 3 are implemented using three plans, namely the *accumulate total plan*, the *counter plan* and the *error check plan*. The individual statements implementing each plan are grouped together into meaningful frames and associated with an annotation describing the plan. Elements of each plan are spread throughout the code. A programmer, when trying to understand the code in Figure 2–4, would scan the code and discover an initialisation of a variable named `count` (lines 3 and 5). The discovery prompts an expectation for a plan to increment the variable and upon further inspection of the code, the line `count++` (line 11) confirms the implementation of the counter plan listed above. The other plans can be discovered using the same technique. A programmer who wishes to recall the algorithm in Figure 2–4 can now recall the three plans and rely on his long-term memory to complete the code for the individual plans.

There are four different ways in which plans can be woven throughout the code, namely abutment, nesting, merging and tailoring (Soloway 1986). Abutment refers to two plans that are joined together sequentially. Nesting refers to the situation where one plan surrounds another plan. The inner plan starts after the outer plan starts and stops before the outer plan is completed. Plans that are merged are interleaved throughout the code. Plans that need to be modified to fit in with existing code are referred to as tailored plans. Plans are, therefore, not only sequential lines of code grouped together, but non-adjacent lines that could be situated anywhere in the code.

Hypotheses

Hypotheses are inferences that drive the plan discovery of a programmer when reading a program (Brooks 1983; Letovsky 1986). Letovsky (1986) identifies three types of hypotheses:

- *Why?* (…is a specific function used),
- *How?* (…is a program goal implemented),
- *What?* (…is this variable/function being studied).

When a programmer is familiar with the problem domain, hypotheses can be created prior to reading the code. In this case, programmers create a list of program goals that should be in the code and use *How?* type hypotheses to guide plan discoveries. These hypotheses create an expectation in the programmer to discover plans typically used to solve problems in that particular problem domain. The programmer reads the code with these hypotheses in mind and tries to find evidence in support of the hypotheses. This evidence takes the form of beacons in the source code.

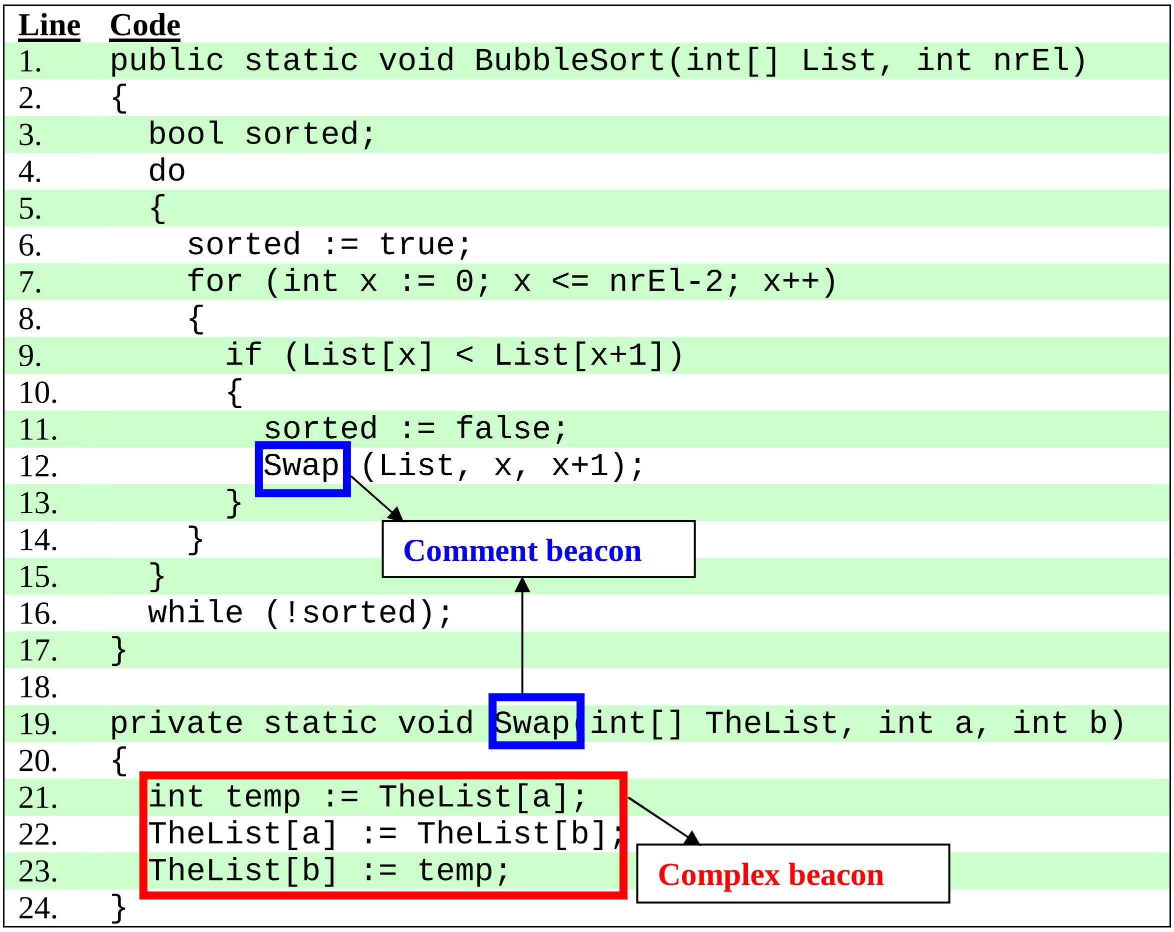
When the problem domain is unfamiliar to the programmer, a bottom-up approach would typically be used to discover the plans. Programmers read the code and use mostly *Why?* and *What?* type of hypotheses to discover plans. Individual statements will be grouped together and hypotheses are developed to discover the higher-level function of the group of statements instead of individual lines. Once all hypotheses are answered, the programmer is able to determine the functionality of the program.

Beacons and Chunking

Beacons are valuable in hypothesis verification when the problem domain is familiar to the programmer. Chunking is useful to discover plans when the problem domain is unfamiliar. From Section 1.2.2, it is also clear that beacon recognition and chunking are two techniques widely used by experts to build their mental model of code being studied.

Beacons are defined as surface features in a program that verify the existence of some structure or operation (Wiedenbeck and Evans 1986). A recent study shows that beacons are extensively used by experienced programmers to facilitate program comprehension (Crosby, Scholtz *et al.* 2002). It was also shown that students do not identify meaningful beacons in code. Beacons are used by programmers to abstract plans from an algorithm (Figure 2-3). Plans form part of the mental model of a programmer during code comprehension. Beacon identification is, therefore, a valuable skill that experts use during program comprehension activities, and it seems reasonable to make an effort to include beacon identification exercises in an introductory programming module.

Two classes of beacons have been identified by Crosby, Scholtz *et al.* (2002), namely comment beacons and complex beacons (Figure 2-5). Comment beacons refer to mnemonic devices in the code that indicate program functionality.



```
public static void BubbleSort(int[] List, int nrEl)
{
  bool sorted;
  do
  {
    sorted := true;
    for (int x := 0; x <= nrEl-2; x++)
    {
      if (List[x] < List[x+1])
      {
        sorted := false;
        Swap (List, x, x+1);
      }
    }
  }
  while (!sorted);
}

private static void Swap(int[] TheList, int a, int b)
{
  int temp := TheList[a];
  TheList[a] := TheList[b];
  TheList[b] := temp;
}
```

Figure 2-5 Example of the two types of beacons in a Bubble Sort algorithm

The function name `Swap` (line 19) and function call to `Swap` (line 12) are examples of a comment beacon. Since comment beacons are direct and simple to detect, they are easy for students to recognise and use. Complex beacons consist of multiple lines of code that are so unique in structure that they are strong indicators of some higher-level operation or function. The three lines of code that perform the swap operation (lines 21 – 23) together are an example of a complex beacon. Complex beacons are more implicit type beacons and only through programming experience would these lines of code become a beacon in future code comprehension activities.

Figure 2-6 shows an extract of an implementation of the Binary Search algorithm. The algorithm is used to find a particular value in a sorted array. The algorithm does this by eliminating half of the data during each phase. A binary search includes the following plans:

1. Calculates the midpoint.
2. Compares the search value to the value in the middle of the array to determine whether the search value falls in the first or last half of the array.
3. Continues searching the relevant section of the array repeatedly.

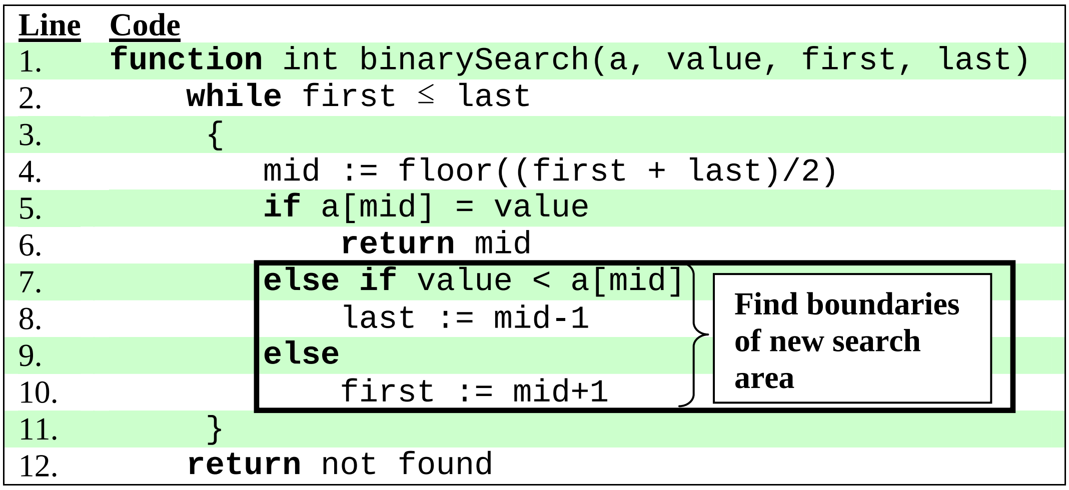
```
Line  Code
1.    function int binarySearch(a, value, first, last)
2.        while first ≤ last
3.         {
4.            mid := floor((first + last)/2)
5.            if a[mid] = value
6.                return mid
7.            else if value < a[mid]
8.                last := mid-1
9.            else
10.               first := mid+1
11.        }
12.       return not found
```

Figure 2-6 Implementation of a Binary Search Algorithm

In this example, `a` is the sorted list, `value` is the item being searched for, `first` and `last` are the indices of the first and last values of the section of the list being scanned.

The name of the function `binarySearch` in line 1 is an example of a comment beacon. Line 2 on its own is an example of a complex beacon and one that is typically found in a binary search (Aschwanden and Crosby 2006). Discovery of this line can prompt an expectation in the program reader that the algorithm might be a binary search. The combination of six lines from lines 5 to 10 is an example of another complex beacon that strengthens the expectation of a binary search. The mnemonic variables `mid`, `first` and `last` are also comment beacons that aid the discovery of plans in this algorithm.

Chunking is a program comprehension strategy that is useful for building mental models during program comprehension in unfamiliar problem domains (Burnstein and Roberson 1997). A chunk has a higher-level meaning and is often associated with a name (Rajlich and Wilde 2002). Large chunks can consist of nested chunks. Chunking can be used to acquire semantic knowledge embedded in the code of an algorithm. Frequently the first step in chunking is to group code on program domain concepts, for example syntactic nesting levels or functional units (Young 1996). A programmer would later revise these chunks to form interrelated units that represent problem domain concepts. A hierarchy of chunks would then be used to build the mental model of the programmer. Often these chunks become a beacon that can be used to identify the same plans in other algorithms.



```
Line  Code
1.    function int binarySearch(a, value, first, last)
2.        while first ≤ last
3.         {
4.            mid := floor((first + last)/2)
5.            if a[mid] = value
6.                return mid
7.            else if value < a[mid]
8.                last := mid-1
9.            else
10.               first := mid+1
11.        }
12.       return not found
```

Figure 2-7 Binary Search Example (Chunking)

Figure 2-7 shows an example of an algorithm where multiple related lines of code are grouped into a named grouping of code associated with a single semantic description. Lines 7 to 10 are used together to make the search area of the array smaller. These lines can be grouped together and instead of having to remember the individual lines of code (which are assumed to be previously learnt material and already in the programmer's long-term memory), the collective description (*Find boundaries of new search area*) can be assimilated into long-term memory. This program chunk is then processed instead of individual statements, simplifying the task of the programmer to recall the overall function of the algorithm.

Chunking can be used to understand control flow and to make a match between the sub-problems of the problem specification for which the program was designed (Rajlich and Wilde 2002). Assigning mental or annotated labels to chunks of code also helps reduce cognitive overload while building a representation of an algorithm in working memory (Shneiderman 1982).

The main difference between beacons and chunks is that beacons are used during top-down code comprehension and chunking during bottom-up comprehension (Aschwanden and Crosby 2006). A chunk of code can only become a beacon to the programmer if (s)he has had enough exposure to similar algorithms containing the same stereotypical grouping of lines. Figure 2-8 shows how beacons (B) and chunks (C) can be used as an index of knowledge when studying source code.

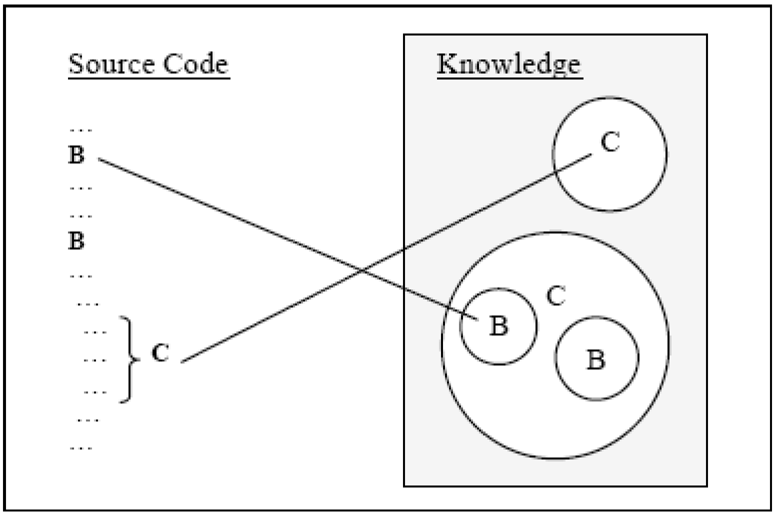


Figure 2-8 Beacons and Chunks (Aschwanden and Crosby 2006)

Instead of assimilating individual lines of code, a programmer first groups together statements that perform a single, higher-level operation. The chunk then becomes knowledge that can be used during future program comprehension activities. Some chunks that were previously assimilated in long-term memory become beacons that aid hypotheses verification when faced with unfamiliar source code. In some instances, a chunk of code can contain beacons that can be used to verify the hypothesis that the particular grouping of code is a specific implementation of a plan.

Programming Constructs and Conventions

In order to facilitate plan recognition, the program reader must have an understanding of the programming constructs used in an algorithm and programming conventions used by expert programmers.

The textual structure of an algorithm is used to organise knowledge of the overall control flow (Pennington 1987). It includes programming constructs, such as loops, conditionals, variables and reserved words in the programming language. To effectively organise knowledge in working memory, it is necessary to have knowledge of the basic elements of the structure of a programming language readily available in long-term memory.

Algorithms written according to standard programming conventions help to simplify the programmer's understanding of a program. Programming conventions create an expectation for the programmer. Examples of programming conventions include coding standards, expected use of certain data structures for certain processes and using mnemonic naming for variables. Programming conventions aid plan discovery by structuring the algorithm in a way that is familiar to most expert programmers.

2.2.3 Navigating Source Code

A programmer builds a conceptual model of a program by navigating the text structure of the source code and by relying on knowledge stored in long-term memory. This knowledge in long-term memory includes knowledge of the problem as well as in the program domain (Pennington 1987).

When reading an unfamiliar program, program knowledge is built up before problem domain knowledge (Pennington 1987). Domain knowledge links the plans from the program model to objects in the real world. Terms used to describe this knowledge come from the problem domain that is modelled in the program. Two strategies to study unfamiliar code are to use a bottom-up or top-down process of comprehension (Pennington 1987).

Top-down comprehension requires in-depth knowledge of the problem domain (Letovsky 1986). This problem domain knowledge is translated into hypotheses that will be used to determine the unfamiliar program's functionality. The text structure is scanned for beacons to accept/reject the hypotheses of the existence of a plan. Throughout the scanning of the unfamiliar code, the hypotheses change as needed.

Bottom-up comprehension assumes knowledge of the syntactic elements of the textual structure and at least the ability to extract semantic knowledge from the unfamiliar algorithm. The programmer reads the algorithm and "chunks" together related statements that perform a single, higher-level operation. These chunks are stored in long-term memory, often with a mental annotation to describe the grouping. The combination of chunks is used to determine the functionality of the program.

The navigation style that a programmer uses to read an algorithm has an impact on whether the algorithm is studied at the in-depth or the superficial levels. Navigation of a program is the process used to collect information about the program. Mosemann and Wiedenbeck (2001) identify three methods of navigation, namely sequential, control flow and data flow.

In sequential navigation, the mental model is constructed line-by-line, from the start of the program to the end. This technique is mostly used by novices (Jeffries 1982). Using this method creates a fragmented image of the program where low-level knowledge is gained, but where hierarchical structure, dynamic behaviours, interactions and purpose are lost (Mosemann and Wiedenbeck 2001). This type of navigation leads to superficial knowledge of the algorithm.

Control flow navigation proceeds through the program in the way that the computer will execute statements. This type of navigation starts from the first line of the main program, then moves to the next statement and steps into the relevant function or procedure that is called from the current point. Upon completion of the function or procedure, control steps back to the point where it was called. While navigating the code in this way, a programmer groups related statements into chunks to reduce cognitive overload and assimilate the abstract chunks to long-term memory. Control flow navigation encourages in-depth knowledge of the algorithm.

Data flow navigation builds a mental representation of the variables and how they change throughout the program (Mosemann and Wiedenbeck 2001). Data flow navigation is useful when a modification is necessary to add a new calculation or introduce new variables into a program; thus, data flow navigation also provides in-depth knowledge of the algorithm.

2.3 Aiding the comprehension of introductory programming algorithms

To improve students' ability to generate new or similar solutions, they are often required to understand a worked example of an algorithm (Koenemann and Robertson 1991; Garner 2003). The algorithm would mostly be an unseen one to show how programming constructs are used to solve a given problem. Students would be required to study the algorithm using a comprehension strategy suited to the situation. In some instances, students would need to debug the algorithm and in other situations, they would be required to read the algorithm for the purpose of understanding how it solves a particular problem or for recalling it at a later stage.

By presenting students with worked examples, they are exposed to properly designed sample solutions to a stated problem (Sweller and Cooper 1985; Renkl 1997). This technique has also been applied to the study of introductory programming (Cook, Bregar and Foote 1984; Garner 2000, 2003). Small worked examples emphasizing a few concepts at a time are often used in introductory programming modules. This is done to support students' active programming skills (Lahtinen, Ala-Mutka *et al.* 2005). Students in introductory programming are presented with a formulation of the problem, steps for a solution and an example of the final solution in a specific programming language (Renkl 1997).

The abundance of worked examples in introductory programming textbooks requires students to develop techniques to read and understand these programs with efficiency (Deimel and Naveda 1990). A recent multinational survey reported that many students have difficulty with programming, even after completing an introductory module (McCracken, Almstrum *et al.* 2001). Lister, Adams *et al.* (2004) conducted a subsequent study to determine if this was due to a lack of problem solving skills, or because of superficial knowledge. Superficial knowledge refers to the phenomenon that a student is capable of articulating particular items of knowledge when prompted, but fails to apply the knowledge in a program generation context. Basic skills that a student should exhibit include the ability of the student to trace and comprehend worked examples. The results of the study showed that students have a fragile grasp of the skills required to solve problems. It seems, therefore, necessary to aid students in their understanding of worked examples.

Subsequently, techniques are needed to promote the in-depth learning of introductory algorithms. These techniques should enable a student to extract semantic meaning from the source code. Over time, their long-term memory should be indexed with a network of semantic plans (Soloway and Woolf 1980; Curtis 1981).

In an effort to continue developing requirements for the in-depth comprehension of algorithms in an introductory programming module, valuable insights can be gained through studying techniques used by educators to aid students in the construction of their mental model during code comprehension activities. It is the responsibility of the educator to guide the student in the construction of a functional mental model so that unfamiliar material can be classified according to the model and correct responses formulated (Ben-Ari 2001). Introductory programming educators use a variety of techniques to assist students to comprehend algorithms.

The techniques typically used by educators to help their students study worked examples of algorithms include the following:

- Reading Tasks (Kimura 1979; Deimel and Naveda 1990)

- Algorithm Animations (Brown 1988; Naps, Eagan and Norton 2001; Yeh, Greyling *et al.* 2006)
- Visualisation of Programming Constructs (Ford 1993; Ala-Mutka 2003; Boyle 2003)
- Control Structure Diagrams (Cross, Hendrix and Barowski 2002; Hendrix and Cross 2002)
- Flowcharts (Shneiderman 1983; Scanlan 1988; Cilliers, Calitz and Greyling 2005; Mendes and Marcelino 2006)
- Object Diagrams (Thomas, Ratcliffe *et al*. 2004)
- Patterns (Clancy and Linn 1999; De Barros, dos Santos Mota, Delgado *et al.* 2005)

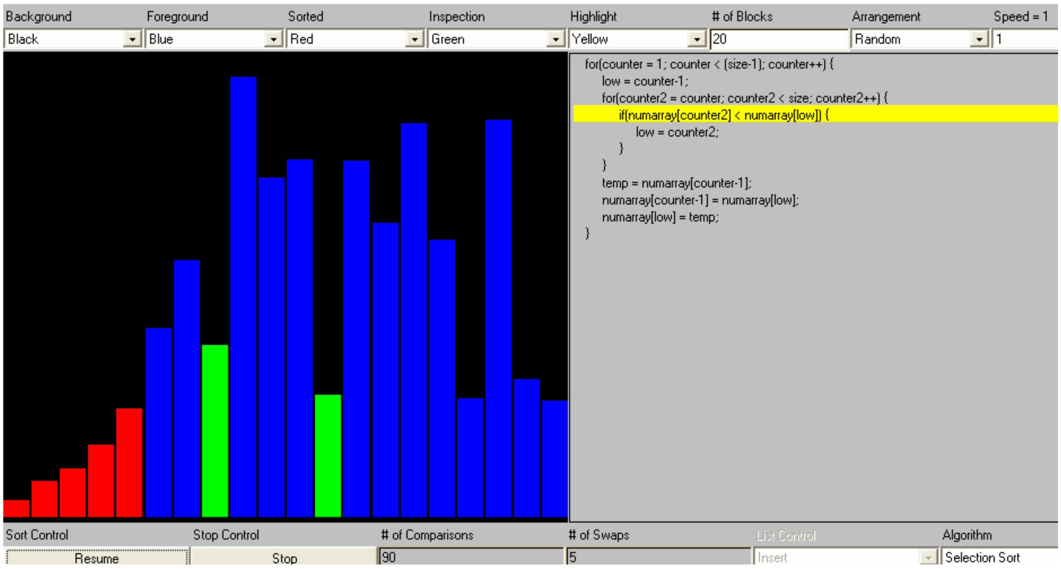
Reading Tasks

One of the earlier techniques for teaching introductory programming involves exposing students to a large enough number of worked examples to teach programming fundamentals before engaging them in code generation exercises (Kimura 1979). It was hypothesized that students are capable of obtaining reading skills with minimal instructional help. In the study, students were given 50 syntactically and semantically correct example programs (written in FORTRAN) as reading exercises and instructed to predict the output of each algorithm. Once the prediction is made, students should run the algorithm and hypothesize the use of constructs used. The hypotheses were then tested by developing and running other algorithms. No assistance was given to students with regards to reading skills and the size of the algorithms was relatively small. Students developed their own techniques to comprehend the code, with some techniques being more successful than others.

Reading skills should be cultivated by including explicit reading exercises into an introductory programming module (Deimel and Naveda 1990). Appropriate control should be taken over the cultivation of reading skills (Carbone and Kaasbøll 1998). The activities students engage in when reading code is different from the tasks when writing code. Students should be provided support for both program reading and program generation.

Algorithm Animations

Animations have been used since the 1980s to interactively visualize algorithms (Brown 1988). Animations can be useful for tracing the control flow in an algorithm or for showing the status of data structures during runtime (Naps, Eagan *et al.* 2001; Yeh, Greyling *et al.* 2006). Code is often highlighted and synchronized with animation actions so that students can relate the code to the animation (Brummund 2001) (Figure 2-9).



Background: Black
Foreground: Blue
Sorted: Red
Inspection: Green
Highlight: Yellow
# of Blocks: 20
Arrangement: Random
Speed = 1: 1

```
for(counter = 1; counter < (size-1); counter++) {
    low = counter-1;
    for(counter2 = counter; counter2 < size; counter2++) {
        if(numarray[counter2] < numarray[low]) {
            low = counter2;
        }
    }
    temp = numarray[counter-1];
    numarray[counter-1] = numarray[low];
    numarray[low] = temp;
}
```

# of Comparisons: 90
# of Swaps: 5
List Control: Insert
Algorithm: Selection Sort

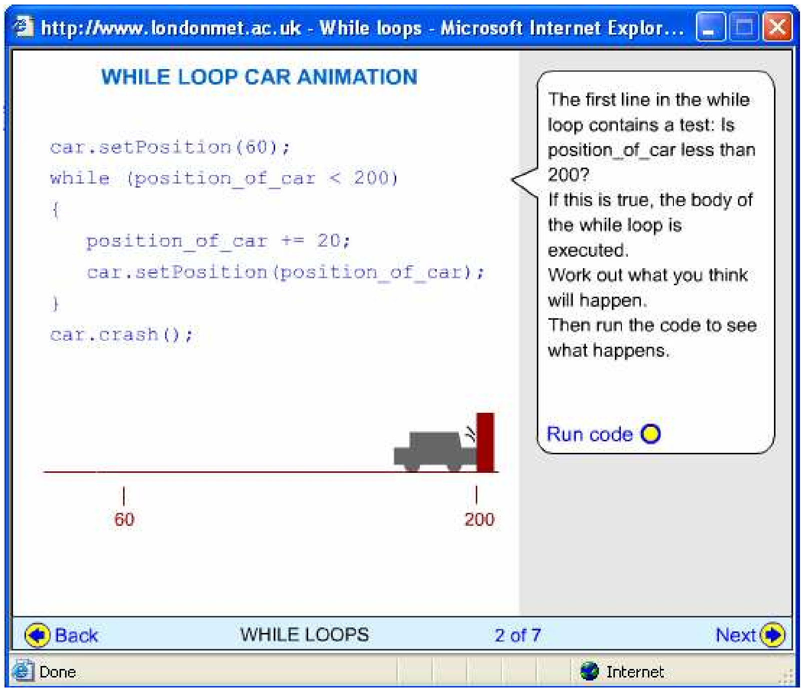
Figure 2-9 Algorithm Animations

Animations can also be used to compare the efficiency of two or more algorithms (Yeh, Greyling *et al.* 2006). However, the working group on the educational impact of algorithm visualization reports that this technique will not be effective unless it engages the students in active learning activities, such as interacting with the animation or answering comprehension questions while playing the animation (Naps, Eagan *et al.* 2001).

Visualisation of Programming Constructs

Dynamic visual learning materials are often used by computer science educators to aid with the learning of basic algorithmic concepts and structures (Ford 1993; Ala-

Mutka 2003; Boyle 2003). A while loop, for example, can be illustrated by showing what happens to a car during execution of the while loop (Figure 2-10).



```
car.setPosition(60);
while (position_of_car < 200)
{
   position_of_car += 20;
   car.setPosition(position_of_car);
}
car.crash();
```

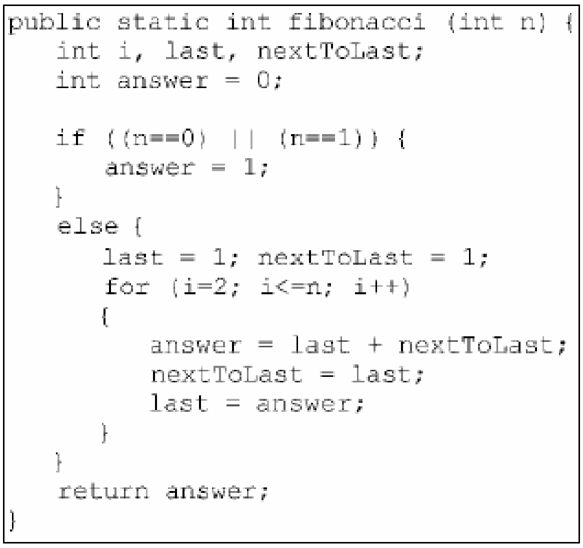
The first line in the while loop contains a test: Is position_of_car less than 200?
If this is true, the body of the while loop is executed.
Work out what you think will happen.
Then run the code to see what happens.

Figure 2-10 Visualisation of a while loop

Ford (1993) conducted experiments to determine how students visualise programs. In these experiments, students were required to animate how they visualise constructs, such as variables, pointers, classes, conditional statements and looping constructs. This study showed, amongst other things, that there is very little misconception regarding these programming concepts, yet numerous results have shown that students find it difficult to apply these concepts to solve a specific problem (McCracken, Almstrum *et al.* 2001; Jenkins 2002; Lahtinen, Ala-Mutka *et al.* 2005).

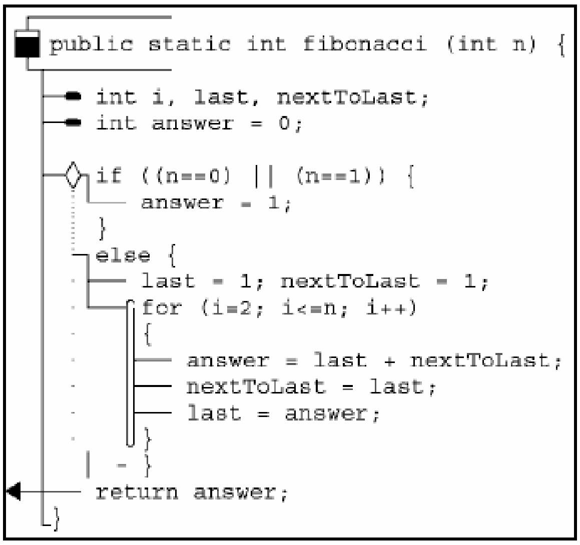
Control Structure Diagrams

Another type of visualisation technique that aids program comprehension is the use of control structure diagrams (CSD) (Cross 1998; Cross, Hendrix *et al.* 2002). In contrast to the program structure visualisation techniques that replace textual code, control structure diagrams use graphical constructs to supplement the text.



```
public static int fibonacci (int n) {
   int i, last, nextToLast;
   int answer = 0;

   if ((n==0) || (n==1)) {
      answer = 1;
   }
   else {
      last = 1; nextToLast = 1;
      for (i=2; i<=n; i++)
      {
         answer = last + nextToLast;
         nextToLast = last;
         last = answer;
      }
   }
   return answer;
}
```

Figure 2-11 Worked Example (Text Only)



```
public static int fibonacci (int n) {
   int i, last, nextToLast;
   int answer = 0;

   if ((n==0) || (n==1)) {
      answer = 1;
   }
   else {
      last = 1; nextToLast = 1;
      for (i=2; i<=n; i++)
      {
         answer = last + nextToLast;
         nextToLast = last;
         last = answer;
      }
   }
   return answer;
}
```

Figure 2-12 Worked Example (with CSD)

(Cross, Hendrix *et al.* 2002)

Figure 2-11 shows an algorithm that calculates the numbers in a Fibonacci sequence. As can be seen from Figure 2-12, a CSD uses the same algorithm with visual cues interspersed within the program text to highlight control structures and control flow. As an illustration of CSDs, the one in Figure 2-12 uses a diamond to indicate conditional structures, and a left-sided arrow to show that control gets passed back to the calling function. It was shown by Hendrix and Cross (2002) that CSDs have a significant positive outcome on program comprehension. The visual cues in a CSD attract the eye of a student and help students follow the control flow navigation.

Flowcharts

Using flowcharts is a visualisation technique that uses icons or symbols to graphically depict algorithms (Figure 2-13). Different shaped symbols convey different semantic meaning within the context of the algorithm it represents.

Scanlan (1988) showed that the time to learn an algorithm and the number of times it is required to be viewed reduces significantly when using a flowchart to understand complex programs as compared to conventional textual notation.

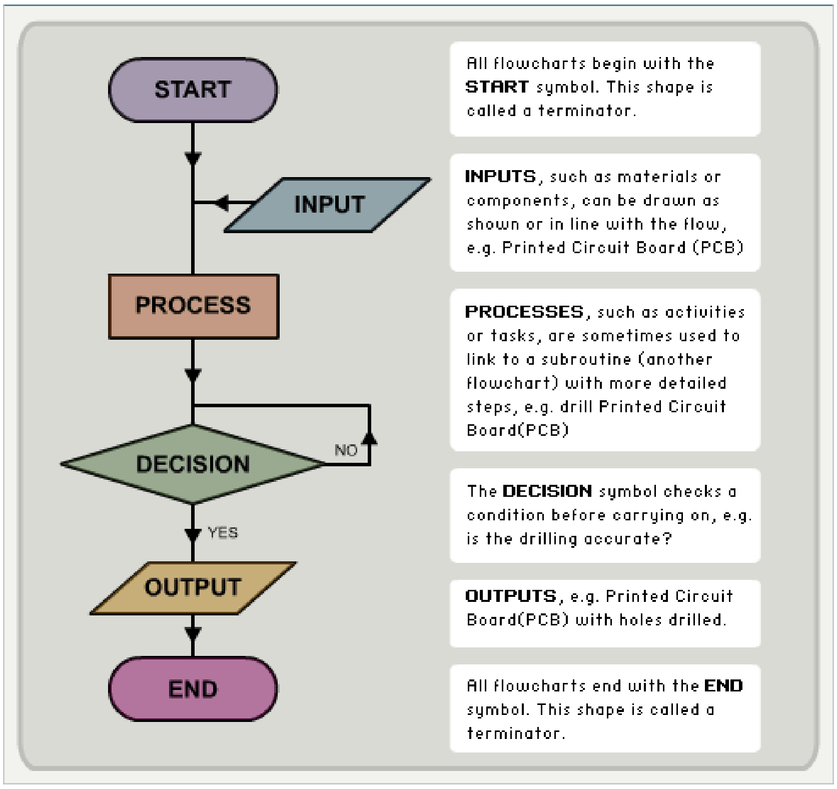
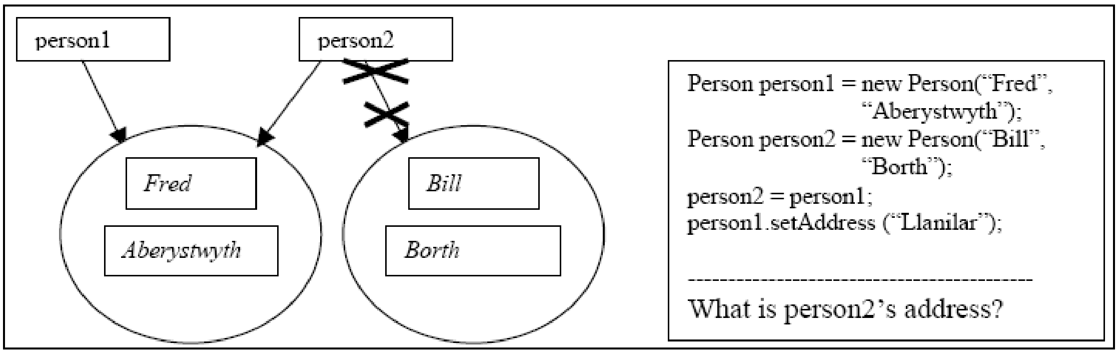

START
INPUT
PROCESS
DECISION
NO
YES
OUTPUT
END
All flowcharts begin with the START symbol. This shape is called a terminator.
INPUTS, such as materials or components, can be drawn as shown or in line with the flow, e.g. Printed Circuit Board (PCB)
PROCESSES, such as activities or tasks, are sometimes used to link to a subroutine (another flowchart) with more detailed steps, e.g. drill Printed Circuit Board(PCB)
The DECISION symbol checks a condition before carrying on, e.g. is the drilling accurate?
OUTPUTS, e.g. Printed Circuit Board(PCB) with holes drilled.
All flowcharts end with the END symbol. This shape is called a terminator.

Figure 2-13 Flowchart (Millner 2002)

Using flowcharts in program generation tasks also increase students' understanding of the problem (Crews and Ziegler 1998). However, a flowchart, where one box corresponds to one statement, is not an effective aid during program comprehension (Shneiderman 1983), since the flowchart becomes lengthier and more complex than the program itself (Waddel and Cross 1988).

Object Diagrams

It is generally accepted that students perform better on code comprehension questions when they draw diagrams while tracing the code (Thomas, Ratcliffe *et al.* 2004). Using this as a basis, the aforementioned encouraged introductory programming students to use object diagrams when tracing worked examples (Figure 2-14). Object diagrams can be used to give a snapshot of objects in a system and how their relationship changes during runtime (Thomas, Ratcliffe *et al.* 2004).



```
Person person1 = new Person("Fred",
                "Aberystwyth");
Person person2 = new Person("Bill",
                "Borth");
person2 = person1;
person1.setAddress ("Llanilar");
------------------------------------------
What is person2's address?
```

Figure 2-14 Object Diagrams (Thomas, Ratcliffe *et al.* 2004)

In the study, students were given partially completed object diagrams together with the code and questions and instructed to complete them and then answer the questions. The results of the study, however, showed unexpectedly that students showed resistance to using object diagrams when the scaffold of partially completed diagrams was taken away.

Patterns

Patterns provide a common vocabulary, documentation and learning support when developing programs (Clancy and Linn 1999). Components of a pattern include a name, motivation for use, structure, results and tradeoffs, related implementation issues, sample code, examples of pattern use and related patterns (Figure 2-15).

Pattern Name	**Uses/Application**	**Syntax**

Loop with Sentinel	You want to repeat a set of actions while a condition is true. In general, the set of actions is related to the processing of a sequence of elements or numbers. The amount of elements is unknown but the end of the sequence is indicated by a sentinel value. The elements can be read or generated.	`<INITIALIZATIONS>`
		`<SENTINEL VARIABLE INIT>`
		`while (<SENTINEL VARIABLE CONDITION>)`
		`{`
		`    <READ/GENERATION OF A SEQUENCE ELEMENT>`
		`    <PROCESS ELEMENT>`
		`    <UPDATE THE SENTINEL VARIABLE>`
		`}`

Figure 2-15 Patterns (adapted from De Barros, dos Santos Mota *et al.* 2005)

Various authors have seen potential in patterns to help students organise examples and code while learning to program (Astrachan, Berry, Cox *et al.* 1997; Wallingford 1998; Clancy and Linn 1999). Clancy and Linn (1999) surveyed various studies of pattern use in introductory programming modules and reported that the inference of patterns does not come naturally to students. Instruction that focused solely on patterns did not have the desired result and expert patterns may be inaccessible to students due to lack of experience. They also found that abstract understanding and a belief in reuse is needed to apply patterns appropriately. The terms patterns and plans are often used interchangeably (Johnson and Soloway 1984). While they have the same goal of solving recurring problems encountered in different programming tasks, a pattern can be seen as a more formalised method of recording plans.

2.4 Methods for testing algorithm comprehension

Bloom's (1956) taxonomy contains a hierarchy of cognitive levels used as a guide to measure learning objectives in a specific learning area. It has been applied by Buck and Stucki (2000) to the teaching of software development and by Buckley and Exton (2003) to the area of system maintenance. A discussion of Bloom's taxonomy and how it applies to this study follows. Lessons learnt from this section are applied to the class tests and controlled experiments discussed in Chapter 4.

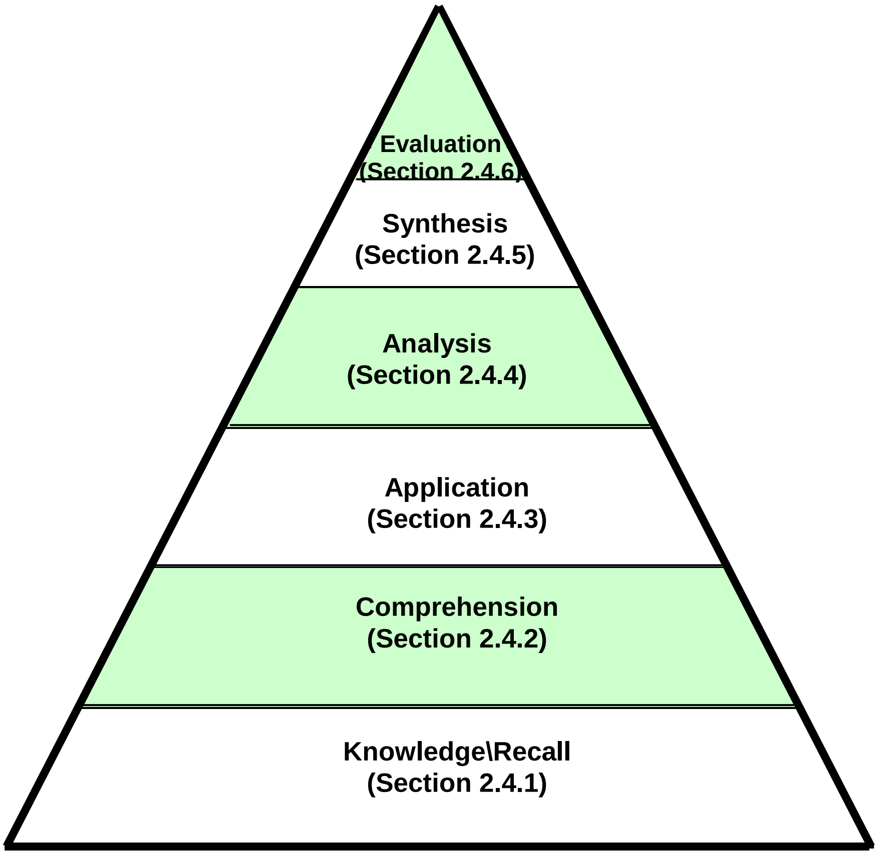


Figure 2-16 Bloom's taxonomy of educational objectives

Six distinct levels can be identified in Bloom’s taxonomy (Figure 2-16), with the levels increasing in complexity from knowledge at the bottom to evaluation at the top. Learning outcomes at higher levels in the hierarchy assumes a solid foundation in the outcomes below and often include elements of all lower levels.

2.4.1 Knowledge

At the knowledge level, students must show the ability to recall learnt information (Bloom 1956). Even though little understanding of the work learnt is necessary, this level is vital in the student’s cognitive development (Bloom 1956; Buck and Stucki 2000; Buckley and Exton 2003).

Applying this level to computer science education, Buck and Stucki (2000) suggests activities such as exposure to standard libraries and syntactic and semantic fluency in a programming language. Other activities include recalling entire algorithms (Wiedenbeck, Ramalingam *et al.* 1999) and identifying functionality of programs through knowledge of certain statements in the code (O’Brien 2003).

Apart from recalling the entire algorithm, cloze procedures can also be used to test this level of cognition (Garner 2002). Cloze activities are not unique to programming. The cloze procedure is a technique widely used in reading exercises and is generally identified by instructions for students to *Fill in the gaps*. Students are often required to choose from a list of possible answers or to determine the missing answer without assistance of a list.

2.4.2 Comprehension

Comprehension refers to the ability of the student to grasp the semantic meaning of the material learnt (Bloom 1956).

Buck and Stucki (2000) and Buckley and Exton (2003) mapped the following computer science activities to the comprehension level of Bloom’s taxonomy:

- Tracing algorithms
- Predicting the results of a program
- Translating the program to a flowchart or another language

A typical type of question from the comprehension level will include questions to predict the value of variables at certain points in a program based on the logic of the algorithm. The algorithm is usually given with an array and a comment in the code instructing students to show the contents of the array at that point. Another type of question will involve showing an algorithm to the student with the instruction to explain, in their own words, what the algorithm does or what specific lines in the algorithm do.

2.4.3 Application

At the application level, students are required to use existing knowledge in new situations. Students must exhibit the ability to apply rules, techniques, concepts, principles, laws and theories to a variety of problem scenarios (Bloom 1956).

Buckley and Exton (2003) suggest that in the context of software maintenance, programmers who exhibit competency at this level should be able to re-use parts of the system during maintenance tasks. The use of library components to implement methods that satisfy a specification can be used to assess competence at the application level of Bloom’s taxonomy (Buck and Stucki 2000).

2.4.4 Analysis

At the analysis level, students must display the ability to dissect material into component parts. Students must also be able to identify component parts, analyse relationships between parts and recognise the organisational structure of a topic (Bloom 1956).

Applying this level to computer science activities, Buck and Stucki (2000) suggest comprehending the code with a goal of modifying the functionality of a system, analysing the performance of algorithms and debugging. In introductory programming, Buckley and Exton (2003) map questions that assess variables and their relationships to the analysis level. Conceptual diagrams of software systems also fall in this level in Bloom’s taxonomy. Questions to assess this level of competence should include new material to ensure that students do not have an opportunity to recall analytical comments that discussed the new material. The new material could be an unseen algorithm that students see for the first time and questions could include an analysis of the algorithm to determine the function.

2.4.5 Synthesis

Bloom (1956) defines competence at the synthesis level to be the ability to construct a new complete structure from different parts. The main emphasis of this level is the formulation of new patterns or structures through creativity.

Buckley and Exton (2003) suggest extending the functionality of an existing system as a test to measure competence at this level. Buck and Stucki (2000) map the following computer science activities to the synthesis level:

- Develop an Application Programming Interface
- Design an Abstract Data Type
- Design fully functional applications (not simple algorithms consisting of a few lines of code) from requirements.

The synthesis level also emphasizes creativity and the application of known plans to novel programming solutions.

2.4.6 Evaluation

Bloom (1956) defines the evaluation level as the ability to judge the value of the learnt material in a particular context. At this level, criteria and standards are used to form opinions about the material. The evaluation level combines some elements of all the previous phases in the hierarchy.

In computer science, Buck and Stucki (2000) map the process of judging inputs into the requirements document during systems analysis to the evaluation level. At this level, Buckley and Exton (2003) suggest getting participants to evaluate characteristics of a system and to defend their evaluation by referencing elements of the code. This type of question is very subjective and therefore, difficult to compare the responses of one participant with another. An example of questions on this level would be for a student to evaluate systems, judging possible problem areas and motivating their answers. Evaluation of input from all sources into the system could also be used to assess the evaluation level.

2.5 Conclusions

Introductory programming can be seen as having two main activities: code generation and code comprehension. A programmer with good program comprehension skills enhances his efficiency when it comes to code generation activities. Students in introductory programming are often required to read and comprehend algorithms presented in lecture notes and textbooks, and then apply the knowledge gained to program generation tasks.

Studies of student performance in introductory programming suggest that it is vital to provide students with techniques to encourage in-depth learning of introductory algorithms. Various techniques are used by educators to guide students in the formulation of an accurate mental model of introductory programs (Section 2.3). Bloom’s taxonomy provides a means of probing students’ mental models to determine whether students achieved the required learning objectives. It can be applied to the study of in-depth code comprehension to determine if the technique advocated in the teaching model of introductory programming encouraged in-depth learning (Section 2.5).

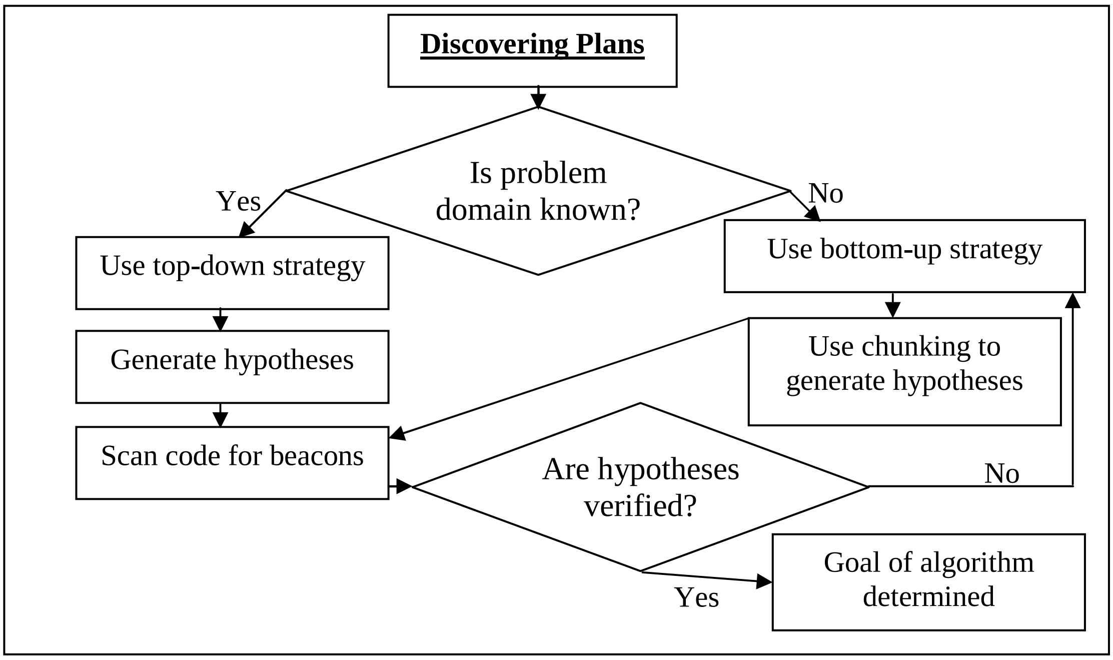


Figure 2-17 The use of beacon recognition and chunking to discover plans

The recognition and novel use of plans emerged as one of the major elements of the mental model of a programmer investigating source code, and as an indication of knowledge on the in-depth level (Section 2.2). Therefore, the problem of developing efficient program comprehension skills in students becomes one of aiding students to discover plans in an algorithm (Figure 2-17).

From the investigation into strategies used by experts to comprehend code, the recognition of beacons and the chunking of code emerged as widely used techniques. Beacon recognition is extensively used to verify hypotheses created during a top-down program comprehension strategy, if the problem domain is familiar to the programmer. In an unfamiliar problem domain, a bottom-up strategy is employed. In this strategy, chunking is used to generate hypotheses. Any hypothesis not verified prompts the programmer to resort to chunking to develop new hypotheses. Once all hypotheses are verified, all plans are discovered and the goal of the program is known.

