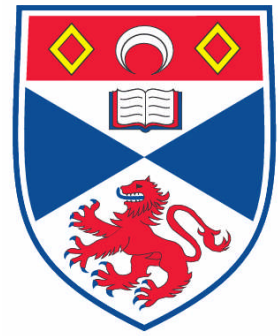
SNOOPIE : DEVELOPMENT OF A LEARNING SUPPORT TOOL FOR NOVICE PROGRAMMERS WITHIN A CONCEPTUAL FRAMEWORK

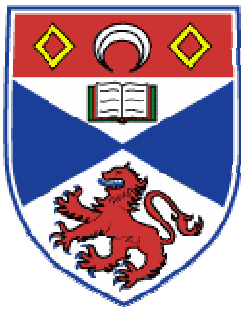
Natalie J. Coull

A Thesis Submitted for the Degree of PhD
at the
University of St. Andrews



2008

SNOOPIE: Development of a Learning Support Tool for Novice Programmers within a Conceptual Framework



A thesis to be submitted to the
UNIVERSITY OF ST ANDREWS
for the degree of
DOCTOR OF PHILOSOPHY

by

Natalie J. Coull

School of Computer Science
University of St Andrews
February 2008

Author's Declaration

I, Natalie J. Coull hereby certify that this thesis, which is approximately 68000 words in length, has been written by me, that it is the record of work carried out by me, and that it has not been submitted in any previous application for a higher degree.

*date*_____*signature of candidate*_____

I was admitted as a research student in October 2002 and as a candidate for the degree of Doctor of Philosophy; the higher study for which this is a record was carried out in the University of St Andrews between 2002 and 2006.

*date*_____*signature of candidate*_____

I hereby certify that the candidate has fulfilled the conditions of the Resolution and Regulations appropriate for the degree of Doctor of Philosophy in the University of St Andrews and that the candidate is qualified to submit this thesis in application for that degree

*date*_____*signature of supervisor*_____

In submitting this thesis to the University of St Andrews I understand that I am giving permission for it to be made available for use in accordance with the regulations of the University Library for the time being in force, subject to any copyright vested in the work not being affected thereby. I also understand that the title and abstract will be published, and that a copy of the work may be made and supplied to any bona fide library or research worker.

*date*_____*signature of candidate*_____

Abstract

Learning to program is recognised nationally and internationally as a complex task that novices find challenging. There exist many endeavours to support the novice in this activity, including software tools that aim to provide a more supportive environment than that provided by standard software facilities, together with schemes that reduce the underlying complexity of programming by providing accessible micro-worlds in which students develop program code. Existing literature recognises that learning to program is difficult because of the need to learn the rules and operation of the language (program formulation), and the concurrent need to interpret problems and recognise the required components for that problem (problem formulation). This thesis describes a new form of learning support that addresses that dual task of program and problem formulation. A review of existing teaching tools that support the novice programmer leads to a set of requirements for a support tool that encompasses the processes of both program and problem formulation. This set of requirements is encapsulated in a conceptual framework for software tool development. The framework demonstrates how the requirements of a support tool can be met by performing a series of automated analyses at different stages in the student's development of a solution. An extended series of observations demonstrates the multi-faceted nature of problems that students encounter whilst they are learning to program and how these problems can be mapped onto the different levels of programs and problem formulation. These observations and the framework were used to inform the development of SNOOPIE, a sample instantiation of the framework for learning Java programming. This software tool has been fully evaluated and demonstrated to have a significant impact on the learning process for novice Java programmers. SNOOPIE is fully integrated into a current introductory programming module and a future programme of work is being established that will see SNOOPIE integrated with other established software tools.

Table of Contents

List of Figures

List of Tables

Chapter 1 Introduction

1.1 Background

1.1.1 The Student's Experience of Learning to Program

Novice programmers, and specifically first year university students in the context of this thesis, are faced with a number of challenges that impact on their experience of learning to program. Many students have no previous experience of programming and those who have programmed at secondary school level have used languages such as True Basic, Comal or Visual Basic 6 but rarely a sophisticated language such as Java, Scheme or VB.NET. Java, for example, is the most commonly used language in Higher Education (Lund, G. 2006, pers. comm. 6th June). To use Java, first year students must familiarise themselves with both a new language and new software to support program development, typically in the form of an Integrated Development Environment (IDE), or use a command line interface. Indeed the first practical classes of the programming modules studied in this research are spent showing the students how to use the new environment, including learning how to create projects and link to external library files, and ensuring that they are able to open and save files to their network space. While this is a complex activity that students find difficult, be it an IDE or command line this is a transient issue (Kölling, 2003). Students face more substantial difficulties in learning to program than are associated with using the software, and the root of the difficulties lies in the multi-faceted nature of learning to program.