By providing students with an environment that supports beacon recognition and chunking, a network of plans can be stored in the long-term memory of the student. Expert programmers frequently use plans to construct programs. The problem for a student in introductory programming is that they have not built sufficient plans in long-term memory to effectively use them. As a result, it seems necessary to build the long-term memory of introductory students with plans. In addition to building their long-term memory with plans, the techniques used to build these plans must be ones that students can use to discover plans on their own after initial instruction. These techniques must also be used by expert programmers in their program comprehension activities.

The discussion in this chapter culminates in a proposal for an alternative teaching model based on techniques used by experts to discover plans during code comprehension activities. Since the discovery of plans relies extensively on the recognition of program beacons and chunking, these techniques are included in the teaching model to support in-depth comprehension of introductory algorithms. Table 2-1 lists requirements for a teaching model to aid the in-depth comprehension of introductory programming algorithms.

	Requirements to support in-depth comprehension of algorithms	**Section**
R1	Allows student to extract semantic information (plans) from worked example	2.2.2
R2	Promotes the use of chunking to learn an unfamiliar algorithm	2.2.2
R3	Coaches students to spot beacons in different algorithms	2.2.2
R4	Provides students with syntactically and semantically correct worked examples	2.3.1
R5	Engages students in active learning activities	2.3.1
R6	Increases ability to recall the semantics of an algorithm	2.4.1
R7	Enables students to successfully transfer knowledge	2.4.2

Table 2-1 Requirements to aid the in-depth comprehension of introductory algorithms

A technological learning support tool can be included in the teaching model to aid code comprehension activities in introductory programming. Table 2–1 emphasises requirements that should be evident in learning support tools that aids introductory programming students in in-depth program comprehension activities. These requirements are derived from investigations into expert programmer behaviour when studying code and educators in introductory programming. This list of requirements (Table 2–1) forms the basis for the rest of the dissertation. Table 2–1 specifically serves as a measuring instrument to review existing support tools used by educators in introductory programming (Chapter 3). Table 2–1 further forms the framework for the investigative study described in Chapter 4, the results of which are reported on in Chapter 5. In particular, requirements 6 and 7 (***R6, R7***) are determined by means of an empirical investigation which involves class tests and controlled experiments.

Chapter 3 Educational Technological Support for In-Depth Comprehension of Introductory Algorithms

3.1 Introduction

The requirements for in-depth code comprehension of introductory algorithms are derived in Chapter 2. These requirements can be used as the selection criteria for a technological support tool that can be incorporated into introductory programming. Chapter 3 expands on the requirements for support tools in introductory programming modules (Section 3.2) and reviews different types of electronic tools used by educators to support the in-depth comprehension of introductory algorithms (Section 3.3). The requirements established in Chapter 2 are used as an evaluation instrument to determine the level of support for these requirements supported by each tool. The experimental tool investigated in this study and how it meets the requirements established in Chapter 2 is also discussed (Section 3.4).

3.2 Requirements for support tools

The challenge for educators is to create a learning environment that goes beyond the simple reading of example code. The learning environment should emphasize reading, comprehension and modification of working programs (Van Merriënboer 1990). Techniques are required to encourage a methodical investigation of code, which is more effective than a haphazard approach (Robbilard, Coelho and Murphy 2004).

One of the major challenges for educators in introductory programming is to provide interactive tools that favour learning (Giannotti 1987). The majority of commercial software comprehension tools are aimed at comprehending large-scale software systems. These comprehension tools, such as CFlow (Poznyakoff 2005), Rigi (Müller, Tilley, Orgun *et al.* 1992) and LCLint (Evans, Guttag, Horning *et al.* 1994) are not suitable for use by students. Since the algorithms taught at introductory level are relatively small, large-scale technological support tools fall outside the scope of this study.

A major contribution of software comprehension tools is to aid the human thinking process (Walenstein 2002). Chapter 2 investigated some cognitive aspects of programmers while they are studying an algorithm. It was established that worked examples are a favoured technique by introductory programming educators to introduce programming concepts to their students (Section 2.3). A tool to aid comprehension of these worked examples should, therefore, guide students' thinking processes. The tool must further facilitate a student to achieve the goal or action without the support in the future (Guzdial 1994). For the purpose of the proposed teaching model, a tool to support comprehension of introductory algorithms needs to meet at least the following requirements derived from Chapter 2 and summarised in Table 3.1.

	Requirements to support in-depth comprehension of introductory algorithms
R1	Allows student to extract semantic information (plans) from worked example
R2	Promotes the use of chunking to learn an unfamiliar algorithm
R3	Coaches students to spot beacons in different algorithms
R4	Provides students with syntactically and semantically correct worked examples
R5	Engages student in active learning activities
R6	Increases ability to recall the semantics of an algorithm
R7	Enables students to successfully transfer knowledge

Table 3-1 Requirements to aid the in-depth comprehension of introductory algorithms

Requirement 1 *(R1)*: Students are required to read a vast amount of worked examples as part of an introductory programming module. It is important for the students to extract appropriate semantic information (plans) from these worked examples (Section 2.2). Plans can then be used across problem domains to develop novel solutions.

Requirement 2 *(R2)*: The use of chunking has been seen to aid programmers using the bottom-up approach when studying algorithms in unfamiliar domains (Section 2.2). Grouping together statements and assigning either a mental or written annotation to related statements also minimises the cognitive load on the programmer. The chunks become knowledge that can be used across different problem domains.

Requirement 3 *(R3)*: Beacons have been shown to aid programmers when answering hypotheses as part of a top-down approach to understanding unfamiliar code. Programmers using the top-down method of code comprehension are usually capable of generating hypotheses when they are familiar with the problem domain. Beacons guide the verification process of these hypotheses. Students do not have an adequate repository of beacons in their long-term memory and need assistance in discovering them (Section 2.2).

Requirement 4 *(R4)*: The worked examples presented to the student must be of such a standard that it is possible to determine syntax rules from the code (Section 2.3). Adhering to standard programming conventions will make it easier to extract semantic information from the worked example. Special care must be taken to ensure the presented solution is semantically correct.

Requirement 5 *(R5)*: Students who simply read an algorithm are less successful than those actively engaged with the code (Section 2.3). Engagement with the algorithm can take the form of completing missing lines, debugging code and answering questions about the semantics of the code. When students are allowed to interact with the algorithm in these ways, it forces them away from using rote-learning techniques.

Requirement 6 *(R6)*: Rote learning an algorithm would result in a student regurgitating code line by line while disregarding slight changes in the problem scenario. Recalling an algorithm and making the necessary changes to suit the difference in problem scenario is evidence of in-depth learning (Section 2.4) and provides an indication that the student has grasped the semantics of the code when it was studied. Chunking is a technique that helps with discovering the semantic plans in the code.

Requirement 7 *(R7)*: If a student studied an algorithm at the in-depth level, (s)he would be capable of using that knowledge in future code comprehension activities (Section 2.4). A repository of plans should be built up in long-term memory while studying algorithms. This would enable the efficient discovery of plans in similar algorithms, and re-use of these plans during code generation exercises. A support tool is also deemed successful in this requirement if students exhibit a tendency to continue with the techniques learnt using the support tool, even after the tool is not used anymore.

The requirements summarised in Table 3-1 will be used to guide the investigation into existing tools used to support in-depth code comprehension.

3.3 Investigation of existing support tools

A study of existing support tools reveals a number of different types of support tools used by educators in introductory programming[1]. These tools can be classified as either code generation tools or code comprehension tools. Since this study focuses on comprehension of introductory programming algorithms, this section will only discuss the latter type of tools. The tools of the type that aid comprehension are further classified as algorithm animation tools (Section 3.3.1), program tracing and debugging tools (Section 3.3.2), programming construct visualisation tools (Section 3.3.3) (also called multimedia learning objects) and flowchart tools (Section 3.3.4).

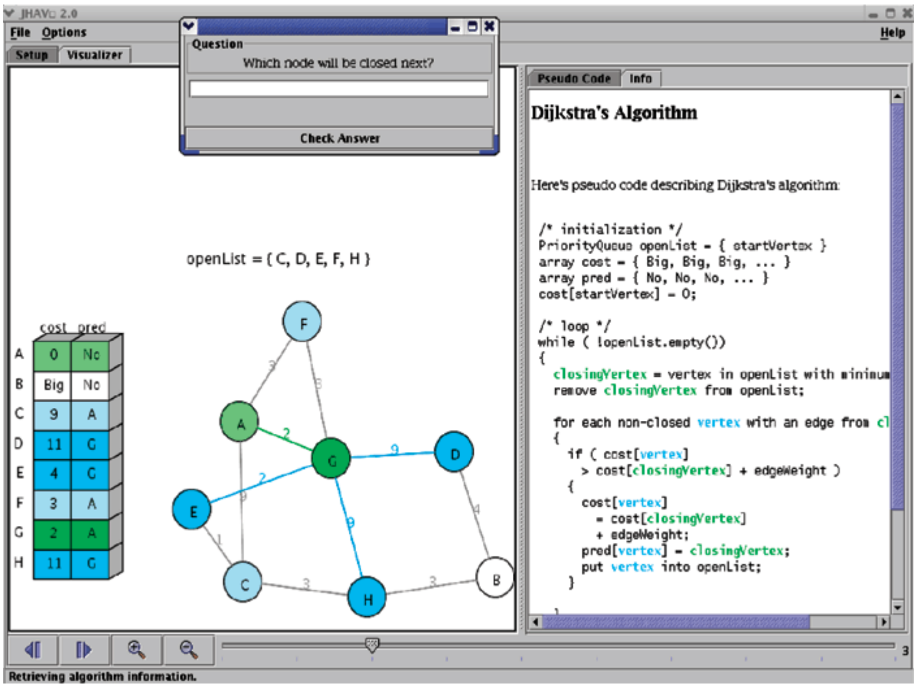
3.3.1. Algorithm Animation Tools

Algorithm animation tools to aid comprehension of algorithms have been in use since the eighties (Brown 1988). Animations are used to demonstrate the behaviour of an algorithm during runtime (McDonald and Ciesielski 2002). A set of animated images

[1] (Naps, Eagan *et al.* 2001; Cross, Hendrix *et al.* 2002; Boyle 2003; Naps 2005; Areias and Mendes 2006; Yeh, Greyling *et al.* 2006)

are shown to demonstrate important events, for instance when the algorithm updates a data structure. A study of algorithm animation tools reports that the result of experiments of their effectiveness in terms of comprehension is not always significantly favourable (Wilson, Katz, Ingargiola *et al.* 1995). Another study indicates that animation tools lose their effectiveness over time, with students comprehending the algorithm while it is shown, but seemingly forgetting aspects of the algorithm after the animation has ended (Stasko, Badre and Lewis 1993). It was also shown that experienced programmers stand to gain more from animations than students, since animations serve more a role of reinforcing and clarifying the knowledge of the algorithm after it has already been understood by the programmer. Textual descriptions of the algorithm would add to the effectiveness of the algorithm (Stasko, Badre *et al.* 1993).

Java-Hosted Algorithm Visualization Environment (JHAVÉ) (Figure 3-1) is an example of an algorithm animation tool used to learn sorting algorithms (Naps, Eagan *et al.* 2001; Naps 2005).



Question

Which node will be closed next?

openList = { C, D, E, F, H }

	cost	pred
A	0	No
B	Big	No
C	9	A
D	11	G
E	4	G
F	3	A
G	2	A
H	11	G

Dijkstra's Algorithm

Here's pseudo code describing Dijkstra's algorithm:

```
/* initialization */
PriorityQueue openList = { startVertex }
array cost = { Big, Big, Big, ... }
array pred = { No, No, No, ... }
cost[startVertex] = 0;

/* loop */
while ( !openList.empty())
{
  closingVertex = vertex in openList with minimum
  remove closingVertex from openList;

  for each non-closed vertex with an edge from cl
  {
    if ( cost[vertex]
      > cost[closingVertex] + edgeWeight )
    {
      cost[vertex]
        = cost[closingVertex]
        + edgeWeight;
      pred[vertex] = closingVertex;
      put vertex into openList;
    }
```

Retrieving algorithm information.

Figure 3-1 Screenshot of JHAVé (Naps, Eagan *et al* 2001)

One of the key findings of a study investigating the effectiveness of this tool is that students' comprehension increases with their level of engagement with the code and the animations. From the student's perspective, the JHAVÉ interface provides explanations of the algorithm in pseudocode in a pane on the right hand side while the animation is viewed in the left hand pane. Students can use controls to step forward and back through the algorithm. Occasionally during the animation a dialogue box appears that forces a student to think about the algorithm by posing a question and requiring feedback interactively.

JHAVÉ is an algorithm animation tool that helps students to understand the functionality and mechanics of the algorithm. Support for the requirements to aid in-depth comprehension of introductory algorithms is summarised in Table 3-2. Individual plans that are used to construct the algorithm are explained in pseudocode ***(R1)***. No evidence could be found of the use of chunking to learn an unfamiliar algorithm, nor of recognising beacons in different algorithms ***(R2, R3)***. JHAVÉ provides students access to a syntactically and semantically correct worked example ***(R4)***. JHAVÉ provides a quiz and the ability to step through the algorithm to allow some level of interaction ***(R5)***. No evidence of the students' ability to transfer knowledge and recall an algorithm was found in available literature about JHAVÉ ***(R6, R7)***.

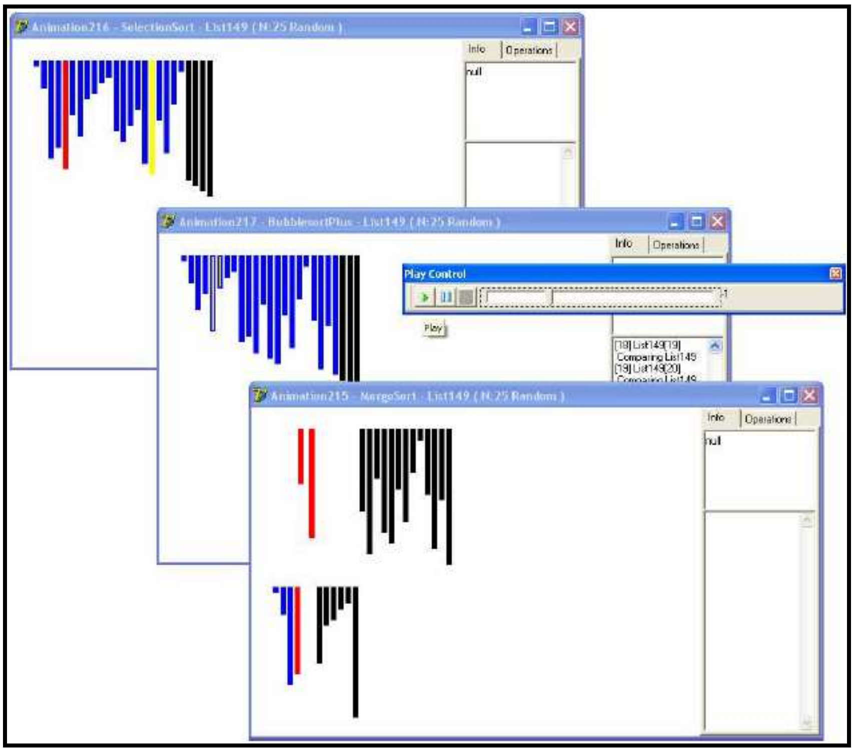
	Requirement	**Supported?**
R1	Allows students to extract semantic information (plans) from worked example	✓
R2	Promotes the use of chunking to learn an unfamiliar algorithm	
R3	Coaches students to spot beacons in different algorithms	
R4	Provides students with syntactically and semantically correct worked examples	✓
R5	Engages student in active learning activities	✓
R6	Increases ability to recall an algorithm	
R7	Enables students to successfully transfer knowledge	

Legend

✓	Requirement is supported
No entry	Inconclusive evidence in available literature in support of requirement

Table 3-2 Support for comprehension requirements in JHAVÉ

A prototype algorithm animation tool (Figure 3-2) was developed in the Department of CS&IS at NMMU as a proof of concept for an extensible framework for algorithm animation systems (Yeh, Greyling *et al.* 2006). The prototype animation tool allows students to view and compare the speed of sorting algorithms in real time. Students are able to set the speed of the algorithm, supply data and construct animations ***(R5)***. The animation tool can be used to visually explain the plans in the algorithm ***(R1)***. Since the code is not displayed at the same time that the animation is running, support for ***R2 – R4*** is not evident (Table 3-3). An investigation into the effectiveness of this tool has not yet been conducted ***(R6, R7)***.


Animation216 - SelectionSort - List149 (N:25 Random)
Info
Operations
null
Animation217 - BubblesortPlus - List149 (N:25 Random)
Info
Operations
Play Control
1
Play
Animation215 - MergeSort - List149 (N:25 Random)
Info
Operations
null

[18] List149[19]
Comparing List149
[19] List149[20]
Comparing List149

Figure 3-2 A prototype algorithm animation tool based on an extensible framework

	Requirement	Supported?
R1	Allows students to extract semantic information (plans) from worked example	✓
R2	Promotes the use of chunking to learn an unfamiliar algorithm	X
R3	Coaches students to spot beacons in different algorithms	X
R4	Provides students with syntactically and semantically correct worked examples	X
R5	Engages student in active learning activities	✓
R6	Increases ability to recall an algorithm	
R7	Enables students to successfully transfer knowledge	

Legend

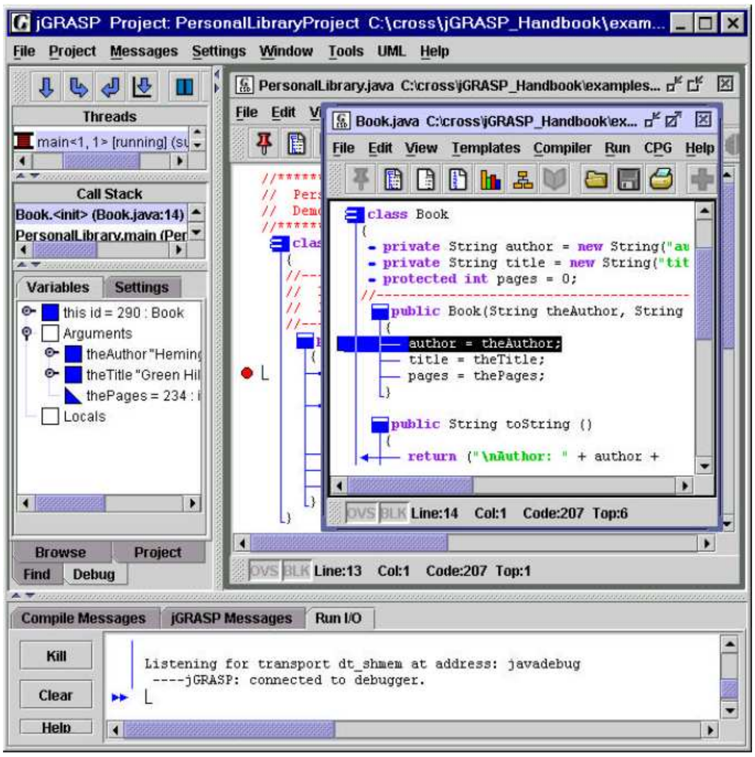
✓	Requirement is supported
X	Requirement is not supported
No entry	Inconclusive evidence in available literature in support of requirement

Table 3-3 Support for comprehension requirements in prototype animation tool

3.3.2. Program Tracing and Debugging Tools

Debugging is a vital component of software development and a key challenge that should be addressed in an introductory programming module (Mathis 1974). However, debuggers are often neglected by instructors of introductory programming or limited to a tool for finding errors in source code (Cross, Hendrix *et al.* 2002). Experiences by Cross, Hendrix *et al.* (2002) suggest that debuggers can be successfully used in introductory programming modules to improve the effectiveness of their teaching and the in-depth comprehension of object-oriented algorithms.

The visual debugger used in Cross, Hendrix *et al.(*2002) is integrated with a Java IDE and called jGRASP (Figure 3-3).


jGRASP Project: PersonalLibraryProject C:\cross\jGRASP_Handbook\exam...
File Project Messages Settings Window Tools UML Help
Threads
main<1, 1> [running] (su
Call Stack
Book.<init> (Book.java:14)
PersonalLibrary.main (Per
Variables Settings
PersonalLibrary.java C:\cross\jGRASP_Handbook\examples...
File Edit Vi
Book.java C:\cross\jGRASP_Handbook\ex...
File Edit View Templates Compiler Run CPG Help
OVS BLK Line:14 Col:1 Code:207 Top:6
OVS BLK Line:13 Col:1 Code:207 Top:1
Browse Project
Find Debug
Compile Messages jGRASP Messages Run I/O
Kill
Clear
Help

this id = 290 : Book
Arguments
theAuthor "Heming
theTitle "Green Hil
thePages = 234 : i
Locals

```
class Book
{
 private String author = new String("au
 private String title = new String("tit
 protected int pages = 0;
//----------------------------------------
  public Book(String theAuthor, String
  {
    author = theAuthor;
    title = theTitle;
    pages = thePages;
  }

  public String toString ()
  {
    return ("\nAuthor: " + author +
```

```
Listening for transport dt_shmem at address: javadebug
 ----jGRASP: connected to debugger.
```

Figure 3-3 jGRASP with associated debugger (Cross, Hendrix *et al.* 2002)

This debugger is used to step through a worked example in contact sessions. The visual debugger explicitly shows how a constructor works and how some methods are implicitly invoked. jGRASP also shows how objects are created, updated and deconstructed. Thus, the debugger shows information normally not visible to the students, thereby aiding their in-depth comprehension processes. Code highlighting and Control Structure Diagrams are also used to guide students' attention.

While jGRASP is useful to show students the effect of the code, is does not explicitly show how plans are used to solve specifications in the problem domain ***(R1)*** (Table 3–4). It also does not promote the use of chunking to learn an unfamiliar algorithm or how to use beacons to verify hypotheses ***(R2, R3)***. It does show syntactically and semantically correct worked examples and makes it easier for students to navigate the control flow of the algorithm by using code highlighting and control structure diagrams ***(R4)***. Students use the debugger to fix errors in the code, so interactive learning is possible ***(R5)***. The jGRASP support tool increases students' ability to recal an algorithm and enables students to successfully transfer knowledge (Cross, Hendrix *et al.* 2002) ***(R6, R7)***.

	Requirement	**Supported?**
R1	Allows students to extract semantic information (plans) from worked example	
R2	Promotes the use of chunking to learn an unfamiliar algorithm	
R3	Coaches students to spot beacons in different algorithms	
R4	Provides students with syntactically and semantically correct worked examples	✓
R5	Engages student in active learning activities	✓
R6	Increases ability to recall an algorithm	✓
R7	Enables students to successfully transfer knowledge	✓

Legend

✓	Requirement is supported
No entry	Inconclusive evidence in available literature in support of requirement

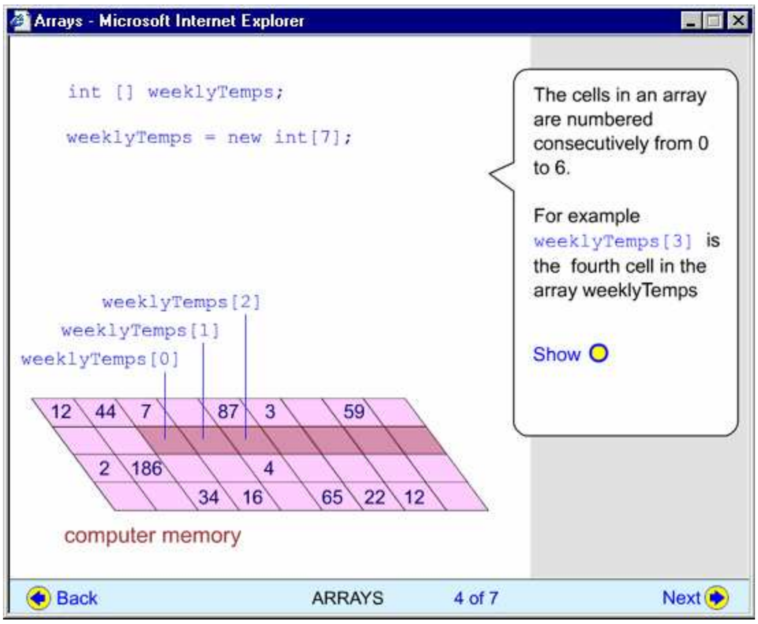
Table 3-4 Support for comprehension requirements in jGRASP

3.3.3. Programming Construct Visualisation (PCV)

Learning objects that use multimedia can be used to cover a wide variety of basic programming constructs, one at a time (Boyle 2003). An example of a multimedia learning object can include a tool to understand a basic looping construct (while loop), or one that illustrates a decision-making construct (if-statement). The main goal of these objects is to improve students' in-depth comprehension of various programming constructs.

The risk of using a learning object to introduce one construct at a time is that it is unsuited to in-depth learning (Wiley 2003) and only promotes learning of surface features. The tool becomes a vehicle to provide decontextualised content in a passive way. An advantage of this type of tool is that students can work through and comprehend each construct at their own pace and it helps them form a more concrete mental representation of each construct.

Figure 3-4 shows a multimedia object that illustrates how the code that defines and initialises an array is implemented in computer memory. The student controls the speed of the animation by clicking "Show" to advance to the next step. Student control is useful to promote engagement and aid cognitive development (Boyle 2003).



```
int [] weeklyTemps;

weeklyTemps = new int[7];
```

The cells in an array are numbered consecutively from 0 to 6.

For example `weeklyTemps[3]` is the fourth cell in the array weeklyTemps

Figure 3-4 Illustrating relationship between arrays and memory

The only two requirements that these multimedia learning objects meet are the provision of active learning activities ***(R5)*** and that it helps with transfer of knowledge ***(R7)*** (Table 3-5). Multimedia learning objects allows students to interact with the object and students gain valuable knowledge of program domain constructs that can be used in program generation and comprehension activities. From the available literature, there seems to be no worked examples to provide a context for these constructs ***(R4)***, which makes it impossible to incorporate extraction of plans, chunking and beacon recognition ***(R1, R2, R3).*** No evidence could be found on their ability to recall an algorithm ***(R6)***.

	Requirement	**Supported?**
R1	Allows students to extract semantic information (plans) from worked example	
R2	Promotes the use of chunking to learn an unfamiliar algorithm	
R3	Coaches students to spot beacons in different algorithms	
R4	Provides students with syntactically and semantically correct worked examples	
R5	Engages student in active learning activities	✓
R6	Increases ability to recall an algorithm	
R7	Enables students to successfully transfer knowledge	✓

Legend

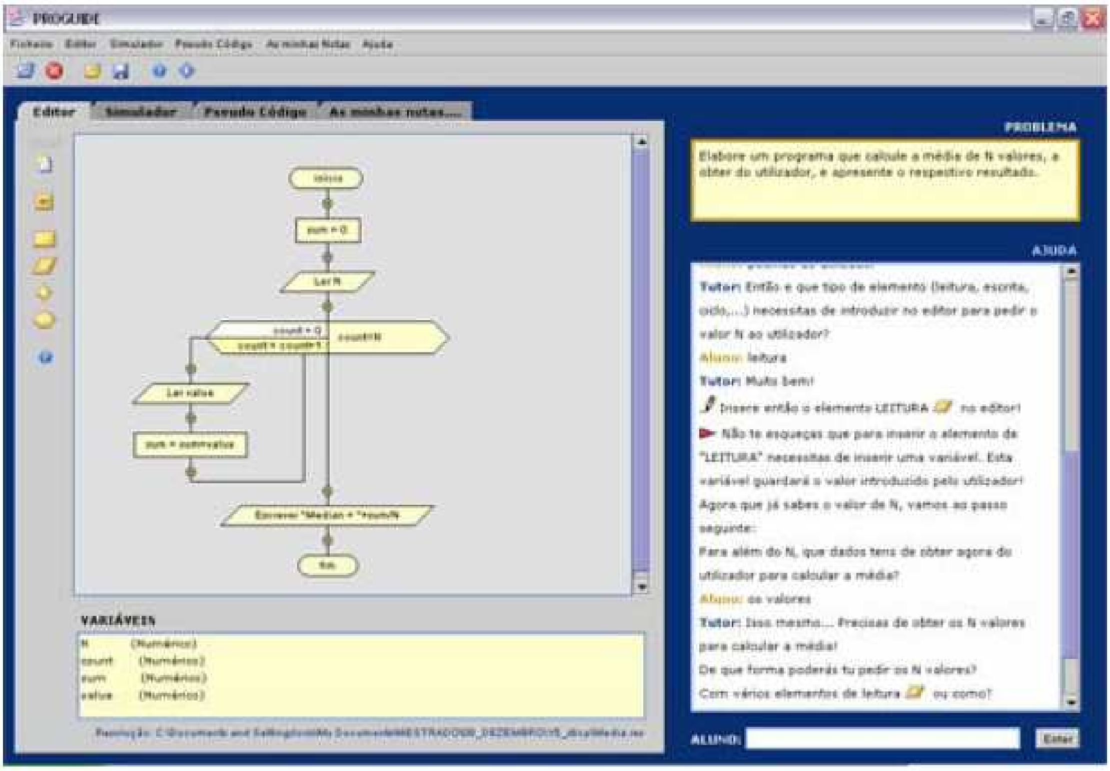
✓	Requirement is supported
No entry	Inconclusive evidence in available literature in support of requirement

Table 3-5 Support for comprehension requirements in program construct visualisation tools

3.3.4. Flowchart Tools

In the early stages of learning how to program, students often find it difficult to create solutions to problems on their first attempt (Mendes and Marcelino 2006). Flowcharts are often used to create solutions by dragging and dropping icons that visually represent programming constructs onto a window (Cilliers 2004; Cilliers, Calitz *et al.* 2005; Mendes and Marcelino 2006). Flowcharts reduce the level of precision and the manual typing required in textual programming environments. The flowchart is usually accompanied by a textual window that shows the underlying code of the constructed flowchart. Flowcharts convey the semantics of an algorithm. Despite an earlier study that reported no significant difference in the ability of students to comprehend code with or without flowcharts (Shneiderman 1983), flowcharts is a useful technique to generate code, especially for students with a higher risk of failing introductory programming (Cilliers, Calitz *et al.* 2005).

Figure 3-5 shows PROGUIDE (Areias and Mendes 2006), a support tool used to guide students in natural language to generate a solution to a problem by using icons to create a flowchart of the solution.



VARIÁVEIS

N (Numérico)
count (Numérico)
sum (Numérico)
value (Numérico)

Elabore um programa que calcule a média de N valores, a obter do utilizador, e apresente o respectivo resultado.

Tutor: Então e que tipo de elemento (leitura, escrita, ciclo,...) necessitas de introduzir no editor para pedir o valor N ao utilizador?

Aluno: leitura

Tutor: Muito bem!

Insere então o elemento LEITURA no editor!

Não te esqueças que para inserir o elemento de "LEITURA" necessitas de inserir uma variável. Esta variável guardará o valor introduzido pelo utilizador!

Agora que já sabes o valor de N, vamos ao passo seguinte:

Para além do N, que dados tens de obter agora do utilizador para calcular a média?

Aluno: os valores

Tutor: Isso mesmo.... Precisas de obter os N valores para calcular a média!

De que forma poderás tu pedir os N valores?

Com vários elementos de leitura ou como?

Figure 3-5 PROGUIDE flowchart tool (Areias and Mendes 2006)

Most of the requirements for in-depth comprehension is evident in PROGUIDE (Table 3–6).

	Requirement	Supported?
R1	Allows students to extract semantic information (plans) from worked example	✓
R2	Promotes the use of chunking to learn an unfamiliar algorithm	
R3	Coaches students to spot beacons in different algorithms	
R4	Provides students with syntactically and semantically correct worked examples	✓
R5	Engages student in active learning activities	✓
R6	Increases ability to recall an algorithm	✓
R7	Enables students to successfully transfer knowledge	✓

Legend

✓	Requirement is supported
No entry	Inconclusive evidence in available literature in support of requirement

Table 3-6 Support for comprehension requirements in PROGUIDE

After a flowchart is created, PROGUIDE translates the solution into pseudocode, JAVA and C ***(R4)***. Natural language prompts guide the student while building the solution ***(R5)***. Educators can create the flowcharts and students can use the icons and natural language prompts to discover the plans in the algorithm ***(R1)***. Students can make changes to the code by moving icons around. Since the focus is on learning semantics instead of syntax, students are able to successfully transfer knowledge ***(R7)*** (Areias and Mendes 2006). This also increases their ability to recall an algorithm ***(R6)***. From the available literature, it is not clear whether PROGUIDE promotes the use of chunking to learn an unfamiliar algorithm or coach a student to spot beacons in different algorithms ***(R2, R3)***.

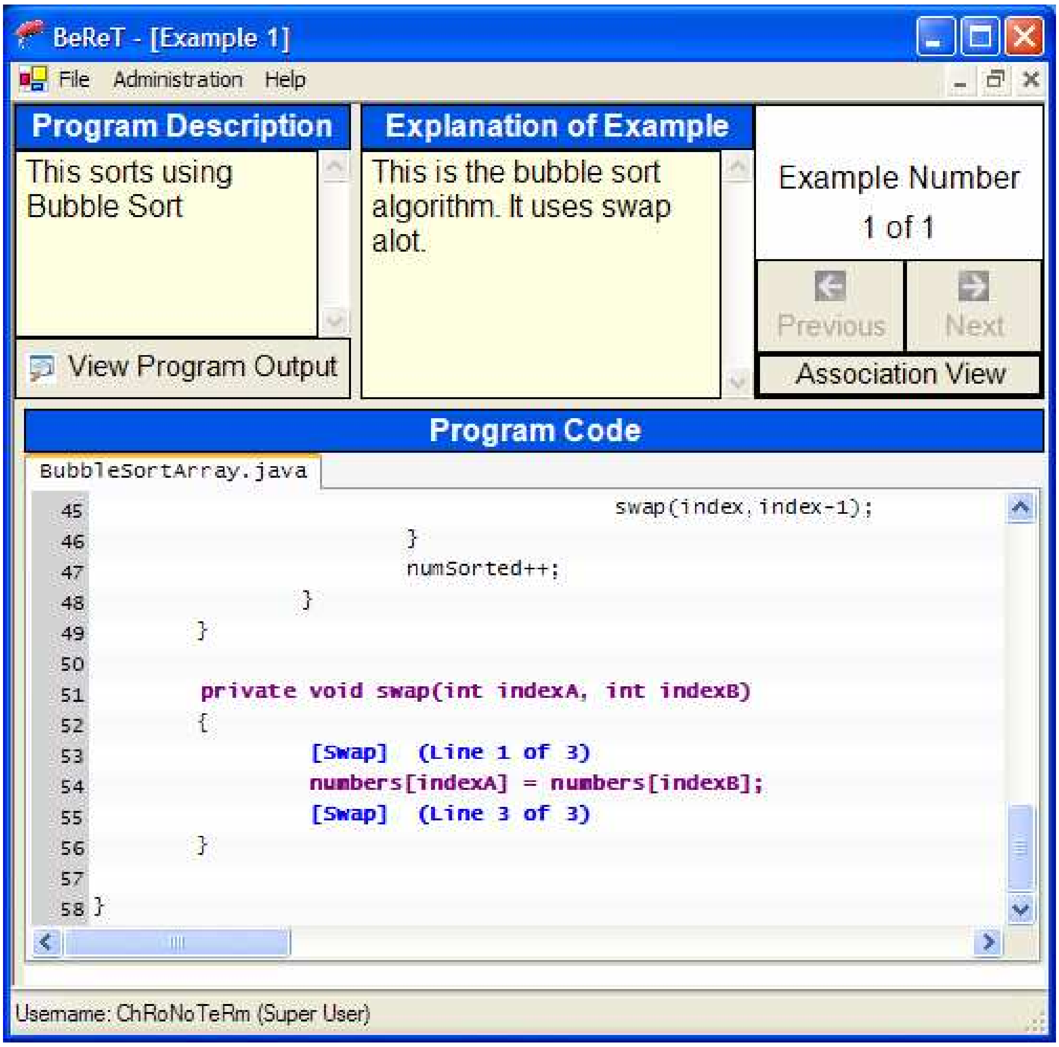
3.4 The Beacon Recognition Tool

The **Be**acon **Re**cognition **T**ool (BeReT) is a tool that helps students recognise and identify plans in the source code (Harris 2005; Harris and Cilliers 2006). The support tool provides tutorials for students that present semantic information (plans) from worked examples (Section 3.4.1). BeReT has exercises that promote the use of chunking to learn an unfamiliar algorithm and it coaches students to spot beacons in different algorithms (Section 3.4.2). The worked examples can be loaded dynamically and are syntactically and semantically correct. Engagement with the worked examples takes the form of code completion, chunking and the definition of beacons.

3.4.1. Tutorials

A tutorial is used to guide students to discover plans in the code and understand how they match the goals of the algorithm ***(R1)***. Students often fail to understand what they see (Mendes and Marcelino 2006). This can be solved by explanatory notes that accompany the algorithm. The tutorial feature in BeReT includes these descriptions of the plans implemented in the algorithm (Figure 3-6).

Each tutorial presents students with a syntactically and semantically correct worked example ***(R4)***. The educator can load an algorithm in any programming language. Since the focus of the tutorial is to help students comprehend the algorithm, chunking is used to guide students through a bottom-up process of comprehension ***(R2)***.



This sorts using Bubble Sort

This is the bubble sort algorithm. It uses swap alot.

```
                                        swap(index, index-1);
                        }
                        numSorted++;
                }
        }

        private void swap(int indexA, int indexB)
        {
                [Swap]   (Line 1 of 3)
                numbers[indexA] = numbers[indexB];
                [Swap]   (Line 3 of 3)
        }

}
```

Username: ChRoNoTeRm (Super User)

Figure 3-6 BeReT Tutorial Screen (Harris 2005)

Chunking is achieved by means of highlighting relevant lines of code. The grouped statements are annotated in a separate frame and a detailed description of the semantic plan behind the highlighted lines is also given in a separate frame. Lines are highlighted by clicking on an annotation. Tutorials that focus on beacon recognition can be created in a similar fashion ***(R3)***.

3.4.2. Exercises

Student engagement is a key component of a successful learning tool (Boyle 2003; Areias and Mendes 2006). The three types of exercises supported by BeReT cater for student engagement and promote an active learning experience (Harris 2005, Harris and Cilliers 2006) ***(R5)***.

Students may be required to do any of the following three types of tasks in BeReT, namely Complete, Match and Define (Harris 2005) (Table 3-7). Exercises can be used to test the students' ability to recognise beacons or any general plan used in an algorithm. One type of exercise is based on the cloze procedure (Section 2.4.1), where students must complete a plan or beacon with statements removed. The match type of exercise is used to test students' ability to recognise the semantics of given lines of code. The define type of exercise tests students' ability to group together related statements into semantic groupings (Harris 2005). These semantic groupings must then be given a name and description.

Task	**Given**	**What must be done**
Complete	Beacon name and description	Fill in missing source code that name and description refers to
Match	Beacon name and description	Select relevant source code lines that matches name and description
Define	Code only	Group statements together and assign a name and description to indicate plan of these statements

Table 3-7 Types of exercises in BeReT

BeReT is an experimental support tool whose design and implementation decisions specifically meet requirements ***R1 – R5*** (Table 3-8). BeReT specifically addresses the lack of support for requirements ***R2*** and ***R3*** which were found to be lacking in the reviewed algorithm comprehension support tools (Section 3.3).

	Requirement	**Supported?**
R1	Allows students to extract semantic information (plans) from worked example	✓
R2	Promotes the use of chunking to learn an unfamiliar algorithm	✓
R3	Coaches students to spot beacons in different algorithms	✓
R4	Provides students with syntactically and semantically correct worked examples	✓
R5	Engages student in active learning activities	✓
R6	Increases ability to recall an algorithm	?
R7	Enables students to successfully transfer knowledge	?

Legend

✓	Requirement is supported
?	To be determined by means of an empirical investigation (Chapters 4 and 5)

Table 3-8 Support for comprehension requirements in BeReT

3.5 Conclusion

An investigation into existing support tools used by educators failed to deliver a support tool that meets all requirements for in-depth code comprehension derived in Chapter 2.

		Existing Tools					
	Requirement	**JHAVé**	**PROTOTYPE ALGORITHM ANIMATION TOOL**	**JGRASP**	**PCV**	**PROGUIDE**	**BERET**
R1	Allow students to extract semantic information (plans) from worked example	✓	✓			✓	✓
R2	**Promotes the use of chunking to learn an unfamiliar algorithm**		X				✓
R3	**Coaches students to spot beacons in different algorithms**		X				✓
R4	Provide students with syntactically and semantically correct worked examples	✓	X	✓		✓	✓
R5	Engage student in active learning activities	✓	✓	✓	✓	✓	✓
R6	Increases ability to recall an algorithm			✓		✓	?
R7	Enable students to successfully transfer knowledge			✓	✓	✓	?

Legend

✓	Requirement is supported
X	Requirement is not supported
No entry	Inconclusive evidence in available literature in support of requirement
?	To be determined by means of an empirical investigation (Chapters 4 and 5)

Table 3-9 Comparison of support for algorithm comprehension evident in various tools

While some tools meet some of the requirements, no tool was found with support for requirements ***R2*** and ***R3*** specifically (Table 3-9). In response, an experimental support tool (BeReT) was developed to specifically meet these requirements, amongst others.

The decisions made in the design and implementation of BeReT takes into consideration requirements ***R1*** to ***R5***. The lack of empirical evidence that shows support for requirements ***R6*** (Increases ability to recall an algorithm) and ***R7*** (Enable students to successfully transfer knowledge) is addressed in a between-groups experiment (Chapter 4).

Chapter 4 Investigative Research Methodology

4.1 Introduction

Chapter 3 failed to uncover a single support tool for in-depth comprehension that meets all of the requirements derived in Chapter 2. In response to the lack of such support tools, BeReT was developed as an appropriate tool that supports most of the requirements for a tool to aid program comprehension in an introductory programming module. Chapter 3 also identified a lack of evidence to show that BeReT increases students' ability to recall an algorithm ***(R6)*** and enables them to transfer knowledge ***(R7)***. This chapter discusses the research methodology used to find empirical evidence to address these requirements.

The context in which the investigative between-groups study takes place and the tools used during the study are presented (Section 4.2). The plan for analysing the data is presented in Section 4.3. The chapter concludes with a discussion of the risks associated with the investigative study and strategies to address these risks (Section 4.4).

4.2 Introductory Programming at NMMU

A framework for an introductory programming module includes the creation of learning resources, learning activities and learning supports (Garner 2003). Learning resources are the material providing the content of the module. Learning activities consist of tasks students have to perform to guide their learning process. Learning supports guide and provide feedback tailored for the individual. Analysing the introductory programming module at NMMU reveals such a framework (CS&IS 2006a, 2006b). Amongst other topics, the second semester of the introductory programming module presented by the Department of CS&IS at NMMU covers the tracing, evaluation and implementation of various algorithms. These algorithms include linear and binary search algorithms, as well as bubble sort and insertion sort algorithms. Algorithms for the union, intersection and difference between two lists are also taught.

The learning resources support the presentation of the topics and include the following:

- Notes provided by the lecturer during lectures. These include comprehension questions students may complete before attempting the code generation practical assignments. The comprehension questions are predominantly tracing type questions, but may include other types of comprehension questions.
- A prescribed textbook.
- Weekly practical assignment sheets. The practical assignments take the form of code generation tasks only, but students are encouraged to complete code comprehension activities prior to starting each practical assignment.

Every week a new topic is introduced by way of learning activities. Students registered for the module have three contact sessions per week. Two types of contact sessions are evident in the module, namely lectures and practical assignments.

Lectures are divided into two sessions per week, one a double lecture of 70 minutes and another lecture of 35 minutes in duration. Lectures are facilitated by an educator not directly involved with the current investigative study. Students receive weekly notes during the lectures containing a worked example illustrating the topic of that week. Students attend lectures over a period of 15 weeks during the second semester of the module.

Practical assignments are included in the module to allow students to solve problems either novel or similar to the ones discussed in the lecture of the previous week. Practical assignments traditionally comprise of only code generation tasks and no code comprehension tasks are included. A practical session is 80 minutes long and takes place in a computer laboratory. Every student has access to their own PC and practical tasks are performed using Microsoft Visual Studio as the development environment and C# as the programming language. Practical assignment sheets are made available to students and may be done either at home or on the computer facilities at the university. Students have a week to complete their practical assignment.

Technological learning support used in the introductory programming module presented by the Department of CS&IS at NMMU is limited to code generation activities. Currently, no support for code comprehension is used other than the discussions during lectures. Algorithm tracing is the main method used to explain the algorithms.

4.3 Investigative Study

The goal of this research is to measure the effect of a Beacon Recognition Tool on the in-depth comprehension of introductory programming algorithms. Data to measure this effect was collected during practical assignments, class tests, controlled experiments and by means of a questionnaire.

Both quantitative and qualitative data were collected in the study. Controlled experiments were conducted, involving a subset of the students enrolled for the introductory algorithm module. The controlled experiments were used to collect data such as accuracy, students’ ability to complete a partial algorithm and time taken to complete a code comprehension task. Eye-tracking data was also collected during the controlled experiments. A questionnaire was also completed by the students to collect qualitative data about the user satisfaction with BeReT.

Experiments assessing programme comprehension typically use two variables to measure comprehension, namely, accuracy and response time (Rajlich and Cowan 1997). Class tests are suitable to measure accuracy; however, it is difficult to accurately collect individual response times in large groups. Controlled experiments can be used to capture individual student responses and evaluate them for accuracy, and to record the time it took to answer a question or respond to an instruction.

Eye-tracking experiments have been successfully used in studies of programming (Crosby and Stelovsky 1990; Crosby, Scholtz *et al.* 2002). One study determined that expert programmers tend to focus more on beacon-like statements when trying to understand a piece of code (Crosby, Scholtz *et al.* 2002). Another study investigated the difference of viewing patterns between prose text and algorithms as well as the difference in code scanning between expert and novice programmers (Aschwanden and Crosby 2006).

In order to determine whether students built up a collection of beacons or plan knowledge to answer comprehension questions, their eye movements were recorded to collect fixations and saccades for this study. Fixations refer to the dwell time on an area of interest and saccades refer to the rapid movement of the eye between fixations (Bartels and Marshall 2006). The controlled experiment with the eye tracker involved showing students an algorithm and asking comprehension questions about the algorithm while capturing their eye movements. Each algorithm has specific areas of interest relating to beacons and plans in the code. The participants were given a list of questions to assess their comprehension of the presented algorithm(s). While studying the code on the computer screen, the eye tracker recorded their eye movements and provided data such as the following:

- The time to first fixation (to determine how long it takes them to spot a beacon)
- The length of time and the number of times areas of interest were looked at (to determine if they are looking at meaningful sections of code when studying an algorithm).

This investigative study made use of various tools and materials for specific purposes; namely a consent form, training manual, practical assignments, class tests, controlled experiments and a questionnaire.

4.3.1. Consent form

Any study that involves human participants has to be approved by the Research Ethics Committee (Human) of NMMU (RECH). The committee is made up of a group of independent experts with the responsibility to ensure that the rights of the participants are protected and that studies are conducted ethically. Therefore, an informed consent form accompanies the application to RECH (Appendix F) for permission to conduct this study with the help of the students in introductory programming. With permission granted by RECH and the consent form approved, the form is completed by the experimental group prior to the start of the study. Each point in the consent form is explained in English by the principal investigator of this study and students are encouraged to ask for clarification of any point not understood. Students initial each point in the form to show that they agree with the stated conditions.

4.3.2. Training manual

A training manual is made available to the students at the first practical assignment of the study (Appendix A). The manual makes extensive use of screenshots of BeReT to show the tutorial screen as well as the three exercise screens. The use of screenshots is a technique used with success in similar studies requiring training of a technological support system (Shih and Alessi 1993; Cilliers 2004). The different areas of the screen are numbered to guide students' focus of attention in the correct order. In addition to the training manual, a demonstration of the system is given at the start of the first practical. The demonstration is done using a data projector and takes the form of guided instruction. The principal investigator showed tasks from a dummy tutorial and exercise and students followed each task to familiarize themselves with the interface of the system. BeReT is developed to make the tutorials and exercises as intuitive as possible to reduce the training time. Each exercise starts with a splash screen, with an instruction explaining how that particular type of exercise is completed. These instructions are repeated in an area of the exercise screen as well.

4.3.3. Practical Assignments

The training aspect of the study was conducted over a three-week period during the second semester of an introductory programming module. Students in the treatment group were exposed to BeReT during three practical assignments. Introductory programming algorithms treated using the experimental support tool include the bubble sort, binary search and the insertion sort. Both groups attended their usual two lectures per week and also attended one of three possible practical sessions. For the practical assignments, the students are split into two groups, the control and treatment groups.

The practical assignments of the control group consist of pen-and-paper code comprehension questions and code generation task. The treatment group performed exactly the same code generation tasks as the control group. During practical assignments, the treatment group had exposure to the algorithm discussed in class by means of BeReT. They studied a tutorial of the algorithm first, and then performed exercises in BeReT. Appendix B shows the notes on which tutorials and exercises are based. The tutorials presented the code, a list of plans and a description of the plans behind selected lines in the code. The exercises included tasks to match code to a beacon and to complete missing beacon lines.

During the practical assignments the treatment and control groups were physically separated in two different computer laboratories. The physical separation during practical assignments effectively minimized the risk of distorting the results due to control group participants also seeing BeReT. Security is also built into BeReT so that only the treatment group participants have access to the BeReT tutorials.

For both groups, the first practical activity is to read and comprehend an algorithm presented in a lecture. The only difference is in the delivery method of the comprehension task. The control group studied the algorithm using a tracing method and answering code comprehension questions, while the treatment group used the tutorials and exercises in BeReT. The second practical activity for both groups is a typical code generation question.

4.3.4. Class tests

During the next available lecture following a particular practical assignment, a follow-up class test is administered to both the treatment and control groups (Appendix C). The class tests are used to monitor the effect of BeReT on the treatment groups' ability to complete, recall and comprehend the algorithms studied in the previous week. The purpose of the class tests is to test students' development on the knowledge and comprehension levels of Bloom's taxonomy of educational objectives (Bloom 1956). Evidence of cognitive development in these two levels will show that BeReT effectively helped students recall and comprehend the algorithms studied ***(R6, R7)***. All class tests are marked by a single individual to ensure consistency in assessment.