The clear aim of any introductory programming module is to teach a student to program, i.e. develop programmed solutions to meet a set of requirements. This is typically mediated through a series of lectures, which define syntax and semantics together with providing examples of construct usage, and a set of associated practical exercises that require the student to translate a specific problem expressed in English into an appropriate solution expressed in a programming language. To solve the problem, the student is required to engage with two concurrent aspects: the production of appropriate syntax to support the

solution and the articulation of the problem in terms of program design. For example, if a student is required to write a program to read in three numbers and print out an average they must be able to, in a general sense, declare variables, define arithmetic operations, make assignments and employ mechanisms for user input and program output. Additionally, these syntactic elements must be utilised in such a way that the required program behaviour is implemented: Variables should exist for each number, a total and the resulting average; those variables for user input are linked to user input statements; arithmetic operations make use of those variables to determine a total and an average; etc. The details of how those general syntactic elements are orchestrated to effect a solution are specific to the problem considered. A further challenge to the learning of programming is that students are expected to be able to assimilate effectively new syntax that relates to often only partial exposure to complex programming concepts, since novices lack a fully integrated view of the underlying paradigm of program development. For example, in Java when learning method invocation it is difficult for students to engage with the syntactic mechanisms of parameters, return types, return values and scope of variables, together with (concurrently) understanding the rationale for methods and the division of computation within a program into cohesive program blocks that constitute individual methods. The former relates to general use of the mechanism; the latter its exploitation in program design.

The two tasks of language detail and programming concepts are conceptually distinct but inherently coupled and so must be taught in parallel. Programming cannot be taught in terms of language rules alone, nor may it be taught by example alone. The number of closely linked cognitive factors involved in the actual task of program development compounds the difficulties associated with any one aspect of learning to program. From understanding the syntax to identifying an appropriate solution to solving a problem, problems in one cognitive area can have a cascade effect on the student’s ability to further their understanding of another area. For those students who have not programmed before, this can be particularly challenging.

1.1.2 The Tutor's Experience of Teaching Students to Program

The development of programming ability in students requires support in a taught context, for example a laboratory session, together with periods of self-study and exploration. In the taught environment, a tutor must face the challenge of delivering a syntactic rule-base and the semantics of single statements together with more abstract concepts relating to the integration of programming components such as variable declarations and loop constructs to implement a particular program behaviour. The aim is to develop a model in the mind of the student of how to develop programs and of program operation. Three operational aspects impede the establishment of this conceptual model during laboratory activity.

First, during practical programming classes, much of the tutor's time may be dedicated to solving trivial syntactic problems, especially at the beginning of the module when the students are not yet familiar with the syntax rules and compiler error messages or when new syntax is introduced – i.e. at the earliest stages of model formulation. The time spent on this activity with any individual student may be substantial. The time taken to assist an individual student, and therefore the time that any student must wait for help, is further extended since students tend to make mistakes similar to those of their peers at particular stages, and so at similar times, in a practical activity and tutors often find themselves solving the same problems for several individuals independently.

Second, some of the problems students encounter may involve deeper issues than are evident simply from the (typically syntax) errors that the student makes, and so the tutor is required to concurrently address multiple problems ranging from trivial syntactic issues to deeper problem solving difficulties. For example, a student that is unable to formulate the syntax for a for loop may generate an error with that line. However, the rest of the program code may reveal a lack of understanding as to the role of that for loop in the program and so the tutor must simultaneously provide guidance on the for loop syntax, the use of the for loop in that solution and the generalised role of for loops in programs. The resolution of a superficially syntactic error can thus take substantial time and multiple iterations.