Both groups are encouraged to do their best in the class tests with the reward of using the class tests marks in their final class mark calculations. Preparation for the class test is only different in the presentation of code comprehension activities prior to code generation practical assignments.

4.3.5. Controlled experiments

The controlled experiment took place the week after the last algorithm was studied. The controlled experiment is designed to test the impact of BeReT with regards to accuracy and time. The controlled experiments include eye-tracking data of participants while answering comprehension questions. The same number of participants from both the treatment and control groups took part in the controlled experiment (Bartels and Marshall 2006; Bednarik and Tukiainen 2006). This procedure was necessary to ensure a valid comparison can be made between the two groups. Participants from both the treatment and control groups were requested to volunteer for the controlled experiments and a subset of students with a similar academic profile were selected from each group of volunteers (Greyling and Calitz 2003, Cilliers 2004, Vogts 2006) (Table 4-1).

Treatment Group		**Control Group**	
Participant	**1st Sem Mark**	**Participant**	**1st Sem Mark**
T1	88	***C1***	94
T2	80	***C2***	86
T3	78	***C3***	83
T4	77	***C4***	72
T5	73	***C5***	70
T6	64	***C6***	60
T7	52	***C7***	58
Mean	73.1		74.7
St Dev	11.8		13.5
p-value = 0.8206 (α = 0.05)			

Table 4-1 Academic profile of control and treatment group participants

Final marks from the first semester module were used to select and balance the participants within the experimental groups. The average first semester marks for the treatment group participants are 73.1% and for the control group 74.7%. From the p-value (0.8206) of a two-tailed pooled variance t-test, it can be concluded that there is no significant difference ($\alpha = 0.05$) in the average first semester mark of the two groups.

The controlled experiments were conducted in the usability laboratory of the Department of CS&IS at NMMU. The experiments were directed by the primary investigator and students participated in each experiment individually.

The results of the controlled experiments provide evidence to determine whether BeReT enables introductory programming students to discover plans in similar algorithms and whether it enables them to do so in a timely fashion. Evidence to show that students exhibit a tendency to continue with the techniques learnt in BeReT was also collected by means of controlled experiments. Eye-tracking data, namely fixations and saccades (the path between fixations) of participants while finding answers in the algorithm after being presented with a question, was also collected.

Experiment A required participants to recognise known plans in a previously unseen algorithm. Experiment B tested comprehension accuracy and comprehension time for a known algorithm, while recording eye-movement data. Experiment C tested participants' ability to recognise fragments of code and the participants' ability to extract semantic meaning from the code.

Experiment A

The goal of experiment A was to measure comprehension and behaviour of participants when reading a previously unseen algorithm for the first time (Appendix D). Tasks in this experiment contributed data to determine whether participants have a repository of plans and beacons in their long-term memory readily available in order to recognise known plans in an unseen algorithm.

For task one, all participants are given an algorithm that they have not studied before, but with plans similar to ones they are familiar with. Participants are instructed to study the algorithm using any method and to make notes while they are learning the algorithm. They are also instructed to let the investigator know the point at which they feel they comprehend the algorithm well enough to be tested on it. The standard quantitative data collected during the first task in the controlled experiments included correctness of participants' responses as well as time taken to answer.

For task two, participants are seated in front of a PC with an eye tracker. The equipment is calibrated to each individual participant's pupil and participants are instructed to familiarise themselves with the on screen version of the selection sort algorithm. Once the participants indicate their readiness, the algorithm is hidden and questions are asked one by one. In an attempt to restrict eye movements to the screen only, the questions are asked verbally by the investigator and responses are also given verbally. Questions in the form of "Which line or lines…" are asked and participants respond with the line number that answers the question. Using a slideshow to present the questions eliminates any distractions such as toolbars that can divert a reader's eye away from the algorithm. It also makes it easier to clear the screen between questions.

The controlled experiment protocol (Appendix G) was used to make sure that the test is conducted in a consistent way for all participants (Pretorius 2005). The Selection Sort algorithm was selected as the unseen algorithm for Experiment A. The selection sort algorithm was selected because participants have not yet been exposed to this particular sorting algorithm, but it contains plans they have seen implemented in other sorting algorithms studied, namely the bubble sort and insertion sort.

Experiment B

The goal of Experiment B was to measure comprehension and behaviour of participants when reading a known algorithm previously studied (Appendix D). The treatment group had the lecture, code generation practical and assistance of BeReT to study the algorithm as part of their preparation. The control group only had the lecture, the code generation practical and lecture notes with associated comprehension questions as part of their preparation. The experiment measured retention of learning as the main theme and also recorded the time taken to respond to comprehension questions. Fixations and saccades were also recorded to augment the accuracy and time variables in order to determine the areas of the code participants look at to determine answers and for how long the participants look at relevant areas of interest before answering specified questions.

Experiment C

The main goal of experiment C (Appendix D) was to test the ability of participants to correctly identify the plan behind the code fragments and the most likely algorithm the lines belong to. Experiment C also recorded the time taken to respond to the questions.

Certain lines from known algorithms were shown on flash cards to the participants, one code fragment at a time (Appendix D). Participants were instructed to determine what the plan or function of the code fragment is and asked to identify the algorithm most likely to contain the fragment shown. The time it takes to correctly identify the algorithm was recorded. The participants' ability to correctly describe the plan implemented in each code fragment in their own words was also recorded.

4.3.6. Questionnaire

Qualitative data in the form of a questionnaire was collected on completion of the final practical assignment (Appendix E). The questionnaire determines, from the point of view of the participants in the treatment group, the usability of BeReT, the perceived effectiveness of BeReT as a learning tool, their attitude/motivation towards code comprehension activities, and the functionality of BeReT.

A number of questions were asked to determine the experience of the treatment group when using BeReT as an algorithm comprehension tool. Some questions elicit a response based on a 5-point Likert scale; some questions a response based on a 3-point Likert scale; some elicit a yes/no response and some questions are open ended.

4.4 Method of Data Analysis

The investigation to find evidence in BeReT in support of the requirements that determine the students' knowledge of an algorithm ***(R6)*** and the students' ability to transfer their knowledge ***(R7)*** was primarily a between-groups experiment. The experiment collected data to compare the accuracy, response time and behaviour of students in an introductory programming module in the Department of CS&IS at NMMU (Section 4.4.1). Hypotheses were formulated to address the primary research question and further refined to guide data collection activities (Section 4.4.2). A plan to analyse the data collected to validate the hypotheses was also developed (Section 4.4.3).

4.4.1. Population

Students enrolled for the second semester of the introductory programming module at NMMU were divided into two groups, namely treatment and control groups. The treatment group followed the same lecture and practical schedule as the control group, but did alternative comprehension activities in BeReT. Students were randomly divided into the two groups.

4.4.2. Formulation of Hypotheses

The primary research question as stated in Chapter 1 is given as *What is the effect of a technological learning tool that supports plan discovery on the in-depth comprehension of introductory programming algorithms typically studied by novice programmers?*

This research question is decomposed into the following hypotheses which will be evaluated by means of the between-groups experiment described in this chapter:

H_0: *BeReT does not have a positive effect on the in-depth comprehension of introductory programming algorithms*

H_1: *BeReT does have a positive effect on the in-depth comprehension of introductory programming algorithms*

Below is a breakdown of the null hypothesis in order to refine the concept of positive effect in terms of in-depth comprehension.

$H_{0.1}$: *Students using BeReT do not perform better than the control group in terms of accuracy on code comprehension questions.*

$H_{0.2}$: *Students using BeReT find recalling the entire algorithm and adapting to any changes in problem scenario more difficult than students that used pen-and-paper exercises.*

$H_{0.3}$: *Students using BeReT are less accurate than the control group when completing a partial algorithm.*

$H_{0.4}$: *Students using BeReT are less successful than the control group in the discovery of known plans in previously unseen algorithms.*

$H_{0.5}$: *Students using BeReT take longer than the control group to find plans in a known and/or previously unseen algorithm.*

$H_{0.6}$: *Students using BeReT do not exhibit evidence of using beacon recognition and chunking when studying a previously unseen algorithm.*

Below is a breakdown of the alternative hypothesis, corresponding to the null hypotheses formulated above.

$H_{1.1}$: *Students using BeReT perform better than the control group in terms of accuracy on code comprehension questions.*

$H_{1.2}$: *Students using BeReT find recalling the entire algorithm and adapting to any changes in problem scenario easier than students that used pen-and-paper exercises.*

$H_{1.3}$: *Students using BeReT are more accurate than the control group when completing a partial algorithm.*

$H_{1.4}$: *Students using BeReT are more successful than the control group in the discovery of known plans in previously unseen algorithms.*

$H_{1.5}$: *Students using BeReT take less time than the control group to find plans in a known and/or previously unseen algorithm.*

$H_{1.6}$: *Students using BeReT exhibit evidence of using beacon recognition and chunking when studying a previously unseen algorithm.*

4.4.3. Techniques to analyse data

The analysis of the results is needed to validate the suitability of BeReT as support in the learning environment of an introductory programming module. Before statistical analysis can take place, the collected data needs to be prepared to ensure valid comparisons can be made between the two groups.

Participants in both the control and the treatment groups had to attend two lectures per week for three weeks, complete all practical assignments and write all class tests during the three weeks when the investigative study took place. The treatment group participants had to complete the BeReT tutorial and exercises during the practical assignments, while the control group completed pen-and-paper comprehension exercises. Any participant missing any of the data collected during this period was discarded from the comparative analysis. Students registered for the second semester of introductory programming who did not write and pass the exam during the first semester of 2006 are also excluded from the study. This was done to ensure that the same exam paper is used to base the sampling on. Only data from the remaining participants have been included in the statistical analysis to compare the performance of the two groups. The data includes results from the class tests and controlled experiments.

Both qualitative and quantitative data were collected during the investigative study. Data from class tests, controlled experiments and a questionnaire needs to be analysed using various techniques to ensure conclusions can be drawn.

The difference in average marks between the control and treatment groups obtained during class tests are tested by means of a two-tailed pooled variance t-test (Neter, Wasserman and Whitmore 1993). Since the resulting sample size of both groups is higher than 30, the assumption of normality holds (Larson 1974).

The controlled experiments results in the following data:

- Classification of different annotation techniques used by participants when reading an algorithm
- Number of participants classified according to the accuracy of their responses

- Average mark in terms of accuracy of responses
- Average time taken to respond
- Visualizations of fixations and saccades resulting from the use of an eye-tracker

The reading and annotation techniques used by the participant to study the unseen algorithm are categorised according to an adapted framework for the classification of annotations made by students while studying an algorithm (developed in Lister, Adams *et al.* (2004) and Fitzgerald, Simon *et al.* (2005)). The classification of reading techniques are statistically analysed using a Chi-squared test to determine the homogeneity of proportions.

The accuracy of responses of each participant are first categorised as follows (McTighe and Seif 2003):

- **High Level:** The participant correctly identified what the algorithm achieves on a high level. This equates to in-depth knowledge of the algorithm, since the focus is on high level semantics rather than on syntax issues. An example response in this category is: "The algorithm creates an array of sorted elements by swapping the first value in the unsorted part of the array, with the smallest value in the rest of the array."
- **Low Level:** The participant correctly explained what the algorithm achieves, but on a low level explaining what each line does. This is similar to superficial knowledge, since the focus is on low level detail of variable names used and other syntax issues. An example response in this category is: "The loop starts at index `i = 0` and terminates when `i` reaches `NrEl-1`."
- **Incorrect:** The participant has the wrong idea about the purpose of the algorithm.

The proportion of participants from each group is then tested for each category of response by means of a chi-squared test for homogeneity of proportions. The average time taken by participants to respond is tested by means of a pooled variance two-tailed t-test for the difference in the two averages from each group. The time taken to respond is measured programmatically using the eye-tracking software. A recording of the eye movements starts as soon as the algorithm is shown on the screen after each question and terminates the moment the participant starts to answer the question. The time therefore corresponds with the recording of eye-tracking data.

The average marks obtained by participants in term of accuracy during the controlled experiments are tested by means of a two-tailed pooled variance t-test (Neter, Wasserman *et al.* 1993). The average time taken by participants to respond to questions during the controlled experiments is also tested by means of a two-tailed pooled variance t-test.

The raw data resulting from the eye-tracker are converted to scanpaths and heatmaps using Begaze software, and superimposed on top of a screen shot of the algorithms studied by the participants. The scanpaths and heatmaps provide visualizations of the saccades and fixations. Average marks and time taken to answer individual questions are computed per group and tested by means of a two-tailed pooled variance t-test. Only scanpaths or heatmaps of results where a statistically significant difference exist between the two groups are reported on (Pretorius 2005).

A questionnaire is used to collect data about the effect of BeReT on the learning of introductory programming algorithms. Since only the treatment group is exposed to the treatment tool, the questionnaire is only completed by this group.

The following rules are applied for the purpose of analysing the results (Table 4-2):

- In the case of a 5-point Likert scale, a response of 4 and 5 is deemed positive and a response of 1, 2 or 3 is deemed negative. A response of 3 is not deemed neutral, but negative as is reported in a similar study (Taljaard 2003).
- In the case of a 3-point Likert scale, a response of 3 is deemed positive and a response of 1 and 2 is deemed negative.
- In some instances of the yes/no responses, no is a positive response and yes a negative response. In others, yes is deemed positive and no a negative response.
- A Chi-squared test for equality of proportions was used for testing whether the students indicated a significant positive or negative response.
- A thematic analysis was performed on open-ended questions. A theme is defined as a statement of meaning that runs through all or most of the pertinent data; or one that carries heavy emotional or factual impact (Ely, Vinz, Downing *et al.* 1999).

Nr	Question	Pos	Neg
1	BeReT is easy to use	4-5	1-3
2	Could you complete **Tutorials** in BeReT without assistance	4-5	1-3
3	If you did not answer "Yes" above, what created the difficulty with **Tutorials**?	Open ended	
4	Could you complete **Exercises** in BeReT without assistance	4-5	1-3
5	If you did not answer "Yes" above, what created the difficulty with **Exercises**?	Open ended	
6	What level of assistance did you need throughout your usage of BeReT?	4-5	1-3
7a	Did you understand the bubble sort algorithm before using BeReT?	4-5	1-3
7b	Did you understand the binary search algorithm before using BeReT?	4-5	1-3
7c	Did you understand the insertion sort algorithm before using BeReT?	4-5	1-3
8a	Did you understand the bubble sort algorithm after using BeReT?	4-5	1-3
8b	Did you understand the binary search algorithm after using BeReT?	4-5	1-3
8c	Did you understand the insertion sort algorithm after using BeReT?	4-5	1-3
9	Did BeReT help programming the algorithms easier from scratch?	4-5	1-3
10	To what level do you think BeReT helped you to better understand what certain lines do in the code?	4-5	1-3
11	To what level do you think BeReT helped you to recognise an algorithm quicker without knowing the name or function of the algorithm?	4-5	1-3
12	To what level do you think BeReT helped you to trace (i.e. show the contents of the variables at certain points) an algorithm more effectively?	4-5	1-3
13	Assuming BeReT was problem free; would you make use of it to gain a better understanding of algorithms?	3	1-2
14	Would you make use of BeReT if it immediately marked your answers and gave you feedback?	3	1-2
15	What other functionality would you like to see included in BeReT?	Open ended	
16	Did you ever run into any unexpected problems in BeReT?	No	Yes
17	If so, briefly describe the task (if possible) and BeReT's reaction.	Open ended	
18	What in your opinion is an advantage of using BeReT?	Open ended	
19	What in your opinion is a disadvantage of using BeReT?	Open ended	
20	Did BeReT encourage you to critically analyse and discuss the code in the given algorithms?	Yes	No
21	Did BeReT encourage you to ask questions about the algorithms?	Yes	No
22	Would you have spent time studying the algorithms before the practical assignments without BeReT?	Yes	No

Table 4-2 Classification of questionnaire responses as Positive or Negative

A number of risks are evident in the design of the experimental study for the current investigation. These identified risks and strategies proposed to address them is the focus of the following section.

4.5 Risks to Validity of Investigative Study

It is important to manage the risks associated with empirical studies to ensure that accurate conclusions can be made on data collected (Applin 2001). The risks identified in this study include the sample size (Section 4.5.1), the possibility of the Hawthorne Effect (Section 4.5.2), and possible problems with administering the practical learning activities (Section 4.5.3).

4.5.1. Sample Size

During the semester when the study was conducted, a total of 90 students registered for the introductory programming module. The entire population is necessary in the study for investigative purposes, to ensure the sample sizes in both groups are large enough for statistical analysis. It is one of the conditions of the RECH application to allow participants to voluntarily withdraw from the experiment. Class and practical attendance in the introductory programming module is also voluntary, which further reduces the number of possible participants for the study.

In order to minimise the risk of small sample sizes, it was decided to limit the duration of the investigative study to a period of four weeks instead of the entire semester. Since the treatment group was expected to learn how to use an experimental tool whose benefits are unknown, there was a higher probability of losing participants in this group. Therefore, the treatment participants were financially rewarded for taking part in the study. The rest of the class formed the participants in the control group.

4.5.2. The Hawthorne Effect

The Hawthorne Effect was first coined by psychologists in the 1930s during a time management study at the Hawthorne Plant of Western Electric (Parsons 1974). This study concluded that each intervention applied to the working environment of the employees had the effect of improving their morale and productivity. It was further concluded that the experiment and not the treatment was responsible for the change in attitude and behaviour. Applied to studies of computer-aided learning, two types of Hawthorne Effects are identified (Draper 1997):

- The effect of using a novel tool in teaching, and
- The effect of the extra attention to the participants in the study.

To test for the existence of the first effect, one can determine the attitude of the participants towards the tool before the start of the treatment. Any participant with a negative attitude towards the tool, but a high achievement in comprehension questions shows that there is no Hawthorne Effect of this type evident in the investigative study (Draper 1997).

The second type of effect can be reduced by letting both groups know that their results are valuable to the study (Draper 1997). Both groups wrote the class tests and selected participants from both groups were recruited for the controlled experiments. All students were made aware from the start that the results of both groups would be compared. Students were also informed at the start of the study that class tests would contribute towards their final module marks. The fact that class test marks would affect their final mark served as a motivator to the control group to also study the algorithms and class notes, and to let them know that their results also had an impact.

4.5.3. Administering Practical Learning Activities

Since BeReT is made available on the Department of CS&IS network, there could be a risk of students from the control group being able to access the tool. However, strict security is built into the system to only allow specific students access to the system. BeReT recognises the students' login username, and only usernames of participants in the treatment group was added to the table of possible users. There was also a risk of students becoming aware of fellow students working on a different system during the practical sessions. To minimise this risk, the treatment group completed their practical assignments in a separate computer laboratory away from the control group. Strict record of attendance was taken at the practical sessions to ensure that groups did not mix.

Students in the treatment group were exposed to the possibility of cognitive overload, since they needed to learn how to use BeReT in addition to the Integrated Development Environment (IDE) used during the code generation tasks. To reduce this risk, students viewed a demonstration of BeReT during the first practical session. The instructions on how to complete specific exercises are also permanently shown on the screen so that students do not need to recall the steps required to complete their tasks.

4.6 Conclusion

The lack of empirical data to validate the incorporation of BeReT into an introductory programming module in the Department of CS&IS at NMMU prompted this investigation. Evidence is specifically required to determine if BeReT increases students' ability to recall the semantics of an algorithm ***(R6)*** and if BeReT enables students to successfully transfer knowledge ***(R7)***. The hypotheses to test for these requirements are:

$H_{0.1}$: *Students using BeReT do not perform than the control group better in terms of accuracy on code comprehension questions.*

$H_{0.2}$: *Students using BeReT find recalling the entire algorithm and adapting to any changes in problem scenario more difficult.*

$H_{0.3}$: *Students using BeReT are less accurate than the control group when completing a partial algorithm.*

$H_{0.4}$: *Students using BeReT are less successful than the control group in the discovery of known plans in previously unseen algorithms.*

$H_{0.5}$: *Students using BeReT takes longer than the control group to find plans in a known and/or previously unseen algorithm quicker.*

$H_{0.6}$: *Students using BeReT do not exhibit evidence of using beacon recognition and chunking when studying a previously unseen algorithm.*

A number of data collection activities are used to verify hypotheses, which in turn are used as confirmation of support for the two requirements for which evidence was not immediately apparent in BeReT.

Students in a second semester introductory programming module in the Department of CS&IS at NMMU participated in the study over a four-week period. The students were divided into two groups (treatment and control) and during three of the four weeks students in the treatment group studied the bubble sort, binary search and insertion sort algorithms in BeReT. Control group participants studied the same algorithms and relied on lecture notes and associated pen-and-paper comprehension questions instead of BeReT tutorials and exercises. Comprehension activities performed by the control group were predominantly tracing, but some comprehension questions were also answered by the students in this group. The controlled experiments were conducted during the fourth and last week of the investigative study.

Requirement	**$H_{0.x}$**	**Data collection activity**	**Data collected**
Increases ability to recall the semantics of an algorithm ***(R6)***	$H_{0.1}$	Class tests	Average mark
	$H_{0.2}$	Class tests	Average mark
	$H_{0.3}$	Class tests	Average mark
Enable students to successfully transfer knowledge ***(R7)***	$H_{0.4}$	Controlled experiments	Proportion of correct responses, Average mark, Saccades between Beacons, Fixation on Beacons
	$H_{0.5}$	Controlled experiments	Response time, Saccades between Beacons, Fixation on Beacons
	$H_{0.6}$	Controlled experiments	No. of students using chunking or beacon recognition

Table 4-3 Mapping between hypotheses, activities and data collected

Table 4-3 summarises the activities used to collect data and the metrics that will be analysed to accept or reject the hypotheses discussed in section 4.4.2. Table 4-3 also matches the hypotheses with the requirement it supports.

The following chapter presents the results obtained in the empirical investigation described in this chapter. To ensure the validity of these results special care is taken in the empirical investigation to minimize the identified risks, namely the sample size the Hawthorne Effect and possible problems with administering the practical learning activities (Section 4.5).

Chapter 5 Results of the Investigation

5.1 Introduction

Program generation is one of the main focuses of introductory programming modules, while program comprehension is often neglected (Deimel and Naveda 1990). While attempts are made to provide students with worked examples, little time is spent on explicitly teaching strategies to study the code. BeReT is a tool designed to address this deficiency.

Chapter 2 established requirements for a tool to support the in-depth comprehension of algorithms typically taught in an introductory algorithms module. The design and implementation decisions made during the development of BeReT (Chapter 3) concluded that requirements ***R1*** to ***R5*** are met (Table 5-1).

	Requirements to support in-depth comprehension of algorithms
R1	Allows student to extract semantic information (plans) from worked example
R2	Promotes the use of chunking to learn an unfamiliar algorithm
R3	Coaches students to spot beacons in different algorithms
R4	Provides students with syntactically and semantically correct worked examples
R5	Engages students in active learning activities
R6	Increases ability to recall the semantics of an algorithm
R7	Enables students to successfully transfer knowledge

Table 5-1 Requirements to aid the in-depth comprehension of introductory algorithms

Evidence to support the remaining requirements is determined by a between-groups experiment (Chapter 4). The methodology described is specifically aimed at collecting data to determine if requirements ***R6*** (Increases ability to recall the semantics of an algorithm) and ***R7*** (Enable students to successfully transfer knowledge) are met. The two main quantitative data collection activities used as measurement tools are class tests and controlled experiments. A questionnaire is used to collect qualitative data.

Hypotheses to confirm support for requirements ***R6*** and ***R7*** were also formulated in Chapter 4 (Table 5-2). The results of the between-groups experiment to reject or accept the hypotheses are presented.

Hypotheses	**§**	**Req**
$H_{0.1}$ Students using BeReT do not perform better in terms of accuracy on code comprehension questions.	5.2.1	***R6***
$H_{0.2}$ Students using BeReT find recalling the entire algorithm and adapting to any changes in problem scenario more difficult.		
$H_{0.3}$ Students using BeReT are less accurate when completing a partial algorithm.		
$H_{0.4}$ Students using BeReT are less successful in the discovery of known plans in previously unseen algorithms.	5.2.2	***R7***
$H_{0.5}$ Students using BeReT takes longer to find plans in a known and/or previously unseen algorithm quicker.		
$H_{0.6}$ Students using BeReT do not exhibit evidence of using beacon recognition and chunking when studying a previously unseen algorithm.		

Table 5-2 Hypotheses and sections where data can be found in support of hypotheses

After discarding the data according to the rules (Section 4.4.3), a sample size of 32 participants per group is achieved for class tests (Section 5.2.1). The assumption of normality, thus, holds for further statistical analysis (Larson 1974).

The remainder of this chapter presents the quantitative data analysis results of the class tests and controlled experiments (Section 5.2). Results of qualitative data analysis to complement quantitative findings appear in Section 5.3.

5.2 Quantitative Results

The quantitative results obtained in the empirical study include class test results (Section 5.2.1) and results from controlled experiments (Section 5.2.2). The quantitative results obtained from the class tests addresses hypotheses $H_{0.1}$, $H_{0.2}$ and $H_{0.3}$, while the quantitative results from the controlled experiments address hypotheses $H_{0.4}$, $H_{0.5}$ and $H_{0.6}$.

5.2.1. Results of Class Tests

The first-year students participating in this research study various algorithms as part of the requirements of the introductory programming module at NMMU. Class tests are written the week after initial exposure to the algorithm in lectures and practical assignments (Section 4.3.2). Three class tests were written to test comprehension of the bubble sort, binary search and insertion sort algorithms respectively (Section 4.3.3). The class test questions presented to the students appear in Appendix C. The results of the class test are presented in Table 5-3.

		Bubble Sort		**Binary Search**		**Insertion Sort**	
		Recall	**Comprehend**	**Recall**	**Comprehend**	**Complete**	**Comprehend**
Treatment Group (*n*=32)	**Mean**	56%	55%	41%	61%	56%	43%
	St Dev	23%	19%	27%	31%	17%	20%
Control Group (*n* = 32)	**Mean**	44%	35%	22%	42%	39%	27%
	St Dev	29%	45%	22%	20%	20%	30%
p-value		0.0169*	0.0003**	0.0017**	0.0004**	0.0005**	0.0002**

* Significant where $p<0.05$; ** Significant where $p<0.01$

Table 5-3 Results of Class Tests

All class test questions (Appendix C) assess students' cognitive development on the knowledge and comprehension levels of Bloom's taxonomy of educational objectives. Random sampling was applied in the determination of the control and treatment groups. To test students' knowledge of each algorithm comprehension questions ($H_{0.1}$), and recall ($H_{0.2}$) or complete ($H_{0.3}$) questions were asked. Table 5-3 presents descriptive statistics of the average for each group per question and algorithm. Table

5-3 also includes test statistics used to make a decision on whether to reject or accept the relevant hypotheses.

Data in support of the hypothesis $H_{0.1}$ *Students using BeReT do not perform better than the control group in terms of accuracy on code comprehension questions* was collected by means of comprehension type questions in the class tests. This hypothesis is used to determine the effect of BeReT on students' cognitive development on level 2 (Comprehension) of Bloom's taxonomy.

From the t-test, it can be concluded at the 99% percentile ($\alpha = 0.01$) that there is a difference in the average mark of the comprehension questions in class tests for all three algorithms. Hypothesis $H_{0.1}$ is, therefore, rejected in the case of class tests. The implication is that students using BeReT performs significantly better in terms of accuracy on code comprehension questions.

Data to support the hypothesis $H_{0.2}$ *Students using BeReT find recalling the entire algorithm and adapting to any changes in problem scenario more difficult* was also collected by means of recall type questions in the class tests. This hypothesis is used to determine the effect of BeReT on students' cognitive development on level 1 (Knowledge/Recall) of Bloom's taxonomy of educational objectives.

Questions instructing students to write an adapted version of the bubble sort and binary search algorithms form part of the class tests written by both groups. From the t-test, it can be concluded at the 95% percentile ($\alpha = 0.05$) that the treatment group performed better in terms of recalling the bubble sort and binary search algorithm. The treatment group's performance in the recall question of the binary search algorithm showed a significant improvement at the 99% percentile ($\alpha = 0.01$). Hypothesis $H_{0.2}$ is, therefore, rejected in the case of class tests. The implication is that students using BeReT find it easier to recall an algorithm and are sensitive to changes in the problem scenario.

Hypothesis $H_{0.3}$ *Students using BeReT are less accurate than the control group when completing a partial algorithm* was tested by means of a complete-type question in the insertion sort algorithm. This hypothesis is used to determine the effect of BeReT on students' cognitive development on level 1 (Knowledge/Recall) of Bloom's taxonomy of educational objectives.

From the t-test, it can be concluded at the 99% percentile ($\alpha = 0.01$) that the treatment group performed better in terms of completing the insertion sort. Hypothesis $H_{0.3}$ is, therefore, rejected in the case of this class test. The implication is that students using BeReT find it easier to fill in the missing lines of code in an incomplete version of a known algorithm.

5.2.2. Results of Controlled Experiments

Three experiments were performed a week after the final class test was written. The three experiments involved an unseen algorithm (Experiment A), an algorithm previously studied (Experiment B) and fragments of known algorithms (Experiment C). The number of students recruited for the controlled experiments was 16 (8 per group), as is reported in similar studies involving the use of an eye-tracker (Bartels and Marshall 2006; Bednarik, Myller, Sutinen *et al.* 2006). Only 7 participants per group provided usable eye-tracking data, since calibration failed on 2 of the participants (one from each group). IViewX software was used to record eye-tracking data. In IViewX, fixations were defined as at least 250 ms in duration in a radius of 50 pixels. Participants were placed in a comfortable chair that allowed minimum movement and seated approximately 60 centimetres from the monitor.

Experiment A – Unseen algorithm

Experiment A (Appendix D) tests the comprehension and interaction of a previously unseen algorithm (specifically the selection sort algorithm). The chosen algorithm uses similar plans used by algorithms that participants learnt in the introductory programming algorithms module. This experiment consisted of three tasks, which required participants to study an unseen algorithm (Task1), to answer comprehension questions about the algorithm (Task 2) and to complete an incomplete version of the unseen algorithm (Task 3).

Task 1 – Study unseen algorithm

Task 1 required participants to explain the purpose of the algorithm in their own words. This task tests their cognitive development on the comprehension and analysis level of Bloom's taxonomy (Buck and Stucki 2000; Buckley and Exton 2003). The goal of Task 1 in Experiment A is to test the hypothesis *$H_{0.6}$ Students using BeReT do not exhibit evidence of using beacon recognition and chunking when studying a previously unseen algorithm*.

During this task, the data recorded include the technique(s) participants used when studying an algorithm, the time it took the participant to study the algorithm (using a stopwatch) and the accuracy of their description of the purpose of the algorithm (Appendix D).

The notes that the participants made while studying the algorithm were collected and scrutinized to discover the techniques used to study the algorithm. An adapted framework for the categorization of annotations students make while studying an algorithm (developed in Lister, Adams *et al.* (2004) and Fitzgerald, Simon *et al.* (2005)) was used to determine participants algorithm reading technique. Two main techniques identified in this study are synchronized tracing and chunking.

Treatment Group Participants				**Control Group Participants**			
	Technique	**Time**	**Accuracy**		**Technique**	**Time**	**Accuracy**
T1	Trace	00:09:13	High Level	***C1***	Trace	00:10:00	Low Level
T2	Trace	00:04:00	High Level	***C2***	Trace	00:08:15	Incorrect
T3	Trace	00:09:40	High Level	***C3***	Trace	00:05:10	Low Level
T4	Chunking	00:10:00	Incorrect	***C4***	Trace	00:10:00	Incorrect
T5	Trace	00:09:00	Low Level	***C5***	Trace	00:10:00	High Level
T6	Trace	00:03:30	Low Level	***C6***	None	00:03:39	Low Level
T7	Trace	00:03:00	Low Level	***C7***	None	00:10:00	Low Level
Proportion High Level			3	**Proportion High Level**			1
Expected Frequency			3.5	**Expected Frequency**			3.5
χ^2 = 1, p-value = 0.317							

Table 5-4 Participants studying a new algorithm

Synchronized tracing shows the values of multiple variables as they change. This is usually done in a table format. Chunking is similar to the description of pattern recognition in Fitzgerald, Simon *et al.* (2005) where higher level meaning is sought in the code. It is identified in the participant notes by the highlighting of code statements that form a higher level operation and describing each chunk with a suitable name or phrase. The participants' responses were analyzed according to the technique discussed in Section 4.4.3 and the results are presented in Table 5-4.

The time column indicates the duration from the time that participants started reading the algorithm until the moment they feel confident enough to answer questions on the algorithm. During this task, time was measured using a stopwatch. The time was started from the moment participants started reading, until they indicated that they were ready for the questions. As can be seen from Table 5-4 the predominant technique used by the participants is tracing. The majority of participants from both groups explained the purpose of the algorithm correctly, either on high (semantic) or low (syntactic) level. From Table 5-4 it can be seen that 6 participants from the treatment group and 3 participants from the control group completed Task 1 of the experiment involving the unseen algorithm within the time limit. From the p-value resulting from a Chi-squared test it can be concluded that no significant difference exists between the proportion of participants that completed the task on time (α = 0.05).

The techniques used by participants when studying an unseen algorithm for the first time appear in Table 5-5, along with the number of participants making use of the technique. A Chi-squared test of the equality of proportions shows no significant preference in the technique used by the participants to study the unseen algorithm (α = 0.05).

Technique	**Treatment (n = 7)**	**Control (n = 7)**	**Expected Frequency**	χ^2	**p-value**
Tracing	6	5	3.5	3.09	0.213
Chunking	1	0	3.5		
None	0	2	3.5		

Table 5-5 Techniques used to study an unseen algorithm

Table 5-6 summarises the accuracy of responses on high level, low level and incorrect. A Chi-squared test of the equality of proportions shows no significant difference in the performance of the participants with regards to the accuracy of their identification of the purpose of the unseen algorithm in Task 1 of Experiment A (α = 0.05).

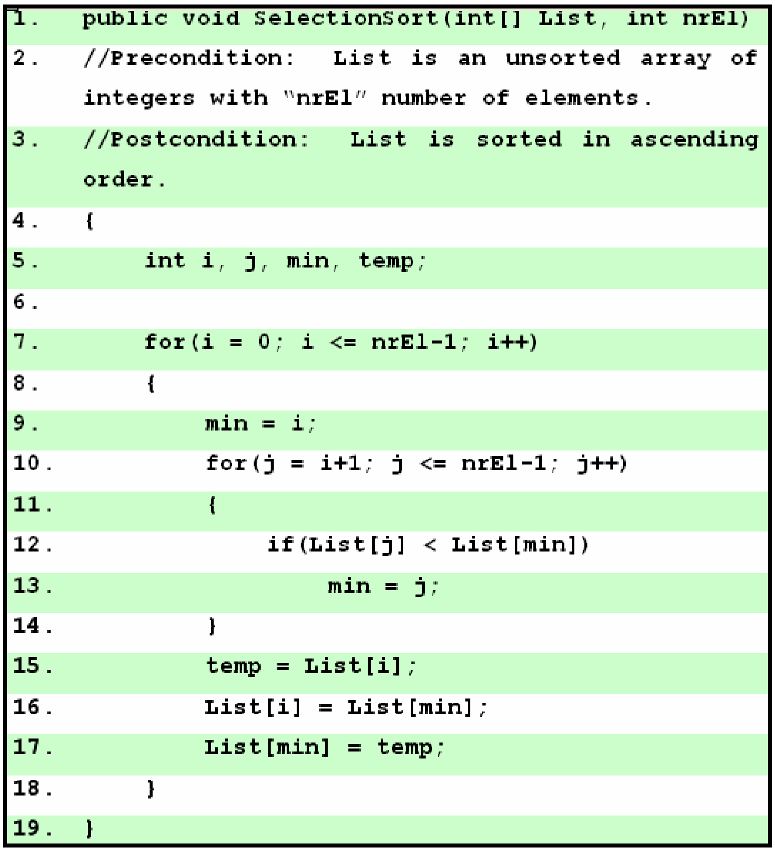
Group	Accuracy		
	High Level	**Low Level**	**Incorrect**
Treatment (*n*=7)	3	3	1
Control (*n*=7)	1	4	2
Expected Frequency	3.5	3.5	3.5
χ^2	1	0.14	0.33
p-value	0.317	0.375	0.564

Table 5-6 Summary of average accuracy of responses

For Task 1 of Experiment A, no significant difference exists between the two groups in terms of the technique used to study the unseen algorithm, the time taken to study the unseen algorithm, and the accuracy of their description of the goal of the unseen algorithm. Hypothesis $H_{0.6}$ *Students using BeReT do not exhibit evidence of using beacon recognition and chunking when studying a previously unseen algorithm* can therefore not be rejected.

Task 2 – Answer comprehension questions

For Task 2, participants were instructed to answer comprehension questions about the unseen algorithm studied in Task 1. This task tests their cognitive development on the comprehension level of Bloom’s taxonomy (Buck and Stucki 2000; Buckley and Exton 2003). The goal of Task 2 in Experiment A is to test two hypotheses, namely $H_{0.4}$ *Students using BeReT are less successful than the control group in the discovery of known plans in previously unseen algorithms* and $H_{0.5}$ *Students using BeReT take longer than the control group to find plans in a known and/or previously unseen algorithm.*



```
public void SelectionSort(int[] List, int nrEl)
//Precondition:  List is an unsorted array of integers with “nrEl” number of elements.
//Postcondition:  List is sorted in ascending order.
{
     int i, j, min, temp;

     for(i = 0; i <= nrEl-1; i++)
     {
          min = i;
          for(j = i+1; j <= nrEl-1; j++)
          {
               if(List[j] < List[min])
                    min = j;
          }
          temp = List[i];
          List[i] = List[min];
          List[min] = temp;
     }
}
```

Figure 5-1 The selection sort algorithm shown on screen during eye tracking

Data collected during this task include the response to comprehension questions, the time it takes to find the answer and eye-movement data to indicate participants’ focus of attention, their navigation and reading style when finding answers. Figure 5-1 illustrates the algorithm that was shown as a slideshow file to the participants.

Treatment Group (*n*=7)			**Control Group (*n*=7)**		
Participant	**Time**	**Accuracy**	**Participant**	**Time**	**Accuracy**
T1	00:00:14	100%	***C1***	00:00:26	60%
T2	00:00:12	60%	***C2***	00:00:11	20%
T3	00:00:15	100%	***C3***	00:00:11	40%
T4	00:00:05	100%	***C4***	00:00:14	20%
T5	00:00:07	80%	***C5***	00:00:16	40%
T6	00:00:05	40%	***C6***	00:00:11	40%
T7	00:00:09	80%	***C7***	00:00:13	60%
Proportion passed		6	**Proportion passed**		2
Expected Frequency		3.5	**Expected Frequency**		3.5
Mean	00:00:09	80%	**Mean**	00:00:15	40%
St Dev	00:00:08	0.41	**St Dev**	00:00:09	0.50

$\chi^2 = 2.00$

p-value (Proportion of participants that passed) = 0.157

Table 5-7 Total performance for Selection Sort algorithm

Table 5-7 summarises the overall performance for the selection sort algorithm between the two groups. The average mark seems to indicate that the treatment group performed better than the control group (80% vs. 40%) on Task 2. A χ^2-test on the proportion of participants who successfully completed Task 2 with a mark of 50% or higher (6 from the treatment group, 2 from the control group) shows no significant difference.

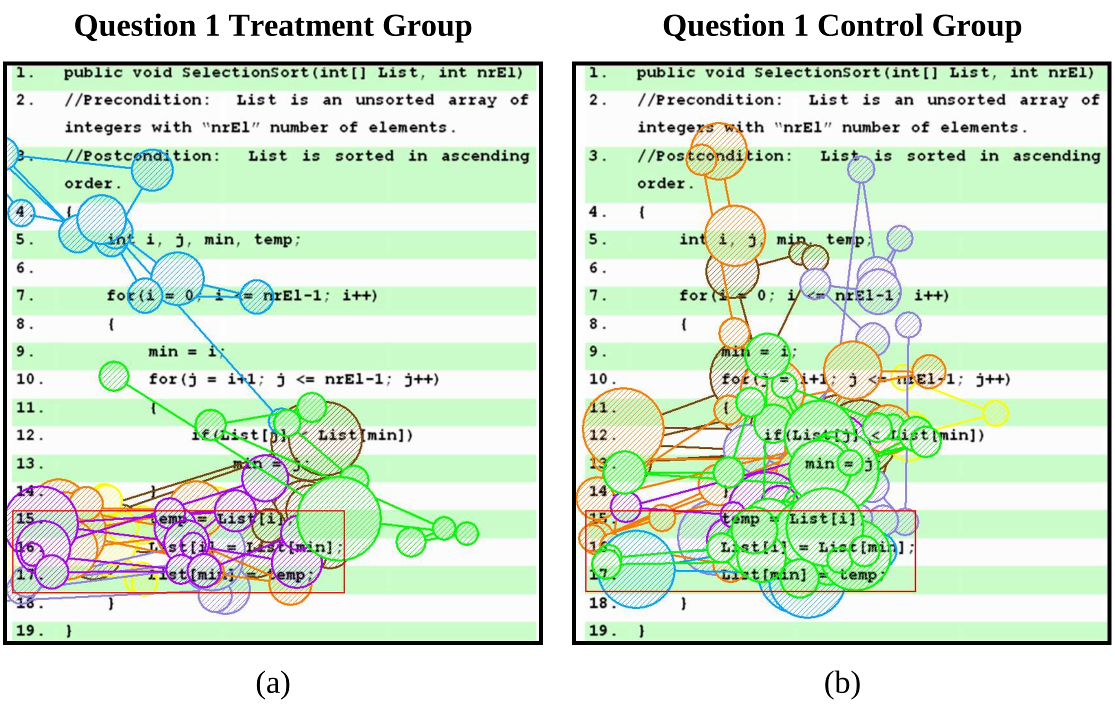
A breakdown of the proportion of correct responses and the average times for all five questions show that only Question 1 demonstrates significantly different results between the two groups (Table 5-8). Eye-tracking data reveal possible reasons for this significant difference.

	Number of correct participants					**Time taken**		
Question	**Treatment (*n*=7)**	**Control (*n*=7)**	**Expected Frequency**	**χ^2**	**p-value**	**Treatment (*n*=7)**	**Control (*n*=7)**	**p-value**
Question 1	7	1	3.5	4.50	0.034*	00:00:04	00:00:12	0.046*
Question 2	5	3	3.5	0.50	0.480	00:00:15	00:00:21	0.427
Question 3	5	3	3.5	0.50	0.480	00:00:09	00:00:14	0.109
Question 4	7	6	3.5	0.08	0.782	00:00:08	00:00:12	0.250
Question 5	4	1	3.5	1.80	0.180	00:00:12	00:00:23	0.125

* Significant where $p<0.05$

Table 5-8 Number of correct responses and response time per question (Selection Sort)

Scanpaths from the gaze analysis of Question 1 ("Which line or lines exchanges (swaps) two elements in an array?") reveals the paths participants take to find the answer for this question (Figure 5-2). Scanpaths and heatmaps resulting from all questions are presented in Appendix I. The different colours represent the saccades and fixations of the different participants. The circles represent fixations, with the diameter of the circle an indication of the duration of fixation at particular points. A larger circle implies a longer the fixation time. The lines between the circles represent saccades, or eye-movements between the fixation points.



Question 1 Treatment Group

```
1.  public void SelectionSort(int[] List, int nrEl)
2.  //Precondition:  List is an unsorted array of
    integers with "nrEl" number of elements.
3.  //Postcondition:  List is sorted in ascending
    order.
4.  {
5.       int i, j, min, temp;
6.
7.       for(i = 0; i <= nrEl-1; i++)
8.       {
9.            min = i;
10.           for(j = i+1; j <= nrEl-1; j++)
11.           {
12.                if(List[j] < List[min])
13.                     min = j;
14.           }
15.           temp = List[i];
16.           List[i] = List[min];
17.           List[min] = temp;
18.      }
19. }
```

(a)

Question 1 Control Group

```
1.  public void SelectionSort(int[] List, int nrEl)
2.  //Precondition:  List is an unsorted array of
    integers with "nrEl" number of elements.
3.  //Postcondition:  List is sorted in ascending
    order.
4.  {
5.       int i, j, min, temp;
6.
7.       for(i = 0; i <= nrEl-1; i++)
8.       {
9.            min = i;
10.           for(j = i+1; j <= nrEl-1; j++)
11.           {
12.                if(List[j] < List[min])
13.                     min = j;
14.           }
15.           temp = List[i];
16.           List[i] = List[min];
17.           List[min] = temp;
18.      }
19. }
```

(b)

Figure 5-2 Scanpaths for Question 1: "Which line or lines exchanges (swaps) two elements in an array?"

From the two images, it is clear that the treatment group follows a more concentrated path around the area of interest containing the answer to question 1 (namely Lines 15, 16 and 17). Only two treatment group participants deviated significantly from the area containing the answer.

The control group scanpaths reveal five participants scanning areas far away from the area of interest. The scanpaths indicate that most of the control group participants did read the correct code, but decided that another line was the correct answer. These images seem to suggest that the treatment group participants were more successful in transferring their knowledge of the "swap" plan from the bubble sort algorithm and recognising it in the previously unseen selection sort algorithm.

Hypothesis $H_{0.4}$ *Students using BeReT are less successful than the control group in the discovery of known plans in previously unseen algorithms* is, therefore, rejected in the case of this controlled experiment. The implication is that students using BeReT found it easier to discover known plans in previously unseen algorithms.

Task 3 – Complete incomplete algorithm and fix errors

Task 3 tests participants' cognitive development on the knowledge level of Bloom's taxonomy. Cloze exercises are used to assess their knowledge of an algorithm

(Section 2.4.1). Small changes are made in the variable names and data types to determine if participants exhibit signs of rote learning. Errors are also intentionally made in the presented code and participants are instructed to find the errors.

Table 5-9 summarises the results of task 3. On average the treatment group participants completed more of the missing lines in the algorithm than the control group (a 9% difference) (Table 5-9). The treatment group managed to complete Task 3 of Experiment A faster than the control group (4 minutes and 16 seconds vs. 6 minutes 35 seconds).

	% of Lines Completed		**% of Errors Corrected**		**Time Taken**	
Participant	**Treatment (*n*=7)**	**Control (*n*=7)**	**Treatment (*n*=7)**	**Control (*n*=7)**	**Treatment (*n*=7)**	**Control (*n*=7)**
1	60%	60%	0%	20%	00:03:00	00:06:00
2	50%	30%	20%	10%	00:03:51	00:11:15
3	100%	100%	20%	20%	00:05:00	00:07:00
4	80%	70%	0%	10%	00:03:00	00:05:24
5	60%	80%	30%	20%	00:04:00	00:03:00
6	60%	30%	0%	10%	00:01:59	00:08:00
7	100%	80%	30%	20%	00:09:00	00:08:23
Mean	73%	64%	14%	16%	00:04:16	00:06:35
St Dev	0.21	0.26	0.14	0.05	00:02:18	00:03:27
Proportion passed	7	5	0	0		
Expected Frequency	3.5	3.5	3.5	3.5		
χ^2	0.33		0.00			
p-value	0.564		1.000			

Table 5-9 Experiment A Task 3 Performance

From the p-values of a χ^2-test on the number of participants who completed more than 50% of the missing lines and corrected more than 50% of the errors, it can be seen that no significant difference exists between the two groups. Although not significant, Table 5-9 suggests that there is a slight improvement in the ability of the treatment group participant to complete this task faster (about 2 minutes on average) than the control group participants. *$H_{0.3}$ Students using BeReT are less accurate than the control group when completing a partial algorithm* was rejected in the case of the class test on the known algorithm, but for the previously unseen algorithm in this controlled experiment hypothesis *$H_{0.3}$* cannot be rejected.

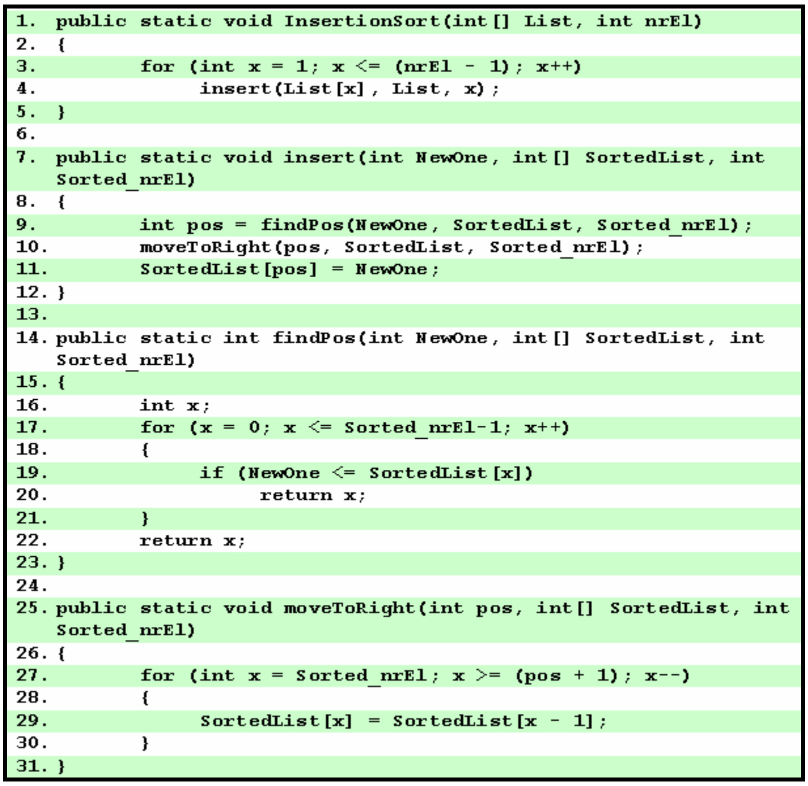
Experiment B – Known algorithm

Experiment B tests comprehension of and interaction with an algorithm studied in the introductory programming module in the Department of CS&IS at NMMU, namely the insertion sort algorithm. The treatment group studied the algorithm in BeReT during one of the weekly practical assignments, while the control group used their class notes only to study the algorithm. Data gathered from this experiment determines the effect of BeReT on first year students' comprehension of the algorithm insertion sort.

Task 1 – Answer comprehension questions

Task 1 of Experiment B assesses whether participants grasp the semantic meaning of the material learnt. This tests their cognitive development on the comprehension level of Bloom's taxonomy (Buck and Stucki 2000; Buckley and Exton 2003).