Third, students can often feel so dependent on the tutor for helping them solve problems relating to their practical exercises that they are often reluctant to work on their programming outside of class time, obviating the benefits of self-study and reflection. Thus, the tutor can often be the only source of academic support. As a result, for some students, class contact is the only mechanism for conceptual model progression. This is such a widely recognised problem that several conferences and workshops focus attention on supporting the novice programmer. For example the recent Disciplinary Commons programme of research (Fincher et al., 2006) chose to focus on the teaching of introductory programming as a challenging study area for its review of teaching practice across a number of higher education institutes in the UK. Further, the number of international conferences and workshops that target the challenges posed by teaching novice programmers, such as the Higher Education Academy's annual workshop on Teaching Programming and the annual ICER conference, highlight the extent of interest in this area.

1.1.3 Problem Definition

Based on the challenges facing both student and tutor in developing programming expertise, three underlying obstacles emerge. First is the engagement with each of language rules and program operation. A combination of reference material, programming practice and the compiler all help the student learn the correct syntax for a given construct and resolve errors made in that syntax. However, exploitation of (static) reference material in the (dynamic) process of writing a program is challenging for students. Further, practice is impeded by frustrations with errors as noted previously. Additionally, the compiler is well recognised as an expert tool for expert users, and so inappropriate for novices. Understanding the operation of a statement such as a for loop or a variable declaration, i.e. its semantics, depends on the development of an operational model. This can be achieved only through trial and error or exploration of the relation between the syntax and how the syntax is executed, and this is impeded by the introduction of syntax errors (which are difficult to resolve).

Secondly the student must be able to determine an appropriate structure to a program in order to develop a solution to a problem. For example, to read in a series of numbers into an array

requires first an array definition, second a for loop to iterate through the array, and third an input statement within that for loop. Determination of an appropriate program structure depends on knowing how each individual programming construct may contribute to a solution to a given problem and in what order those constructs should be arranged. This in turn depends on the ability to determine the contribution of that component to the solution (semantics) and the successful integration of those components, which depends on their correct formulation (syntax). Consequently, the student may not develop solutions successfully without the underpinning syntactic and semantic knowledge. Importantly, this underlying knowledge of syntax and semantics is most strongly promoted by development of solutions to specific problems, yet a limited architectural understanding impedes this solution development.

Thirdly, the difficulty introduced by the need to concurrently engage across these cognitive/ conceptual levels to develop programs means that novice programmers are unwilling to learn on their own. This then means that, because of lack of practice on the part of the student, tutors devote too much time to the (often repetitive) correcting of syntax and this limits time available for the exploration of the generic semantic aspects and specific problem-based structural and design levels of programming for implementing solutions.

1.2 Thesis Statement

This thesis considers the development of programming expertise as two interrelated aspects of learning. For one aspect, hereafter referred to as program formulation, the novice must learn to formulate individual lines of code that are in accordance with language rules and perform an intended operation. For the other aspect, hereafter referred to as problem formulation, the novice must learn how to arrange these individual lines of code within the context of a larger program to form a solution to a prescribed problem.

The thesis presents SNOOPIE (**S**upporting **N**ovices in an **O**bject **O**riented **P**rogramming **I**ntegrated **E**nvironment), a program development learning support tool that supports students with both program and problem formulation. SNOOPIE is underpinned by, and is one

instantiation of, a novel conceptual framework that defines how holistic support across program and problem formulation may be developed. The framework is a synthesis of two bodies of existing literature. First, three current models of student learning are unified into a single model that defines four knowledge levels at which students encounter problems when learning to program, i.e. syntactic and semantic levels defining program formulation, and schematic and strategic levels defining problem formulation. Second, a wide range of existing learning support tools are reviewed and used to define nine requirements for holistic support of student learning in programming. The framework defines the component parts required in any program development tool that provides support consistent with both the four knowledge levels and the nine requirements.

SNOOPIE has been developed, deployed and evaluated within the classroom context over two consecutive academic sessions. SNOOPIE performs generalised checks on the student's program and provides advice and feedback on compiler errors and common semantic mistakes in student-friendly terminology. These generic checks implement the program formulation support in SNOOPIE. Additionally, SNOOPIE has knowledge of the programming exercises that the students are trying to complete and may then provide hints and tips on missing constructs or relationships between constructs for a given exercise. It is this exercise-specific support that effects the support for problem formulation.

Existing tools have supported program formulation or problem formulation alone, even though this is at odds with the existing literature on the knowledge levels associated with programming. This framework, and so SNOOPIE, is the first approach to integrate program and problem formulation into a holistic support tool. Consequently, this thesis demonstrates, through evaluation of SNOOPIE, that combined support for both program and problem formulation improves student performance over short and long timescales in developing programmed solutions to problems. To demonstrate this, four hypotheses are tested:

1. The program formulation support provided by SNOOPIE improves short-term student performance.

2. The problem formulation support provided by SNOOPIE improves short-term student performance.

3. The problem formulation support provided by SNOOPIE reduces the time taken to complete an ideal solution.

4. A combination of problem and program formulation improves long-term student performance when compared with program formulation alone.

The thesis demonstrates that support with program formulation alone improves short-term student performance (H1). The additional inclusion of problem formulation support is seen to further improve short-term student performance in terms of task completion (H2) but not time taken (H3). In fact the results demonstrate that problem formulation support allows more students to complete a given task than would have done so without that support, even though the mean time to completion increases. Finally, program and problem formulation combined is shown to improve long-term student performance relative to program formulation alone (H4). A fifth hypothesis is tested to determine the impact that SNOOPIE is perceived to have on the students' experience of learning:

5. Students perceive that SNOOPIE has a positive impact on the learning process.

The aim of SNOOPIE is to support students' learning from the outset and across an extended period of time. For any software product to achieve such initial and extended usage, it is important that the software tool maintains credibility with its user base. In this case, if the cohort of students recognises SNOOPIE as valuable from an early stage it may then become part of their common toolkit for software development. In contrast, if the view held by students is that SNOOPIE is of no use, there will be no opportunity to impact on the learning process through this tool. The thesis shows that students are aware that SNOOPIE can help them produce working solutions both in terms of program and problem formulation. This results in a positive reaction to SNOOPIE and a range of usage patterns for the majority of students over the period of SNOOPIE use.

The key contributions of this thesis are:

- A conceptual framework that integrates both the four knowledge levels required to learn to program and the nine requirements for holistic support;
- An implementation of that framework, SNOOPIE, that highlights the operational aspects of the tool, i.e. the analyses of student code undertaken, the feedback given and the way in which the knowledge associated with a question is encapsulated within SNOOPIE;
- An evaluation of that framework, via SNOOPIE, that demonstrates the impact of holistic support, i.e. program and problem formulation, on student performance.

1.3 Methodology

The conceptual architecture and nine requirements are informed by a review of the relevant literature. The review (Chapter 2) considers the nature of learning generally and the specific challenges posed by learning to program. Drawing on existing work, these challenges are encapsulated into the two distinct but interrelated activities of program formulation and problem formulation. Models of student learning are used to refine further the programming activities into four distinct levels of knowledge: syntactic, semantic, schematic and strategic, where syntactic and semantic contribute to program formulation and schematic and strategic to problem formulation.

In addition to this, the review appraises a number of existing learning support tools. These are categorised according to where they (most) contribute to learning in terms of the software phases of analysis, design, implementation and testing together with those tools that offer support in a more general sense. In this appraisal, key features of each approach are identified and good practice highlighted. The review of existing support tools is exploited to rationalise the list of nine requirements for a learning support tool (Chapter 3).

The components of the conceptual framework are defined (Chapter 3). The analyses necessary to identify the learning support required may be mapped directly on to one of program or problem formulation activities depending on the state of the program undergoing that analysis, and in the case of program formulation be further separated into syntactic and semantic levels of knowledge. However, the assumption that any feedback given to the student to support a given difficulty may also be mapped on to only one of program or problem formulation, depending on where that difficulty is manifest in the analysis, seems at odds with the recognised interrelations between program and problem formulation.

To explore the complex relationship between manifest problem and the underlying root-cause of that problem, an extended observational study was undertaken at both universities associated with this programme of research in the academic sessions 2002-2003 and 2003-2004 (Chapter 4). The study identifies a wide range of problems which students are unable to resolve without assistance and, with individual students, explores the underlying cause of the problem encountered and in turn identifies the knowledge needed to solve the problem. The observation reveals, as evidenced by a series of case study examples, that problems identified at one knowledge level may have a root cause at one or more different knowledge levels. In addition to recognising that multi-level feedback is required to support students, the dialogue observed by students in posing questions and staff in framing guidance is shown to make a valuable contribution to the form of the feedback offered by a support tool.