Figure 5-3 illustrates the insertion sort algorithm presented as a slideshow to participants. Participants in both groups were exposed to this version of the algorithm in their class notes. The function and variables names used in this implementation of the algorithm were the same as used in the class notes.



```
public static void InsertionSort(int[] List, int nrEl)
{
        for (int x = 1; x <= (nrEl - 1); x++)
                insert(List[x], List, x);
}

public static void insert(int NewOne, int[] SortedList, int
Sorted_nrEl)
{
        int pos = findPos(NewOne, SortedList, Sorted_nrEl);
        moveToRight(pos, SortedList, Sorted_nrEl);
        SortedList[pos] = NewOne;
}

public static int findPos(int NewOne, int[] SortedList, int
Sorted_nrEl)
{
        int x;
        for (x = 0; x <= Sorted_nrEl-1; x++)
        {
                if (NewOne <= SortedList[x])
                        return x;
        }
        return x;
}

public static void moveToRight(int pos, int[] SortedList, int
Sorted_nrEl)
{
        for (int x = Sorted_nrEl; x >= (pos + 1); x--)
        {
                SortedList[x] = SortedList[x - 1];
        }
}
```

Figure 5-3 The insertion sort algorithm shown on screen during eye tracking

Table 5-10 summarises for the two groups the average accuracy and time taken to find answers relating to the insertion sort algorithm. In terms of accuracy, the treatment group scored 3% better on average than the control group. The treatment group participants answered the comprehension questions on average 10 seconds faster than the control group.

Treatment Group (*n*=7)			**Control Group (*n*=7)**		
Participant	**Time**	**Accuracy**	**Participant**	**Time**	**Accuracy**
T1	00:00:16	60%	***C1***	00:00:33	80%
T2	00:00:14	80%	***C2***	00:00:25	40%
T3	00:00:25	80%	***C3***	00:00:17	40%
T4	00:00:08	20%	***C4***	00:00:28	40%
T5	00:00:04	80%	***C5***	00:00:14	20%
T6	00:00:10	0%	***C6***	00:00:22	80%
T7	00:00:13	60%	***C7***	00:00:18	60%
Mean	00:00:13	54%	**Mean**	00:00:23	51%
St Dev	00:00:07	0.32	**St Dev**	00:00:07	0.23
Proportion passed		5	**Proportion passed**		3
Expected Frequency		3.5	**Expected Frequency**		3.5

$\chi^2 = 0.5$
p-value (Accuracy) = 0.480

Table 5-10 Total performance for Insertion Sort algorithm

From a χ^2-test on the number of participants who passed Task 1 with an average mark of 50% or higher, it can be concluded no significant difference exist between the two groups.

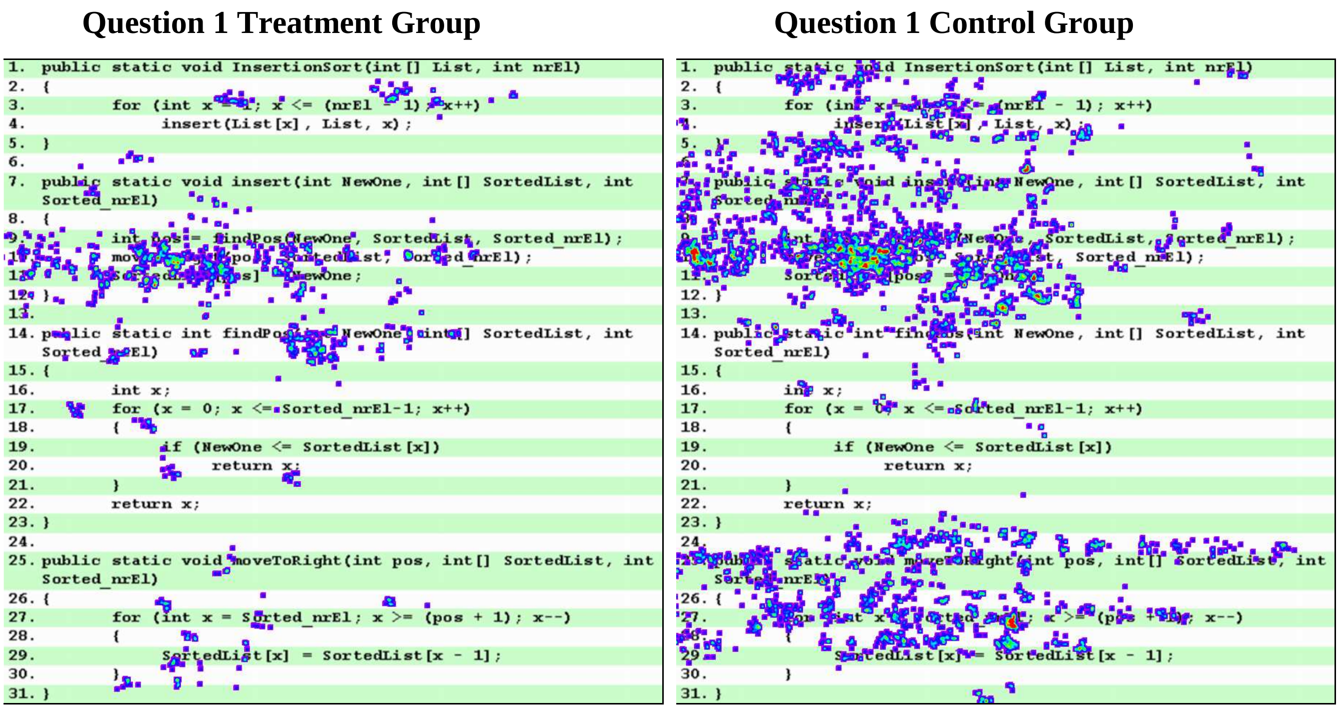
Due to the small number of participants, the average time taken to study the known algorithm can also not be tested for significance, therefore hypothesis *$H_{0.5}$ Students using BeReT takes longer than the control group to find plans in a known and/or previously unseen algorithm* is, can not be rejected in the case of the controlled experiment involving the known algorithm (insertion sort). However, there seems to be an improvement in the time it takes participants from the treatment group in finding the answer. Reasons for this can be further explored by analyzing individual questions and the eye-tracking data obtained from this experiment.

	Number of correct participants					Time taken		
Question	Treatment (*n*=7)	Control (*n*=7)	χ^2	Expected Frequency	p-value	Treatment (*n*=7)	Control (*n*=7)	p-value
Question 1	5	5	0	3.5	1	00:00:04	00:00:21	0.040*
Question 2	1	3	1	3.5	0.317	00:00:17	00:00:18	0.827
Question 3	5	5	0	3.5	1	00:00:11	00:00:20	0.178
Question 4	5	3	0.5	3.5	0.480	00:00:13	00:00:19	0.254
Question 5	3	2	0.2	3.5	0.655	00:00:19	00:00:34	0.068

* Significant where p<0.05

Table 5-11 Number of correct responses and response time per question (Insertion Sort)

A breakdown of the proportion of correct responses for each question shows no significant difference between the two groups (Table 5-11). A question-by-question breakdown of the average time to find answers to comprehension questions shows significant difference between the two groups in Question 1 only (Table 5-11).



Question 1 Treatment Group

```
1.  public static void InsertionSort(int[] List, int nrEl)
2.  {
3.          for (int x = 1; x <= (nrEl - 1); x++)
4.                  insert(List[x], List, x);
5.  }
6.
7.  public static void insert(int NewOne, int[] SortedList, int
    Sorted_nrEl)
8.  {
9.          int pos = findPos(NewOne, SortedList, Sorted_nrEl);
10.         moveToRight(pos, SortedList, Sorted_nrEl);
11.         SortedList[pos] = NewOne;
12. }
13.
14. public static int findPos(int NewOne, int[] SortedList, int
    Sorted_nrEl)
15. {
16.         int x;
17.         for (x = 0; x <= Sorted_nrEl-1; x++)
18.         {
19.                 if (NewOne <= SortedList[x])
20.                         return x;
21.         }
22.         return x;
23. }
24.
25. public static void moveToRight(int pos, int[] SortedList, int
    Sorted_nrEl)
26. {
27.         for (int x = Sorted_nrEl; x >= (pos + 1); x--)
28.         {
29.                 SortedList[x] = SortedList[x - 1];
30.         }
31. }
```

(a)

Question 1 Control Group

```
1.  public static void InsertionSort(int[] List, int nrEl)
2.  {
3.          for (int x = 1; x <= (nrEl - 1); x++)
4.                  insert(List[x], List, x);
5.  }
6.
7.  public static void insert(int NewOne, int[] SortedList, int
    Sorted_nrEl)
8.  {
9.          int pos = findPos(NewOne, SortedList, Sorted_nrEl);
10.         moveToRight(pos, SortedList, Sorted_nrEl);
11.         SortedList[pos] = NewOne;
12. }
13.
14. public static int findPos(int NewOne, int[] SortedList, int
    Sorted_nrEl)
15. {
16.         int x;
17.         for (x = 0; x <= Sorted_nrEl-1; x++)
18.         {
19.                 if (NewOne <= SortedList[x])
20.                         return x;
21.         }
22.         return x;
23. }
24.
25. public static void moveToRight(int pos, int[] SortedList, int
    Sorted_nrEl)
26. {
27.         for (int x = Sorted_nrEl; x >= (pos + 1); x--)
28.         {
29.                 SortedList[x] = SortedList[x - 1];
30.         }
31. }
```

(b)

Figure 5-4 Heat Map Question 1: "Which line or lines calls a function to make space for a new element in an array?"

Question 1 asked: *"Which line/lines calls a function to make space for a new element in an array?"* The correct answer is line 10. The heat map (Figure 5-4) resulting from Question 1 clearly shows that the treatment group participants focus mostly on the correct area. The control group on the other hand spends a large amount of time reading the code in the `moveToRight` function. These figures seem to indicate that the control group of participants did not view the function call `moveToRight` as a comment beacon that indicates the plan of making space for a new element in an array. They still had to read the code inside the function to determine what it does.

Task 2 – Complete incomplete algorithm and fix errors

Task 2 assesses participants' cognitive development on the knowledge level of Bloom's taxonomy (Bloom 1956; Buck and Stucki 2000; Buckley and Exton 2003). Assessment methods take the form of the following:

- Cloze exercises to assess their knowledge of an algorithm previously studied.
- Making small changes in variable names and data types to test for rote learning.
- Manual debugging of the code.

Table 5-12 summarises the results of this task.

	% of Lines Completed		**% of Errors Corrected**		**Time Taken**	
Participant	**Treatment (*n*=7)**	**Control (*n*=7)**	**Treatment (*n*=7)**	**Control (*n*=7)**	**Treatment (*n*=7)**	**Control (*n*=7)**
1	65%	55%	40%	10%	00:07:00	00:08:25
2	65%	30%	40%	10%	00:05:07	00:10:58
3	95%	35%	50%	20%	00:06:58	00:06:00
4	55%	0%	0%	0%	00:04:00	00:08:00
5	70%	50%	20%	10%	00:05:15	00:10:25
6	75%	30%	0%	0%	00:03:30	00:10:14
7	90%	55%	40%	20%	00:10:00	00:10:00
Mean	73.6%	36.4%	27%	10%	00:05:59	00:09:09
St Dev	0.29	0.30	0.21	0.08	00:02:13	00:01:46
Proportion passed	7	3	1	0		
Expected Frequency	3.5	3.5	3.5	3.5		
χ^2	1.60		1			
p-value	0.206		0.317			

Table 5-12 Experiment B Task 3 Performance

Table 5-12 shows the p-values of a χ^2-test on the number of participants who completed more than 50% of the missing lines and corrected more than 50% of the errors. It can be seen that no significant difference exists between the two groups when using 50% as the benchmark. However, it is worth noting that significantly more treatment group participants obtained a mark of 60% or higher ($\chi^2 = 6$, p-value = 0.014, $\alpha = 0.05$). This again rejects hypothesis *$H_{0.3}$ Students using BeReT are less accurate than the control group when completing a partial algorithm*. Although not significant, Table 5-12 also shows that the treatment group participants are capable of completing incomplete algorithms quicker than control group participants in the case of known algorithms.

Experiment C – Code fragments

Experiment C consists of code fragments of the three algorithms previously studied by the participants, namely the bubble sort, the insertion sort and the binary search algorithms. The data recorded for each participant includes the following:

- The participants' response of the most likely algorithm that uses the code displayed (between the three algorithms selected for this empirical study).
- The time taken to realise what the most likely algorithm is.
- A semantic description of the plan implemented by the code fragments.

This experiment consequently assesses participants' ability to quickly recognise stereotypical lines of code, and their ability to describe the plan behind the code in their own words. Participants' cognitive development on both knowledge and comprehension level of Bloom's taxonomy are assessed in this experiment (Buck and Stucki 2000; Buckley and Exton 2003). The results of Experiment C are summarised in Table 5-13 and Table 5-14, including the average mark obtained during the task of identifying the algorithms based on the code fragments.

	Identification of algorithm					Response Time			
Participant	**Treatment (*n*=7)**	**Control (*n*=7)**	χ^2	**p-value**	**Participant**	**Treatment (*n*=7)**	**Control (*n*=7)**	χ^2	**p-value**
1	100%	83%	0	1	***1***	00:00:06	00:00:21	6	0.014*
2	100%	100%			***2***	00:00:07	00:00:22		
3	83%	50%			***3***	00:00:09	00:00:12		
4	100%	100%			***4***	00:00:11	00:00:17		
5	100%	67%			***5***	00:00:08	00:00:20		
6	100%	100%			***6***	00:00:07	00:00:13		
7	100%	100%			***7***	00:00:04	00:00:10		
Mean	98%	86%			**Mean**	00:00:07	00:00:16		
St Dev	0.06	0.2			**St Dev**	00:00:02	00:00:05		
Proportion Passed	7	7			**Proportion less than 10 seconds**	6	0		
Expected Frequency	3.5	3.5			**Expected Frequency**	3.5	3.5		

Table 5-13 Summary of performance in Experiment C

Program comprehension is a combination of searching and problem solving. Readers generally take between 2 to 2.5 seconds to inspect an object and the rest of the time to process the information (Aschwanden and Crosby 2006). According to Aschwanden and Crosby (2006) it takes about 10 seconds to process the information in a beacon type statement. This time period is therefore used as a benchmark of the time it should take participants to identify the appropriate algorithm that the code fragments belong to. From a χ^2-test on the number of participants that identified the algorithm in less than 10 seconds, it can be stated at the 95% percentile ($\alpha = 0.05$) that the proportion of participants from the treatment group that identified the algorithm within this benchmark time are significantly higher than the control group (Table 5-13). Hypothesis $H_{0.5}$ *Students using BeReT takes longer than the control group to find plans in a known and/or previously unseen algorithm quicker* is, therefore, rejected in the case of the controlled experiment involving the code fragments. However, it can be concluded that the improvement in the ability of the treatment group to correctly identify the algorithm which the fragment belongs to is not significant.

Table 5-14 lists for each question in Experiment C the proportions of each type of description given by participants in each group along with the p-values from a χ^2-test for the equality of proportions. In each χ^2-test the expected frequency was 50%.

From Table 5-14 in can be concluded at the stated confidence levels that the code fragments in questions 1, 3 and 5 were better described by the treatment group. In this case “better” is described as more high level (semantic) descriptions and less low level or incorrect descriptions (Section 4.4.3).

Description of plan for:	**Type of description**	**Frequency of Participants**			**p-value**
		Treatment	**Control**	**Expected**	
Question 1	High Level	3	0	3.5	0.05*
	Low Level	4	4		
	Incorrect	0	3		
Question 2	High Level	2	3	3.5	0.175
	Low Level	5	2		
	Incorrect	0	2		
Question 3	High Level	5	0	3.5	0.007**
	Low Level	2	2		
	Incorrect	0	5		
Question 4	High Level	3	1	3.5	0.237
	Low Level	4	6		
	Incorrect	0	0		
Question 5	High Level	2	0	3.5	0.005**
	Low Level	5	1		
	Incorrect	0	6		
Question 6	High Level	5	2	3.5	0.270
	Low Level	1	3		
	Incorrect	1	2		

* Significant where p<0.05, ** Significant where p<0.01

Table 5-14 Number of high level, low level and incorrect descriptions

5.3 Qualitative Results

On completion of the last practical assignment during the empirical study period, a questionnaire (Appendix E) was circulated to treatment group participants (*n* = 32). The questions in the questionnaire are designed to determine the usability of BeReT (Section 5.3.1), the perceived effectiveness of BeReT as a learning tool (Section 5.3.2), the students’ motivation towards code comprehension activities (Section 5.3.3), and the functionality of BeReT (Section 5.3.4).

5.3.1. Usability of BeReT

The following questions are posed to determine if BeReT is easy to use or not.

- Q1 determines whether BeReT is easy to use,
- Q2 and Q4 determine whether students can complete Tutorials and Exercises without human intervention,

- Q6 determines if students can perform all required steps (apart from Tutorials and Exercises) in BeReT without human intervention,
- Q16 determines if students experience any unexpected errors in BeReT.
- Q17 is an open-ended question to determine the nature of the errors experienced in Q16.

Table 5-15 shows the number of positive and negative responses of the 32 treatment group participants that completed the questionnaire. Expected frequencies are given in brackets in the p-value column.

Question	**Positive**	**Negative**	**Expected Frequency**	χ^2	**p-value**
Q1	31	1	16	28.13	0.000**
Q2	32	0	16	32	0.000**
Q4	32	0	16	32	0.000**
Q6	23	9	16	6.13	0.013*
Q16	6	26	16	12.5	0.000**
Q17	Open Ended				

* Significant where p<0.05, ** Significant where p<0.01

Table 5-15 Questions to determine ease of use

From Table 5-15, it can be concluded at the stated confidence levels that BeReT tutorials and exercises are easy to use without assistance (Q1, Q2, Q4, and Q6).

Question 16, however, reveals there are some problems that need to be fixed. A theme-based analysis of the open-ended question following Question 16 identifies a bug in the Exercise component of BeReT. Example responses for Question 17 are:

> *"While attempting an exercise, the system closed after clicking 'Next Task'."*
>
> *"Sometimes an exercise was unavailable, even though I haven't completed it yet."*

5.3.2. Perceived effectiveness of BeReT as a learning tool

The following questions are posed to determine how the students perceive their own progress when using BeReT.

- Q7(a-c) and Q8(a-c) addresses students' perception of their understanding of the three algorithms before and after BeReT intervention
- Q9 to Q12 concentrates on the perceived benefit students will get from BeReT intervention. The perceived benefits include:

- BeReT makes program generation easier (Q9),
- BeReT aids understanding of the plans used in an algorithm (Q10),
- BeReT makes recognition of an unseen algorithm with similar plans easier (Q11), and
- BeReT makes tracing the algorithm easier (Q12).

Table 5-16 shows the number of positive and negative responses of the 32 treatment group participants that completed the questionnaire.

Question	Positive	Negative	Expected Frequency	χ^2	p-value
Q7a	10	22	16	4.5	0.034*
Q7b	8	22	16	8	0.005**
Q7c	4	28	16	18	0.000**
Q8a	30	2	16	24.5	0.000**
Q8b	29	3	16	21.3	0.000**
Q8c	20	12	16	2	0.175
Q9	23	9	16	6.13	0.013*
Q10	25	7	16	10.13	0.001**
Q11	23	9	16	6.13	0.013*
Q12	8	24	16	8	0.005**

* Significant where $p<0.05$, ** Significant where $p<0.01$

Table 5-16 Questions to determine perceived effectiveness of BeReT as a learning tool

From Table 5-16, it can be concluded at the stated confidence levels that students perceive BeReT to have a generally positive effect on their learning of the algorithms studied. A consequence of the positive attitude towards the treatment tool is that no conclusion can be drawn with regards to the impact of the Hawthorne effect of using a novel tool in teaching (Section 4.5.2). No conclusion can be made with regards to the effect BeReT had on students' understanding of the Insertion Sort algorithm (Q8c). BeReT also had no perceived positive effect on students' ability to perform a trace on the algorithms studied (Q12).

5.3.3. Motivation towards code comprehension activities

The following questions were posed to determine how motivated students are to perform code comprehension activities in addition to code generation activities.

- Q13 and Q14 determine if students would use BeReT in future code comprehension activities,
- Q20 and Q21 captures the atmosphere in the computer laboratory during practical assignments,
- Q22 determines the motivation of students to perform program comprehension tasks before program generation tasks without the use of a tool such as BeReT.

Table 5-17 shows the number of positive and negative responses of the 32 treatment group participants that completed the questionnaire.

Question	Positive	Negative	Expected Frequency	χ^2	p-value
Q13	27	5	16	15.13	0.000**
Q14	28	4	16	18	0.000**
Q20	31	1	16	28.13	0.000**
Q21	27	5	16	15.13	0.000**
Q22	16	16	16	0	1

* Significant where $p<0.05$, ** Significant where $p<0.01$

Table 5-17 Questions to determine motivation towards code comprehension

From Table 5-17 it can be concluded at the stated confidence levels that students have a positive attitude towards using BeReT for code comprehension tasks (Q13 and Q14). There is also a clear indication that BeReT stimulates critical discussion about the algorithms amongst students and between the students and the tutor/lecturer (Q20 and Q21) during practical assignments. No conclusion can be made regarding the motivation of students to perform code comprehension tasks prior to a practical assignment without BeReT (Q22).

Two open-ended questions (Q18 and Q19) were posed to determine the perceived advantages and disadvantages of using BeReT. A thematic analysis of the responses to the open-ended questions of Q18 revealed three advantages, namely BeReT tutorials enhances comprehension, BeReT exercises train students to think fast and BeReT allows for self paced studying. Three disadvantages also emerged from the thematic analysis of Q19, namely BeReT is inaccessible at home, there is restricted functionality in BeReT, and system errors in BeReT hinder progress.

As can be seen from the list of disadvantages, a theme of restricted functionality in BeReT emerges as one of the disadvantages. Question 15 is an attempt to address this disadvantage by allowing the students to suggest functionality they would like to see included in BeReT. The following are some of the suggestions from the open ended questions given by students to improve BeReT's functionality:

> *"Allow Exercises to be marked for immediate feedback",*
> *"Include an animation screen to visualise the execution of each plan",*
> *"Provide a comparison view of different ways of achieving the same plan",*
> *"Show arrows to link beacon description with corresponding lines of code",*
> *"Allow stepping through code, showing memory status at various points",*
> *"Allow selection of lines during Completion Exercises".*

These suggestions form a basis for subsequent revisions of BeReT (Section 6.4).

5.4 Conclusion

Class tests, controlled experiments and a questionnaire were used to collect data to determine whether BeReT increases the students' ability to recall the semantics of an algorithm ***(R6)*** and enables students to successfully transfer knowledge ***(R7)***.

Three class tests contribute data used to measure the effect BeReT has on the treatment group. The students' level of comprehension is tested for the three algorithms used in the study, namely bubble sort, binary search, and insertion sort. Recall, complete and comprehension type questions are used to determine the students' development on the knowledge level of Bloom's taxonomy. Results of a pooled-variance, two-tailed t-test for testing of the difference between average marks obtained in class tests show statistically significant variations in the averages between the two groups (Table 5-3). The average marks for comprehension, recall and complete questions are all in favour of the treatment group. It can, therefore, be concluded that the following hypotheses are rejected:

> $H_{0.1}$ *Students using BeReT do not perform than the control group better in terms of accuracy on code comprehension questions.*
>
> $H_{0.2}$ *Students using BeReT find recalling the entire algorithm and adapting to any changes in problem scenario more difficult.*
>
> $H_{0.3}$ *Students using BeReT are less accurate than the control group when completing a partial algorithm.*

This implies that BeReT does indeed increase the ability of students in introductory programming to recall the semantics of an algorithm ***(R6)***.

Results of a Chi-squared test for testing of the proportion of students who obtained a pass mark in comprehension questions asked in a controlled experiment (involving known plans in an unseen algorithm) show no statistical significance (Table 5-7). It can, therefore, be concluded that $H_{0.4}$ *Students using BeReT are less successful than the control group in the discovery of known plans in previously unseen algorithms* can not be rejected.

A Chi-squared test for testing the proportion of students who successfully identified plans within a benchmark of 10 seconds shows that the treatment group identified the plans significantly quicker than the control group (Table 5-10). It can, therefore, be concluded that $H_{0.5}$ *Students using BeReT takes longer than the control group to find plans in a known and/or previously unseen algorithm quicker* is rejected. The rejection of $H_{0.5}$ implies that BeReT partially increase the ability of students in introductory programming to transfer knowledge of plans between algorithms ***(R7)***.

It is not possible to reject hypothesis $H_{0.6}$ *Students using BeReT do not exhibit evidence of using beacon recognition and chunking when studying a previously unseen algorithm*. An inspection of the notes made by students when studying an algorithm for the first time shows tracing to be the favoured technique used by students.

Chi-squared tests to determine the equality of proportions of positive versus negative responses were used to verify statistical significance of the data collected by means of a questionnaire. The Chi-squared test statistics from the questionnaire reveal that despite one or two system errors identified in BeReT, the students find the system easy to use. They also perceive BeReT to be an effective learning tool. Results also indicate an increase in the motivation of students to discuss the algorithm with their peers and to ask critical questions about the algorithm.

The results of the empirical investigation, therefore, complete the body of evidence to show that BeReT supports all seven requirements of program comprehension identified in Chapter 2. It can, therefore, be concluded that incorporating a tool, such as BeReT, into the introductory programming module can be of benefit to the students.

The evidence in favour of BeReT shows that a technological support tool based on beacon recognition and chunking will add benefit if incorporated into the introductory programming module. The following chapter concludes this dissertation and contributes a plan for the incorporation of BeReT into the teaching model of introductory programming in the Department of CS&IS at NMMU.

Chapter 6 Conclusions and Recommendations

6.1 Introduction

The difficulties facing students in introductory programming prompted investigations into modifying the teaching model to improve student performance and pass rates. The primary goal of this dissertation is to determine the effect of an experimental technological support tool on the in-depth code comprehension of students in an introductory programming module. An analysis of various cognition models of expert programmers during in-depth code comprehension revealed beacon recognition and chunking as the techniques commonly used to build the mental model of expert programmers during code comprehension.

An experimental Beacon Recognition Tool (BeReT) was developed to train students in chunking and the identification of programming beacons in source code. BeReT was developed to meet the requirements derived in Chapter 2. After BeReT was used for training, the students were assessed to determine the impact of BeReT on their in-depth comprehension of introductory programming algorithms.

This chapter concludes the dissertation by examining the research done (Section 6.2) and proposing a methodology for incorporating BeReT in the teaching model of an introductory programming module (Section 6.5). The limitations (Section 6.3) and possibilities for future research are also addressed (Section 6.4).

6.2 Research achievements

The achievements of this research are apparent as components of both theoretical and practical contributions.

Theoretical contributions are:

- A derived list of requirements to support in-depth comprehension of introductory algorithms (Chapter 2), and
- An experimental design which incorporated the use of an eye-tracker to supplement performance results (Chapter 4).

Practical contributions are:

- An evaluation of existing technological support tools using the derived requirements to support in-depth comprehension of introductory algorithms (Chapter 3), and
- Results of the investigation to determine the impact of an in-depth code comprehension tool (BeReT) in an introductory programming module (Chapter 5).

Each of the above-mentioned contributions adds to the existing body of knowledge on code comprehension in introductory programming. This research also resulted in a peer reviewed article presented at the 2007 SAICSIT conference and published in the conference proceedings (Appendix J).

6.2.1. Theoretical contributions

An investigation into the mental model of expert programmers during code comprehension established beacon recognition and chunking as techniques used by expert programmers when trying to comprehend algorithms (Section 2.2). This analysis of the behaviour of expert programmers as well as an overview of various techniques used by educators to aid their students in their code comprehension activities (Section 2.3) resulted in a list of requirements for a tool to support program comprehension in introductory programming (Table 6-1).

Chapter 2 also acknowledged that any study involving a comparison of the achievement of students in a module requires a method for assessing their knowledge.

Bloom's taxonomy is presented as a comprehensive technique for assessing student's knowledge on a variety of levels. This taxonomy is adapted to assessing code comprehension for the purpose of this study (Section 2.4). The scope of this investigation is limited to the first two levels of Bloom's taxonomy, namely Knowledge (Section 2.4.1) and Comprehension (Section 2.4.2).

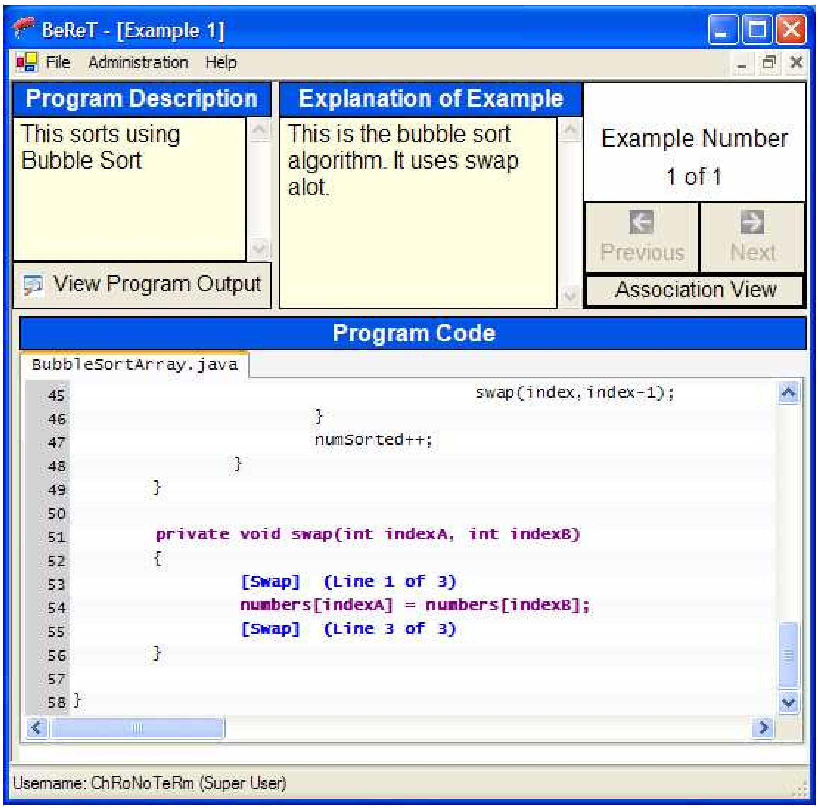
The in-depth comprehension requirements derived in Chapter 2 (Table 6-1) were derived from an investigation of expert program comprehension ***(R1, R2*** and ***R3)***, techniques used by educators to aid program comprehension in introductory programming ***(R4*** and ***R5)*** and educational objectives evident in Bloom's taxonomy ***(R6*** and ***R7)***.

	Requirements to support in-depth comprehension of algorithms
R1	Allows student to extract semantic information (plans) from worked example
R2	Promotes the use of chunking to learn an unfamiliar algorithm
R3	Coaches students to spot beacons in different algorithms
R4	Provides students with syntactically and semantically correct worked examples
R5	Engages students in active learning activities
R6	Increases ability to recall the semantics of an algorithm
R7	Enables students to successfully transfer knowledge

Table 6-1 Requirements to aid the in-depth comprehension of introductory algorithms

BeReT (Figure 6-1) was designed and implemented to specifically satisfy requirements ***R1 – R5*** (Section 3.4).

The tutorials in BeReT present students with an algorithm and explain plans used in the construction of the algorithm ***(R1).*** The exercises in BeReT provide the same algorithm, and students must show their knowledge of the plans implemented in the code ***(R1).*** Chunking ***(R2)*** and beacon recognition ***(R3)*** are the predominant techniques to assimilate the plans. Any syntactically and semantically correct algorithm can be loaded by the lecturer, in any programming language ***(R4).*** Exercises engage students in active learning activities ***(R5).***



This sorts using Bubble Sort

This is the bubble sort algorithm. It uses swap alot.

Example Number
1 of 1

```
                                        swap(index,index-1);
                            }
                            numSorted++;
                  }
         }

         private void swap(int indexA, int indexB)
         {
                  [Swap]  (Line 1 of 3)
                  numbers[indexA] = numbers[indexB];
                  [Swap]  (Line 3 of 3)
         }

}
```

Username: ChRoNoTeRm (Super User)

Figure 6-1 BeReT Tutorial

Evidence of the support provided by BeReT to increase students' ability to recall the semantics of an algorithm ***(R6)*** and the support in BeReT to enable students to successfully transfer knowledge ***(R7)*** is not immediately apparent in BeReT's design and implementation. Empirical data is required to provide evidence in support of these requirements. The empirical study is, therefore, used to provide evidence of student comprehension of the algorithms studied ***(R6)*** and the ability of the students to transfer the knowledge gained in the tool to future program comprehension activities ***(R7)***.

Chapter 4 focused on the methodology used in the investigative study to collect the necessary empirical data. The investigative study took place in the context of the learning environment of the introductory programming module in the Department of CS&IS at NMMU. Various materials were developed and used during the investigative study. In order to ensure data is valid and usable for deliberation, Chapter 4 presented a plan to analyse the data and strategies to address the risks associated with the study. The methodology incorporated the use of an eye tracker to collect metrics other than time and accuracy to assess the performance of students (Section 4.3.5).

Studies of program comprehension rarely incorporate the use of eye-tracking to enhance understanding of programmer behaviour (Section 4.3.5). Cognitive processes during reading tasks are mostly done using techniques such as think-aloud protocols and observational studies, and student comprehension are mostly assessed using performance measures such as accuracy and time (Section 4.3.4). In this study, eye-tracking visualizations such as scanpaths and heat maps suggest further confirmation of the accuracy and time taken to find answers to comprehension questions (Section 4.3.5).

6.2.2. Practical contributions

Chapter 3 employed the requirements in Table 6-1 as an evaluation tool to investigate from available literature a subset of representative categories of tools in order to determine their level of support for these requirements for in-depth code comprehension.

		Existing Tools					
	Requirement	**JHAVé**	**PROTOTYPE ALGORITHM ANIMATION TOOL**	**JGRASP**	**PCV**	**PROGUIDE**	**BERET**
R1	Allow students to extract semantic information (plans) from worked example	✓	✓			✓	✓
R2	Promotes the use of chunking to learn an unfamiliar algorithm		X				✓
R3	Coaches students to spot beacons in different algorithms		X				✓
R4	Provide students with syntactically and semantically correct worked examples	✓	X	✓		✓	✓
R5	Engage student in active learning activities	✓	✓	✓	✓	✓	✓
R6	Increases ability to recall an algorithm			✓		✓	?
R7	Enable students to successfully transfer knowledge			✓	✓	✓	?

<u>Legend</u>

✓	Requirement is supported
X	Requirement is not supported
No entry	Inconclusive evidence in available literature in support of requirement
?	Determined by means of an empirical investigation (Chapters 4 and 5)

Table 6-2 Comparison of support for algorithm comprehension evident in various tools

This comparative study of support tools used by educators identified tools that meet most requirements (Table 6-2), except for requirements ***R2*** (Promotes the use of chunking to learn an unfamiliar algorithm) and ***R3*** (Coaches students to spot beacons in different algorithms) specifically. An experimental support tool (BeReT) was designed and implemented with specific support for requirements ***R1 – R5***. This research has therefore shown that the list of requirements can be used as a measuring instrument to evaluate existing technological support tools that can be used for code comprehension in introductory programming (Section 3.3). The lack of support for the requirements ***(R2*** and ***R3)*** prompted the design and implementation of BeReT. An investigative study to find evidence in support for requirements ***R6*** and ***R7*** showed that students who used BeReT to study three algorithms (binary search, bubble sort and insertion sort) performed significantly better on code comprehension questions than the students that relied on pen-and-paper exercises (Table 6-3).

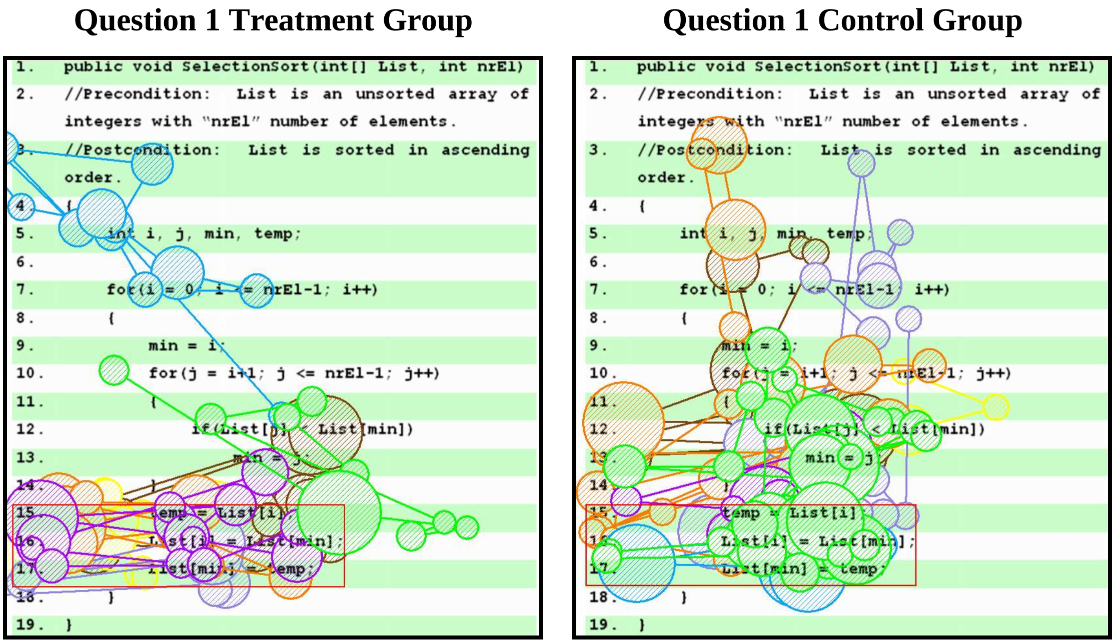
Ref No.	**Assessment Performance Measure**	**Treatment**	**Control**	**p-value**	**Requirement**
1	Average mark Bubble Sort Recall (*n*=32)	56%	44%	0.0169*	***R6***
2	Average mark Bubble Sort Comprehension (*n*=32)	55%	35%	0.0003**	***R6***
3	Average mark Binary Search Recall (*n*=32)	41%	22%	0.0017**	***R6***
4	Average mark Binary Search Comprehension (*n*=32)	61%	42%	0.0004**	***R6***
5	Average mark Insertion Sort Complete (*n*=32)	56%	39%	0.0005**	***R6***
6	Average mark Insertion Sort Comprehension (*n*=32)	43%	27%	0.0002**	***R6***
7	Known Algorithm Plan Identification Time (*n*=7)	6 completed within 10 sec	0 completed within 10 sec	0.014*	***R6 & Partially R7***

* Significant where $p<0.05$; ** Significant where $p<0.01$

Table 6-3 Overview of significant results from investigative study

Students that used BeReT also found recalling and completing algorithms much easier (Ref No. 1, 3 and 5). The empirical data showed that students that used BeReT find answers to comprehension questions quicker in known algorithms (Ref No 2, 4 and 6) (Table 6-3). Students in the treatment group also performed the task of identifying a known algorithm from beacon like code fragments quicker than the control group (Ref No. 7).

Scanpaths and heat maps generated from eye tracking data revealed the students that studied with the help of BeReT suggest evidence of plan knowledge and beacon recognition. The eye tracking images showed that these successful students spent little time reading irrelevant sections of code, and instead focused mostly on known programming plans used in previously unseen algorithms (Figure 6-2). They were also able to identify these plans quicker than the students that did not use BeReT during the training phase (Figure 6-3). Since training only differed in the treatment group performing code comprehension exercises in BeReT and the control group performing pen-and-paper based code comprehension exercises, it can be concluded that BeReT assisted in this improvement in performance.



Question 1 Treatment Group

```
public void SelectionSort(int[] List, int nrEl)
//Precondition: List is an unsorted array of
integers with "nrEl" number of elements.
//Postcondition: List is sorted in ascending
order.
{
      int i, j, min, temp;

      for(i = 0; i <= nrEl-1; i++)
      {
            min = i;
            for(j = i+1; j <= nrEl-1; j++)
            {
                  if(List[j] < List[min])
                        min = j;
            }
            temp = List[i];
            List[i] = List[min];
            List[min] = temp;
      }
}
```

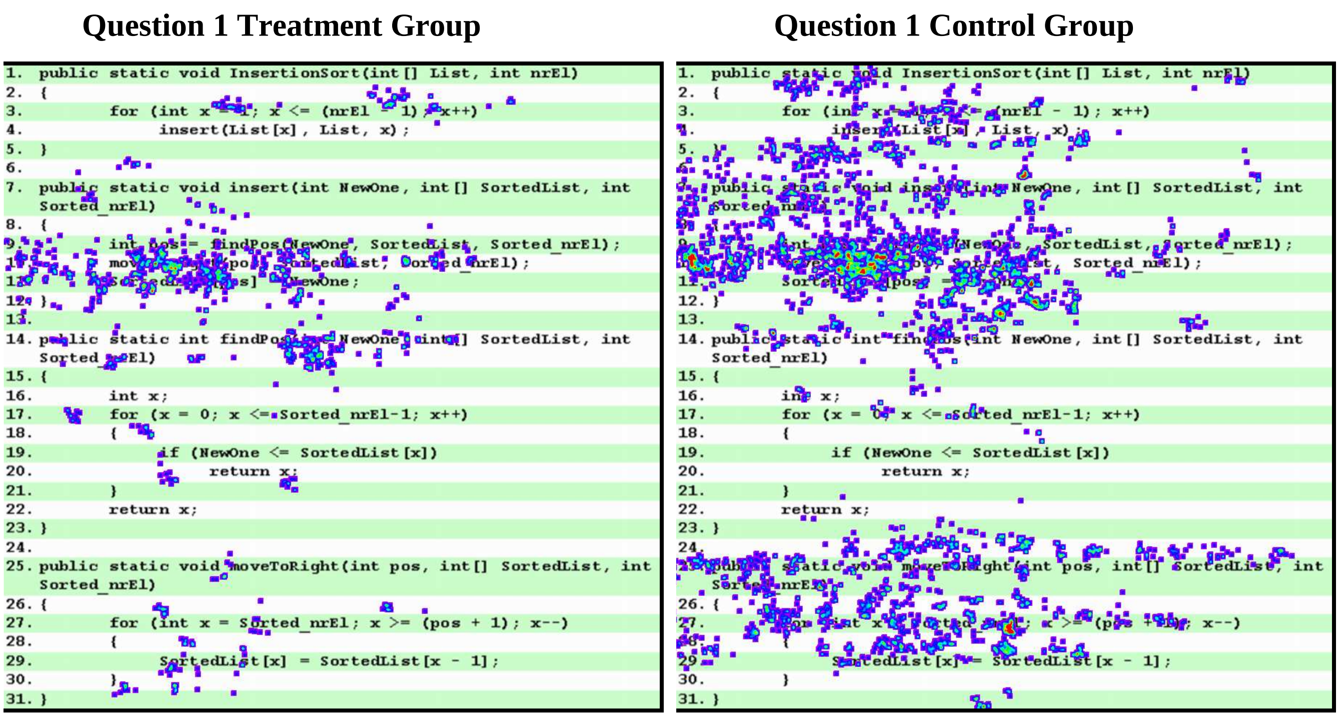
(a)

Question 1 Control Group

```
public void SelectionSort(int[] List, int nrEl)
//Precondition: List is an unsorted array of
integers with "nrEl" number of elements.
//Postcondition: List is sorted in ascending
order.
{
      int i, j, min, temp;

      for(i = 0; i <= nrEl-1; i++)
      {
            min = i;
            for(j = i+1; j <= nrEl-1; j++)
            {
                  if(List[j] < List[min])
                        min = j;
            }
            temp = List[i];
            List[i] = List[min];
            List[min] = temp;
      }
}
```

(b)

Figure 6-2 Scanpaths indicating plan discovery in unseen algorithm



Question 1 Treatment Group

```
public static void InsertionSort(int[] List, int nrEl)
{
        for (int x = 1; x <= (nrEl - 1); x++)
            insert(List[x], List, x);
}

public static void insert(int NewOne, int[] SortedList, int
Sorted_nrEl)
{
        int pos = findPos(NewOne, SortedList, Sorted_nrEl);
        moveToRight(pos, SortedList, Sorted_nrEl);
        SortedList[pos] = NewOne;
}

public static int findPos(int NewOne, int[] SortedList, int
Sorted_nrEl)
{
        int x;
        for (x = 0; x <= Sorted_nrEl-1; x++)
        {
            if (NewOne <= SortedList[x])
                return x;
        }
        return x;
}

public static void moveToRight(int pos, int[] SortedList, int
Sorted_nrEl)
{
        for (int x = Sorted_nrEl; x >= (pos + 1); x--)
        {
            SortedList[x] = SortedList[x - 1];
        }
}
```

Question 1 Control Group

```
public static void InsertionSort(int[] List, int nrEl)
{
        for (int x = 1; x <= (nrEl - 1); x++)
            insert(List[x], List, x);
}

public static void insert(int NewOne, int[] SortedList, int
Sorted_nrEl)
{
        int pos = findPos(NewOne, SortedList, Sorted_nrEl);
        moveToRight(pos, SortedList, Sorted_nrEl);
        SortedList[pos] = NewOne;
}

public static int findPos(int NewOne, int[] SortedList, int
Sorted_nrEl)
{
        int x;
        for (x = 0; x <= Sorted_nrEl-1; x++)
        {
            if (NewOne <= SortedList[x])
                return x;
        }
        return x;
}

public static void moveToRight(int pos, int[] SortedList, int
Sorted_nrEl)
{
        for (int x = Sorted_nrEl; x >= (pos + 1); x--)
        {
            SortedList[x] = SortedList[x - 1];
        }
}
```

(a) (b)

Figure 6-3 Heat Map indicating beacon recognition in known algorithm

6.3 Limitations of research

The limitations of this research are related to the limited exposure to the treatment tool, system errors in BeReT, limited information regarding which lines of code are regarded by experts as beacons for comprehension, the limitations imposed by eye-tracking and insufficient data to draw accurate conclusions on one aspect of the Hawthorne effect.

The training period employed in the current investigation only spanned three weeks. This limited exposure to BeReT could inhibit the motivation of students from using chunking and beacon recognition during code comprehension after the treatment period concluded. Lack of discussion around the topic of code comprehension during lectures could possibly further exacerbate the problem.

Apart from the limited exposure, the time spent in BeReT was further hampered by errors in the software. Although BeReT is fully functional according to the initial requirements, the biggest problem manifested itself when multiple users accessed the exercises at different times. This problem was discovered only after the treatment period started and negated the self-paced nature of BeReT. Students had to start each exercise together, otherwise the exercise was removed from the list of possible exercises as soon as someone completed an exercise. The exercises had to be reset every time somebody accidentally opened and closed an exercise before everyone started, resulting in a loss of student responses to exercises already completed.

Due to the scope of the current investigation, one of the exercises supported in BeReT was not used. The lack of following a rigorous approach to determine exactly which lines of code are deemed beacons in the chosen algorithms meant that the *Define a Beacon* exercise was not yet used to its full potential. This exercise possibly has the potential to train students better in the skill of beacon recognition.

Due to the nature of the experiments involving eye-tracking, only a small subset of both groups could participate. The small sample sizes (n=7) for the controlled experiments result in a limit of the application of formal statistical testing for some of the results obtained during these experiments.

One aspect of the Hawthorne Effect identified as the effect of using a novel tool in teaching is not effectively measured in the current investigation. The impact of this effect is unknown. The limitations identified in this section give an indication of future research directions.

6.4 Future research

Reasons for students not displaying a tendency towards using beacon recognition and chunking when studying an unseen algorithm can be investigated. Experiments similar to the current investigation can be performed, but over a longer treatment period. The training period in the current investigation was only three weeks due to logistical reasons. A controlled experiment whereby students have to use any technique they feel comfortable with to study an unseen algorithm containing known plans revealed only one treatment group participant that used some technique promoted in BeReT (Section 5.2.2). This reluctance to use chunking and beacon recognition can possibly be attributed to the short exposure students have had to BeReT. The lecturer used tracing as the predominant comprehension technique during lectures. It is, therefore, reasonable to assume that tracing will be the preferred technique used by students when confronted by an unseen algorithm. This preference for the tracing technique can also be credited to the small amount of contact time students had with BeReT as well as the lack of explicit alternative code comprehension skills covered in lectures.

The effect of formally including the topic of code comprehension in lectures should, therefore, also be investigated. Part of these follow-up investigations should include the exercise type (*Define a Beacon*) not used in the current study.

To make the best possible use of the *Define a Beacon* type BeReT exercise, an investigation into what specifically can be considered as beacons in introductory programming algorithms should be conducted. It is envisaged that such an investigation could possibly involve the study of experts in the eye-tracking laboratory to determine where their focus is during code comprehension activities. The expert programmers could be given a number of different introductory programming algorithms to study and a comparison of their answers to comprehension questions and their visual attention could reveal insights into what can be considered as beacons.

The study of expert programmers should include additional algorithms not used during this investigation. The results of the current investigation only concluded on the effect of BeReT on algorithms typically taught during introductory programming. The effect of BeReT on algorithms of different sizes and complexity would further focus the proposal for incorporating BeReT in the teaching model of an introductory programming module. The effect of BeReT was measured against pen-and-paper-based comprehension exercises in the current study. A follow-up study can be conducted that compares BeReT against other forms of interventions, for example, additional lectures or peer-led learning exercises. The current investigation only included data with regards to the transfer of plan knowledge to the comprehension of unseen algorithms with known plans. Data to determine the effect of BeReT on the transfer of plans to code generation exercises could also follow. Subsequent investigations could also determine the effect of BeReT on the cognitive development of the students on the application, analysis, synthesis and evaluation levels of Bloom’s taxonomy (Section 2.4).

Further investigations into the effect of BeReT on the comprehension of introductory programming should include a proper usability study of BeReT. This study can determine how students of varying experience levels and academic ability interacts with BeReT. The questionnaire completed by treatment group participants further identified the following features students want included in BeReT:

- Allow exercises to be marked for immediate feedback,
- Include an animation screen to visualise the execution of each plan,
- Provide a comparison view of different ways of achieving the same plan,
- Show arrows to link beacon description with corresponding lines of code,
- Allow stepping through code, showing memory status at various points,
- Allow selection of lines during completion exercises.