In line with the conceptual framework and feedback study, the SNOOPIE implementation of that framework (Chapter 5) is described in two phases: Version 1, which offers program formulation support, and Version 2, which offers problem formulation support. The range of support provided and its implementation and feedback given are detailed. For support tool evaluation, and following the observational studies of 2002-2003 and 2003-2004, Version 1 was used within the class context in the academic session 2004 -2005.

Version 2 incorporates the functionality of Version 1 and implements a new form of learning support to address the requirements for assisting in problem formulation. To support the student in their development of a solution to a problem, the current state of the program must

be assessed for both the presence of key constructs, e.g. a ‘for loop’ and an ‘if statement’, and the interrelations among those constructs, e.g. the ‘if statement’ is placed within the ‘for loop’. Further, as per the identified requirements, this assessment must be sensitive to both the context, i.e. the question, and the progress that the student has made with that question. This latter sensitivity determines the implementation strategy: support is effected in terms of a series of increasingly fine-scaled composition and structural tests that steer the student toward a solution. Where appropriate, the tool accommodates support for variant solutions to a given problem. The overall scheme is presented and indicative case study examples are worked through to support a detailed description, including one example that explicitly highlights the capacity of the approach to support multiple solutions to a given problem. For evaluation, Version 2 was used within the class context in the academic session 2005-2006.

Chapter 6 describes the evaluation of each version of SNOOPIE. The challenges associated with evaluating any learning support tool for introductory programming are discussed and, by drawing on established literature, a suite of evaluation mechanisms is identified. The evaluation of SNOOPIE using these mechanisms, including formal questionnaires, module results and experiments, is described independently. Additionally, the manner in which SNOOPIE meets each of the nine requirements is addressed. Shortcomings of the SNOOPIE implementation are also identified from this evaluation.

The concluding Chapter (7) considers the strengths and weaknesses of the work presented in the thesis, together with immediate extensions to the SNOOPIE implementation. New research and new collaborations are outlined, within the context of the appraisal of this programme of research and with a view to future directions.

Chapter 2 Literature Review

2.1 The Learning Process

Learning is "the acquisition of knowledge or skills through experience, practice, study or by being taught" (Hanks, 1998). The methods necessary for successful learning depend very much on the nature of the task. For a relatively simple cognitive task, learning may simply involve reviewing the steps often enough for them to pass through short-term memory into long-term memory (Ellington and Earl, 1996). However, Clark and Harrelson (2002) state that for learning of a more complex activity to occur the student must initially form a coherent idea of the concept being taught. This idea then has to be integrated into schemas (existing memories) stored in long-term memory. The student must then be able to review and practice this complex activity to reinforce its integration with existing schemas so that the student can successfully perform the activity at a later date.

Two recognised approaches are evident in the strategies employed to address the task of learning: surface and deep. Surface learning is simply the practice of storing new facts and ideas without any attempt to connect or process information. Ellington and Earl (1996) note that: "Students who adopt such a surface approach tend to work according to the following general pattern:

- Concentrating purely on assessment requirements
- Accepting information and ideas passively
- Memorising facts and procedures routinely
- Ignoring guiding principles or patterns
- Failing to reflect on underlying purpose or strategy."

Deep learning in contrast involves a distinct effort to engage with the material being learned, by critically examining each new fact or idea and linking them to existing knowledge. Ellington and Earl (1996) characterise deep learners as:

- "Attempting to understand material for themselves
- Interacting vigorously and critically with content
- Relating new ideas to previous knowledge and experience
- Using organisation principles to integrate ideas
- Relating evidence to conclusions."

The Higher Education Academy, Engineering Subject Centre (2000) state that students should be encouraged to adopt a deep approach to learning. This can be achieved by concentrating on key concepts and acknowledging students' misconceptions. Students need to be engaged in the learning process and able to relate new material to existing ideas. Students should also be allowed to make mistakes without penalty.

There are additional factors, which affect learning. Individual student learning will be affected to a greater or lesser extent by factors such as motivation, interest, age, maturity, background and confidence in personal ability (Jenkins, 2001). Many of these factors are intrinsically related and academic research has been conducted in the area of motivation, which is undoubtedly a key aspect of student learning and particularly so in self-study practices of first year university students (Watson et al., 2004). Academic motivation and the differences shown among students are a reflection of many influences, ranging from experiences of upbringing to success and failure of activities in previous learning environments. The learning approach chosen by the student will also depend on their personality and how this suits the chosen teaching method and style used by the tutor. In summary, learning any complex subject material requires regular and repeated engagement with that material and the activities comprising that engagement promote learning beyond the surface level.