Future revisions of BeReT could also include features not necessarily requested by the students, but which proved to be successful in existing code comprehension tools. Incorporating visual cues associated with control structure diagrams would be one example of a feature that could add value to BeReT (Section 2.3). This could aid control flow navigation, which in turn has the potential of improving in-depth code comprehension (Section 2.2.3). Recall that programmers following a top-down model of code comprehension typically start by generating hypotheses about an algorithm, and then search for beacons to confirm their hypotheses (Section 2.2.2). Adding functionality into BeReT to record hypotheses generated by students would provide additional data that could help with an investigation into the students' mental model while reading algorithms. This functionality of storing hypotheses in BeReT can also guide students when following the top-down model of code comprehension.

A feasibility study is required to determine which of these features are possible to implement in the current version of BeReT. This study can incorporate the use of the eye-tracker and other formal usability equipment to determine how students use and interact with BeReT.

6.5 Incorporation of BeReT into introductory programming

The significantly positive effect BeReT has on students' comprehension of introductory algorithms (Table 6-3) necessitates a proposal for the incorporation of

BeReT into the teaching model of introductory programming at NMMU. The proposal focuses on the selection of algorithms treatable in BeReT.

While program generation is the most appropriate technique for teaching and learning programming, arguments can be made in support of the benefits of incorporating explicit program comprehension activities (Chapter 1). Currently, in the introductory programming module in the Department of CS&IS at NMMU, program comprehension activities are limited to the instructor tracing algorithms, prior to a code generation practical assignment (Section 4.2). BeReT is an attempt to incorporate explicit interactive program comprehension activities into an introductory programming module.

After initial exposure to an algorithm, students are usually required to read the code for general understanding. Since this requires a systematic reading approach, it is proposed that BeReT tutorials and exercises should first focus on how the algorithm implements various plans (Section 2.2.1). The initial tutorial should include a formulation of the problem, steps towards a solution for the problem and an example of the final solution in a relevant programming language (Section 2.3). A guide for the lecturer explaining how to set up a tutorial in BeReT appears in Appendix H. Subsequent tutorials and exercises can then focus on beacon recognition in a variety of similar algorithms (Section 2.2.2).

Based on the theory reported on in this investigative study, the following set of criteria may be used when selecting algorithms for treatment in BeReT (Chapters 2 and 3):

- The algorithms exhibit a fine balance between simplicity and complexity.
- The algorithms contain constructs mastered by the students.
- The algorithms contain stereotypical plans used in different algorithms.
- The algorithms contain beacon-like statements.

1. The algorithms exhibit a fine balance between simplicity and complexity.

The lecturers involved in all years of programming should investigate which algorithms students experience the most problems with. For example in this study, the insertion sort, in particular, was identified by students to be a difficult one to comprehend (Section 5.3.2). The ultimate goal of the investigation should be to derive a list of algorithms (on all year levels of study) suitable for treatment in BeReT. The simplicity of the selected algorithms in the current investigation is derived from the fact that they are not full-scale software applications. BeReT would not be a suitable tool for comprehension of such large-scale applications, since no functionality for visualisation of control flow or data flow typically found in code comprehension tools for such large scale applications, exists in BeReT.

2. The algorithms contain constructs mastered by the students.

It is assumed that students using BeReT should have at least mastered the constructs used in the algorithms presented in BeReT. Should BeReT, for instance, be used to comprehend worked examples during the initial weeks of the first semester, the algorithm selected must not contain looping constructs if they have not been covered yet (Section 2.2.2). Moreover, it must still be investigated whether BeReT has a significant positive effect if the algorithm selected is small. Adhering to standardized programming conventions would structure the algorithm in a familiar way for the students, which would aid plan discovery (Section 2.2.2).

3. The algorithms contain stereotypical plans used in different algorithms.

One of the aims of BeReT is to develop a repository of plans in students' long-term memory that they can transfer between different algorithms (Section 2.2). This knowledge must also be readily available during program generation exercises. It is obvious that BeReT will only be successful in this goal if suitable plans are implemented in the selected algorithm. BeReT tutorials should be used to visually show students how the plans are implemented and BeReT exercises should be used to assess their knowledge of the plans in each algorithm (Section 3.4).

4. The algorithms contain beacon-like statements.

This criterion assumes knowledge of exactly which statements can be regarded as lines of code that can be regarded as beacons. The current investigation relied on experience of educators and the limited literature providing insights into which lines are considered beacons by experts to apply this criterion (Section 2.2). It is suggested that a more rigorous analysis of introductory programming algorithms be conducted to determine which lines of code can be classified as beacons.

6.6 Summary

A lack of explicit code comprehension activities evident in the introductory programming module in the Department of CS&IS at NMMU gave rise to an investigation into changing the teaching model to include a support tool that focuses on code comprehension. A literature review of code comprehension performed by expert programmers, techniques used by educators to aid code comprehension and Bloom's taxonomy of educational objectives resulted in a list of requirements for a code comprehension tool. These requirements were used to collect evidence from available literature for a single technological support tool that supports all the requirements. The lack of existence of such a support tool resulted in the design and implementation of an experimental tool that supports all requirements (BeReT). It was clear from a heuristic evaluation of BeReT that it meets requirement ***R1 – R5*** (Table 6-4), but more investigation was needed to collect evidence in support for two requirements that are not immediately evident in the design of BeReT ***(R6*** and ***R7)***.

An empirical investigation successfully determined a significantly positive effect of BeReT on the in-depth comprehension of algorithms in an introductory programming module. The results of the empirical investigation contribute to a body of evidence that proves the incorporation of BeReT in an introductory programming module is beneficial in respect of the following:

- BeReT increases the ability of students to recall the semantics of an algorithm,
- BeReT enables students to successfully transfer plan knowledge between algorithms.

	Requirements to support in-depth comprehension of introductory algorithms	**BERET**	**Supporting Evidence**
R1	Allows students to extract semantic information (plans) from worked example	✓	**Section 3.4**
R2	Promotes the use of chunking to learn an unfamiliar algorithm	✓	
R3	Coaches students to spot beacons in different algorithms	✓	
R4	Provides students with syntactically and semantically correct worked examples	✓	
R5	Engages student in active learning activities	✓	
R6	Increases ability to recall an algorithm	✓	**Ch 5**
R7	Enables students to successfully transfer knowledge	✓	

Table 6-4 Support for algorithm comprehension evident in BeReT

This dissertation contributes a proposal for the incorporation of BeReT into the introductory programming module in the Department of CS&IS at NMMU. The theoretical and practical contributions of this research offer educators in introductory programming an alternative way of incorporating program comprehension skills into their teaching model.

References

ALA-MUTKA, K. (2003): Problems in learning and teaching programming. http://www.cs.tut.fi/~edge/literature_study.pdf, Date Accessed: June 2005

APPLIN, A.G. (2001): Second Language Acquisition and CS1: Is * = = **? *ACM SIGCSE Bulletin* **33**(1):174 - 178

AREIAS, C.M. and MENDES, A. (2006): ProGuide: A Dialogue-Based Tool To Support Initial Programming Learning. In *Proceedings 3rd E-Learning Conference – Computer Science Education*, Coimbra, Portugal.

ASCHWANDEN, C. and CROSBY, M. (2006): Code Scanning Patterns in Program Comprehension. In *Proceedings of the 39th Hawaii International Conference on System Sciences*, Kauai, Hawaii.

ASTRACHAN, O., BERRY, G., COX, L. and MITCHENER, G. (1997): Design Patterns: An essential component of CS Curricula. In *Proceedings 28th SIGCSE Technical Symposium on Computer Science Education*. 153 - 160

BARTELS, M. and MARSHALL, S.P. (2006): Eye Tracking Insights into Cognitive Modelling. In *Proceedings ETRA 2006*, San Diego, California. 141 - 178

BEDNARIK, R., MYLLER, N., SUTINEN, E. and TUKIAINEN, M. (2006): Program Visualization: Comparing Eye-Tracking Patterns with Comprehension Summaries and Performance. In *Proceedings 18th Workshop of the Psychology of Programming Interest Group*, University of Sussex. 68 - 82, ROMERO, P., GOOD, J., CHAPARRO, E.A. and BRYANT, S. (eds).

BEDNARIK, R. and TUKIAINEN, M. (2006): An eye-tracking methodology for characterizing program comprehension processes. In *Proceedings of the 2006 symposium on Eye tracking research & applications*, San Diego, California. 125 - 132, ACM Press

BEN-ARI, M. (2001): Constructivism in Computer Science Education. *Journal of Computers and Mathematics in Science Teaching* **20**(1):45 - 73

BLOOM, B.S. (1956): *Taxonomy of Educational Objectives*. David McKay Co.

BOYLE, T. (2003): Design principles for authoring dynamic reusable learning objects. *Australian Journal of Educational Technology* **19**(1):46-58

BROOKS, R. (1983): Towards a theory of the comprehension of computer programs. *International Journal of Man-Machine Studies* **18**(6):543 - 554

BROWN, M.H. (1988): Exploring algorithms using BALSA-II. *IEEE Computer* **21**(5):14 - 36

BRUMMUND, P. (2001): The Complete Collection of Algorithm Animations. http://www.cs.hope.edu/alganim/ccaa/ Last updated: 26 June 2001, Date Accessed: May 2006

BUCK, D. and STUCKI, D.J. (2000): Design early considered harmful: graduated exposure to complexity and structure based on levels of cognitive development. *SIGCSE Bulletin* **32**(1):75 - 79

BUCKLEY, J. and EXTON, C. (2003): Blooms' Taxonomy: A Framework for Assessing Programmers' Knowledge of Software Systems. In *Proceedings 11th IEEE International Workshop on Program Comprehension*. 165

BURNSTEIN, I. and ROBERSON, K. (1997): Automated Chunking to Support Program Comprehension. In *Proceedings 5th International Workshop on Program Comprehension*. 40, IEEE Computer Society

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.

CARBONE, A. and KAASBØLL, J.J. (1998): A Survey of Methods Used to Evaluate Computer Science Teaching. In *Proceedings Information Technology in Computer Science Education*, Dublin, Ireland. 41 - 45

CHAN, P.S. and MUNRO, M. (1997): PUI: A Tool to Support Program Understanding. In *Proceedings IEEE 5 th International Workshop on Program Comprehension*. 192-198

CILLIERS, C.B. (2004): A Comparison of Programming Notations for a Tertiary Level Introductory Programming Course. Doctoral thesis. Department of Computer Science and Information Systems, UPE. Port Elizabeth.

CILLIERS, C.B., CALITZ, A.P. and GREYLING, J.H. (2005): The effect of integrating an iconic programming notation into CS1. In *Proceedings of the 10th annual SIGCSE conference on Innovation and technology in computer science education*, Caparica, Portugal. 108 - 112, ACM Press, http://doi.acm.org/10.1145/1067445.1067478, Date Accessed: May 2006

CLANCY, M.J. and LINN, M.C. (1999): Patterns and pedagogy. In *Proceedings 30th Technical Symposium on Computer Science Education*, New Orleans, Louisiana, United Stated. 37 - 42

COOK, C., BREGAR, W. and FOOTE, D. (1984): A Preliminary Investigation of the use of the Cloze Procedure as a Measure of program Understanding. *Information Processing & Management* **20**(1 - 2):199 - 208

CREWS, T. and ZIEGLER, U. (1998): The Flowchart Interpreter for Introductory Programming Courses. In *Proceedings 1998 Frontiers in Education Conference*, Tempe, Arizona.

CROSBY, M.E., SCHOLTZ, J. and WIEDENBECK, S. (2002): The roles beacons play in comprehension for novice and expert programmers. In *Proceedings of PPIG*, Brunel University, UK. **14**:58 - 73, J. KULJIS, L.B.R.S. (ed)

CROSBY, M.E. and STELOVSKY, J. (1990): How do we read algorithms? A case study. *Computer* **23**(1):24 - 35

CROSS, J.H. (1998): The Control Structure Diagram: An Overview and Initial Evaluation. *Empirical Software Engineering* **3**(2):131 - 158

CROSS, J.H., HENDRIX, T.D. and BAROWSKI, L.A. (2002): Using the debugger as an integral part of teaching CS1. In *Proceedings 32nd ASEE/IEEE Frontiers in Education Conference*, Boston, Massachusetts.

CS&IS (2006a): WRA101 (Algorithmics 1.1) Module Guide. Port Elizabeth, Department of Computer Science and Information Systems, NMMU.

CS&IS (2006b): WRA102 (Algorithmics 1.2) Module Guide. Port Elizabeth, Department of Computer Science and Information Systems, NMMU.

CURTIS, B. (1981): A Review of Human Factors Research on Programming Languages and Specifications. ACM Press:212 - 218

DE BARROS, L.N., DOS SANTOS MOTA, A.P., DELGADO, K.V. and MATSUMOTO, P.M. (2005): A tool for programming learning with pedagogical patterns. In *Proceedings 2005 OOPSLA workshop on Eclipse technology eXchange*, San Diego, California. 125 - 129

DEIMEL, L. and NAVEDA, J. (1990): Reading Computer Programs: Instructor's Guide and Exercises. *Educational Materials CMU/SEI-90-EM-3*. Pennsylvania.

DRAPER, S.W. (1997): Integrative evaluation: An emerging role for classroom studies of CAL. http://www.psy.gla.ac.uk/~steve/IE.html Last updated: 12 July 2006

ELY, M., VINZ, R., DOWNING, M. and ANZUL, M. (1999): *On Writing Qualitative Research: Living by Words*. Falmer Press.

ENGEBRETSON, A. and WIEDENBECK, S. (2002): Novice comprehension of programs using task-specific and non-task-specific constructs. In *Proceedings IEEE 2002 Symposia on Human Centric Computing Languages and Environments*. 11

EVANS, D., GUTTAG, J., HORNING, J. and TAN, Y. (1994): LCLint: a tool for using specifications to check code. In *Proceedings ACM SIGSOFT Foundations of Software Engineering*, New Orleans.

FITZGERALD, S., SIMON, B. and THOMAS, L. (2005): Strategies that students use to trace code: an analysis based in grounded theory. In *Proceedings 2005 international workshop on Computing education research*, Seattle, WA, USA. 69 - 80, ACM Press

FIX, V., WIEDENBECK, S. and SCHOLTZ, J. (1993): Mental Representations of Programs by Novices and Experts. In *Proceedings ACM CHI 93 Human Factors in Computing Systems*, Amsterdam, The Netherlands. 74 - 79, ASHLUND, S., MULLT, KEVIN, HENDERSON, AUSTIN, HOLLNAGEL, ERIK AND WHITE, TED (ed)

FORD, L. (1993): Interactive learning and researching with visualizations. Technical Report 274. University of Exeter. atlas.ex.ac.uk

GARNER, S. (2000): A Code Restructuring Tool to help Scaffold Novice Programmers. In *Proceedings International Conference in Computer Education (ICCE)*.

GARNER, S. (2002): The learning of plans in programming: A program completion approach. In *Proceedings of the International Conference on Computers in Education*. **2**:1053 - 1057

GARNER, S. (2003): Learning Resources and Tools to Aid Novices Learn Programming. In *Proceedings Informing Science & Information Technology Education Joint Conference (INSITE)*, Pori, Finland. 213 - 222

GARNER, S., HADEN, P. and ROBINS, A. (2005): My program is correct but it doesn't run: a preliminary investigation of novice programmers' problems. In *Proceedings of the 7th Australasian conference on Computing education*, Newcastle, New South Wales, Australia. **42**:173 - 180

GIANNOTTI, E. (1987): Algorithm animator: a tool for programming learning. *SIGCSE Bulletin* **19**(1):308-314

GREYLING, J.H. and CALITZ, A.P. (2003): The Implementation of a Computerised Placement Battery for First Year IT Courses. In *Proceedings 3rd International Conference on Science, Mathematics and Technology Education*, East London, South Africa.

GUZDIAL, M. (1994): Software-Realized Scaffolding to Facilitate Programming for Science Learning. *Interactive Learning Environments* **4**(1):1 - 44

HARRIS, N. (2005): Beacon Recognition Tool. Honours Treatise. Computer Science and Information Systems, University of Port Elizabeth. Port Elizabeth.

HARRIS, N. and CILLIERS, C. (2006): A Program Beacon Recognition Tool. In *Proceedings 7th International Conference on Information Technology Based Higher Education and Training, 2006. (ITHET '06)*, Ultimo, New South Wales, Australia. 216 - 225

HENDRIX, D. and CROSS, J.H. (2002): Effectiveness of Control Structure Diagrams in Source Code Comprehension Activities. *IEEE Transactions on Software Engineering* **28**(5):463 - 477

JEFFRIES, R. (1982): A comparison of the debugging behaviour of expert and novice programmers. In *Proceedings American Educationa/Research Association annual meeting*.

JENKINS, T. (2002): On the difficulty of Learning to Program. http://www.ics.heacademy.ac.uk/Events/conf2002/jenkins.html Last updated: 2002).

JOHNSON, W.L. and SOLOWAY, E. (1984): PROUST: Knowledge-based program understanding. In *Proceedings 7th international conference on Software Engineering*, Orlando, Florida, United States. 369 - 380

KIMURA, T. (1979): Reading before Composition. In *Proceedings of the tenth SIGCSE technical symposium on Computer science education*. **11**:162 - 166, ACM Press

KOENEMANN, J. and ROBERTSON, S.P. (1991): Expert problem solving strategies for program comprehension. In *Proceedings SIGCHI conference on Human factors in computing systems: Reaching through technology*, New Orleans, Louisiana. 125 - 130, ACM Press

LAHTINEN, E., ALA-MUTKA, K. and JÄRVINEN, H.-M. (2005): A study of the difficulties of novice programmers. In *Proceedings 10th annual SIGCSE conference on Innovation and technology in computer science education*, Caparica, Portugal. 14 - 18, ACM Press

LARSON, H.J. (1974): *Introduction to Probability Theory and Statistical Inference*. 2nd Edn, Wiley International Edition.

LETOVSKY, S. (1986): Cognitive processes in program comprehension. *Empirical studies of programmers*.58 - 79.

LISTER, R., ADAMS, E.S. FITZGERALD, S, FONE, W, HAMER, J, LINDHOLM, M, MCCARTNEY, R, MOSTRÖM, J.E., SANDERS, K, SEPPÄLÄ, O, SIMON, B, THOMAS, L (2004): A multi-national study of reading and tracing skills in novice programmers. SIGCSE BULLETIN 36(4):119 - 150

MATHIS, R.F. (1974): Teaching debugging. *SIGCSE Bulletin* **6**(1):59 - 63

MAYER, R.E. (1981): The Psychology of How Novices Learn Computer Programming. *ACM Computing Surveys* **13**(1):121 - 141

MCCRACKEN, M., ALMSTRUM, V, DIAZ, D, GUZDIAL, M, HAGAN, D, KOLIKANT, Y.B., LAXER, C, THOMAS, L, UTTING, I, WILUSZ, T (2001): A multi-national, multi-institutional study of assessment of programming skills of first-year CS students. ACM SIGCSE Bulletin 33(4):125 - 180.December 2001.

MCDONALD, P. and CIESIELSKI, V. (2002): Design and Evaluation of an Algorithm Animation of State Space Search Methods. *Computer Science Education* **12**(4):301 - 324

MCTIGHE, J. and SEIF, E. (2003): A Summary of Underlying Theory and Research Base for Understanding by Design. *Nebraska Association for Supervision and Curriculum Development (ASCD) Journal* **16**

MENDES, A. and MARCELINO, M. (2006): PESEN - A Visual Programming Environment to Support Initial Programming Learning. In *Proceedings CompSysTech' 2006- International Conference on Computer Systems and Technologies*, Veliko-Turnovo, Bulgaria.

MILLNER, J. (2002): Bitesize Revision. http://www.bbc.co.uk/schools/gcsebitesize/design/systemscontrol/workingwithsystemsrev3.shtml Last updated:

MOSEMANN, R. and WIEDENBECK, S. (2001): Navigation and comprehension of programs by novice programmers. In *Proceedings 9th International Workshop on Program Comprehension*, Toronto, Canada. 79 - 88

MÜLLER, H., TILLEY, S., ORGUN, M., CORRIE, B. and MADHAVJI, N. (1992): A reverse engineering environment based on spatial and visual software interconnection models. In *Proceedings Fifth ACM SIGSOFT Symposium on Software Development Environments*, Tyson's Corner, Virginia. **17**:88 - 98, ACM Software Engineering Notes

NAPS, T., EAGAN, J. and NORTON, L. (2001): JHAVÉ: An Environment to Actively Engage Students in Web-based Algorithm Visualizations. *ACM SIGCSE*:109 - 113

NAPS, T.L. (2005): JHAVé Supporting Algorithm Visualization. *IEEE Comput. Graph. Appl.* **25**(5):49 - 55

NETER, J., WASSERMAN, W. and WHITMORE, G.A. (1993): *Applied Statistics*. 4th Edn, Allyn and Bacon.

O'BRIEN, M.P. (2003): Software Comprehension – A Review & Research Direction. Technical Report UL-CSIS-03-3. University of Limerick. Ireland.

PANE, J.F. and MYERS, B.A. (1996): Usability issues in the design of novice programming interfaces. Technical Report CMU-HCI-96-101. Human-Computer Interaction Institute.

PARSONS, H.M. (1974): What happened at Hawthorne. *Science* **183**:922 - 932

PENNINGTON, N. (1987): Comprehension strategies in programming. In *Proceedings Empirical studies of programmers: Second Workshop*. 100 - 112, OLSON, S.A.S. (ed) Ablex Publishing Corp.

POZNYAKOFF, S. (2005): GNU cflow. http://www.gnu.org/software/cflow/ (Last updated: 2006/11/24).

PRETORIUS, M.C. (2005): The Added Value of Eye Tracking in the Usability Evaluation of a Network Management Tool. Masters Dissertation. Department of Computer Science and Information Systems, Nelson Mandela Metropolitan University. Port Elizabeth.

RAJLICH, V. and COWAN, G.S. (1997): Towards Standards for Experiments in Program Comprehension. In *Proceedings IEEE International Workshop on Program Comprehension*, Dearborn, MI, USA.

RAJLICH, V. and WILDE, N. (2002): The role of concepts in Program Comprehension. In *Proceedings International Workshop on Program Comprehension*. 271 - 278, IEEE Computer Society Press

RENKL, A. (1997): Learning from worked-out examples: A study on individual differences. *Cognitive Science* **21**:1 - 29

RIST, R.S. (1986): Plans in programming: definition, demonstration and development. In *Proceedings First Workshop Empirical Studies of Programmers*, Norwood, N.J. 28 - 47, Ablex Publishing

ROBBILARD, M., COELHO, W. and MURPHY, G. (2004): How effective developers investigate source code: An exploratory study. *IEEE Transactions on Software Engineering* **30**(12):889 - 903

SCANLAN, D. (1988): Should short, relatively complex algorithms be taught using both graphical and verbal methods? Six replications. *SIGCSE Bulleting* **20**(1):185 - 189

SHIH, Y.-F. and ALESSI, S.M. (1993): Mental models and transfer of learning in computer programming. *Journal of Computing in Education* **26**(2):154 - 176

SHNEIDERMAN, B. (1982): Control Flow and Data Structure Documentation: Two Experiments. *Communications of ACM*:55 - 63

SHNEIDERMAN, B. (1983): Direct Manipulation: A step beyond Programming Languages. *IEEE Computer* **16**(8):56 - 69

SHNEIDERMAN, B. and MAYER, R. (1979): Syntactic/Semantic Interaction in Programmer Behavior: A Model and Experimental Results. *International Journal of Computer and Information Sciences* **8**(3):219 - 238

SOLOWAY, E. (1986): Learning to program = learning to construct mechanisms and explanations. *Communications ACM* **29**(9):850 - 858

SOLOWAY, E. and EHRLICH, K. (1984): Empirical Studies of Programming Knowledge. *IEEE Transactions on Software Engineering* **SE-10**(5):595 - 609

SOLOWAY, E. and WOOLF, B. (1980): Problems, plans, and programs. In *Proceedings of the eleventh SIGCSE technical symposium on Computer science education*, Kansas City, Missouri, United States. 16 - 24, ACM Press

STASKO, J., BADRE, A. and LEWIS, C. (1993): Do algorithm animations assist learning? an empirical study and analysis. In *Proceedings INTERACT '93 and CHI '93 conference on Human factors in computing systems*, Amsterdam, The Netherlands. 61 - 66

SWELLER, J. and COOPER, G.A. (1985): The use of worked examples as a substitute for problem solving in learning algebra. *Cognition and Instruction* **2**(1):59 - 89

TALJAARD, M (2003): Using E-learning to Support IT Education in a University Environment: A Case Study Approach. Masters Dissertation. Department of Computer Science and Information Systems, University of Port Elizabeth. Port Elizabeth.

THOMAS, L.A., RATCLIFFE, M.B. and THOMASSON, B.J. (2004): Scaffolding with Object Diagrams in first Year Programming Classes: Some Unexpected Results. In *Proceedings 35th Technical Symposium on Computer Science Education*, Norfolk, Virginia, USA. **36**

UPCHURCH, R. (1997): Program Comprehension. Computer and Information Science Department, University of Massachusetts Dartmouth.

VAN GREUNEN, D. (2002): Formal Usability Testing of Interactive Educational Software: A Case Study Approach. Masters Dissertation thesis. Department of Computer Science and Information Systems, University of Port Elizabeth. Port Elizabeth.

VAN MERRIËNBOER, J.J.G. (1990): Strategies for programming instruction in high school: Program completion vs. program generation. *Journal of Computing Research* **6**:265 - 285

VOGTS, D. (2006): A Simplified Programming Language for Novice Programmers. Doctoral Thesis. Department of Computer Science and Information Systems, Nelson Mandela Metropolitan University. Port Elizabeth.

VON MAYRHAUSER, A. and VANS, A.M. (1994): Program Understanding: A Survey. Technical Report CS-94-120. Colorado State University. Fort Collins, Colorado.

WADDEL, K.C. and CROSS, J.H. (1988): Survey of empirical studies of graphical representations for algorithms. In *Proceedings 1988 ACM sixteenth annual conference on Computer science*, Atlanta, Georgia, United States. 696

WALENSTEIN, A. (2002): Cognitive Support in Software Engineering Tools: A Distributed Cognition Framework. PhD Thesis thesis. School of Computing Science, Simon Fraser University.

WALLINGFORD, E. (1998): Toward a First Course Based on Object-Oriented Patterns. In *Proceedings 27th SIGCSE Technical Symposium on Computer Science Education*. 27 - 31

WARREN, P.R. (2003): Learning to Program: Spreadsheets, Scripting and HCI. In *Proceedings Southern African Computer Lecturers Association*.

WIEDENBECK, S. and EVANS, N.J. (1986): Beacons in Program Comprehension. *SIGCHI Bulletin* **18**(2):56 - 57

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**:225 - 282

WIEDENBECK, S. and SCHOLTZ, J. (1989): Beacons and Initial Program Comprehension. *SIGCHI Bulletin* **21**(1):90 - 91

WILEY, D.A. (2003): Learning objects: difficulties and opportunities. http://wiley.ed.usu.edu/docs/lo_do.pdf Last updated:

WILSON, J., KATZ, I.R., INGARGIOLA, G., AIKEN, R. and HOSKIN, N. (1995): Students' use of animations for algorithm understanding. In *Proceedings Human factors in computing systems*, Denver, Colorado, United States. 238 - 239

YEH, C.L., GREYLING, J.H. and CILLIERS, C.B. (2006): A framework proposal for algorithm animation systems. In *Proceedings 2006 annual research conference of the South African institute of computer scientists and information technologists*, Somerset West, South Africa. **204**:155 - 163, South African Institute for Computer Scientists and Information Technologists

YOUNG, P. (1996): Program Comprehension. Centre for Software Maintenance, University of Durham.

Appendix A: Introduction to BeReT (Student Guide)

BeReT (short for **Be**acon **Re**cognition **T**ool) is a programme that helps you recognise and identify beacons in the source code. It can also be used as a tool to create a stepwise abstraction of an algorithm in order to simplify studying algorithms. You may be required to do any of the following three types of tasks in BeReT:

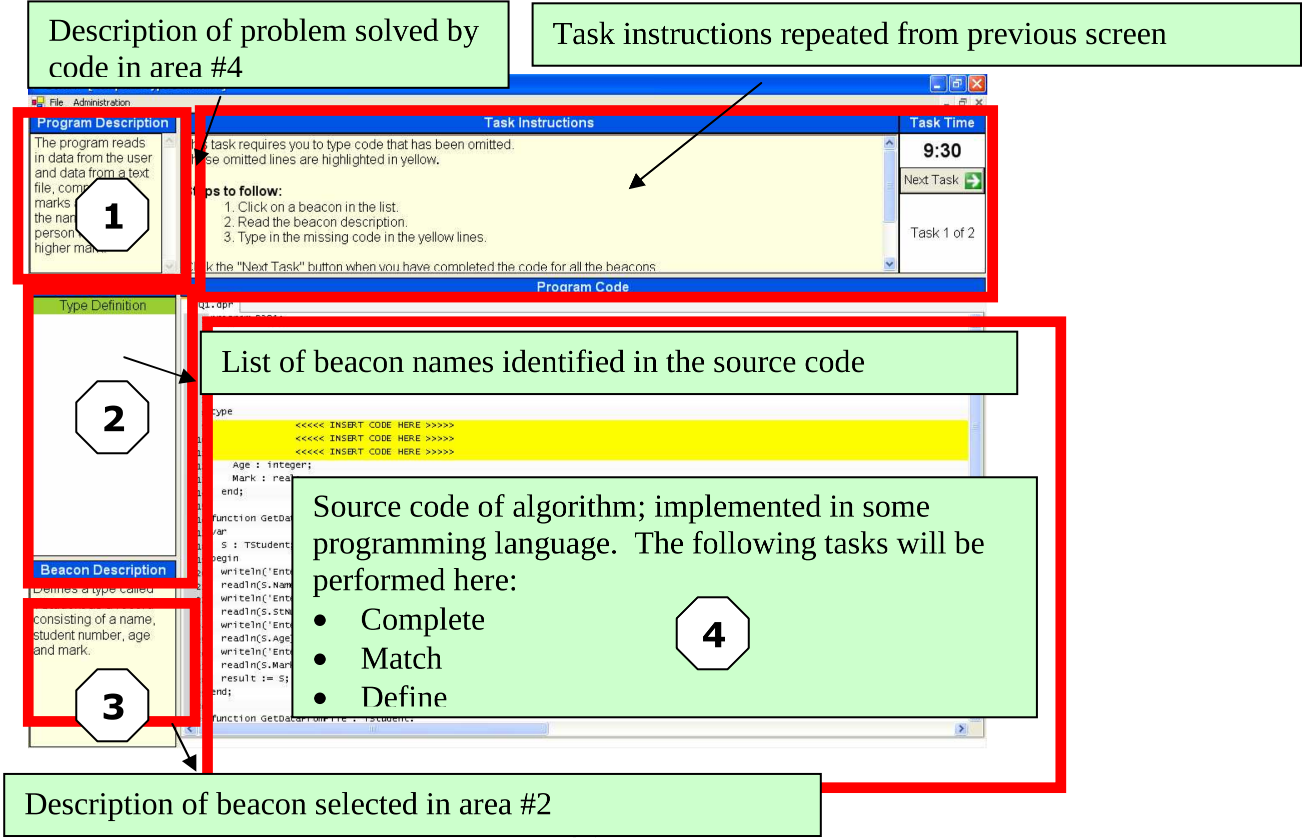
Task	**Given**	**What must be done**
Complete	Beacon name and description	Fill in missing source code that name and description refers to
Match	Beacon name and description	Select relevant source code lines that matches name and description
Define	Code only	Group statements together and assign a name and description to indicate higher level of operation of these statements

General Layout of BeReT Exercise screens

Before each task begins, an instruction page tells you about the type of task that you need to perform and lists the steps to complete the tasks. These instructions are repeated at the top of the exercise screen, but you need not read that again if you are familiar with the steps to complete each task.

From any exercise screen it is advisable that you follow the following steps in sequence:

1. Read the description of the algorithm solved by the source code in area #4.
2. Look at the list of beacon names given in area #2.
3. Click on a beacon name to read a description in area #3.
4. Complete the required task in area #4.
5. Work through all the beacon names in area #2 until “Done” appears next to all of them. Keep an eye on the “Task Time” to complete the task on time.
6. Click “Next Task” when you’ve completed all beacon names.



Description of problem solved by code in area #4

Task instructions repeated from previous screen

The program reads in data from the user and data from a text file, comp marks the nar person higher ma...

s task requires you to type code that has been omitted.
se omitted lines are highlighted in yellow.

ps to follow:
1. Click on a beacon in the list.
2. Read the beacon description.
3. Type in the missing code in the yellow lines.

k the "Next Task" button when you have completed the code for all the beacons

List of beacon names identified in the source code

```
type
            <<<<< INSERT CODE HERE >>>>>
            <<<<< INSERT CODE HERE >>>>>
            <<<<< INSERT CODE HERE >>>>>
    Age : integer;
    Mark : rea
  end;

function GetDa
var
  S : TStudent
begin
  writeln('Ent
  readln(S.Nam
  writeln('Ent
  readln(S.StN
  writeln('Ent
  readln(S.Age
  writeln('Ent
  readln(S.Mar
  result := S;
end;

function GetDataFromFile : TStudent;
```

Source code of algorithm; implemented in some programming language. The following tasks will be performed here:

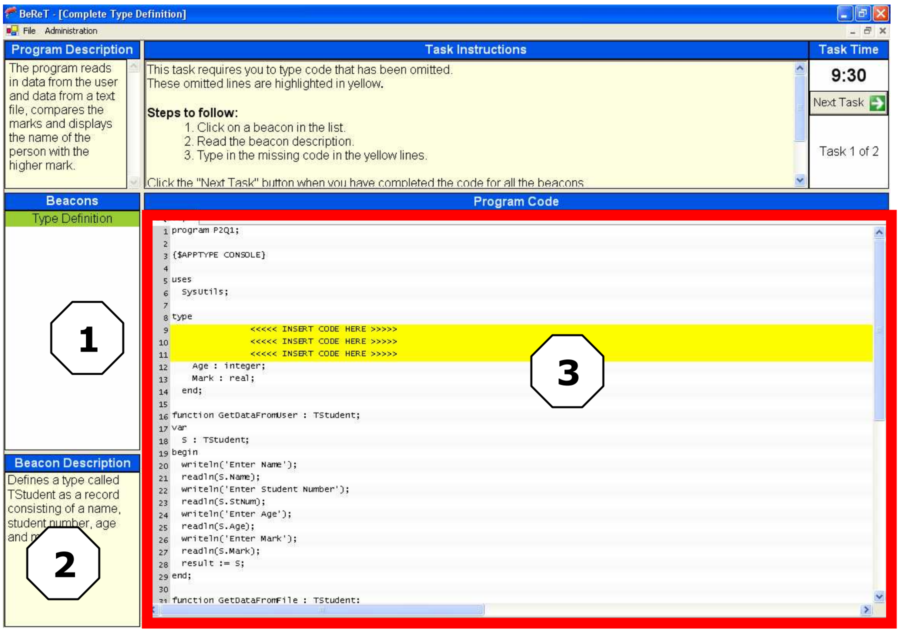
- Complete
- Match
- Define

Defines a type called consisting of a name, student number, age and mark.

Description of beacon selected in area #2

Complete missing code

1. Click on a beacon name in area #1.
2. Read the corresponding description in area #2.
3. Click on a line highlighted in yellow (in area #3), and type in the missing line(s) that matches the selected beacon name and description.
4. Press enter after each line is completed.



The program reads in data from the user and data from a text file, compares the marks and displays the name of the person with the higher mark.

This task requires you to type code that has been omitted.
These omitted lines are highlighted in yellow.

Steps to follow:

1. Click on a beacon in the list.
2. Read the beacon description.
3. Type in the missing code in the yellow lines.

Click the "Next Task" button when you have completed the code for all the beacons

9:30

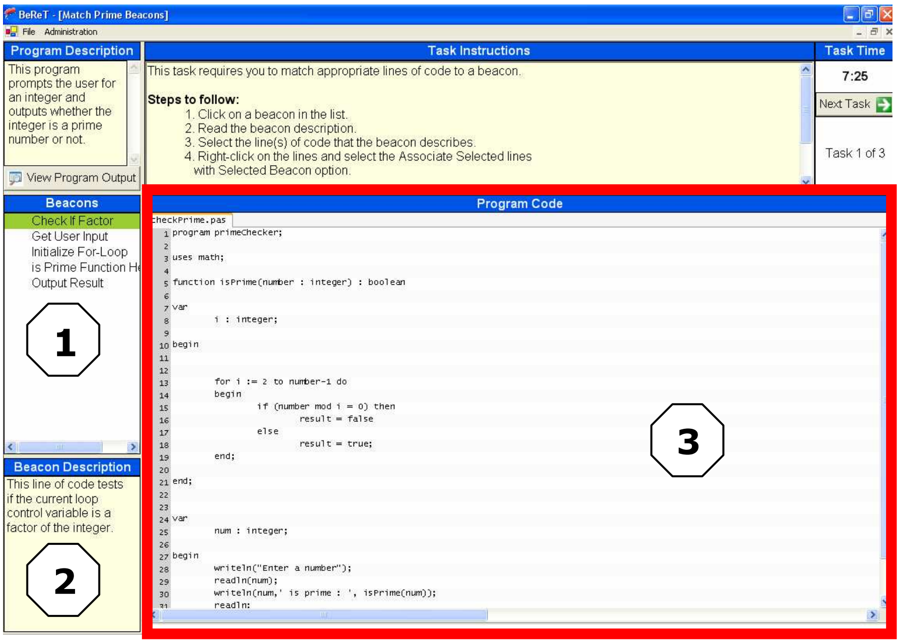
Task 1 of 2

Defines a type called TStudent as a record consisting of a name, student number, age and m

```
program P2Q1;

{$APPTYPE CONSOLE}

uses
  SysUtils;

type
                <<<<< INSERT CODE HERE >>>>>
                <<<<< INSERT CODE HERE >>>>>
                <<<<< INSERT CODE HERE >>>>>
    Age : integer;
    Mark : real;
  end;

function GetDataFromUser : TStudent;
var
  S : TStudent;
begin
  writeln('Enter Name');
  readln(S.Name);
  writeln('Enter Student Number');
  readln(S.StNum);
  writeln('Enter Age');
  readln(S.Age);
  writeln('Enter Mark');
  readln(S.Mark);
  result := S;
end;

function GetDataFromFile : TStudent;
```

Match Code to a Beacon

1. Click on a beacon name in area #1.
2. Read the corresponding description in area #2.
3. Select the line(s) of code described by the name and description in area #3.
4. Right-click on selected line and select “Associate selected lines with selected beacon” from popup menu.



Program Description: This program prompts the user for an integer and outputs whether the integer is a prime number or not.

Task Instructions: This task requires you to match appropriate lines of code to a beacon.

Steps to follow:

1. Click on a beacon in the list.
2. Read the beacon description.
3. Select the line(s) of code that the beacon describes.
4. Right-click on the lines and select the Associate Selected lines with Selected Beacon option.

Task Time: 7:25

Task 1 of 3

Beacons:

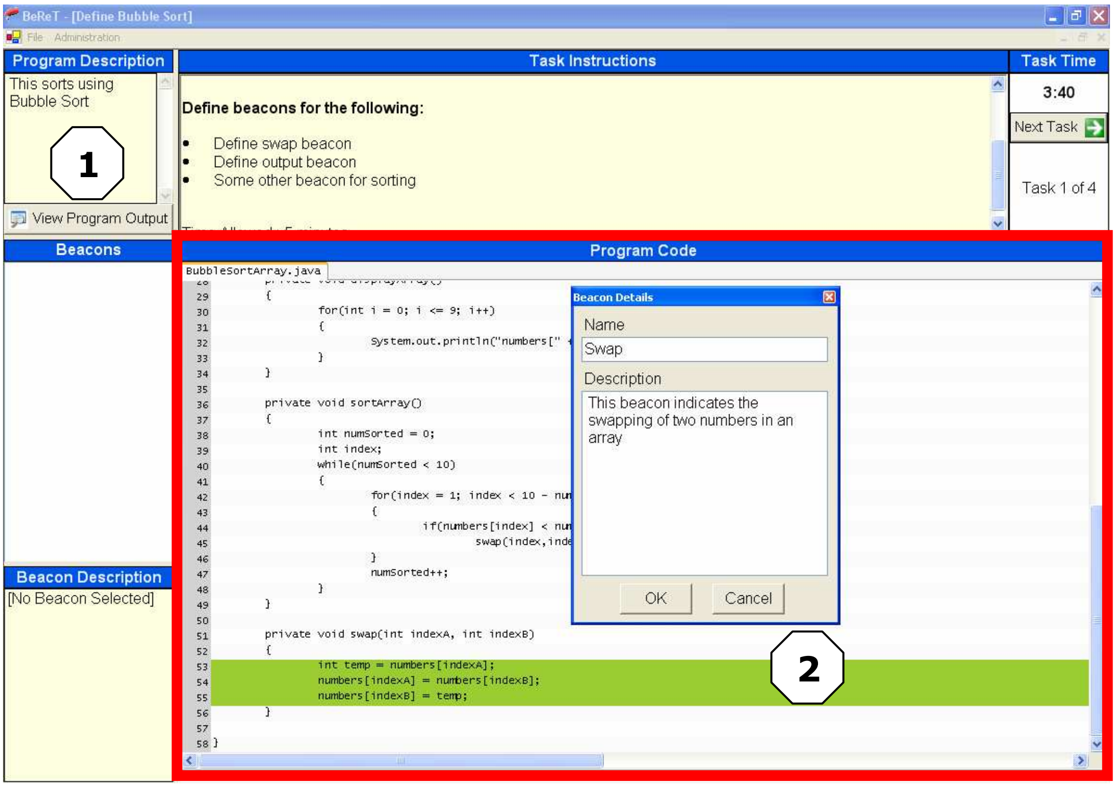
- Check If Factor
- Get User Input
- Initialize For-Loop
- is Prime Function He
- Output Result

Beacon Description: This line of code tests if the current loop control variable is a factor of the integer.

```
program primeChecker;

uses math;

function isPrime(number : integer) : boolean

var
        i : integer;

begin


        for i := 2 to number-1 do
        begin
                if (number mod i = 0) then
                        result = false
                else
                        result = true;
        end;

end;


var
        num : integer;

begin
        writeln("Enter a number");
        readln(num);
        writeln(num,' is prime : ', isPrime(num));
        readln;
```

Beacon Definition

1. Read the problem description that is solved by the algorithm
2. Scan the code to understand what each individual statement does (area #2)
3. Select the lines of code you want to group together
4. Right-click one of the selected lines and click “Create New Beacon” on the popup menu
5. Complete the details by entering a name and description appropriate for the selected lines of code



Program Description: This sorts using Bubble Sort

Task Instructions: **Define beacons for the following:**

- Define swap beacon
- Define output beacon
- Some other beacon for sorting

Task Time: 3:40

Task 1 of 4

Beacon Description: [No Beacon Selected]

```
        {
                for(int i = 0; i <= 9; i++)
                {
                        System.out.println("numbers[" +
                }
        }

        private void sortArray()
        {
                int numSorted = 0;
                int index;
                while(numSorted < 10)
                {
                        for(index = 1; index < 10 - num
                        {
                                if(numbers[index] < num
                                        swap(index,inde
                        }
                        numSorted++;
                }
        }

        private void swap(int indexA, int indexB)
        {
                int temp = numbers[indexA];
                numbers[indexA] = numbers[indexB];
                numbers[indexB] = temp;
        }

}
```

Beacon Details

Name: Swap

Description: This beacon indicates the swapping of two numbers in an array

Appendix B: Practical Assignments

The following notes were used to create BeReT Tutorials and Exercises for each algorithm.

Bubble Sort Algorithm

```
procedure BubbleSort;
var
    i : integer;
    sorted : boolean;
    TheArray : array[1..7] of integer;
Begin
    repeat
    sorted := true;

     for i := 1 to 6 do
       begin
        if TheArray[i] > TheArray[i+1] then
           begin
              sorted := false;
              Swap(TheArray[i],TheArray[i+1]);
           end;
        end;
    until sorted;
End.

procedure Swap(var a, b : integer);
var
    temp : integer;
begin
    temp := a;
    a := b;
    b := temp;
end;
```

Bubble Sort Plans

Repeat the following until the entire list is in sorted order:

1. Repeat until end of the list is reached.
    a. Determine if adjacent values are in sorted order in relation to each other.
    b. Swap values if not sorted.
        i. Make a copy of one value.
        ii. Swap two values.
        iii. Reinstate original value.

Binary Search Algorithm

```
procedure BnrySrch(Ar: Tarray; num: integer; var pos:
integer);
var
    First, Mid, Last : integer;
    Cur : integer;
    Found : boolean;
begin
    pos := 0;
    Found := false;
    First := 1;
    Last := 10000;

    while (First <= Last) and not Found do
    begin

        Mid := (First + Last) div 2;
        Cur := Ar[Mid];

        if num = Cur then
        begin
            Pos := Mid;
            Found := true;
        end
        else begin
            if num < Cur then
            begin
                Last := Mid - 1;
            end
            else begin
                First := Mid + 1;
            end;
        end;
    end;
end;
```

Binary Search Plans

1. Find middle position in a list.
2. Store the value in the middle position in a temporary variable.
3. Compare the search value with the value in the middle position.
    a. If it is the same, the value is found and the position returned.
    b. If search value is smaller than value in middle position, set the last index one less than the middle position.
    c. Otherwise, set the first index one higher than the middle position.
4. Repeat while value is not found and the first index is less than or equal to last index.

Insertion Sort Algorithm

```
public static void InsertionSort(int[] List, int nrEl)
{
       for (int x = 1; x <= (nrEl - 1); x++)
            insert(List[x], List, x);
}

public static void insert(int NewOne, int[] SortedList, int
nrEl)
{
       int pos = findPos(NewOne, SortedList, nrEl);
       moveToRight(pos, SortedList, nrEl);
       SortedList[pos] = NewOne;
}

public static int findPos(int NewOne, int[] SortedList, int
nrEl)
{
       int x;
       for (x = 0; x <= nrEl-1; x++)
       {
            if (NewOne <= SortedList[x])
                 return x;
       }
       return x;
}

public static void moveToRight(int pos, int[] SortedList,
int nrEl)
{
       for (int x = nrEl; x >= (pos + 1); x--)
       {
            SortedList[x] = SortedList[x - 1];
       }
}
```

Insertion Sort Plans

1. Loop through unsorted list, and build a new sorted list by starting with a list of 1 element and inserting the next value in the unsorted list in its proper place in the growing sorted list.
2. Find the correct position for an element to be inserted into a sorted list.
3. Starting from end of list, move elements one up until an element smaller than element to insert is found.
4. Insert element at the correct position in the sorted list.

Appendix C - Class Tests

Question 1: Recall Bubble Sort

Write a Bubble Sort Algorithm that sorts an unsorted array of integers in **descending** order. Hand in the answer sheet when you are done.

Question 2: Bubble Sort Comprehension

1. What data structure is used to store the data?
2. Which line compares two adjacent values in an array?
3. What does line 12 do?
4. Under which condition would two adjacent values be swapped?

Question 1: Recall Binary Search

Write a Binary Search Algorithm that searches for an **integer** in an **array of integers** sorted in **descending** order. The algorithm will receive the array, the number of elements in the array and the wanted number. It must return the **position** of the wanted number in the array, or **-1** if the number is not found.

Question 2: Binary Search Comprehension

1. What data structure is used to store data of what type?
2. Under which condition(s) must the search continue?
3. What is updated when the *wanted_value* is less than the middle value?
4. What is updated when the *wanted_value* is greater than the middle value?
5. What happens when the first index is greater than the last index? Under what condition will this happen?
6. What data type should the variable in line 6 be?
7. Which line(s) is/are used to determine where the *wanted_value* occur in the list?

Question 1: Insertion Sort Completion

Complete the partially complete algorithm. Fix any errors that you can identify.

```
1.   public static void InsertionSort(double[] Prices, int nrEl)
2.   {
3.       for (int x = 0; x <= (nrEl - 1); x++)
4.   
5.   }
6.   public static void insert(int NewElement, int[] SortedList, int SortednrEl)
7.   {
8.   
9.       moveToRight(the_pos, SortedList, SortednrEl);
10.      SortedList[the_pos] = ;
11.  }
12.  public static int findPos(int NewOne, double[] SortedList, int SortednrEl)
13.  {
14.      double x;
15.      for (x = 0; x <= nrEl-1; x++)
16.      {
17.  
18.  
19.      }
20.      return x;
21.  }
22.  public static void moveToRight(int pos, double[] SortedList, int SortednrEl)
23.  {
24.      for (int x = SortednrEl; x >= (pos + 1); x++)
25.      {
26.  
27.      }
28.  }
```

Question 2: Insertion Sort Comprehension

1. Write down in your own words what the following line(s) of the Insertion Sort Algorithm do.
    a. Line 24 and 26
    b. Line 20
    c. Line 4
    d. Line 15, 17 and 18
2. Which line inserts an element in the appropriate place in an array?
3. Which line(s) creates a space to insert a new element into a suitable position in an array?
4. Which line(s) loops through the unsorted part of the array?

Appendix D – Controlled Experiments

Controlled Experiment A

Task 1: Paper-based (To determine technique(s) used to study an algorithm)

1. Give a printout of the complete version of the Selection Sort to the participant (Algorithm A.doc).
2. *"Read the algorithm in the handout. You may make notes and use any technique while determining the purpose of the algorithm."*
3. Take the algorithm and any additional notes made by the participant.
4. *"Explain* ***in your own words****, what the algorithm does."*
5. Record the response of the participant.
6. Explain the algorithm to the participant.
7. Inform participant that the eye-tracker experiment will start and move over to the observer side of the room.

Time taken: ____

Task 2: Eye-tracker 1 (To determine reading pattern when studying an algorithm)

1. Instruct participant to sit comfortably in front of the PC. Calibrate the participant.
2. *"I'm going to open the same algorithm on the screen of the computer. Please read the algorithm on the screen of the computer. Familiarise yourself with the code and try to determine how the code achieve the goals of the algorithm. Tell me when you are ready to answer five questions about the algorithm."*
3. Open Selection Sort.pps via VNC Viewer.
4. Create new data file in IViewX. Start a recording. Stop the recording as soon as the participant indicates he/she is ready. Save data file as **01Study.idf** in participant folder. Show white screen in Selection Sort.pps before asking each question.

Time taken: ____

Task 3: Eye-tracker 2 (To determine ability to recognise and use beacons)

1. *"I will now ask you a set of five questions. Each time I ask a question, please study the algorithm on screen to find the answer. If you didn't understand or hear the question, ask for a clarification. Let me know when you are ready to find the answer. I will then show the algorithm and you can start scanning the code. As soon as you have the answer, tell me the corresponding line number."*
2. Perform the following steps for each question:
   - Create new data file in IViewX.
   - Read question. Ask if participant understood question.
   - Show algorithm, start recording and stopwatch.
   - When answer is given, stop recording, stop stopwatch and clear screen.
   - Write down answer. Write down time taken. Save data file as select_<question #>.idf in <Participant #>\Exp 1-Selection Sort.
3. Ask the following questions:

Q1. "Which line or lines exchanges (swaps) the current value in the array, with the smallest value in the unsorted part of the array?"

Correct answer: Lines 15, 16, 17

Participant answer:____

Time taken: ____

Q2. "Which line or lines finds the index of the smallest number in the array?

Correct answer: Lines 10, 12
Participant answer:____
Time taken: ____

Q3. Which line or lines loops through the unsorted part of the array to determine the smallest value?"

Correct answer: Line 10
Participant answer:____
Time taken: ____

Q4. "Which line or lines determines if the current value is smaller than the current smallest value?"

Correct answer: Line 12
Participant answer:____
Time taken: ____

Q5. "Which line or lines sets the position of the smallest value in the unsorted part of the array?"

Correct answer: Line 13
Participant answer:____
Time taken: ____

Task 4: Paper-based (To determine ability to complete and debug code)

1. Hand out an incomplete version of Selection Sort (Incomplete Selection Sort.doc).
2. *"Complete the algorithm in the handout. Fix any errors that you can identify."*
3. Start stopwatch.
4. When participant is done, stop stopwatch and collect the answer sheet.
5. Take a 5 minute break before the next experiment.

Time taken: ____

Controlled Experiment B

Task 1: Eye-tracker 1 (To determine retention of learning)

1. *"I will now ask you five questions on an algorithm you've studied before. Each time I ask a question, study the algorithm on screen to find the answer. If you didn't understand or hear the question, ask for a clarification. Let me know when you are ready to find the answer. I will show the algorithm and you can start scanning the code. As soon as you have the answer, tell me the corresponding line number."*
2. Perform the following steps for each question:
    - Create new data file in IViewX.
    - Read question. Ask if participant understood question.
    - Show algorithm, start recording and stopwatch.
    - When answer is given, stop recording, stop stopwatch and clear screen.
    - Write down answer. Write down time taken. Save data file as insert_<question #>.idf in <Participant #>\Exp B-Insertion Sort.
3. Ask the following questions:

Q1. "Which line/lines calls a function to make space for a new element in an array?"

Correct answer: Line 10
Participant answer:____
Time taken: ____

Q2. "Which line or lines calls a function to insert a new element into a sorted array, keeping it sorted?"

Correct answer: Line 4

Participant answer:____

Time taken: ____

Q3. "Which line or lines will return a position in the array if the new value is higher than all the values in the sorted list?"

Correct answer: Line 22

Participant answer:____

Time taken: ____

Q4. "Which line or lines will loop through the unsorted part of the array?"

Correct answer: Line 3

Participant answer:____

Time taken: ____

Q5. "Which line or lines will loop through the sorted part of the array and determine a suitable position to insert a new element?

Correct answer: Lines 17, 19

Participant answer:____

Time taken: ____

Task 2: Paper-based (To determine ability to complete and debug code)

1. Hand out an incomplete version of Insertion Sort (Incomplete Insertion Sort.doc).
2. *"Complete the algorithm in the handout. Fix any errors that you can identify."*
3. Start stopwatch.
4. When participant is done, stop stopwatch and collect the answer sheet.
5. Take a 5 minute break before the next experiment.

Time taken: ____

Controlled Experiment C

"I am going to show you 6 code fragments, one at a time. Study them, then describe in your own words what each code fragment does and identify the one algorithm from the list most likely to contain each fragment.

Code Fragment 1

```
for (x = 0; x <= nrEl-1; x++)
{
        if (NewOne <= SortedList[x])
                return x;
}
return x;
```

Function: Returning a position of a certain value in an array. If the value is less than the first value in an array, the loop will terminate. If the value is greater than all values in the array, it will return the position after the last value in the array.

The likely algorithm:	Bubble Sort	Binary Search	**Insertion Sort**	Don't know

Time taken: ____

Code Fragment 2

```
mid_index = (first_index + last_index) / 2;
```

Function: We are calculating the middle position in an array.

The likely algorithm:

Bubble Sort	**Binary Search**	Insertion Sort	Don't know

Time taken: ______________________________

Code Fragment 3

```
if (TheList[m] > TheList[m+1])
{
        sorted = false;
        int temp = TheList[m];
        TheList[m] = TheList[m+1];
        TheList[m+1] = temp;
}
```

Function: Two adjacent values will be swapped if they are in descending order in relation to each other. The list should be sorted in ascending order.

The likely algorithm:

Bubble Sort	Binary Search	Insertion Sort	Don't know

Time taken: ______________________________

Code Fragment 4

```
//insert() is a function that insert a new value into a sorted array, while keeping the
array sorted
for (int x = 1; x <= (nrEl - 1); x++)
        insert(List[x], List, x);
```

Function: Scrolling through the unsorted part of an array and inserting the first element of the unsorted list in its correct position.

The likely algorithm:

Bubble Sort	Binary Search	**Insertion Sort**	Don't know

Time taken: ______________________________

Code Fragment 5

```
cur_value = List[mid_index);
if (wanted < cur_value)
        last_index = (mid_index - 1);
```

Function: Updating the last position of an array to work in the first half when the value being searched for is less than the value in the middle position of the array.

The likely algorithm:

Bubble Sort	**Binary Search**	Insertion Sort	Don't know

Time taken: ______________________________

Code Fragment 6

```
for (int x = nrEl; x >= (pos + 1); x--){
        SortedList[x] = SortedList[x - 1];
```

Function: Moving elements in an array one position to the right.

The likely algorithm:

Bubble Sort	Binary Search	**Insertion Sort**	Don't know

Time taken: ______________________________

Appendix E – BeReT Questionnaire

Surname, Init ______ **Student Nr** ______

1. BeReT is easy to use

Strongly disagree	Disagree	Undecided	Agree	Strongly agree

2. Could you complete **Tutorials** in BeReT without assistance

Can't remember	No	Hardly	Almost	Yes

3. If you did not answer "Yes" above, what created the difficulty with **Tutorials**?

____

____

4. Could you complete **Exercises** in BeReT without assistance

Can't remember	No	Hardly	Almost	Yes

5. If you did not answer "Yes" above, what created the difficulty with **Exercises**?

____

____

6. What level of assistance did you need throughout your usage of BeReT?

Very low	Low	Average	High	Very high

7. Did you understand the following algorithms **before** using BeReT?

Bubble Sort	Not at all	Very little	Some of it	Well	Very well	Can't remember
Binary Search	Not at all	Very little	Some of it	Well	Very well	Can't remember
Insertion Sort	Not at all	Very little	Some of it	Well	Very well	Can't remember

8. Did you understand the following algorithms **after** using BeReT?

Bubble Sort	Not at all	Very little	Some of it	Well	Very well	Can't remember
Binary Search	Not at all	Very little	Some of it	Well	Very well	Can't remember
Insertion Sort	Not at all	Very little	Some of it	Well	Very well	Can't remember

9. Did BeReT help programming the algorithms easier from scratch?

Made it more difficult	Made no difference	Made some difference	Made it easier	Made it a lot easier

10. To what level do you think BeReT helped you to better understand what certain lines do in the code?

Very low	Low	Average	High	Very high

11. To what level do you think BeReT helped you to recognise an algorithm quicker without knowing the name or function of the algorithm?

Very low	Low	Average	High	Very high

12. To what level do you think BeReT helped you to trace (i.e. show the contents of the variables at certain points) an algorithm more effectively?

Very low	Low	Average	High	Very high

13. Assuming BeReT was problem free; would you make use of it to gain a better understanding of algorithms?

Never	Sometimes	Always

14. Would you make use of BeReT if it immediately marked your answers and gave you feedback?

Never	Sometimes	Always

15. What other functionality would you like to see included in BeReT?

____

____

16. Did you ever run into any unexpected problems in BeReT?

Yes	No

17. If so, briefly describe the task (if possible) and BeReT's reaction.

____

____

18. What in your opinion is an advantage of using BeReT?

____

____

19. What in your opinion is a disadvantage of using BeReT?

____

____

20. Did BeReT encourage you to critically analyse and discuss the code in the given algorithms?

Yes	No

21. Did BeReT encourage you to ask questions about the algorithms?

Yes	No

22. Would you have spent time studying the algorithms before the practical assignments without BeReT?

Yes	No

Appendix F – Human Ethics Application Form

NELSON MANDELA METROPOLITAN UNIVERSITY

APPLICATION FOR APPROVAL FROM NMMU RESEARCH ETHICS COMMITTEE (HUMAN)

(ETHICAL STANDARDS: RESEARCH PROTOCOL)

1. Any project, in which humans are the subjects of research, requires completion of this form and submission for approval to the ETHICS COMMITTEE.
2. The faculty through the Faculty Research Committee **and** Head of Department should approve research proposals before submission to the Ethics Committee.
3. Each faculty should have the primary responsibility for ensuring that human subjects used in social research in their faculties are protected adequately by the application of the appropriate code applicable to the relevant profession.
4. The application form, after **being** completed in **typescript**, to be handed in at the Department of Research Management.

FOR OFFICE USE ONLY		REF NO:	
DEPARTMENT		**DATE RECEIVED**	
DATE SUBMITTED TO THE RESEARCH ETHICS COMMITTEE (HUMAN)		**DATE APPROVED BY THE RESEARCH ETHICS COMMITTEE (HUMAN)**	
AUTHORIZED Chairperson of the Research Ethics Committee (Human)			

1. GENERAL PARTICULARS

a) Name of principal investigator/researcher:

Mr Ronald George Leppan

b) Gender of principal investigator/researcher:

Male

c) Contact number of principal investigator/researcher:

504 2763

d) Type of research:

STAFF							STUDENT					
Multinational [MN]		National [N]		Local Research [LR]			Multinational [MN]		National [N]		Local Research [LR]	
Undergraduate [U]		Honours [H]		Master's [M]		X	Undergraduate [U]		Honours [H[		Master's [M]	
Doctoral [D]		Other [O]					Doctoral [D]		Other [O]			

e) **If medical research**:

Therapeutic Research [TR]		Non-therapeutic Research [NTR]	

f) **Funding**

External Grant [E]	X	NMMU Research Grant [RG]	

Privately Funded [P]		Not specifically funded [N]	

If external funding, state source of funds:

NRF Development Grant

Are there any restrictions or conditions attached to publication and/or presentation of the study results?

None

Does the contract specifically recognise the independence of the researchers involved?

Yes

(Note that any such restrictions or conditions contained in funding contracts must be made available to the Committee)

g) **Summary of research**

(i) What is the purpose of the research?

This study aims at finding a strategy (or strategies) for code comprehension that is based on techniques used by experts to trace code. In an attempt to bridge the gap between novice and expert programmers, an emphasis is placed on technological tools to scaffold learner competence in using these strategies to comprehend code.

(iii) Briefly state the methodology and the procedure in which subjects will be asked to participate (attach protocol)

The effect of the technological support tools are to be determined by means of controlled experiments to test student comprehension of known and unseen algorithms. Formal assessment in the form of standardized tests and exams are also conducted and the results of comprehension type questions extracted from these assessments. The effect of the support tools on program comprehension will also be determined by means of an eye tracker experiment to gain a deeper understanding of the cognitive processes of introductory programming students during code comprehension exercises. The data collected by the controlled experiment, the standardized assessment and the practical assignments will be interpreted and conclusions will be drawn from the analysis. Any data collected from the participants during the controlled experiment, the standardized assessment and the practical assignments will be seen as strictly confidential and no names will be associated with the data.

Students in introductory programming (WRA102) will be randomly divided into two groups, a control group and a treatment group. Both groups will attend the same lecture in the same venue at the same time. During practical assignment periods, they will be split into two groups in different computer labs. The control group will perform their practical assignments as in the past. The treatment group will be exposed to the experimental tool during three practical assignments. Exposure to the technological tool will thus be minimal (3 contact sessions out of a total of 28) with as little disruption of their normal practical activities as possible. The workload of the two groups will be balanced, with no group expected to do more work than the other.

h) Name of the investigator/researcher (whether student or staff member) mostly involved in this project:

Ronald George Leppan

i) Name(s) of co-investigator/assistant researchers:	Affiliation

j) Name(s) of supervisor/co-supervisor or promoter/co-promoter: **Dr Charmain Cilliers** **Ms Marinda Taljaard**	Affiliation **Senior Lecturer – CS & IS Department (Supervisor)** **Lecturer – CS & IS Department (Co-supervisor)**

2. INFORMATION TO PARTICIPANT

a) What information will be offered to the participant before he/she consents to participate? (Append both the written and any oral information given)

Informed Consent Form

b) Who will provide this information? (Give name(s) and state whether such a person is the principle investigator/ researcher, student, research assistant etc.)

Ronald George Leppan – Principle investigator

c) Will the information provided be complete and accurate?

☒ YES □ NO

If NO, describe the nature and extent of the deception involved and explain the rationale for the necessity of this deception. (If necessary, attach separate schedule)

3. TARGET PARTICIPANT GROUP

		Answer YES or NO	If necessary, explain in space provided or attach appendix
a)	Are particular characteristics of any kind required in the target group? (e.g. age, cultural derivation, background, physical characteristics, disease status etc.)	**NO**	Specify the characteristics:
b)	Are participants drawn from NMMU students?	**YES**	
c)	Are participants drawn from specific groups of NMMU students?	**YES**	Identify the group: **First–year Introductory Programming students**
d)	Are participants drawn from a school population?	**NO**	Identify: (State whether pre-primary, primary, secondary, etc.)
e)	Are participants drawn from an institutional population? (e.g. hospital, prison, mental institution)	**NO**	Identify:
f)	Will any records be consulted for information?	**NO**	Specify source of records:
g)	Will each individual participant know his/her records are being consulted?	**N/A**	State how these records will be obtained:
h)	Are all participants over 21 years of age?	**NO**	If NO, state justification for inclusion of minors in study: **Participants register for first year programming typically a year after finishing Grade 12.**

State the minimum and maximum number of participants involved (Minimum number should reflect the number of participants necessary to make the study statistically viable)	**Minimum** | **60** | **Maximum** | **120**

4.	**RISKS AND BENEFITS OF PROJECT**		
		Answer YES or NO	**If necessary, explain in space provided or attach appendix**
a)	Is there any risk of harm, embarrassment or offence, however slight or temporary, to third parties or to the community at large?	**NO**	If YES, specify:
b)	Are all risks reversible?	**N/A**	If NO, specify:
c)	Are remedial measures available?	**N/A**	
d)	Are alternative procedures available?	**N/A**	
e)	Has the person administering the project previous experience with the particular risk factors involved?	**NO**	**Supervisor and co-supervisor have experience with the risk factors involved in the study**
f)	Are any benefits expected to accrue to the participant personally? (e.g. improved health, mental state, financial etc.)	**N/A**	If YES, specify: **Identifying the benefits, if any, forms part of the intended study**
g)	Will you be using equipment of any sort?	**YES**	If YES, specify: **Personal Computers** **Eye tracking equipment**
h)	Will any article of property, personal or cultural be collected in the course of the project?	**NO**	If YES, specify:

5.	**CONSENT OF PARTICIPANTS**		
		Answer YES or NO	**If necessary, explain in space provided or attach appendix**
a)	Is consent to be given in writing?	**YES**	If YES, attach consent form. If NO, state reasons why written consent is not appropriate in this study:
b)	Are any participant(s) subject to legal restrictions preventing them from giving effective informed consent?	**NO**	If YES, justify:
c)	Do any participant(s) operate in an institutional environment, which may cast doubt on the voluntary aspect of consent?	**NO**	If YES, state what special precautions will be taken to obtain a legally effective informed consent:
d)	Will participants receive remuneration for their participation?	**NO**	If YES, state on what basis the remuneration is calculated:
e)	Do you require consent of an institutional	**NO**	If YES, specify:

	authority for this project?		

6.	**PRIVACY, ANONYMITY AND CONFIDENTIALITY OF DATA**		
		Answer YES or NO	**If necessary, explain in space provided or attach appendix**
a)	Will the participant be identified by name in your research?	**NO**	
b)	Are provisions made to protect subject's rights to privacy and anonymity and to preserve confidentiality with respect to data?	**YES**	Specify: **No names will be attached to any data, and for the most part data will be gathered and analysed per group rather than per individual.**
c)	Will mechanical methods of observation be used? (e.g. one-way mirrors, recordings, videos etc.)	**YES**	**A subgroup (16 participants) of the population will voluntarily participate in a special assessment. In this assessment, unobtrusive remote eye tracking equipment will be used in a usability lab with two rooms separated by a one-way mirror. Video and audio recording will also take place.**
d)	Will participant's consent to such mechanical methods of observation be obtained?	**YES**	
e)	Will data collected be stored in any way?	**YES**	If YES: (i) By whom? **Dept. of CS & IS, NMMU** (ii) How many copies? **Two, one for analysis and one for backup.** (iii) For how long? **For the duration of the study. (Expected completion time: December 2006)** (iv) For what reasons? **Statistical Analysis** (v) How will participant's anonymity be protected? **No names will be attached to the data**
f)	Will stored data be made available for re-use?	**NO**	If YES, how will participant's consent be obtained for such re-usage?
g)	Will any part of the project be conducted on private property (including shopping centres)?	**NO**	If YES, specify and state how consent of property owner is to be obtained:
h)	Are there any contractual secrecy or confidentiality constraints on this data?	**NO**	If YES, specify:

7.	**FEEDBACK**		
		Answer YES or NO	**If necessary, explain in space provided or attach appendix**
a)	Will feedback be given to participants?	**NO**	If YES, describe whether this is to be given: (i) to each individual immediately after participation □ (ii) to each participant after the

			entire project is completed □ (iii) to all participants in a group setting □ (iv) other □ Specify whether feedback will be written, oral or by other means (attach your information)
b)	If you are working in a school or other institutional setting, will you be providing teachers, school authorities or equivalent a copy of your results?	**N/A**	If YES, specify:

8. STATEMENT ON CONFLICT OF INTEREST

The researcher is expected to declare to the Committee the presence of any potential or existing conflict of interest that may potentially pose a threat to the scientific integrity and ethical conduct of any research. The Committee will decide whether such conflicts are sufficient as to warrant consideration of their impact on the ethical conduct of the study.

Disclosure of conflict of interest does not imply that a study will be deemed unethical, as the mere existence of a conflict of interest does not mean that a study cannot be conducted ethically. However, failure to declare to the Committee a conflict of interest known to the researcher at the outset of the study will be deemed to be unethical conduct.

Researchers are therefore expected to sign either of the two declarations below.

a) As the principal researcher in this study **MR RONALD GEORGE LEPPAN**…………………………………(name) I hereby declare that I am not aware of any potential conflict of interest that may influence my ethical conduct of this study.

Signature:………………………….……….………… Date:……………………………..……………..

b) As the principal researcher in this study……………………………………………………………………(name) I hereby declare that I am aware of the following potential conflicts of interest which should be considered by the Committee:

Signature:…………………………………………………… Date:……………………………………………………

9. ETHICAL AND LEGAL ASPECTS

The Declaration of Helsinki version 2000 to be included in the references.

10. DECLARATION

If any changes are made to the above arrangements or procedures, I will bring these to the attention of the Chairperson of the Research Ethics Committee (Human).

SIGNATURE OF PRINCIPAL INVESTIGATOR/RESEARCHER	**DATE**
SIGNATURE OF SUPERVISOR	**DATE**
SIGNATURE OF HEAD OF DEPARTMENT	**DATE**
Noted application **SIGNATURE OF CHAIRPERSON: FACULTY RESEARCH COMMITTEE**	**DATE**

Appendix G: Controlled Experiment Protocol

Before the test:

Participant room check list:	
	Check that the room is neat and presentable.
	Switch on the test pc.
	Switch on the eye tracker.
	Check eye tracker, video and audio cabling.
	Test the Win cal software / check calibration points.
	Test the microphone.

Observer room check list:	
	Check that the room is neat and presentable.
	Switch on the IView-X PC.
	Switch on the equipment: • Audio amplifier; Audio mixer; Video mixer; • TV monitor; Camera boards.
	Check video feeds for the IView-X PC and the TV monitor.
	Adjust the cameras as needed.
	Test the microphone.
	Setup the IView-X PC for the test.

When the participant arrives:

Check list:	
	Welcome the test participant.
	Brief the participant. • Introduce the observers. • Show and explain the equipment to the participant. • Explain the task process to the participant. • Ask to give verbal comments after task completions (recordings).
	The participant conducts a training task.
	Subject calibration.
	Conduct the test. • Record video and audio. • Record eye tracking data. • Log participant activities. • Save the data files after the test.
	Debrief the participant.
	Thank the participant.

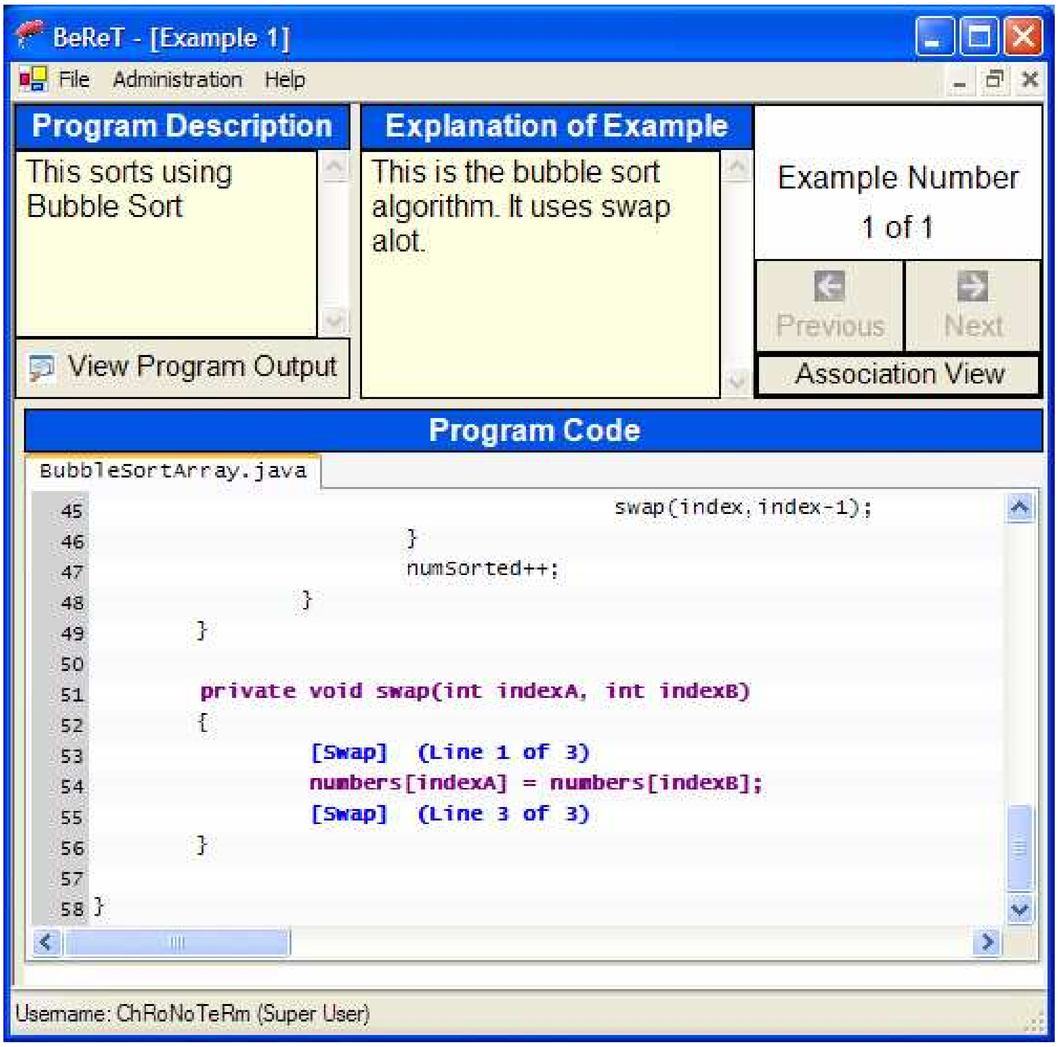
After the test:

Check list:	
	Collect the test paperwork.
	Put all the forms in a folder for the specific test participant.
	Backup the saved data.
	Have a brief session with the test team to collect their thoughts of the test.
	Clear the participant room from any documentation left behind.
	Setup the logging software for the next participant.

End of the day:

Check list:	
	Have a brief review with the test team to summarise the day’s events.
	Write the saved data to CD as an additional backup.
	Turn off the equipment.
	Turn off the PCs.
	Check that the room is neat and presentable.
	Switch the lights off.
	Lock the usability laboratory.

Appendix H: Creation of Tutorials and Exercises in BeReT (Lecturer Guide)



This sorts using Bubble Sort

This is the bubble sort algorithm. It uses swap alot.

Example Number
1 of 1

```
                                        swap(index, index-1);
                              }
                              numSorted++;
                    }
          }

          private void swap(int indexA, int indexB)
          {
                    [Swap]   (Line 1 of 3)
                    numbers[indexA] = numbers[indexB];
                    [Swap]   (Line 3 of 3)
          }

}
```

Usemame: ChRoNoTeRm (Super User)

Tutorials in BeReT consist of four main areas: program description, explanation of example, program code and navigation information. Lecturers have control over what appears in the program description, explanation of example and program code pages. In the current investigation, the “program description” pane is used to describe the algorithm presented in the “program code” pane. It usually contains a bulleted list of the plans used in the algorithm. The “explanation of example” pane is used to provide an example of the parameters passed to the algorithm and the final result upon completion of the algorithm. Using an example of what happens to the input provided to an algorithm and viewing the output is a subset of the tracing technique used in contact sessions. The BeReT “explanation of example” pane is set up in this way in order to provide some level of familiarity for the student, since tracing is the predominant technique used to explain algorithms in lectures.

The “program code” pane contains the algorithm that the lecturer uploaded. The algorithm can be in any language the lecturer prefers. The “navigation information” pane contains buttons to navigate between examples; it shows the number of the current example and allows users to switch between the two tutorial modes. In “context view”, learners see the beacon names in the context of the code. Pointing to these lines brings up a tooltip with the description of the semantic plan behind that line and clicking on it, displays the inner workings of the code that accomplishes the plan. The association view provides the same information as the context view. Lecturers have no control over what happens in the “navigation information” pane. It is suggested that students first use the association view to learn about the plans implemented in the code. The context view can be used by students to test their knowledge of the actual code and description of the plan linked to the annotation. To test their understanding of various plans, they first view a pseudo-code type annotation of the plan in context and then recall from memory what the statement looks like that implements the plan by clicking on the annotation. Students can also test their knowledge of a high-level semantic description by looking at either the annotation or the implementation of the plan and then pointing to the line to view the tooltip with the description.

The type of tutorial created depends on what the lecturer wants to use the tutorial for. If the students must learn about all the plans that are implemented in the algorithm, the tutorial should include descriptions and annotations for each plan. If the lecturer wants to focus on only beacon-like statements, descriptions and annotations for only these statements should be included in the tutorial.

It is suggested that students start with a tutorial before attempting an exercise. Students may be required to do any of the following three types of exercises in BeReT: Complete, Match and Define. These three types of tasks are summarised in Table 1.

Task	**Given**	**What must be done**
Complete	Beacon name and description	Fill in missing source code that name and description refers to
Match	Beacon name and description	Select relevant source code lines that matches name and description
Define	Code only	Group statements together and assign a name and description to indicate plan of these statements

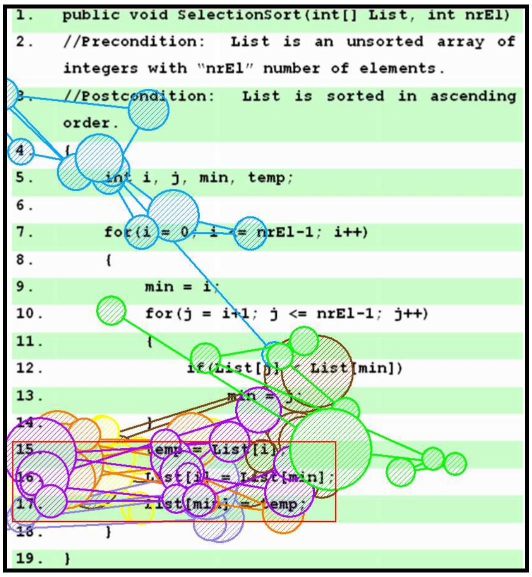
Table 1 Types of exercises in BeReT

Appendix I – Eye-tracking images from Controlled Experiments A and B

Experiment A

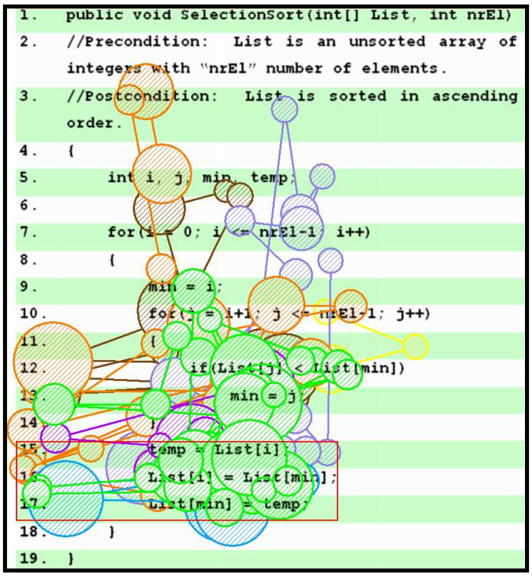
Q1. "Which line or lines exchanges (swaps) the current value in the array, with the smallest value in the unsorted part of the array?"
Correct answer: Lines 15, 16, 17

Question 1 Treatment Scanpaths



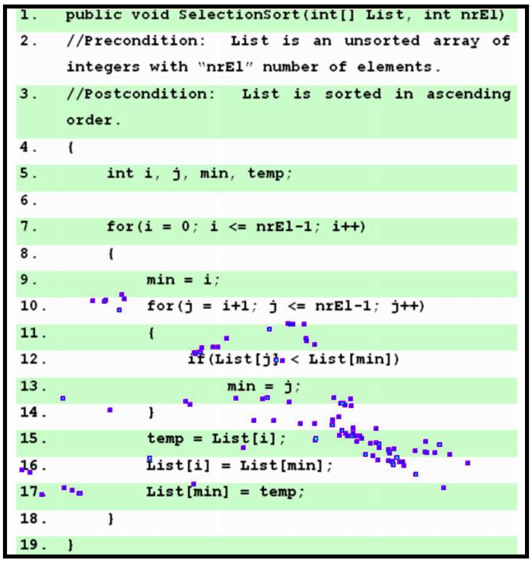
```
1.  public void SelectionSort(int[] List, int nrEl)
2.  //Precondition:  List is an unsorted array of
    integers with "nrEl" number of elements.
3.  //Postcondition:  List is sorted in ascending
    order.
4.  {
5.      int i, j, min, temp;
6.
7.      for(i = 0; i <= nrEl-1; i++)
8.      {
9.          min = i;
10.         for(j = i+1; j <= nrEl-1; j++)
11.         {
12.             if(List[j] < List[min])
13.                 min = j;
14.         }
15.         temp = List[i];
16.         List[i] = List[min];
17.         List[min] = temp;
18.     }
19. }
```

Question 1 Control Scanpaths



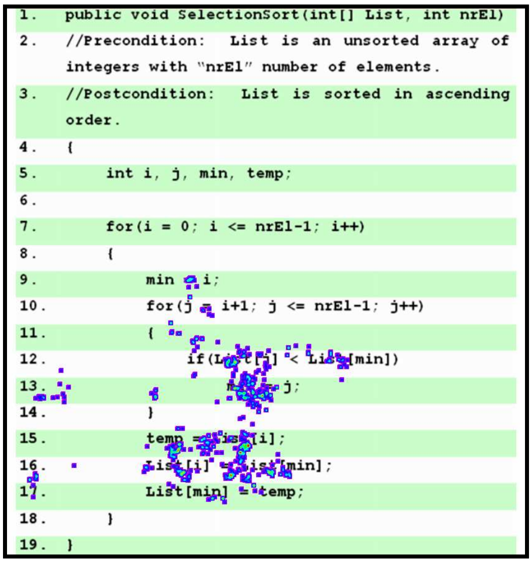
```
1.  public void SelectionSort(int[] List, int nrEl)
2.  //Precondition:  List is an unsorted array of
    integers with "nrEl" number of elements.
3.  //Postcondition:  List is sorted in ascending
    order.
4.  {
5.      int i, j, min, temp;
6.
7.      for(i = 0; i <= nrEl-1; i++)
8.      {
9.          min = i;
10.         for(j = i+1; j <= nrEl-1; j++)
11.         {
12.             if(List[j] < List[min])
13.                 min = j;
14.         }
15.         temp = List[i];
16.         List[i] = List[min];
17.         List[min] = temp;
18.     }
19. }
```

Question 1 Treatment Heat Maps



```
1.  public void SelectionSort(int[] List, int nrEl)
2.  //Precondition:  List is an unsorted array of
    integers with "nrEl" number of elements.
3.  //Postcondition:  List is sorted in ascending
    order.
4.  {
5.      int i, j, min, temp;
6.
7.      for(i = 0; i <= nrEl-1; i++)
8.      {
9.          min = i;
10.         for(j = i+1; j <= nrEl-1; j++)
11.         {
12.             if(List[j] < List[min])
13.                 min = j;
14.         }
15.         temp = List[i];
16.         List[i] = List[min];
17.         List[min] = temp;
18.     }
19. }
```

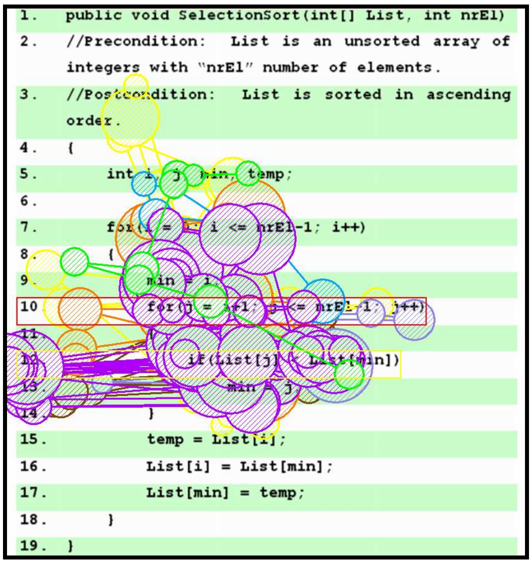
Question 1 Control Heat Maps



```
1.  public void SelectionSort(int[] List, int nrEl)
2.  //Precondition:  List is an unsorted array of
    integers with "nrEl" number of elements.
3.  //Postcondition:  List is sorted in ascending
    order.
4.  {
5.      int i, j, min, temp;
6.
7.      for(i = 0; i <= nrEl-1; i++)
8.      {
9.          min = i;
10.         for(j = i+1; j <= nrEl-1; j++)
11.         {
12.             if(List[j] < List[min])
13.                 min = j;
14.         }
15.         temp = List[i];
16.         List[i] = List[min];
17.         List[min] = temp;
18.     }
19. }
```

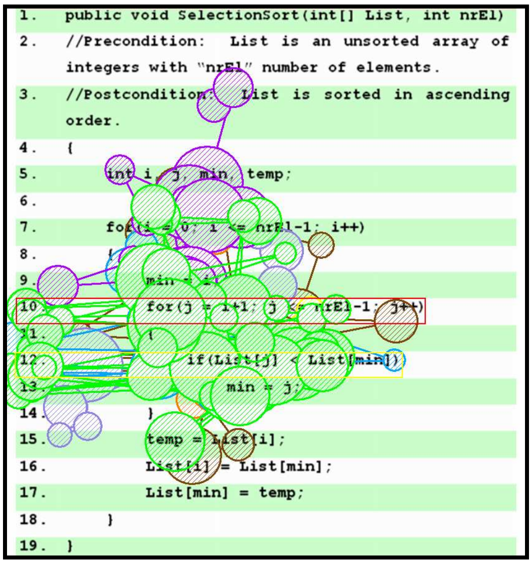
Q2. "Which line or lines finds the index of the smallest number in the array?
Correct answer: Lines 10, 12

Question 2 Treatment Scanpaths



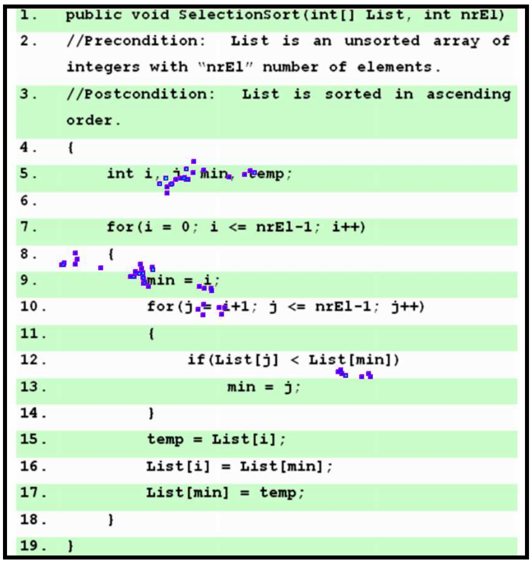
```
1.  public void SelectionSort(int[] List, int nrEl)
2.  //Precondition:  List is an unsorted array of
    integers with "nrEl" number of elements.
3.  //Postcondition:  List is sorted in ascending
    order.
4.  {
5.      int i, j, min, temp;
6.
7.      for(i = 0; i <= nrEl-1; i++)
8.      {
9.          min = i;
10          for(j = i+1; j <= nrEl-1; j++)
11.         {
12.             if(List[j] < List[min])
13.                 min = j;
14.         }
15.         temp = List[i];
16.         List[i] = List[min];
17.         List[min] = temp;
18.     }
19. }
```

Question 2 Control Scanpaths



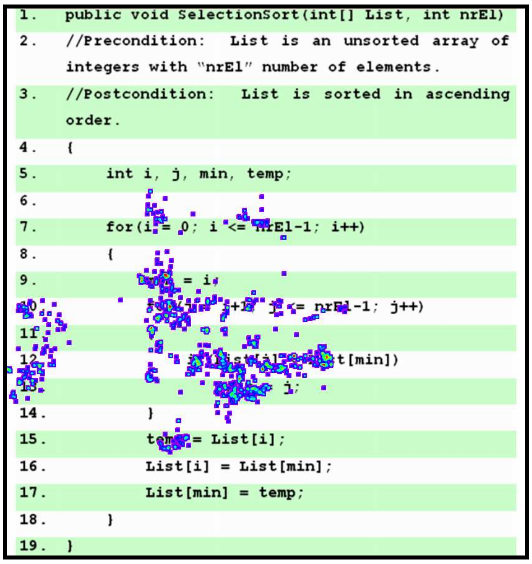
```
1.  public void SelectionSort(int[] List, int nrEl)
2.  //Precondition:  List is an unsorted array of
    integers with "nrEl" number of elements.
3.  //Postcondition:  List is sorted in ascending
    order.
4.  {
5.      int i, j, min, temp;
6.
7.      for(i = 0; i <= nrEl-1; i++)
8.      {
9.          min = i;
10          for(j = i+1; j <= nrEl-1; j++)
11.         {
12.             if(List[j] < List[min])
13.                 min = j;
14.         }
15.         temp = List[i];
16.         List[i] = List[min];
17.         List[min] = temp;
18.     }
19. }
```

Question 2 Treatment Heat Maps



```
1.  public void SelectionSort(int[] List, int nrEl)
2.  //Precondition:  List is an unsorted array of
    integers with "nrEl" number of elements.
3.  //Postcondition:  List is sorted in ascending
    order.
4.  {
5.      int i, j, min, temp;
6.
7.      for(i = 0; i <= nrEl-1; i++)
8.      {
9.          min = i;
10.         for(j = i+1; j <= nrEl-1; j++)
11.         {
12.             if(List[j] < List[min])
13.                 min = j;
14.         }
15.         temp = List[i];
16.         List[i] = List[min];
17.         List[min] = temp;
18.     }
19. }
```

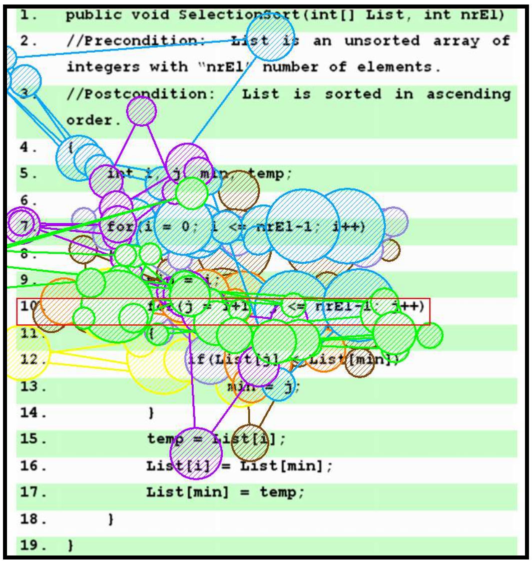
Question 2 Control Heat Maps



```
1.  public void SelectionSort(int[] List, int nrEl)
2.  //Precondition:  List is an unsorted array of
    integers with "nrEl" number of elements.
3.  //Postcondition:  List is sorted in ascending
    order.
4.  {
5.      int i, j, min, temp;
6.
7.      for(i = 0; i <= nrEl-1; i++)
8.      {
9.          min = i;
10.         for(j = i+1; j <= nrEl-1; j++)
11.         {
12.             if(List[j] < List[min])
13.                 min = j;
14.         }
15.         temp = List[i];
16.         List[i] = List[min];
17.         List[min] = temp;
18.     }
19. }
```

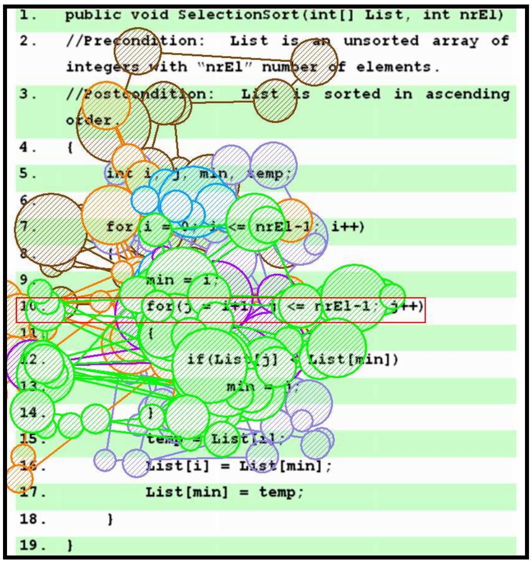
Q3. Which line or lines loops through the unsorted part of the array to determine the smallest value?"
Correct answer: Line 10

Question 3 Treatment Scanpaths



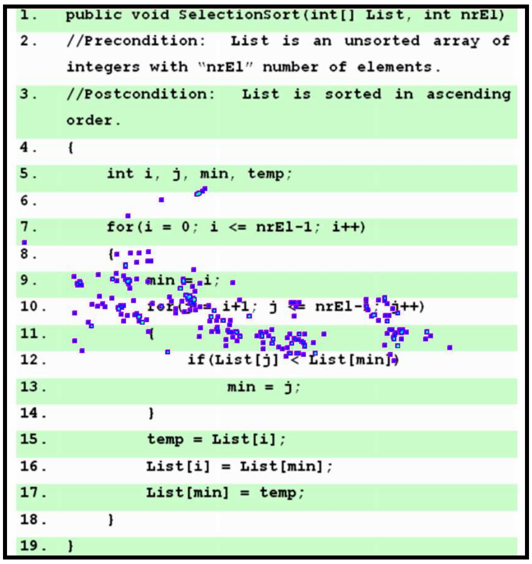
```
1.  public void SelectionSort(int[] List, int nrEl)
2.  //Precondition: List is an unsorted array of
    integers with "nrEl" number of elements.
3.  //Postcondition: List is sorted in ascending
    order.
4.  {
5.      int i, j, min, temp;
6.
7.      for(i = 0; i <= nrEl-1; i++)
8.      {
9.          min = i;
10.         for(j = i+1; j <= nrEl-1; j++)
11.         {
12.             if(List[j] < List[min])
13.                 min = j;
14.         }
15.         temp = List[i];
16.         List[i] = List[min];
17.         List[min] = temp;
18.     }
19. }
```

Question 3 Control Scanpaths



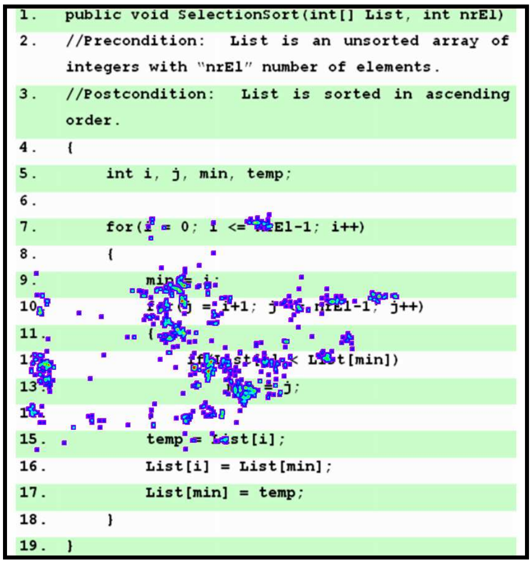
```
1.  public void SelectionSort(int[] List, int nrEl)
2.  //Precondition: List is an unsorted array of
    integers with "nrEl" number of elements.
3.  //Postcondition: List is sorted in ascending
    order.
4.  {
5.      int i, j, min, temp;
6.
7.      for(i = 0; i <= nrEl-1; i++)
8.      {
9.          min = i;
10.         for(j = i+1; j <= nrEl-1; j++)
11.         {
12.             if(List[j] < List[min])
13.                 min = j;
14.         }
15.         temp = List[i];
16.         List[i] = List[min];
17.         List[min] = temp;
18.     }
19. }
```

Question 3 Treatment Heat Maps



```
1.  public void SelectionSort(int[] List, int nrEl)
2.  //Precondition: List is an unsorted array of
    integers with "nrEl" number of elements.
3.  //Postcondition: List is sorted in ascending
    order.
4.  {
5.      int i, j, min, temp;
6.
7.      for(i = 0; i <= nrEl-1; i++)
8.      {
9.          min = i;
10.         for(j = i+1; j <= nrEl-1; j++)
11.         {
12.             if(List[j] < List[min])
13.                 min = j;
14.         }
15.         temp = List[i];
16.         List[i] = List[min];
17.         List[min] = temp;
18.     }
19. }
```

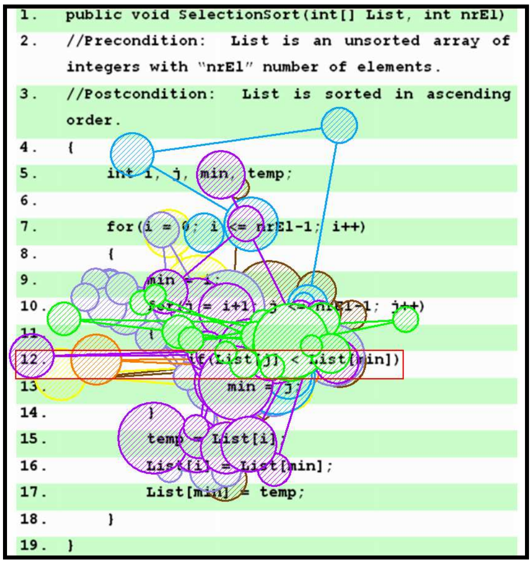
Question 3 Control Heat Maps



```
1.  public void SelectionSort(int[] List, int nrEl)
2.  //Precondition: List is an unsorted array of
    integers with "nrEl" number of elements.
3.  //Postcondition: List is sorted in ascending
    order.
4.  {
5.      int i, j, min, temp;
6.
7.      for(i = 0; i <= nrEl-1; i++)
8.      {
9.          min = i;
10.         for(j = i+1; j <= nrEl-1; j++)
11.         {
12.             if(List[j] < List[min])
13.                 min = j;
14.         }
15.         temp = List[i];
16.         List[i] = List[min];
17.         List[min] = temp;
18.     }
19. }
```

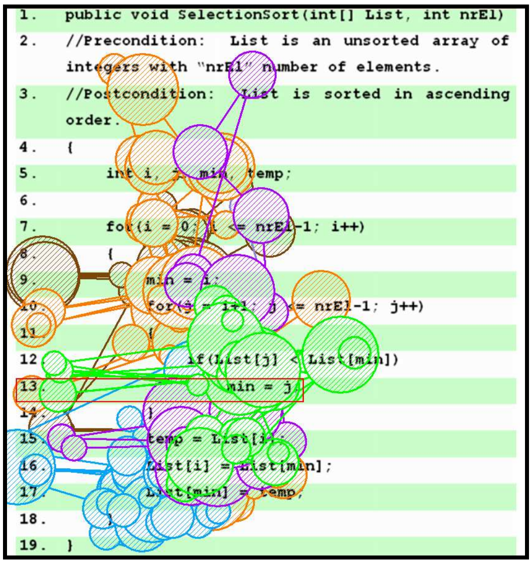
Q4. "Which line or lines determines if the current value is smaller than the current smallest value?"
Correct answer: Line 12