2.2 Learning to Program

This treatment of learning is general and therefore true for all subject domains. The specific task of learning to program attracts particular challenges, and it is well recognised as a time-consuming and frustrating process for the majority of novice programmers (Johnson, 1990). Hereafter, in line with other work (e.g. Mayer, 1997), the term "novice programmers" is taken to mean those who are learning to program, and the term "expert programmers" is taken to mean those who can program.

The underlying difficulty in teaching programming has been acknowledged since the teaching of programming within mainstream higher education began. Almost 20 years ago,

Solway and Ehrlich (1989) explored the difference between novice programmers and experts. They state that the process of programming requires at least two different skill sets: program comprehension (i.e. the ability to understand each program part) and secondly plan recognition (i.e. the ability to understand the interactions among each program part). To demonstrate these two skill sets they introduce the notion of a program narrative, where well-structured programs in terms of constructs and naming conventions are easier to understand. In an observational study, both experts and novices were exposed to well structured and poorly structured programs and assessed on their program comprehension. In the case of well-structured programs, expert programmers performed significantly better than novices. In the case of poorly structured programs there was no significant difference between the performances of the two groups. Clearly program comprehension is dependent on an understanding of the form and meaning of each symbol and each line, and this skill is necessarily evident in experts. This study also demonstrates that understanding arises as a result of recognition of the interaction among program lines in the form of a systematic plan. This plan recognition is evidently present in experts but not in novices.

Mayer (1981) presents a useful analogy to programming in terms of playing chess. Citing Chase (1973), Mayer discusses a study where subjects were presented with chessboard configurations and then asked to reconstruct them from memory. Although chess masters performed much better than less experienced players when the chessboards came from real games, there was little difference when the chessboards contained randomly placed pieces. This demonstrates that the chess masters do not have better memories than the less experienced players, but an understanding of the patterns that can exist among pieces on the board. Mayer compares this to a study by Shneiderman (1980) where expert programmers were better able to recall lines of code than novices in a meaningful program but performed no better than novices when the program contained random lines. Mayer suggests that it could be useful to teach novices to program using “chunks” or “schemas” to help them learn the patterns evident in programming.

du Boulay (1986) undertakes a more thorough analysis, and determines five fundamental challenges in learning to program. One underlying challenge is “orientation” where students

must see the purpose of programming. This clearly relates to the importance of motivation in learning as above. Two challenges relate to the general process of developing programs: the "notional machine" and programming "pragmatics". The notional machine represents the abstraction of computer operations into a programming language, an editor and a compilation process. It was noted that students find difficulty in identifying the exact relationship between these three artefacts and the running program. Pragmatics relates to the broad lifecycle of program development, including testing and debugging. Two challenges are analogous to those presented by Soloway and Ehrlich (1989): "notation", relating to syntax and underlying semantics and "structures", relating to the development of plans to implement solutions.

Ebrahimi (1994) undertakes a review of existing work on the difficulties faced by novice programmers and clearly states that the two challenges noted by both studies above impact most on the learning process for novices. That knowledge of the syntax and underlying semantics is required to develop programs is clear. Ebrahimi states unambiguously that program plans are a fundamental requirement to successful program development. Ebrahimi further shows, through observational study, that the concepts of "language constructs" and "plan composition" are central to programmer development and intimately linked. Ebrahimi concludes that programmers cannot form plans without language and cannot use language without plans and that, consequently, these must be taught concurrently.

The above issues relate to the initial writing of programs and their subsequent debugging. As the writing of a program involves more than stringing together syntax, debugging is not simply removing syntactic errors; it is the process of 're-architecting' the entire program to solve the problem (Brna & Matheson 1993), and so requires knowledge of the individual lines, as per the language constructs of Ebrahimi, and the program architecture, as per Ebrahimi's plan composition. Research has been conducted (Vessey 1985, Kessler & Anderson, 1986) to suggest that both expert and novice programmers are able to formulate hypotheses about what has gone wrong in their program in terms of responding to syntax errors. Experts however, have a substantially better understanding of what a program is supposed to do and are therefore able to better resolve syntax errors in a program.