Question 4 Treatment Scanpaths



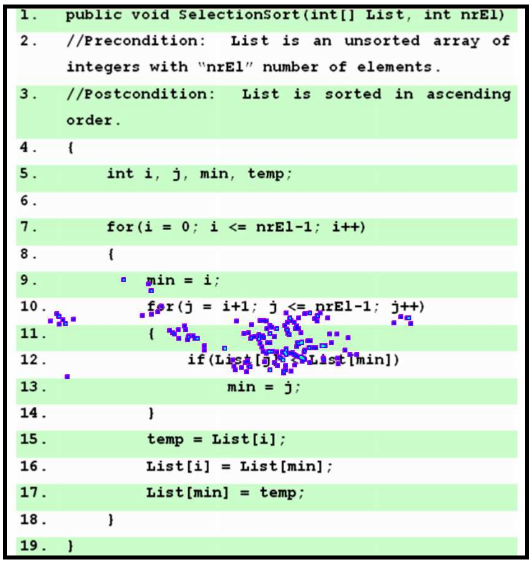
```
1.  public void SelectionSort(int[] List, int nrEl)
2.  //Precondition:  List is an unsorted array of
    integers with "nrEl" number of elements.
3.  //Postcondition:  List is sorted in ascending
    order.
4.  {
5.      int i, j, min, temp;
6.
7.      for(i = 0; i <= nrEl-1; i++)
8.      {
9.          min = i;
10.         for(j = i+1; j <= nrEl-1; j++)
11.         {
12.             if(List[j] < List[min])
13.                 min = j;
14.         }
15.         temp = List[i];
16.         List[i] = List[min];
17.         List[min] = temp;
18.     }
19. }
```

Question 4 Control Scanpaths



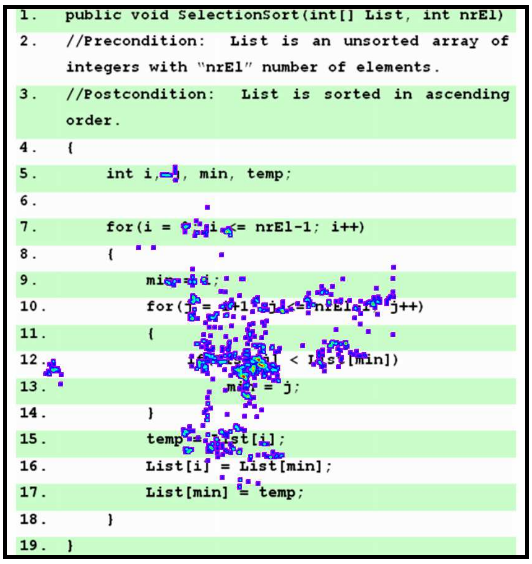
```
1.  public void SelectionSort(int[] List, int nrEl)
2.  //Precondition:  List is an unsorted array of
    integers with "nrEl" number of elements.
3.  //Postcondition:  List is sorted in ascending
    order.
4.  {
5.      int i, j, min, temp;
6.
7.      for(i = 0; i <= nrEl-1; i++)
8.      {
9.          min = i;
10.         for(j = i+1; j <= nrEl-1; j++)
11.         {
12.             if(List[j] < List[min])
13.                 min = j;
14.         }
15.         temp = List[i];
16.         List[i] = List[min];
17.         List[min] = temp;
18.     }
19. }
```

Question 4 Treatment Heat Maps



```
1.  public void SelectionSort(int[] List, int nrEl)
2.  //Precondition:  List is an unsorted array of
    integers with "nrEl" number of elements.
3.  //Postcondition:  List is sorted in ascending
    order.
4.  {
5.      int i, j, min, temp;
6.
7.      for(i = 0; i <= nrEl-1; i++)
8.      {
9.          min = i;
10.         for(j = i+1; j <= nrEl-1; j++)
11.         {
12.             if(List[j] < List[min])
13.                 min = j;
14.         }
15.         temp = List[i];
16.         List[i] = List[min];
17.         List[min] = temp;
18.     }
19. }
```

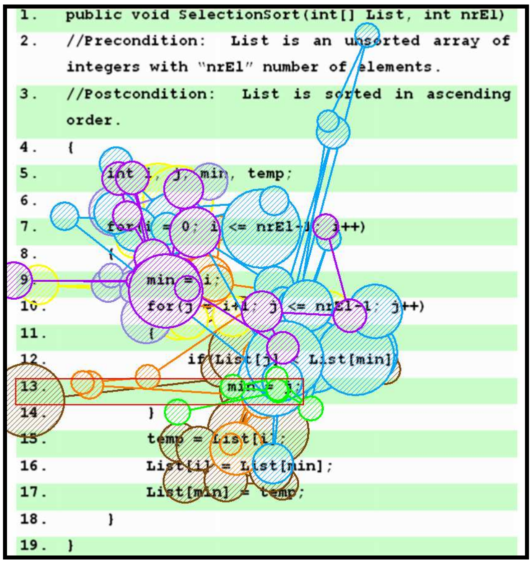
Question 4 Control Heat Maps



```
1.  public void SelectionSort(int[] List, int nrEl)
2.  //Precondition:  List is an unsorted array of
    integers with "nrEl" number of elements.
3.  //Postcondition:  List is sorted in ascending
    order.
4.  {
5.      int i, j, min, temp;
6.
7.      for(i = 0; i <= nrEl-1; i++)
8.      {
9.          min = i;
10.         for(j = i+1; j <= nrEl-1; j++)
11.         {
12.             if(List[j] < List[min])
13.                 min = j;
14.         }
15.         temp = List[i];
16.         List[i] = List[min];
17.         List[min] = temp;
18.     }
19. }
```

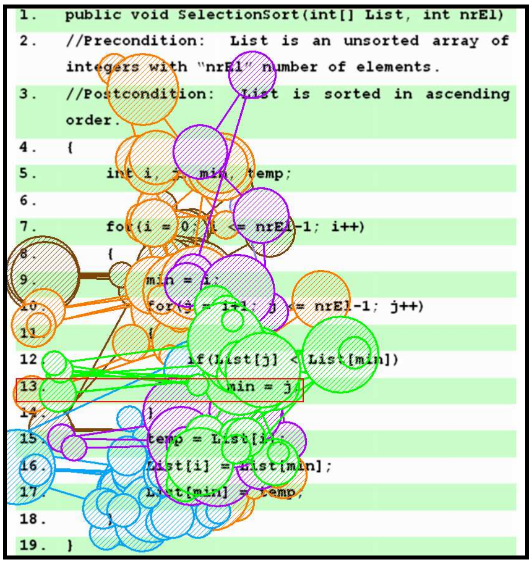
Q5. "Which line or lines sets the position of the smallest value in the unsorted part of the array?"
Correct answer: Line 13

Question 5 Treatment Scanpaths



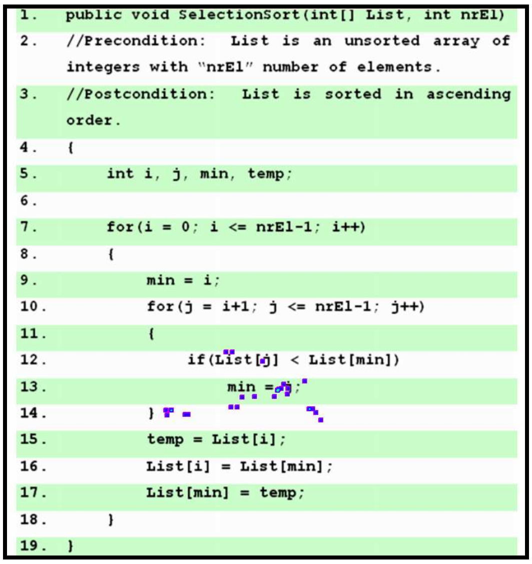
```
1.  public void SelectionSort(int[] List, int nrEl)
2.  //Precondition:  List is an unsorted array of
    integers with "nrEl" number of elements.
3.  //Postcondition:  List is sorted in ascending
    order.
4.  {
5.      int i, j, min, temp;
6.
7.      for(i = 0; i <= nrEl-1; i++)
8.      {
9.          min = i;
10.         for(j = i+1; j <= nrEl-1; j++)
11.         {
12.             if(List[j] < List[min])
13.                 min = j;
14.         }
15.         temp = List[i];
16.         List[i] = List[min];
17.         List[min] = temp;
18.     }
19. }
```

Question 5 Control Scanpaths



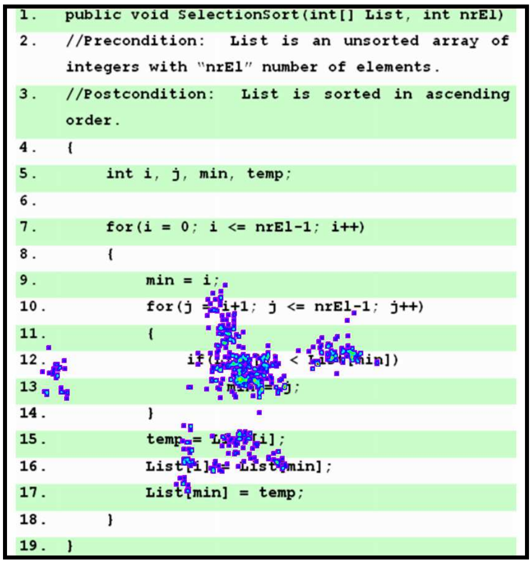
```
1.  public void SelectionSort(int[] List, int nrEl)
2.  //Precondition:  List is an unsorted array of
    integers with "nrEl" number of elements.
3.  //Postcondition:  List is sorted in ascending
    order.
4.  {
5.      int i, j, min, temp;
6.
7.      for(i = 0; i <= nrEl-1; i++)
8.      {
9.          min = i;
10.         for(j = i+1; j <= nrEl-1; j++)
11.         {
12.             if(List[j] < List[min])
13.                 min = j;
14.         }
15.         temp = List[i];
16.         List[i] = List[min];
17.         List[min] = temp;
18.     }
19. }
```

Question 5 Treatment Heat Maps



```
1.  public void SelectionSort(int[] List, int nrEl)
2.  //Precondition:  List is an unsorted array of
    integers with "nrEl" number of elements.
3.  //Postcondition:  List is sorted in ascending
    order.
4.  {
5.      int i, j, min, temp;
6.
7.      for(i = 0; i <= nrEl-1; i++)
8.      {
9.          min = i;
10.         for(j = i+1; j <= nrEl-1; j++)
11.         {
12.             if(List[j] < List[min])
13.                 min = j;
14.         }
15.         temp = List[i];
16.         List[i] = List[min];
17.         List[min] = temp;
18.     }
19. }
```

Question 5 Control Heat Maps

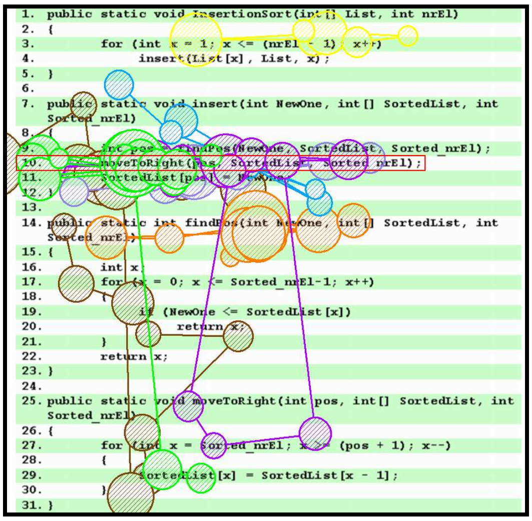


```
1.  public void SelectionSort(int[] List, int nrEl)
2.  //Precondition:  List is an unsorted array of
    integers with "nrEl" number of elements.
3.  //Postcondition:  List is sorted in ascending
    order.
4.  {
5.      int i, j, min, temp;
6.
7.      for(i = 0; i <= nrEl-1; i++)
8.      {
9.          min = i;
10.         for(j = i+1; j <= nrEl-1; j++)
11.         {
12.             if(List[j] < List[min])
13.                 min = j;
14.         }
15.         temp = List[i];
16.         List[i] = List[min];
17.         List[min] = temp;
18.     }
19. }
```

Experiment B

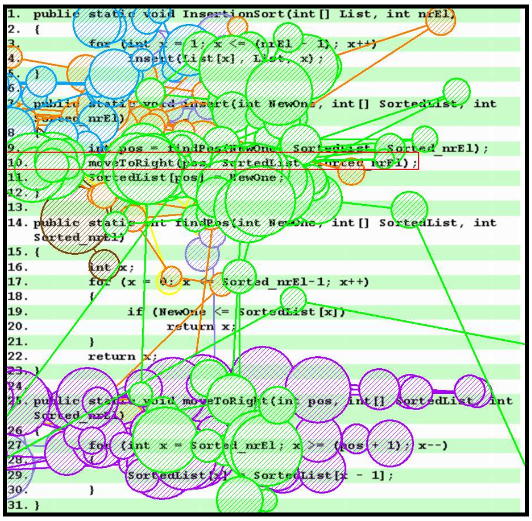
Q1. "Which line/lines calls a function to make space for a new element in an array?"
Correct answer: Line 10

Question 1 Treatment Scanpaths



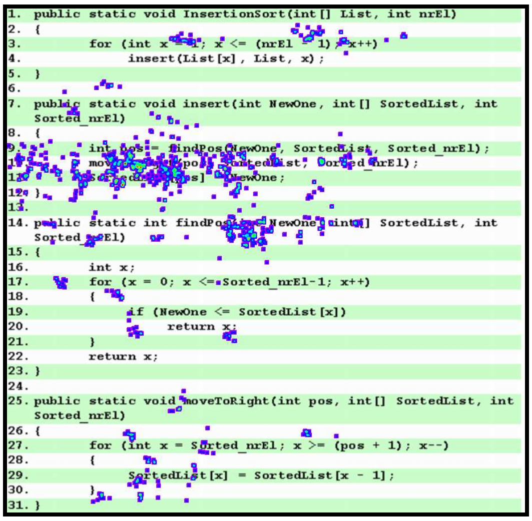
```
1.  public static void InsertionSort(int[] List, int nrEl)
2.  {
3.          for (int x = 1; x <= (nrEl - 1); x++)
4.               insert(List[x], List, x);
5.  }
6.
7.  public static void insert(int NewOne, int[] SortedList, int
    Sorted_nrEl)
8.  {
9.          int pos = findPos(NewOne, SortedList, Sorted_nrEl);
10.         moveToRight(pos, SortedList, Sorted_nrEl);
11.         SortedList[pos] = NewOne;
12. }
13.
14. public static int findPos(int NewOne, int[] SortedList, int
    Sorted_nrEl)
15. {
16.         int x;
17.         for (x = 0; x <= Sorted_nrEl-1; x++)
18.         {
19.              if (NewOne <= SortedList[x])
20.                   return x;
21.         }
22.         return x;
23. }
24.
25. public static void moveToRight(int pos, int[] SortedList, int
    Sorted_nrEl)
26. {
27.         for (int x = Sorted_nrEl; x >= (pos + 1); x--)
28.         {
29.              SortedList[x] = SortedList[x - 1];
30.         }
31. }
```

Question 1 Control Scanpaths



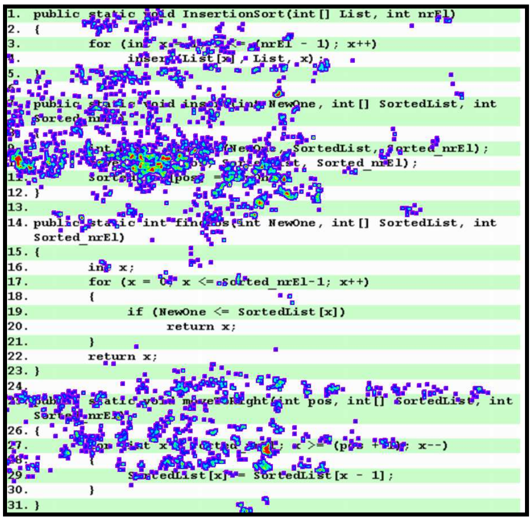
```
1.  public static void InsertionSort(int[] List, int nrEl)
2.  {
3.          for (int x = 1; x <= (nrEl - 1); x++)
4.               insert(List[x], List, x);
5.  }
6.
7.  public static void insert(int NewOne, int[] SortedList, int
    Sorted_nrEl)
8.  {
9.          int pos = findPos(NewOne, SortedList, Sorted_nrEl);
10.         moveToRight(pos, SortedList, Sorted_nrEl);
11.         SortedList[pos] = NewOne;
12. }
13.
14. public static int findPos(int NewOne, int[] SortedList, int
    Sorted_nrEl)
15. {
16.         int x;
17.         for (x = 0; x <= Sorted_nrEl-1; x++)
18.         {
19.              if (NewOne <= SortedList[x])
20.                   return x;
21.         }
22.         return x;
23. }
24.
25. public static void moveToRight(int pos, int[] SortedList, int
    Sorted_nrEl)
26. {
27.         for (int x = Sorted_nrEl; x >= (pos + 1); x--)
28.         {
29.              SortedList[x] = SortedList[x - 1];
30.         }
31. }
```

Question 1 Treatment Heat Maps



```
1.  public static void InsertionSort(int[] List, int nrEl)
2.  {
3.          for (int x = 1; x <= (nrEl - 1); x++)
4.               insert(List[x], List, x);
5.  }
6.
7.  public static void insert(int NewOne, int[] SortedList, int
    Sorted_nrEl)
8.  {
9.          int pos = findPos(NewOne, SortedList, Sorted_nrEl);
10.         moveToRight(pos, SortedList, Sorted_nrEl);
11.         SortedList[pos] = NewOne;
12. }
13.
14. public static int findPos(int NewOne, int[] SortedList, int
    Sorted_nrEl)
15. {
16.         int x;
17.         for (x = 0; x <= Sorted_nrEl-1; x++)
18.         {
19.              if (NewOne <= SortedList[x])
20.                   return x;
21.         }
22.         return x;
23. }
24.
25. public static void moveToRight(int pos, int[] SortedList, int
    Sorted_nrEl)
26. {
27.         for (int x = Sorted_nrEl; x >= (pos + 1); x--)
28.         {
29.              SortedList[x] = SortedList[x - 1];
30.         }
31. }
```

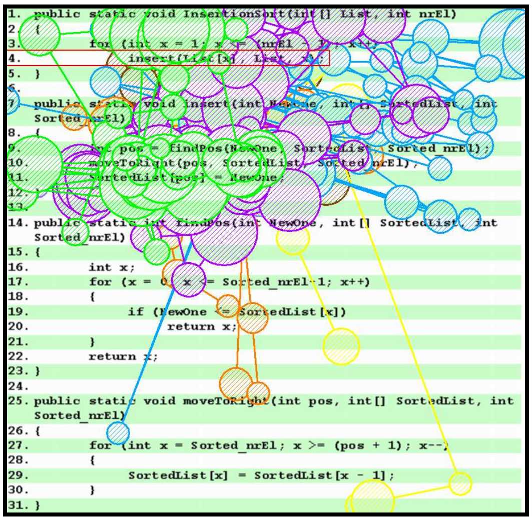
Question 1 Control Heat Maps



```
1.  public static void InsertionSort(int[] List, int nrEl)
2.  {
3.          for (int x = 1; x <= (nrEl - 1); x++)
4.               insert(List[x], List, x);
5.  }
6.
7.  public static void insert(int NewOne, int[] SortedList, int
    Sorted_nrEl)
8.  {
9.          int pos = findPos(NewOne, SortedList, Sorted_nrEl);
10.         moveToRight(pos, SortedList, Sorted_nrEl);
11.         SortedList[pos] = NewOne;
12. }
13.
14. public static int findPos(int NewOne, int[] SortedList, int
    Sorted_nrEl)
15. {
16.         int x;
17.         for (x = 0; x <= Sorted_nrEl-1; x++)
18.         {
19.              if (NewOne <= SortedList[x])
20.                   return x;
21.         }
22.         return x;
23. }
24.
25. public static void moveToRight(int pos, int[] SortedList, int
    Sorted_nrEl)
26. {
27.         for (int x = Sorted_nrEl; x >= (pos + 1); x--)
28.         {
29.              SortedList[x] = SortedList[x - 1];
30.         }
31. }
```

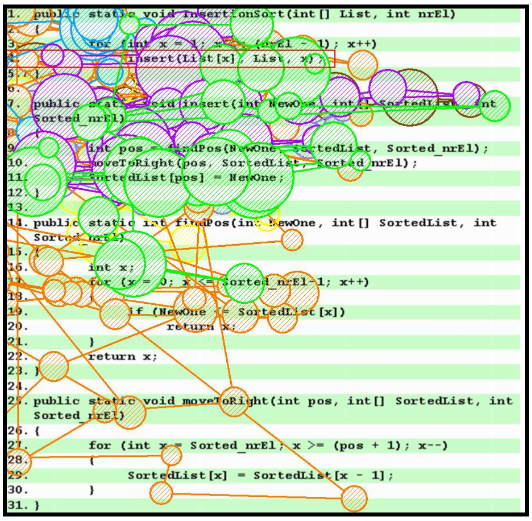
Q2. “Which line or lines calls a function to insert a new element into a sorted array, keeping it sorted?”
Correct answer: Line 4

Question 2 Treatment Scanpaths



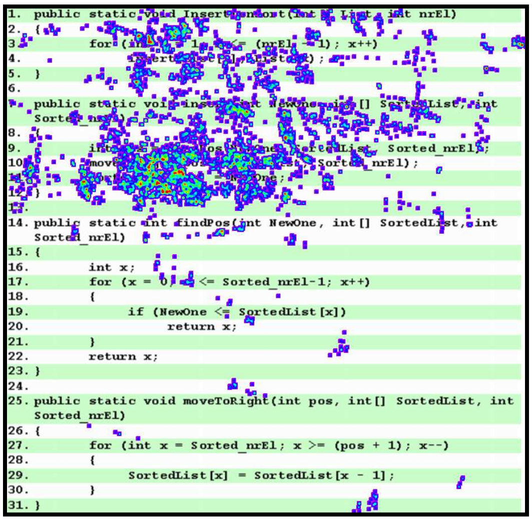
```
1.  public static void InsertionSort(int[] List, int nrEl)
2. {
3.          for (int x = 1; x < (nrEl - 1); x++)
4.                 insert(List[x], List, x);
5. }
6.
7. public static void insert(int NewOne, int[] SortedList, int
   Sorted_nrEl)
8. {
9.          int pos = findPos(NewOne, SortedList, Sorted_nrEl);
10.         moveToRight(pos, SortedList, Sorted_nrEl);
11.         SortedList[pos] = NewOne;
12. }
13.
14. public static int findPos(int NewOne, int[] SortedList, int
   Sorted_nrEl)
15. {
16.         int x;
17.         for (x = 0; x <= Sorted_nrEl-1; x++)
18.         {
19.                if (NewOne <= SortedList[x])
20.                       return x;
21.         }
22.         return x;
23. }
24.
25. public static void moveToRight(int pos, int[] SortedList, int
   Sorted_nrEl)
26. {
27.         for (int x = Sorted_nrEl; x >= (pos + 1); x--)
28.         {
29.                SortedList[x] = SortedList[x - 1];
30.         }
31. }
```

Question 2 Control Scanpaths



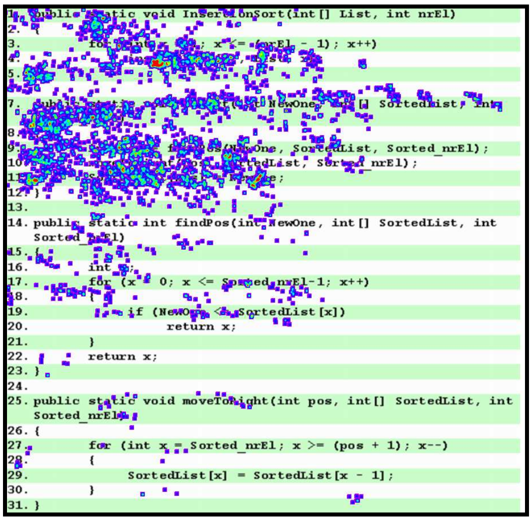
```
1.  public static void InsertionSort(int[] List, int nrEl)
2. {
3.          for (int x = 1; x < (nrEl - 1); x++)
4.                 insert(List[x], List, x);
5. }
6.
7. public static void insert(int NewOne, int[] SortedList, int
   Sorted_nrEl)
8. {
9.          int pos = findPos(NewOne, SortedList, Sorted_nrEl);
10.         moveToRight(pos, SortedList, Sorted_nrEl);
11.         SortedList[pos] = NewOne;
12. }
13.
14. public static int findPos(int NewOne, int[] SortedList, int
   Sorted_nrEl)
15. {
16.         int x;
17.         for (x = 0; x <= Sorted_nrEl-1; x++)
18.         {
19.                if (NewOne <= SortedList[x])
20.                       return x;
21.         }
22.         return x;
23. }
24.
25. public static void moveToRight(int pos, int[] SortedList, int
   Sorted_nrEl)
26. {
27.         for (int x = Sorted_nrEl; x >= (pos + 1); x--)
28.         {
29.                SortedList[x] = SortedList[x - 1];
30.         }
31. }
```

Question 2 Treatment Heat Maps



```
1.  public static void InsertionSort(int[] List, int nrEl)
2. {
3.          for (int x = 1; x < (nrEl - 1); x++)
4.                 insert(List[x], List, x);
5. }
6.
7. public static void insert(int NewOne, int[] SortedList, int
   Sorted_nrEl)
8. {
9.          int pos = findPos(NewOne, SortedList, Sorted_nrEl);
10.         moveToRight(pos, SortedList, Sorted_nrEl);
11.         SortedList[pos] = NewOne;
12. }
13.
14. public static int findPos(int NewOne, int[] SortedList, int
   Sorted_nrEl)
15. {
16.         int x;
17.         for (x = 0; x <= Sorted_nrEl-1; x++)
18.         {
19.                if (NewOne <= SortedList[x])
20.                       return x;
21.         }
22.         return x;
23. }
24.
25. public static void moveToRight(int pos, int[] SortedList, int
   Sorted_nrEl)
26. {
27.         for (int x = Sorted_nrEl; x >= (pos + 1); x--)
28.         {
29.                SortedList[x] = SortedList[x - 1];
30.         }
31. }
```

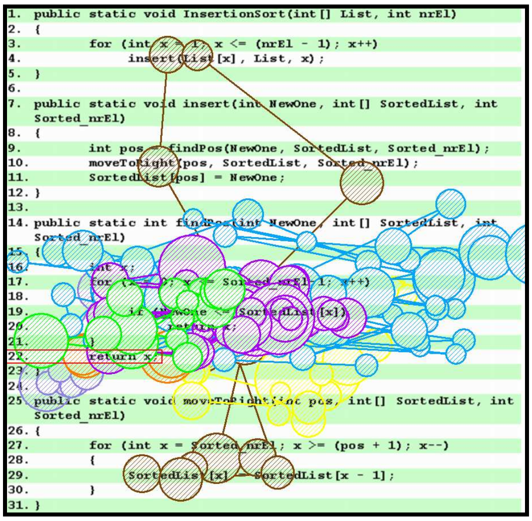
Question 2 Control Heat Maps



```
1.  public static void InsertionSort(int[] List, int nrEl)
2. {
3.          for (int x = 1; x < (nrEl - 1); x++)
4.                 insert(List[x], List, x);
5. }
6.
7. public static void insert(int NewOne, int[] SortedList, int
   Sorted_nrEl)
8. {
9.          int pos = findPos(NewOne, SortedList, Sorted_nrEl);
10.         moveToRight(pos, SortedList, Sorted_nrEl);
11.         SortedList[pos] = NewOne;
12. }
13.
14. public static int findPos(int NewOne, int[] SortedList, int
   Sorted_nrEl)
15. {
16.         int x;
17.         for (x = 0; x <= Sorted_nrEl-1; x++)
18.         {
19.                if (NewOne <= SortedList[x])
20.                       return x;
21.         }
22.         return x;
23. }
24.
25. public static void moveToRight(int pos, int[] SortedList, int
   Sorted_nrEl)
26. {
27.         for (int x = Sorted_nrEl; x >= (pos + 1); x--)
28.         {
29.                SortedList[x] = SortedList[x - 1];
30.         }
31. }
```

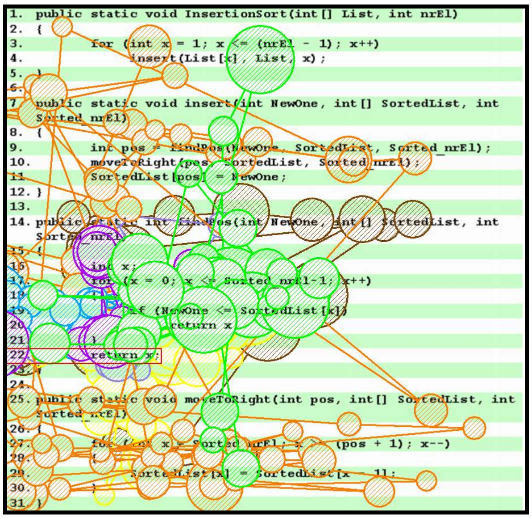
Q3. "Which line or lines will return a position in the array if the new value is higher than all the values in the sorted list?"
Correct answer: Line 22

Question 3 Treatment Scanpaths



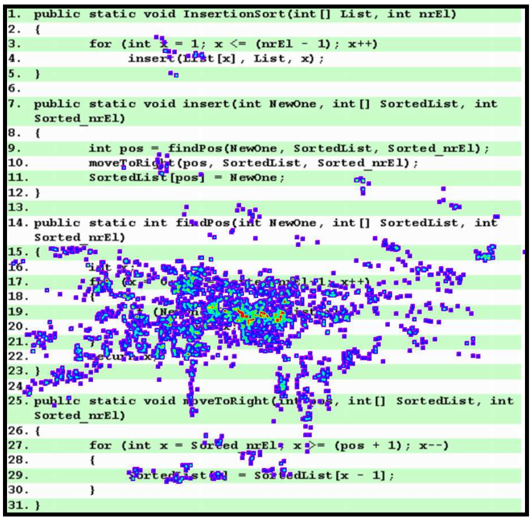
```
1.  public static void InsertionSort(int[] List, int nrEl)
2.  {
3.          for (int x = 1; x <= (nrEl - 1); x++)
4.               insert(List[x], List, x);
5.  }
6.
7.  public static void insert(int NewOne, int[] SortedList, int
    Sorted_nrEl)
8.  {
9.          int pos = findPos(NewOne, SortedList, Sorted_nrEl);
10.         moveToRight(pos, SortedList, Sorted_nrEl);
11.         SortedList[pos] = NewOne;
12. }
13.
14. public static int findPos(int NewOne, int[] SortedList, int
    Sorted_nrEl)
15. {
16.         int x;
17.         for (x = 0; x <= Sorted_nrEl-1; x++)
18.         {
19.              if (NewOne <= SortedList[x])
20.                   return x;
21.         }
22.         return x;
23. }
24.
25. public static void moveToRight(int pos, int[] SortedList, int
    Sorted_nrEl)
26. {
27.         for (int x = Sorted_nrEl; x >= (pos + 1); x--)
28.         {
29.              SortedList[x] = SortedList[x - 1];
30.         }
31. }
```

Question 3 Control Scanpaths



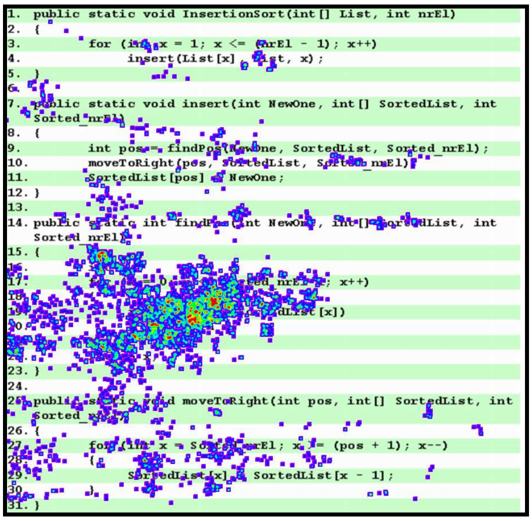
```
1.  public static void InsertionSort(int[] List, int nrEl)
2.  {
3.          for (int x = 1; x <= (nrEl - 1); x++)
4.               insert(List[x], List, x);
5.  }
6.
7.  public static void insert(int NewOne, int[] SortedList, int
    Sorted_nrEl)
8.  {
9.          int pos = findPos(NewOne, SortedList, Sorted_nrEl);
10.         moveToRight(pos, SortedList, Sorted_nrEl);
11.         SortedList[pos] = NewOne;
12. }
13.
14. public static int findPos(int NewOne, int[] SortedList, int
    Sorted_nrEl)
15. {
16.         int x;
17.         for (x = 0; x <= Sorted_nrEl-1; x++)
18.         {
19.              if (NewOne <= SortedList[x])
20.                   return x;
21.         }
22.         return x;
23. }
24.
25. public static void moveToRight(int pos, int[] SortedList, int
    Sorted_nrEl)
26. {
27.         for (int x = Sorted_nrEl; x >= (pos + 1); x--)
28.         {
29.              SortedList[x] = SortedList[x - 1];
30.         }
31. }
```

Question 3 Treatment Heat Maps



```
1.  public static void InsertionSort(int[] List, int nrEl)
2.  {
3.          for (int x = 1; x <= (nrEl - 1); x++)
4.               insert(List[x], List, x);
5.  }
6.
7.  public static void insert(int NewOne, int[] SortedList, int
    Sorted_nrEl)
8.  {
9.          int pos = findPos(NewOne, SortedList, Sorted_nrEl);
10.         moveToRight(pos, SortedList, Sorted_nrEl);
11.         SortedList[pos] = NewOne;
12. }
13.
14. public static int findPos(int NewOne, int[] SortedList, int
    Sorted_nrEl)
15. {
16.         int x;
17.         for (x = 0; x <= Sorted_nrEl-1; x++)
18.         {
19.              if (NewOne <= SortedList[x])
20.                   return x;
21.         }
22.         return x;
23. }
24.
25. public static void moveToRight(int pos, int[] SortedList, int
    Sorted_nrEl)
26. {
27.         for (int x = Sorted_nrEl; x >= (pos + 1); x--)
28.         {
29.              SortedList[x] = SortedList[x - 1];
30.         }
31. }
```

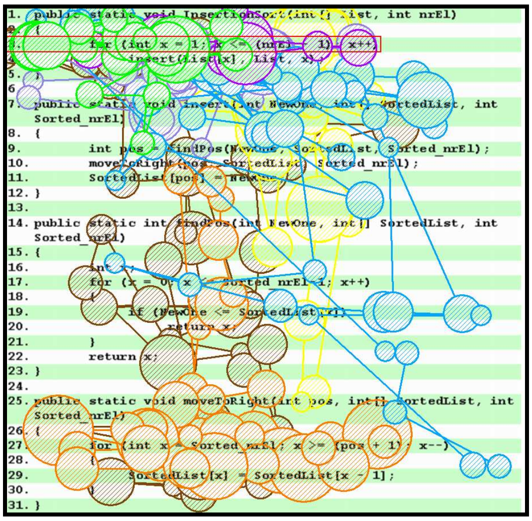
Question 3 Control Heat Maps



```
1.  public static void InsertionSort(int[] List, int nrEl)
2.  {
3.          for (int x = 1; x <= (nrEl - 1); x++)
4.               insert(List[x], List, x);
5.  }
6.
7.  public static void insert(int NewOne, int[] SortedList, int
    Sorted_nrEl)
8.  {
9.          int pos = findPos(NewOne, SortedList, Sorted_nrEl);
10.         moveToRight(pos, SortedList, Sorted_nrEl);
11.         SortedList[pos] = NewOne;
12. }
13.
14. public static int findPos(int NewOne, int[] SortedList, int
    Sorted_nrEl)
15. {
16.         int x;
17.         for (x = 0; x <= Sorted_nrEl-1; x++)
18.         {
19.              if (NewOne <= SortedList[x])
20.                   return x;
21.         }
22.         return x;
23. }
24.
25. public static void moveToRight(int pos, int[] SortedList, int
    Sorted_nrEl)
26. {
27.         for (int x = Sorted_nrEl; x >= (pos + 1); x--)
28.         {
29.              SortedList[x] = SortedList[x - 1];
30.         }
31. }
```

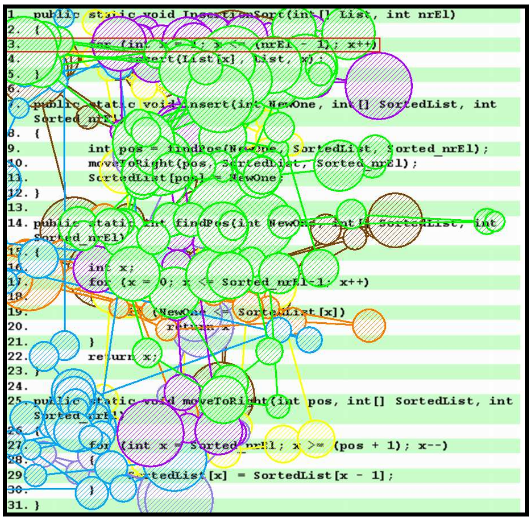
Q4. "Which line or lines will loop through the unsorted part of the array?"
Correct answer: Line 3

Question 4 Treatment Scanpaths



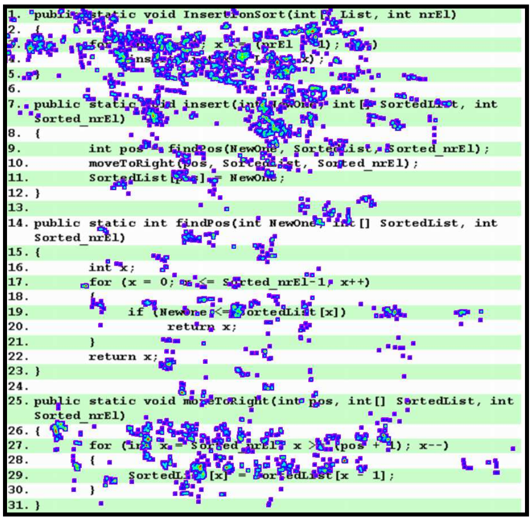
```
1.  public static void InsertionSort(int[] List, int nrEl)
2.  {
3.        for (int x = 1; x <= (nrEl - 1); x++)
4.             insert(List[x], List, x);
5.  }
6.
7.  public static void insert(int NewOne, int[] SortedList, int
    Sorted_nrEl)
8.  {
9.        int pos = findPos(NewOne, SortedList, Sorted_nrEl);
10.       moveToRight(pos, SortedList, Sorted_nrEl);
11.       SortedList[pos] = NewOne;
12. }
13.
14. public static int findPos(int NewOne, int[] SortedList, int
    Sorted_nrEl)
15. {
16.       int x;
17.       for (x = 0; x <= Sorted_nrEl-1; x++)
18.       {
19.            if (NewOne <= SortedList[x])
20.                 return x;
21.       }
22.       return x;
23. }
24.
25. public static void moveToRight(int pos, int[] SortedList, int
    Sorted_nrEl)
26. {
27.       for (int x = Sorted_nrEl; x >= (pos + 1); x--)
28.       {
29.            SortedList[x] = SortedList[x - 1];
30.       }
31. }
```

Question 4 Control Scanpaths



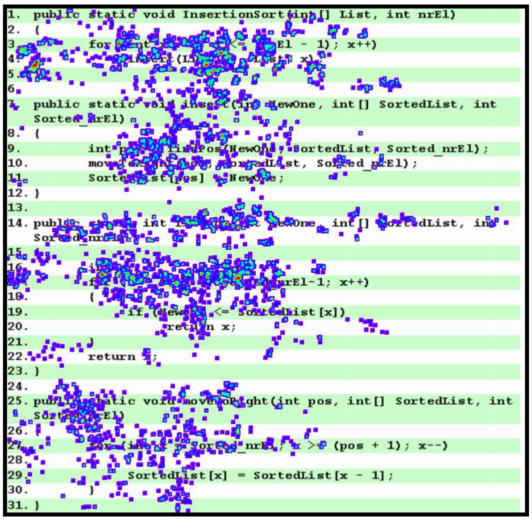
```
1.  public static void InsertionSort(int[] List, int nrEl)
2.  {
3.        for (int x = 1; x <= (nrEl - 1); x++)
4.             insert(List[x], List, x);
5.  }
6.
7.  public static void insert(int NewOne, int[] SortedList, int
    Sorted_nrEl)
8.  {
9.        int pos = findPos(NewOne, SortedList, Sorted_nrEl);
10.       moveToRight(pos, SortedList, Sorted_nrEl);
11.       SortedList[pos] = NewOne;
12. }
13.
14. public static int findPos(int NewOne, int[] SortedList, int
    Sorted_nrEl)
15. {
16.       int x;
17.       for (x = 0; x <= Sorted_nrEl-1; x++)
18.       {
19.            if (NewOne <= SortedList[x])
20.                 return x;
21.       }
22.       return x;
23. }
24.
25. public static void moveToRight(int pos, int[] SortedList, int
    Sorted_nrEl)
26. {
27.       for (int x = Sorted_nrEl; x >= (pos + 1); x--)
28.       {
29.            SortedList[x] = SortedList[x - 1];
30.       }
31. }
```

Question 4 Treatment Heat Maps



```
1.  public static void InsertionSort(int[] List, int nrEl)
2.  {
3.        for (int x = 1; x <= (nrEl - 1); x++)
4.             insert(List[x], List, x);
5.  }
6.
7.  public static void insert(int NewOne, int[] SortedList, int
    Sorted_nrEl)
8.  {
9.        int pos = findPos(NewOne, SortedList, Sorted_nrEl);
10.       moveToRight(pos, SortedList, Sorted_nrEl);
11.       SortedList[pos] = NewOne;
12. }
13.
14. public static int findPos(int NewOne, int[] SortedList, int
    Sorted_nrEl)
15. {
16.       int x;
17.       for (x = 0; x <= Sorted_nrEl-1; x++)
18.       {
19.            if (NewOne <= SortedList[x])
20.                 return x;
21.       }
22.       return x;
23. }
24.
25. public static void moveToRight(int pos, int[] SortedList, int
    Sorted_nrEl)
26. {
27.       for (int x = Sorted_nrEl; x >= (pos + 1); x--)
28.       {
29.            SortedList[x] = SortedList[x - 1];
30.       }
31. }
```

Question 4 Control Heat Maps

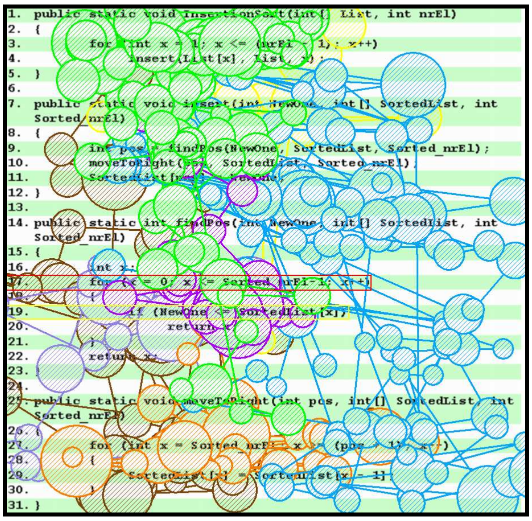


```
1.  public static void InsertionSort(int[] List, int nrEl)
2.  {
3.        for (int x = 1; x <= (nrEl - 1); x++)
4.             insert(List[x], List, x);
5.  }
6.
7.  public static void insert(int NewOne, int[] SortedList, int
    Sorted_nrEl)
8.  {
9.        int pos = findPos(NewOne, SortedList, Sorted_nrEl);
10.       moveToRight(pos, SortedList, Sorted_nrEl);
11.       SortedList[pos] = NewOne;
12. }
13.
14. public static int findPos(int NewOne, int[] SortedList, int
    Sorted_nrEl)
15. {
16.       int x;
17.       for (x = 0; x <= Sorted_nrEl-1; x++)
18.       {
19.            if (NewOne <= SortedList[x])
20.                 return x;
21.       }
22.       return x;
23. }
24.
25. public static void moveToRight(int pos, int[] SortedList, int
    Sorted_nrEl)
26. {
27.       for (int x = Sorted_nrEl; x >= (pos + 1); x--)
28.       {
29.            SortedList[x] = SortedList[x - 1];
30.       }
31. }
```

Q5. "Which line or lines will loop through the sorted part of the array and determine a suitable position to insert a new element?

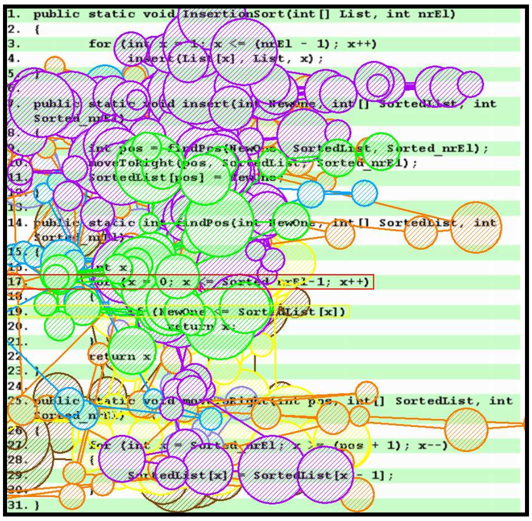
Correct answer: Lines 17, 19

Question 5 Treatment Scanpaths



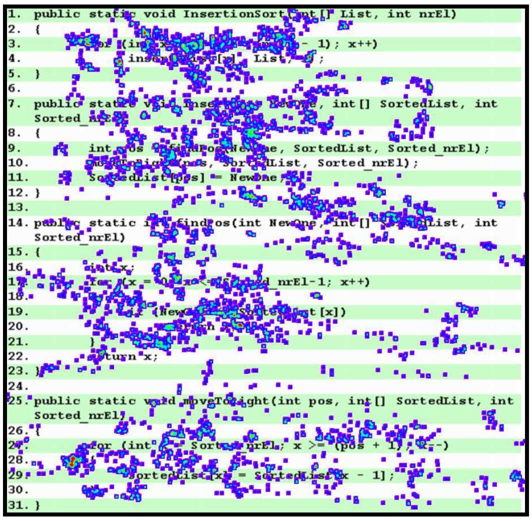
```
1.  public static void InsertionSort(int[] List, int nrEl)
2.  {
3.          for (int x = 1; x <= (nrEl - 1); x++)
4.              insert(List[x], List, x);
5.  }
6.
7.  public static void insert(int NewOne, int[] SortedList, int
    Sorted_nrEl)
8.  {
9.          int pos = findPos(NewOne, SortedList, Sorted_nrEl);
10.         moveToRight(pos, SortedList, Sorted_nrEl);
11.         SortedList[pos] = NewOne;
12. }
13.
14. public static int findPos(int NewOne, int[] SortedList, int
    Sorted_nrEl)
15. {
16.         int x;
17.         for (x = 0; x <= Sorted_nrEl-1; x++)
18.         {
19.             if (NewOne <= SortedList[x])
20.                 return x;
21.         }
22.         return x;
23. }
24.
25. public static void moveToRight(int pos, int[] SortedList, int
    Sorted_nrEl)
26. {
27.         for (int x = Sorted_nrEl; x >= (pos + 1); x--)
28.         {
29.             SortedList[x] = SortedList[x - 1];
30.         }
31. }
```

Question 5 Control Scanpaths



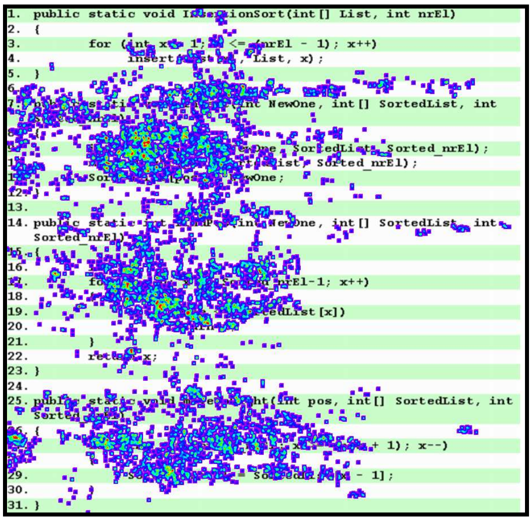
```
1.  public static void InsertionSort(int[] List, int nrEl)
2.  {
3.          for (int x = 1; x <= (nrEl - 1); x++)
4.              insert(List[x], List, x);
5.  }
6.
7.  public static void insert(int NewOne, int[] SortedList, int
    Sorted_nrEl)
8.  {
9.          int pos = findPos(NewOne, SortedList, Sorted_nrEl);
10.         moveToRight(pos, SortedList, Sorted_nrEl);
11.         SortedList[pos] = NewOne;
12. }
13.
14. public static int findPos(int NewOne, int[] SortedList, int
    Sorted_nrEl)
15. {
16.         int x;
17.         for (x = 0; x <= Sorted_nrEl-1; x++)
18.         {
19.             if (NewOne <= SortedList[x])
20.                 return x;
21.         }
22.         return x;
23. }
24.
25. public static void moveToRight(int pos, int[] SortedList, int
    Sorted_nrEl)
26. {
27.         for (int x = Sorted_nrEl; x >= (pos + 1); x--)
28.         {
29.             SortedList[x] = SortedList[x - 1];
30.         }
31. }
```

Question 5 Treatment Heat Maps



```
1.  public static void InsertionSort(int[] List, int nrEl)
2.  {
3.          for (int x = 1; x <= (nrEl - 1); x++)
4.              insert(List[x], List, x);
5.  }
6.
7.  public static void insert(int NewOne, int[] SortedList, int
    Sorted_nrEl)
8.  {
9.          int pos = findPos(NewOne, SortedList, Sorted_nrEl);
10.         moveToRight(pos, SortedList, Sorted_nrEl);
11.         SortedList[pos] = NewOne;
12. }
13.
14. public static int findPos(int NewOne, int[] SortedList, int
    Sorted_nrEl)
15. {
16.         int x;
17.         for (x = 0; x <= Sorted_nrEl-1; x++)
18.         {
19.             if (NewOne <= SortedList[x])
20.                 return x;
21.         }
22.         return x;
23. }
24.
25. public static void moveToRight(int pos, int[] SortedList, int
    Sorted_nrEl)
26. {
27.         for (int x = Sorted_nrEl; x >= (pos + 1); x--)
28.         {
29.             SortedList[x] = SortedList[x - 1];
30.         }
31. }
```

Question 5 Control Heat Maps



```
1.  public static void InsertionSort(int[] List, int nrEl)
2.  {
3.          for (int x = 1; x <= (nrEl - 1); x++)
4.              insert(List[x], List, x);
5.  }
6.
7.  public static void insert(int NewOne, int[] SortedList, int
    Sorted_nrEl)
8.  {
9.          int pos = findPos(NewOne, SortedList, Sorted_nrEl);
10.         moveToRight(pos, SortedList, Sorted_nrEl);
11.         SortedList[pos] = NewOne;
12. }
13.
14. public static int findPos(int NewOne, int[] SortedList, int
    Sorted_nrEl)
15. {
16.         int x;
17.         for (x = 0; x <= Sorted_nrEl-1; x++)
18.         {
19.             if (NewOne <= SortedList[x])
20.                 return x;
21.         }
22.         return x;
23. }
24.
25. public static void moveToRight(int pos, int[] SortedList, int
    Sorted_nrEl)
26. {
27.         for (int x = Sorted_nrEl; x >= (pos + 1); x--)
28.         {
29.             SortedList[x] = SortedList[x - 1];
30.         }
31. }
```

Appendix J: SAICSIT Paper

Supporting CS1 with a Program Beacon Recognition Tool

Ronald Leppan
Department of CS&IS, NMMU
PO Box 1600, NMMU, 6031
(+2741) 504 9109
Ronald.Leppan@nmmu.ac.za

Charmain Cilliers
Department of CS&IS, NMMU
PO Box 1600, NMMU, 6031
(+2741) 504 2235
Charmain.Cilliers@nmmu.ac.za

Marinda Taljaard
Department of CS&IS, NMMU
PO Box 1600, NMMU, 6031
(+2741) 504 2668
Marinda.Taljaard@nmmu.ac.za

ABSTRACT

Reading and understanding algorithms is not an easy task and often neglected by educators in an introductory programming course. One proposed solution to this problem is the incorporation of a technological support tool to aid program comprehension in CS1. One such support tool (BeReT) is primarily designed to encourage a student to correctly identify beacons within provided program extracts. A between-groups experiment is described which compares the program comprehension of students that used BeReT to study various introductory algorithms, with students that relied solely on traditional lecturing materials. The use of an eye tracker was incorporated into the empirical study to provide additional data to measure the effect of BeReT. The results indicate that a technological support tool like BeReT can have a positive effect on student comprehension of introductory algorithms traditionally taught in CS1.

Categories and Subject Descriptors

K.3. [**Computers and Education**]

General Terms

Algorithms, Measurement, Performance, Experimentation, Verification

Keywords

Program Plans, Program Beacons, Program Comprehension, Introductory Programming, CS1

1. INTRODUCTION

Students in CS1 often find it difficult to cope with programming [15, 24]. The lack of support provided by educators in specifically program comprehension further compounds the problem [8]. Studies of the fragile knowledge of students in introductory programming [11, 21, 26, 33] indicate the need for these courses to explicitly include not only code generation strategies, but also code comprehension strategies when studying algorithms. Introductory programming courses focus mainly on program generation and techniques to improve students' ability to write programs. Little effort is put into explicitly improving their ability to read programs [8]. Program generation can be described as the construction of plans to solve a specific problem [28]. During program comprehension, the focus is on the discovery of these plans implemented in the program [11, 19, 28].

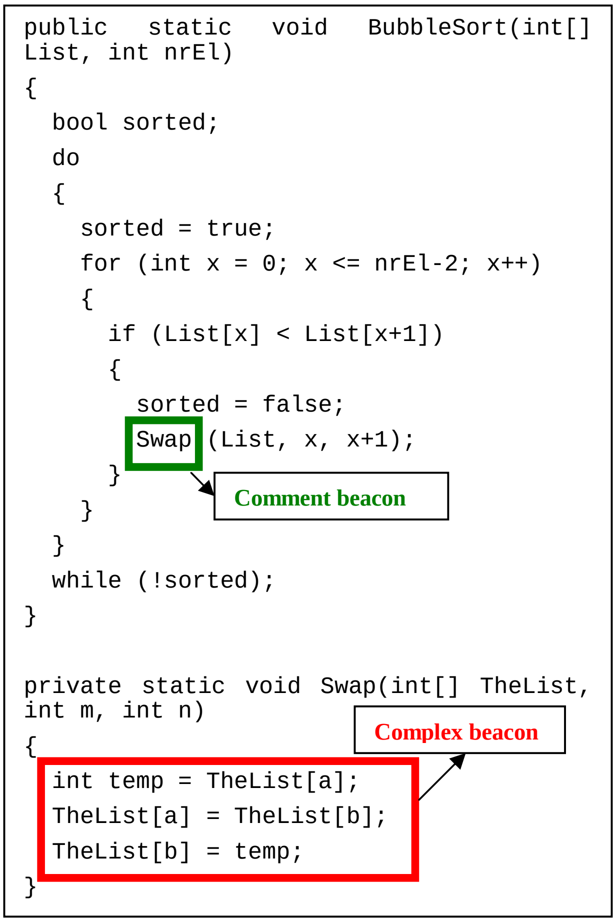
Educators in introductory programming use several techniques to support their students to comprehend algorithms. Some techniques, like animations, flowcharting, control structure diagrams and the visualisation of programming constructs and data structures, are often used for the comprehension of a specific algorithm only. The development of program reading skills is therefore not a primary outcome of these techniques [3, 5, 34]. Giving students a collection of algorithms as reading exercises leads to a number of different reading skills [17], but some students need more guidance to be successful [20]. Educators often investigate expert programmers in order to guide students to become experts themselves. Since expert programmers spend a large portion of their time on maintenance, program reading skills are of vital importance to them. Valuable insights can be gained by investigating expert programmers and the techniques they use to study unfamiliar programs.

This paper highlights two techniques used by expert programmers to read unfamiliar programs (Section 2). An experimental technological support tool that incorporates these two techniques is also described (Section 3). The main focus of the paper however is on an empirical investigation that determines the effectiveness of using this technological tool on the comprehension of introductory programming algorithms (Sections 4 and 5). This paper therefore contributes evidence to educators that the use of a support tool such as the one described in Section 3 can be used to improve program comprehension in CS1.

2. RELATED WORK

Program plans are one of the elements of the mental model of an expert programmer investigating source code [9, 10, 25, 28]. The problem of fostering efficient program comprehension skills in students thus becomes one of aiding them to identify plans in an algorithm. Expert programmers frequently use clichéd plans to construct programs [9]. The difficulty for a student in introductory programming is that they haven't built sufficient plans in long term memory to effectively use them. As a result it seems necessary to build the long term memory of introductory students with plans. In addition to building their long term memory with plans, the techniques used to build these plans must be ones that students can use to discover plans on their own after initial instruction.

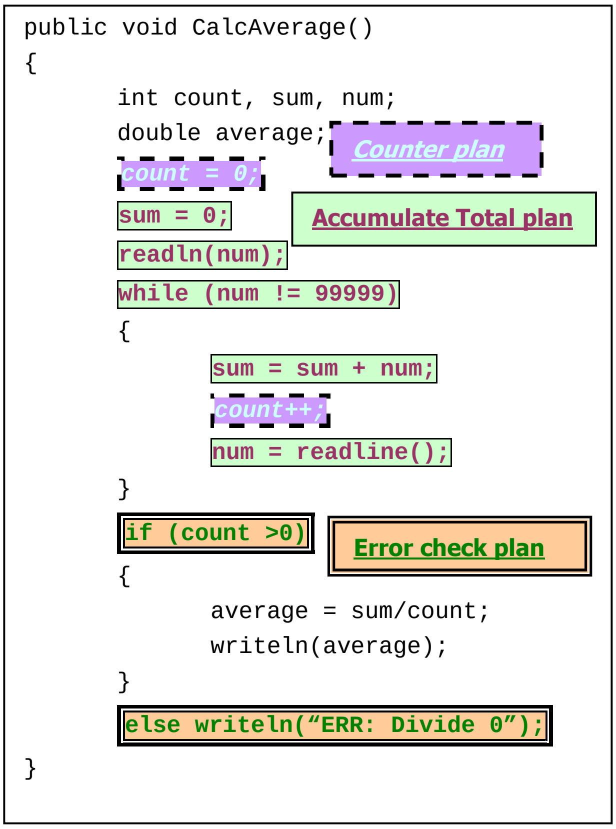
Beacons play a prominent role in the recognition of plans if the programmer has knowledge of the problem domain. Beacons can be defined as surface features in a program that verifies the existence of some structure or operation [31, 32]. A recent study shows that beacons are extensively used by experienced programmers to facilitate program comprehension [6]. It was also shown that novices do not identify meaningful beacons in code. As illustrated in Figure 1 two types of beacons exist, comment and complex [6]. Comment beacons refer to lines in the algorithm that use a naming convention that describes the functionality of the code. Complex beacons refer to a line or multiple lines in the code that contains a key to the functionality of the algorithm. In the example used in Figure 1 the use of the function name “Swap” is an example of a comment beacon. The actual code that performs the swapping of two variables is an example of a complex beacon. Complex beacons are often unique in structure and used so frequently in specific contexts that expert programmers use them to validate the existence of a program plan. A programmer that performs a surface scan of the code of the bubble sort algorithm could discover any one of the two types of beacons. This discovery would prompt the programmer to hypothesize the existence of a plan to perform some sorting operation. The programmer would then search for more beacons to support this hypothesis [24].



```
public  static  void  BubbleSort(int[]
List, int nrEl)
{
  bool sorted;
  do
  {
    sorted = true;
    for (int x = 0; x <= nrEl-2; x++)
    {
      if (List[x] < List[x+1])
      {
        sorted = false;
        Swap (List, x, x+1);
      }
    }
  }
  while (!sorted);
}

private static void Swap(int[] TheList,
int m, int n)
{
  int temp = TheList[a];
  TheList[a] = TheList[b];
  TheList[b] = temp;
}
```

Figure 1 Examples of the two types of beacons

Expert programmers investigating source code in unfamiliar domains frequently revert to a technique that can be described as the “chunking” of individual code fragments into related groups. Often these “chunks” can be associated with an intended plan in the algorithm (Figure 2). For instance, a function that is used to calculate the average of a sum of numbers provided by the user could make use of three plans, a “Counter” plan, an “Accumulate Total” plan and an “Error Check” plan. The “Counter” plan is used to keep track of the number of values provided by the user; the “Accumulate Total” plan adds up the values and the “Error Check” plan ensures that values are provided by the user before calculating the average. Chunking is required to maximize the capacity of the short-term memory during the activity of program reading and to add new chunks into long-term memory. Instead of trying to memorize individual statements making up the entire algorithm, programmers can group related statements together into the three plans. The three plans can then be learnt as logical units. If the chunks are unique in structure or if they include mnemonic hints about program functionality, they become beacons for future algorithm reading activities.



```
public void CalcAverage()
{
        int count, sum, num;
        double average;
        count = 0;
        sum = 0;
        readln(num);
        while (num != 99999)
        {
                sum = sum + num;
                count++;
                num = readline();
        }
        if (count >0)
        {
                average = sum/count;
                writeln(average);
        }
        else writeln(“ERR: Divide 0”);
}
```

Figure 2 Use of chunking to discover plans in a program

Experts use two strategies to study unfamiliar code; a top-down or bottom-up process of comprehension [30]. Top-down comprehension requires knowledge of the problem domain [4, 27]. This problem domain knowledge is translated into hypotheses that will be used to determine the functionality of the unfamiliar program. The text structure is scanned for beacons to accept/reject the hypotheses. Throughout the scanning of the unfamiliar code, the hypotheses change as needed. Bottom-up comprehension assumes knowledge of the syntactic elements of the text structure and at least the ability to extract semantic knowledge from the unfamiliar algorithm [25]. The programmer reads the algorithm and "chunks" together related statements that performs a single higher level operation. These chunks are stored in long term memory, often with a mental annotation to describe the grouping. The combination of chunks is used to determine the functionality of the program.

Since beacon recognition and chunking are two techniques used by expert programmers in familiar comprehension processes (top-down and bottom-up), they are therefore regarded in this paper as suitable techniques to aid the comprehension of introductory algorithms. By providing students with an environment that supports beacon recognition and chunking, a network of plans can be stored in long term memory of the student. This could prove invaluable in future program reading activities. Requirements need to be in place to aid the process of selecting or creating a suitable environment that supports beacon recognition and chunking.

Combining the results of a study of program comprehension activities of expert programmers, and techniques used by educators in CS1 to aid comprehension of introductory algorithms, produced the requirements summarised in Table 1 [18].

Table 1 Requirements to aid the comprehension of introductory algorithms

	Requirements
1.	Allow student to extract semantic information (plans) from worked example.
2.	Promotes the use of chunking to learn an unfamiliar algorithm.
3.	Coaches students to spot beacons in different algorithms.
4.	Provide students with syntactically and semantically correct algorithms.
5.	Engage students in active learning activities.
6.	Increases ability to recall the semantics of an algorithm.
7.	Enable students to successfully transfer knowledge.

The goal of requirement 1 is to provide an environment that focuses on program comprehension of algorithms, instead of program generation. The two techniques of beacon recognition and chunking that are extensively used by expert programmers during program comprehension must be advocated in this environment (requirements 2 and 3). Students must see the process of chunking and the benefit it holds when studying an unfamiliar algorithm. The goal behind requirement 2 is to motivate students to use chunking in future program comprehension activities. The environment must highlight beacons (especially complex beacons) in the algorithm and explain the plan behind the beacons. The goal of requirement 3 is to build a collection of beacons students can use in future program comprehension activities. Requirement 4 ensures that proper programming standards are maintained in the algorithm to be studied. This requirement ensures that students are able to determine syntax rules from the algorithm. Requirement 4 also promotes the use of standard programming conventions to make it easier for students to study the code. Requirement 5 is necessary to force students away from passive and rote learning. Interaction is necessary to actively engage students with the algorithm. Requirement 6 necessitates students to display comprehension of the algorithm studied. After exposing students to algorithms in this environment, they must be able to transfer the knowledge gained to future program comprehension and program generation activities (Requirement 7). The transfer of knowledge refers to both using the techniques of chunking and beacon recognition in program comprehension as well as the ability to recognise similar plans in unseen algorithms.

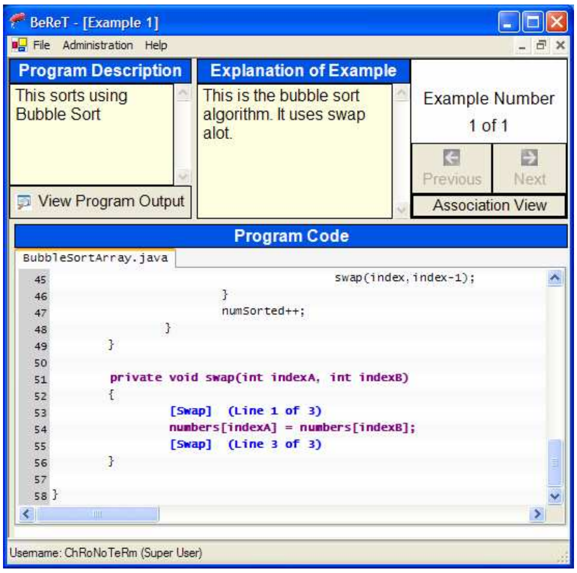
An investigation into program comprehension support tools used by educators identified tools that met most requirements in Table 1, except for requirements 2 and 3 specifically [13, 18]. An experimental support tool was developed to meet requirements 1 – 5 [12, 13] (Section 3). An empirical study was undertaken, which incorporated the collection of eye tracking data, to determine if the experimental tool meets requirements 6 and 7. The empirical study was therefore used to provide evidence of student comprehension of the algorithms studied and the ability of the students to transfer the knowledge gained in the tool to future program comprehension activities.

Eye tracking provides more insight about what a programmer is focusing on during their comprehension process [1, 2, 7]. The focus of a programmer's attention can provide supportive evidence of the cognitive processes when reading an algorithm [23]. Eye movement is described as saccades and fixations [16]. A saccade is a rapid movement of the eye when focus shifts between areas. A fixation refers to the eye resting on a specific area after a saccade. An eye tracker follows the eye around during its saccades and tracks the location of the fixation points. Especially gaze fixation and fixation duration can be used when testing cognitive progresses [14].

During a fixation a person extracts visual information that needs to be processed [22]. The fixation duration on a specific line of code is an indication of how long it takes a programmer to process that line [29]. The amount of time it takes programmers to focus on an area relevant to the program comprehension task is a demonstration of their understanding of the code. A long time spent reading irrelevant sections of code can be an indication that the programmer first has to scan the code to identify candidate lines relevant to the task they have to perform on the code [29]. Fixation duration on specific lines and time to first fixation on an area of interest can therefore be used to assess students' understanding of an algorithm.

3. A BEACON RECOGNITION TOOL

In 2005 an experimental tool, a **Be**acon **Re**cognition **T**ool (BeReT) [12, 13] was created to support plan discovery in introductory algorithms. The primary purpose of the tool is to encourage a novice programmer to correctly identify beacons within syntactically and semantically correct algorithms. BeReT is an attempt to aid novice programmers while learning introductory algorithms so that they develop techniques they can use in future program comprehension activities. Support in the form of beacon recognition and chunking is provided in BeReT. The two main functional components in BeReT are tutorials used for training, and exercises used for reinforcement and assessment.



This sorts using Bubble Sort

This is the bubble sort algorithm. It uses swap alot.

```
                                        swap(index,index-1);
                                }
                                numSorted++;
                        }
                }

                private void swap(int indexA, int indexB)
                {
                        [Swap]  (Line 1 of 3)
                        numbers[indexA] = numbers[indexB];
                        [Swap]  (Line 3 of 3)
                }

}
```

Username: ChRoNoTeRm (Super User)

Figure 3 Example of a Tutorial in BeReT [12]

BeReT was specifically designed to meet requirements 1 to 5 listed in Table 1. Lecturers can create a tutorial in BeReT which includes a specific algorithm and highlights plans associated with this algorithm (Requirement 1). The tutorial screen (Figure 3) is divided into four sections. The "Program Description" area is used to provide an overview of the function of the algorithm. The "Explanation of Example" area is used to list the plans associated with the algorithm. A system feedback area (top right hand corner) shows the progress and number of tutorials available to the students as well as navigation buttons. The bulk of the screen consists of the algorithm loaded by the lecturer. Tutorials make use of chunking to map higher level semantic plans in the algorithm to grouped lines of code (Requirement 2). Tutorials can also be created to focus specifically on those lines of code that can be seen as beacons in the code (Requirement 3). When setting up the tutorials and exercises in BeReT, the lecturer must ensure that standard programming conventions are used and that the algorithms used in the tutorials and exercises are syntactically and semantically correct (Requirement 4). Algorithms loaded in BeReT can be written in any programming language, since BeReT makes use of a database to store the code. The advantage of using a database is that BeReT can therefore support any future change in programming language in CS1. Algorithms loaded in BeReT can be as simple and complex as the lecturer requires. Three types of exercises provide interaction in BeReT; "complete", "match" and "define" type exercises (Requirement 5). In a "complete" type exercise students are given a beacon name and description, and they must fill in the missing lines of code described by the name and description. In "match" type exercise students are also given a beacon name and description and they are required to select relevant source code lines that match name and description. In "define" type exercises students are given the code only and are instructed to identify lines of code that seem good candidates to classify as beacons. To accomplish this, students group statements together and assign a name and description to indicate the semantic plan of these statements.

4. EMPIRICAL STUDY

An empirical study was conducted during a three week period in the second semester of 2006 to determine to what extent BeReT meets requirements 6 and 7 listed in Table 1 [18]. In order to meet these two requirements, BeReT must:

- Increase their ability to recall the semantics of an algorithm.
- Enable students to successfully transfer knowledge.

Participants in the study were CS1 students in the Department of Computer Science and Information Systems (CS&IS) at the Nelson Mandela Metropolitan University (NMMU) (Section 4.1). The two requirements were broken into research questions to guide the empirical study (Section 4.2). The empirical study followed acknowledged procedures to collect evidence in support of requirements 6 and 7 (Section 4.3).

4.1 Participant Population and Sample Size

A total of 64 first year students participated in a between-groups study. Participants were divided into two groups, with 32 students in a control and 32 students in a treatment group. The main difference between the two groups is that the treatment group made use of class notes and had exposure to the algorithms in BeReT, while the control group only relied on the class notes, which included program comprehension questions. Both groups had the same number of program generation problems. The data collected in the empirical study is thus an accurate reflection of the impact made by BeReT on students in an introductory programming course in the Department of CS&IS at NMMU in respect of requirements 6 and 7 in Table 1.

4.2 Research Questions

To test for requirement 6 there is a need to test for comprehension of the semantics of the algorithms studied. Evidence of rote learning would indicate learning on a superficial level. Rote learning an algorithm would result in a student regurgitating code line by line while disregarding changes in the problem scenario. Recalling an algorithm and making the necessary changes to suit the difference in problem scenario is evidence of that the student has grasped the semantics of the algorithm when it was studied. The research questions that were used to guide the empirical study to find support for requirement 6 are:

RQ6.1: Do students that used BeReT perform better in terms of accuracy on code comprehension questions?

RQ6.2: Are students that used BeReT more successful in recalling the entire algorithm and adapting to any changes in problem scenario?

RQ6.3: Are students that used BeReT more accurate when completing a partial algorithm?

BeReT would be deemed successful with respect to requirement 7 if students display evidence of using existing plan knowledge in future code comprehension activities of previously unseen algorithms. It was therefore necessary to determine if BeReT managed to build a repository of plans in the long-term memory of students in the treatment group. A bank of plans at students' disposal would enable them to discover plans in similar algorithms and re-use these plans during code generation exercises. BeReT will also be deemed successful in this requirement if students exhibit a tendency to continue with the techniques of beacon recognition and chunking when studying a previously unseen algorithm. The following research questions were used to guide the empirical study to determine the level of support for requirement 7:

RQ7.1: Are students that used BeReT more successful in the discovery of known plans in previously unseen algorithms?

RQ7.2: Do students that used BeReT find plans in a known and/or previously unseen algorithm quicker?

RQ7.3: Do students that used BeReT exhibit evidence of using beacon recognition and chunking when studying a previously unseen algorithm?

4.3 Procedure of empirical study

During the empirical study, students from the treatment and control group attended the same lectures. The concepts beacons and chunking were not explicitly discussed in lectures. For practical assignments they were split into their respective groups in different laboratories. Strict attendance monitoring was applied to ensure that the groups did not mix. By separating students during practical assignments it was ensured that only the treatment group had exposure to BeReT. The separation effectively minimized the risk of distorting the results due to control group participants also seeing BeReT. Security is also built into BeReT so that only the treatment group participants have access to the BeReT tutorials. Tutorials were constructed from the class notes, which the control group participants also received during lectures. Exercises were constructed by adapting comprehension questions that formed part of the class notes. For the purpose of this empirical study, exercises in BeReT could only be done during the practical assignment. Once an exercise was completed, it was not available anymore. The practical assignments of the control group consisted of only code generation tasks, as has been the practice in previous years. However, the students in the control group were also encouraged to study the algorithms as discussed in class prior to attempting the practical. This required control group students to study the class notes and attempt to answer the comprehension questions. Student assistants and the lecturer were available for consultation should the control group students have any difficulty studying the class notes.

The treatment group performed exactly the same code generation tasks as the control group. Treatment group students had access to an adapted version of the class notes and the associated comprehension questions in BeReT. They studied a tutorial of the algorithm first, and then performed exercises in BeReT. The tutorials presented the code, a list of plans and a description of the plans behind selected lines in the code. When students clicked on a plan, specific lines of code that matched the description were highlighted in the algorithm. The exercises included tasks to match code to a plan and to complete missing lines. Three algorithms, namely the bubble sort, insertion sort and binary search were used in the study. One algorithm was covered per week.

During the subsequent lecture following any practical assignment, follow-up class tests were administered to monitor both groups' ability to recall, complete and comprehend the algorithm studied in the previous week. The recall and complete questions were structured in such a way so as to determine whether students used rote learning to study each algorithm, or whether they were sensitive to any changes in context. The comprehension questions determined students' ability to extract proper plans from each algorithm. The purpose of the class tests was to answer primarily the research questions RQ6.1, RQ6.2 and RQ6.3. Both groups participated in the class tests and data was used to indicate trends in the performance of the two groups. All class tests were marked by the same assessor to ensure consistency in final results was achieved.

Two controlled experiments were conducted a week after the final class test was written. A representative sample from each group participated in the controlled experiments. The experiments were performed in a controlled environment under strict conditions using an eye tracker in both experiments to collect additional data:

- Experiment A involved an unseen algorithm (Selection Sort).
- Experiment B involved a known algorithm (Insertion Sort).

The purpose of the controlled experiments was to answer primarily the research questions RQ7.1, RQ7.2 and RQ7.3. When using fixation duration to compare two students reading the same algorithm, the longer fixation would indicate that a student takes longer to determine the plan behind that line of code. When answering comprehension questions about an algorithm, a student that spends a long time reading irrelevant lines of code is an indication that the student still has to read the algorithm to determine the plans behind the lines. A student that has built up a hierarchy of plans would discover the plan quicker in a known and/or previously unseen algorithm.

Eye-tracking data was collected during the controlled experiment using a remote eye-tracking device. IViewX software was used to calibrate the student and record fixations and saccades while finding answers to comprehension questions from an algorithm displayed on screen. BeGaze was used for a fixation analysis of all participants in the study. Scanpaths and heat maps were produced in BeGaze and are presented in the next section.

5. RESULTS OF STUDY

Three class tests were administered over the period of the empirical study. Each class test evaluated the ability of students to recall, comprehend and complete the algorithm studied the week before. Results for the class tests for each algorithm used in this study are presented in Table 2.

In the case of all three class tests, the results for the recall and comprehension questions between the control and treatment groups show a clear improvement in the observed means for all questions. The improvement in performance therefore provided evidence of positive responses to RQ6.1, RQ6.2 and RQ6.3. The improved performance therefore suggests evidence in support of requirement 6.

Table 2 Average marks obtained in Class Tests

		Bubble Sort		Binary Search		Insertion Sort	
		Recall	Comprehend	Recall	Comprehend	Complete	Comprehend
Treatment (n = 32)	Mean	56%	41%	61%	56%	43%	55%
	St Dev	23%	27%	31%	17%	20%	19%
Control (n = 32)	Mean	44%	22%	42%	39%	27%	35%
	St Dev	29%	22%	20%	20%	30%	45%

Experiment A of the controlled experiments tests comprehension and interaction of a previously unseen algorithm (the selection sort algorithm). The chosen algorithm uses similar plans used by algorithms that participants learnt in the introductory programming algorithms course. For the first task of this experiment participants had to study the selection sort algorithm for a maximum of 10 minutes using any technique they feel comfortable with. Students made notes while studying the algorithm, and notes were analyzed to determine the techniques used by them when studying an unseen algorithm. It is interesting to note however that all the students, regardless of the group used some sort of tracing technique to determine the function of the algorithm. This is an indication that students that used BeReT failed to exhibit evidence of using chunking or beacon recognition when studying a previously unseen algorithm. There is therefore no evidence in support of RQ7.3 in this regard. It seems therefore that the three weeks of exposure to BeReT was not enough to promote the use of chunking and beacon recognition when reading a previously unseen algorithm.

Immediately after studying the unseen algorithm, participants had to explain in their own words what the algorithm does. Table 3 summarizes the number of participants from each group that either gave high-level (semantic) explanations, explanations on syntactic level or incorrectly identified the goal of the selection sort algorithm after studying it for the first time. The table shows no clear trend in favour of any group. However, it is promising to see slightly more treatment group participants giving high-level semantic explanations instead of low level syntactic explanations of the algorithm.

Table 3 Number of participants identifying the goal of the selection sort algorithm

	Treatment (n = 7)	Control (n = 7)
High level explanations	3	1
Low level explanations	3	4
Incorrect explanations	1	2

For the second task of controlled experiment A, participants had to sit in front of a PC connected to a remote eye tracker. Students were first given an opportunity to scan the algorithm on screen again, this time with the instruction to become comfortable reading from the screen and to also try to recognise plans used in algorithms studied before. As soon as the students indicated they were ready to proceed, they had to answer a series of five comprehension questions about specific lines or groups of lines in the program displayed on-screen. Participants only had to verbally provide the line number(s) that contained the answer. As soon as an answer was given, the code was hidden from view. The code was shown again after the next question was asked. Responses and time taken were recorded by the observer. Eye tracking data was recorded only from the time the code was shown to the time an answer was given.

A summary of the proportion of correct responses for all five questions in controlled experiment A appears in Table 4. The average time it took each group to answer appears in Table 5.

Table 4 Proportion of correct responses per question (Experiment A)

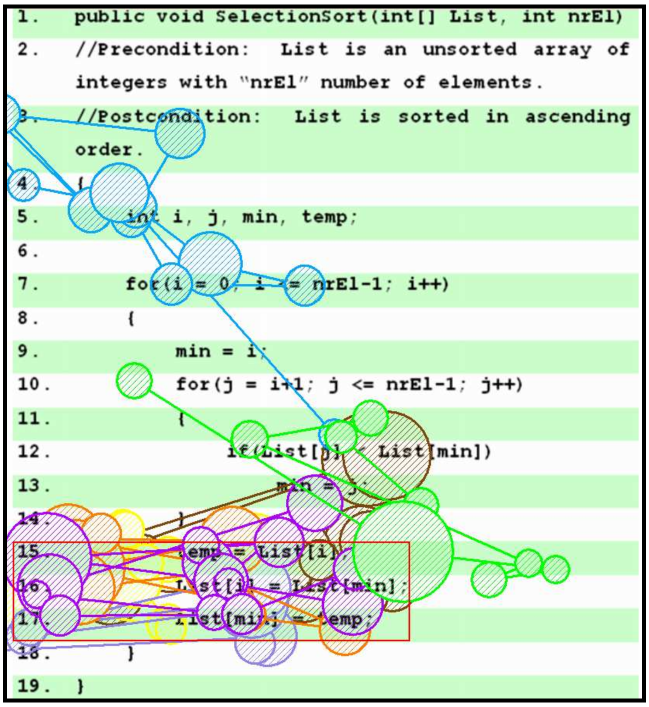
	Q1	Q2	Q3	Q4	Q5
Treatment (n = 7)	**100%** **(n = 7)**	71% (n = 5)	71% (n = 5)	100% (n = 7)	57% (n = 4)
Control (n = 7)	**14%** **(n = 1)**	43% (n = 3)	43% (n = 3)	86% (n = 6)	14% (n = 1)

Table 5 Time (in seconds) per question (Experiment A)

		Mean	Min	Max	Median
Treatment	**Q1**	**4**	**3**	**6**	**4**
	Q2	15	6	36	11
	Q3	9	4	16	7
	Q4	8	2	17	7
	Q5	12	3	34	4
Control	**Q1**	**12**	**4**	**25**	**8**
	Q2	12	3	33	10
	Q3	14	8	26	12
	Q4	12	3	26	13
	Q5	23	7	41	22

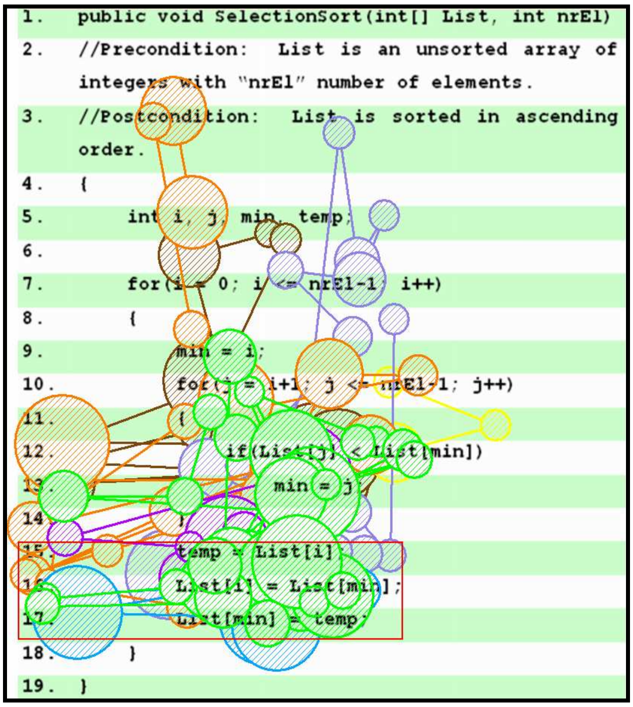
A breakdown of **accuracy** and **times** for all five questions showed that only one question demonstrated a clear difference in results between the two groups. It is worth looking at the eye-tracking data to find reasons for this difference. Scanpaths from the gaze analysis of the question to determine the line or lines that swaps two elements in an array revealed the paths participants took to find the answer for this question. The scanpaths resulting from the eye tracking data for the participants in the treatment group appear in Figure 4 and the scanpaths of the control group participants appear in Figure 5. From the two images it is clear that the treatment group follows a more concentrated path around the area of interest containing the answer to this question (Lines 15 – 17). Only two participants deviated significantly from the area containing the answer. The control group scanpaths reveal five participants scanning areas far away from the area of interest. It can be seen in Table 4 that all treatment group participants gave the correct answer, while only one of the control group participants answered correctly.

The scanpaths indicate that most of the control group participants did read the correct code, but decided that another line was the correct answer. This is an indication that the treatment group participants were more successful in transferring their knowledge of the "swap" plan from the bubble sort algorithm and recognise it in the previously unseen selection sort algorithm. This shows that in at least one question there is evidence to support RQ7.1.



```
1.   public void SelectionSort(int[] List, int nrEl)
2.   //Precondition:  List is an unsorted array of
     integers with "nrEl" number of elements.
3.   //Postcondition:  List is sorted in ascending
     order.
4.   {
5.       int i, j, min, temp;
6.
7.       for(i = 0; i <= nrEl-1; i++)
8.       {
9.           min = i;
10.          for(j = i+1; j <= nrEl-1; j++)
11.          {
12.              if(List[j] < List[min])
13.                  min = j;
14.          }
15.          temp = List[i];
16.          List[i] = List[min];
17.          List[min] = temp;
18.      }
19.  }
```

Figure 4 Treatment group scanpaths



```
1.   public void SelectionSort(int[] List, int nrEl)
2.   //Precondition:  List is an unsorted array of
     integers with "nrEl" number of elements.
3.   //Postcondition:  List is sorted in ascending
     order.
4.   {
5.       int i, j, min, temp;
6.
7.       for(i = 0; i <= nrEl-1; i++)
8.       {
9.           min = i;
10.          for(j = i+1; j <= nrEl-1; j++)
11.          {
12.              if(List[j] < List[min])
13.                  min = j;
14.          }
15.          temp = List[i];
16.          List[i] = List[min];
17.          List[min] = temp;
18.      }
19.  }
```

Figure 5 Control group scanpaths

Controlled experiment B featured one of the algorithms previously studied in class (the insertion sort algorithm). The process followed for the first task of this experiment is the same as for the second task of experiment A. Students first scanned the algorithm on the computer screen, and then answered a series of five questions. Apart from recording the accuracy and response time, eye-tracking data was also collected in this experiment. Table 6 summarizes the proportion of correct answers for the five questions in experiment B, and Table 7 the average time taken before giving the answer. Both tables show trends in favour of the treatment group, which suggests some support for RQ7.1 and RQ7.2.

Table 6 Proportion of correct responses per question (Experiment B)

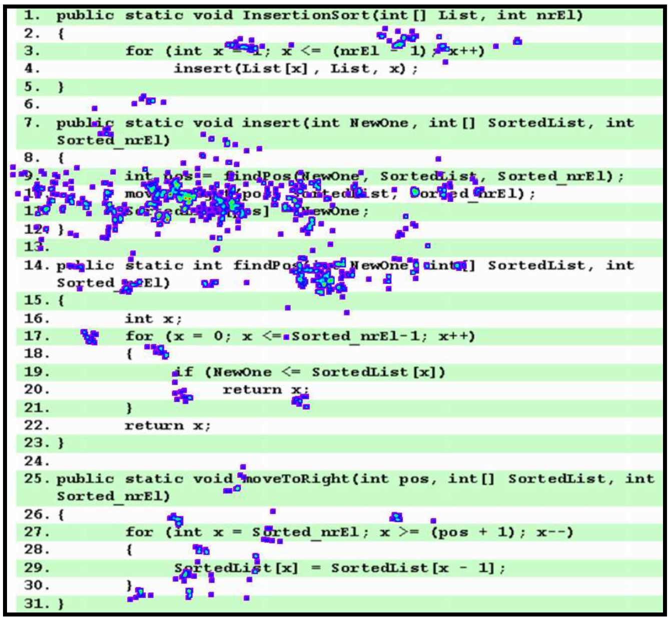
	Q1	Q2	Q3	Q4	Q5
Treatment	**100%**	14%	71%	71%	43%
Control	**100%**	43%	71%	43%	29%

It can be seen from Table 7 that one question (Q1) in particular showed a clear difference in response **times** between the groups. Table 6 shows 100% **accuracy** for both groups for Q1. This question relates to the plan to make space for a new element in an array. Since the question was constructed in such a way to elicit the function call that invokes the plan, the correct answer can be found in line 10 of Figure 6 and Figure 7.

Table 7 Time (in seconds) per question (Experiment B)

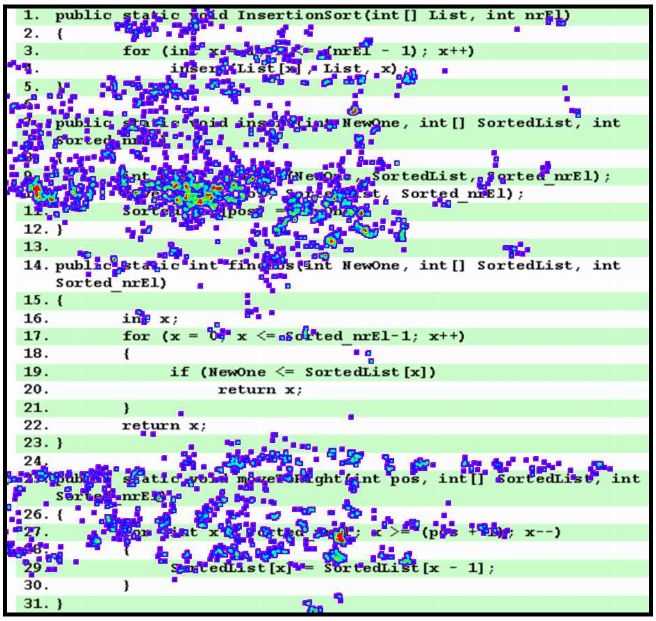
		Mean	Min	Max	Median
Treatment	**Q1**	**4**	**1**	**8**	**4**
	Q2	17	3	35	10
	Q3	11	4	28	8
	Q4	13	2	20	12
	Q5	19	6	43	11
Control	**Q1**	**21**	**3**	**55**	**20**
	Q2	18	5	37	18
	Q3	20	3	34	21
	Q4	19	7	40	14
	Q5	34	11	56	33

The heat map in Figure 6 that resulted from this question clearly shows that the treatment group participants focus mostly on the function calls. The control group on the other hand spends a large amount of time reading the code in the “moveToRight” function as well (Figure 7). These figures seem to indicate that the control group of students did not view the function call “moveToRight” as a comment beacon that indicates the plan of making space for a new element in an array. They still had to read the code inside the function to figure out what it does.



```
1.  public static void InsertionSort(int[] List, int nrEl)
2.  {
3.           for (int x = 0; x <= (nrEl - 1); x++)
4.                  insert(List[x], List, x);
5.  }
6.
7.  public static void insert(int NewOne, int[] SortedList, int
    Sorted_nrEl)
8.  {
9.           int pos = findPos(NewOne, SortedList, Sorted_nrEl);
10.          moveToRight(pos, SortedList, Sorted_nrEl);
11.          SortedList[pos] = NewOne;
12. }
13.
14. public static int findPos(int NewOne, int[] SortedList, int
    Sorted_nrEl)
15. {
16.          int x;
17.          for (x = 0; x <= Sorted_nrEl-1; x++)
18.          {
19.               if (NewOne <= SortedList[x])
20.                     return x;
21.          }
22.          return x;
23. }
24.
25. public static void moveToRight(int pos, int[] SortedList, int
    Sorted_nrEl)
26. {
27.          for (int x = Sorted_nrEl; x >= (pos + 1); x--)
28.          {
29.                 SortedList[x] = SortedList[x - 1];
30.          }
31. }
```

Figure 6 Treatment group heatmap



```
1.  public static void InsertionSort(int[] List, int nrEl)
2.  {
3.           for (int x = 0; x <= (nrEl - 1); x++)
4.                  insert(List[x], List, x);
5.  }
6.
7.  public static void insert(int NewOne, int[] SortedList, int
    Sorted_nrEl)
8.  {
9.           int pos = findPos(NewOne, SortedList, Sorted_nrEl);
10.          moveToRight(pos, SortedList, Sorted_nrEl);
11.          SortedList[pos] = NewOne;
12. }
13.
14. public static int findPos(int NewOne, int[] SortedList, int
    Sorted_nrEl)
15. {
16.          int x;
17.          for (x = 0; x <= Sorted_nrEl-1; x++)
18.          {
19.               if (NewOne <= SortedList[x])
20.                     return x;
21.          }
22.          return x;
23. }
24.
25. public static void moveToRight(int pos, int[] SortedList, int
    Sorted_nrEl)
26. {
27.          for (int x = Sorted_nrEl; x >= (pos + 1); x--)
28.          {
29.                 SortedList[x] = SortedList[x - 1];
30.          }
31. }
```

Figure 7 Control group heatmap

6. CONCLUSIONS

To improve the comprehension of introductory algorithms taught in CS1, a list of requirements for program comprehension was used as basis for the development of an experimental tool (BeReT). These requirements are:

- Allow student to extract semantic information (plans) from worked example.
- Promotes the use of chunking to learn an unfamiliar algorithm.
- Coaches students to spot beacons in different algorithms.
- Provide students with syntactically and semantically correct algorithms.
- Engage students in active learning activities.

An empirical study was conducted to determine whether the following requirements were met in BeReT:

- Increases ability to recall the semantics of an algorithm.
- Enable students to successfully transfer knowledge.

Research questions were used to guide the empirical investigation to collect evidence in support of these two requirements. Data collected by means of class tests and controlled experiments were specifically used to find answers to these research questions. The result of the data collected in the class tests indicates an improvement in performance of students using BeReT during the treatment period. BeReT therefore completely satisfies the requirement that it should increase the ability of a student to recall semantics of an algorithm. The controlled experiments provided only partial evidence in support of requirement 7, being students' ability to transfer knowledge. Treatment group participants displayed the existence of some plan and beacon knowledge readily available, but no participant in this group exhibited evidence of using chunking or beacon recognition when studying the unfamiliar code for the first time. These results contribute evidence that a program comprehension tool such as BeReT can support CS1 students in their program comprehension activities.

7. RECOMMENDATIONS AND FUTURE WORK

Despite one of the requirements being satisfied only partially there is sufficient evidence in favour of BeReT to continue further investigation into incorporating it as a support tool in a CS1 course to improve students' comprehension of introductory algorithms. A rigorous hypothesis testing methodology should follow to determine if the results show significant statistical support in favour of BeReT users. Further controlled experiments are required with bigger sample sizes to confirm results for requirement 7.

During the treatment period only two of the three possible exercises were used in BeReT. A follow-up study is required which uses the "Create a beacon" type exercise to determine if this will have an effect on students' motivation to use beacon recognition and chunking in future program comprehension activities. Future work thus also includes a more rigorous analysis of the introductory programming algorithms to determine which lines of code can be classified as beacons. This can be done by using similar methods used by Aschwanden [1] and Uwano [29].

The alternative tutoring approach used by the control group in this experiment can be described as pen and paper exercises. Future experiments can be conducted to compare the effect of BeReT against other approaches.

8. ACKNOWLEDGMENTS

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 authors and therefore the NRF does not accept any liability in regard thereto.

9. REFERENCES

[1] Aschwanden, C. and Crosby, M., Code Scanning Patterns in Program Comprehension. in *Proceedings of the 39th Hawaii International Conference on System Sciences*, (Kauai, Hawaii, 2006).

[2] Bednarik, R. and Tukiainen, M., An eye-tracking methodology for characterizing program comprehension processes. in *Proceedings of the 2006 symposium on Eye tracking research & applications*, (San Diego, California, 2006), ACM Press, 125 - 132.

[3] Bradley, C. and Boyle, T. Students' use of learning objects. *Interactive Multimedia Electronic Journal of Computer-Enhanced Learning, 6* (2). 2004.

[4] Brooks, R. Towards a theory of the comprehension of computer programs. *International Journal of Man-Machine Studies, 18* (6). 1983. 543 - 554.

[5] Cilliers, C.B., Calitz, A.P. and Greyling, J.H., The effect of integrating an iconic programming notation into CS1. in *Proceedings of the 10th annual SIGCSE conference on Innovation and technology in computer science education*, (Caparica, Portugal, 2005), ACM Press, 108 - 112.

[6] Crosby, M.E., Scholtz, J. and Wiedenbeck, S., The roles beacons play in comprehension for novice and expert programmers. in *Proceedings of PPIG*, (Brunel University, UK, 2002), 58 - 73.

[7] Crosby, M.E. and Stelovsky, J. How Do We Read Algorithms? A Case Study. *IEEE Computer, 23* (1). 1990. 24 - 35.

[8] Deimel, L. and Naveda, J. Reading Computer Programs: Instructor's Guide and Exercises *Educational Materials CMU/SEI-90-EM-3*, Pennsylvania, 1990.

[9] Garner, S., The learning of plans in programming: A program completion approach. in *Proceedings of the International Conference on Computers in Education*, (2002), 1053 - 1057.

[10] Garner, S., Learning Resources and Tools to Aid Novices Learn Programming. in *Informing Science & Information Technology Education Joint Conference (INSITE)*, (Pori, Finland, 2003), 213 - 222.

[11] Garner, S., Haden, P. and Robins, A., My program is correct but it doesn't run: a preliminary investigation of novice programmers' problems. in *Proceedings of the 7th Australasian conference on Computing education*,

(Newcastle, New South Wales, Australia, 2005), Australian Computer Society, Inc., 173 - 180.

[12] Harris, N. *Beacon Recognition Tool* Honours Treatise, University of Port Elizabeth, Port Elizabeth, 2005

[13] Harris, N. and Cilliers, C., A Program Beacon Recognition Tool. in *7th International Conference on Information Technology Based Higher Education and Training, 2006. (ITHET '06)*, (Ultimo, New South Wales, Australia, 2006), 216 - 225.

[14] Hornhof, A.J. and Halverson, T. Cleaning up systematic error in eye tracking data by using required fixation locations. *Behavior Research Methods, Instruments, and Computers*, *34* (4). 2002. 596 - 604.

[15] Jenkins, T. *On the difficulty of Learning to Program* Available at http://www.ics.heacademy.ac.uk/Events/conf2002/jenkins.html, 2002. (Last accessed - 2005)

[16] Karn, K.S. and Jacob, R.J.K., Eye tracking in human-computer interaction and usability research: Ready to deliver the promises. in *The mind's eye, cognitive and Applied Aspects of Eye movement Research*, (Oxford, 2003), Elsevier Science.

[17] Kimura, T., Reading before Composition. in *Proceedings of the tenth SIGCSE technical symposium on Computer science education*, (1979), ACM Press, 162 - 166.

[18] Leppan, R.G. *The impact of a plan discovery tool on in-depth introductory algorithm comprehension* Unpublished Masters Dissertation, NMMU, Port Elizabeth, 2007

[19] Letovsky, S. and Soloway, E. Delocalized plans and program comprehension. *IEEE Software*, *19* (3). 1986. 41 - 48.

[20] Linn, M.C. and Clancy, M.J. The Case for Case Studies of Programming Problems. *Communications of ACM*, *35* (3). 1992. 121 - 132.

[21] Lister, R., Adams, E.S., Fitzgerald, S., Fone, W., Hamer, J., Lindholm, M., McCartney, R., Moström, J.E., Sanders, K., Seppälä, O., Simon, B. and Thomas, L. A multi-national study of reading and tracing skills in novice programmers. *SIGCSE BULLETIN*, *36* (4). 2004. 119 - 150.

[22] Liversedge, S.P. and Findlay, J.M. Saccadic eye movements and cognition. *Trends in Cognitive Sciences*, *4* (4). 2000. 6 - 14.

[23] Nevalainen, S. and Sajaniemi, J., Comparison of Three Eye Tracking Devices in Psychology of Programming Research. in *Proceedings of the 16th Annual Workshop of the Psychology of Programming Interest Group*, (Carlow, Ireland, 2004), 151-158.

[24] O'Brien, M.P. *Software Comprehension – A Review & Research Direction* Technical Report UL-CSIS-03-3 2003.

[25] Pennington, N. Comprehension strategies in programming. in *Empirical studies of programmers: Second Workshop*, Ablex Publishing Corp., 1987, 100 - 113.

[26] Sajaniemi, J. and Hu, C., Teaching Programming: Going beyond "Objects First". in *Proceeding of 18th Workshop of the Psychology of Programming Interest Group*, (University of Sussex, 2006), 255 - 265.

[27] Soloway, E., Adelson, B. and Ehrlich, K. *Knowledge and Processes in the Comprehension of Computer Programs*. A. Lawrence Erlbaum Associates, Hillsdale, N.J., 1988.

[28] Soloway, E.M. and Woolf, B., Problems, plans, and programs. in *Proceedings of the eleventh SIGCSE technical symposium on Computer science education*, (Kansas City, Missouri, United States, 1980), ACM Press, 16 - 24.

[29] Uwano, H., Nakamura, M., Monden, A. and Matsumoto, K.-i., Analyzing Individual Performance of Source Code Review Using Reviewers' Eye Movement. in *Proceedings of the 2006 Symposium on Eye tracking research and applications*, (San Diego, California, 2006), ACM Press, 133 - 140.

[30] Von Mayrhauser, A. and Vans, A.M. *Program Understanding: A Survey* Technical Report CS-94-120 1994.

[31] Wiedenbeck, S. and Evans, N.J. Beacons in Program Comprehension. *SIGCHI Bulletin*, *18* (2). 1986. 56 - 57.

[32] Wiedenbeck, S. and Scholtz, J. Beacons and Initial Program Comprehension. *SIGCHI Bulletin*, *21* (1). 1989. 90 - 91.

[33] Winslow, L.E. Programming Pedagogy -- A Psychological Overview. *SIGCSE BULLETIN*, *28* (3). 1996. 17-22.

[34] Yeh, C.L., Greyling, J.H. and Cilliers, C.B., A framework proposal for algorithm animation systems. in *Proceedings of the 2006 annual research conference of the South African institute of computer scientists and information technologists*, (Somerset West, South Africa, 2006), South African Institute for Computer Scientists and Information Technologists, 155 - 163.