To address the problem of poor hypothesis formulation (relating to syntax error messages) by novice programmers, Chmiel and Loui (2004) conducted a study demonstrating that formal training in debugging (which they use to describe syntax error removal) helps students develop skills in diagnosing and removing defects from computer programs. The activities used to enhance the students' debugging skills included debugging exercises, logs and reflective memos. The debugging exercises involved identifying and solving errors in existing programs. All students were required to document each program development process in a log, including designs, plans and debugging experiences. Based on these logs each student was required to submit a reflective memo for each programming exercise stating time spent on the exercise and describing defects identified in their program. As a result of the study, the students agreed that the exercises had enhanced their debugging skills. In particular, they found that novice problem solvers fall victim to problems without making reference to the context, even though this context is simplistic initially. In contrast experts generate measurable criteria such as cause-effect relationships and procedures, which they use to make informed choices in order to solve problems. Further, Spohrer and Soloway (1986) observed that many novice programmers inject high-frequency defects into their programs. Debugging training, which focuses on those high frequency defects, could help students reduce the occurrences of these defects in their programs. While the details of the activity of debugging are different to those of programming, program and problem formulation present themselves in the activity of debugging. Clearly debugging requires exploitation of strategies to identify and address errors in individual lines of code, for example using simple output statements and program breaks to trace values. Additionally, debugging requires problem formulation level activities which potentially extend over the whole program and must take account of the properties of the specific problem being addressed, for example the construction of test sets and execution of those test sets.

The literature described above demonstrates that there are two challenges in learning to program. The first, here termed *program formulation,* is in essence the "notation" of Soloway and Erlich; the "language constructs" of Ebrahimi. It is the syntax and semantics of the individual program parts that are fundamental to the formulation of any program. The

second, here termed *problem formulation*, is the concept of "structures" from Soloway and Erlich and the "plan compositions" of Ebrahimi. Problem formulation is therefore the challenge of identifying and structuring the necessary program components to formulate a programmed solution to a specific problem.

Clearly, fundamental to the development of programming ability is the establishment of knowledge of the rules of program formulation and skills in problem formulation in long-term memory. The learning of program formulation may be effected through regular reviews of the available teaching material. Syntax and program operation, e.g. scope, the rules of a particular language may be learned by a surface learning approach, such as memory of the 'shape' of constructs. The development of problem formulation skills may come about only by a deep approach. Crucially, being able to recognise and reuse patterns depends on being able to integrate across different programs. Students then require distinct support for each program and problem formulation skill development, but this support must be integrated since, as noted by Ebrahimi (1994), concurrent teaching is required. Lahtinen et al. (2005) similarly comment that concept knowledge, i.e. program formulation, and strategies, i.e. problem formulation, must be combined but that they do represent both different aspects of programming and different levels of learning – surface and deep respectively.

2.3 Models of Student Learning

In order to improve the learning process, which underpins the development of expertise in programming, a number of models of student learning have been devised. These models are typically formulated in terms of stages in progressing from a novice to an expert programmer and can be used to identify and describe the composite aspects of the activity of learning to program. In turn, these activities, and the accordant skill base, can be used to determine the direction that we expect students will follow and provide guidance and structure on how we need to help students begin this process of learning.

McGill and Volet (1997) have produced a three level framework of programming knowledge (Table 2-1). This table arose from analysis and synthesis of work by educational computing and cognitive psychology groups which focus on the importance of knowledge that allows

recognition of meaningful patterns, access to relevant information and generation of appropriate solutions based on that knowledge. McGill and Volet categorise the knowledge required for programming into declarative and procedural, i.e. know what and know how.

Table 2-1 McGill and Volet (1997): A Conceptual Framework of the Various Components of Programming Knowledge

	Declarative Knowledge	Procedural Knowledge
Syntactic Knowledge	Knowledge of syntactic facts related to a particular language	Ability to apply rules of syntax when programming
Conceptual Knowledge	Understanding of and ability to explain the semantics of the actions that take place as a program executes	Ability to design solutions to programming problems
Strategic/Conditional Knowledge	Ability to design, code, and test a program to solve a novel problem	

Table 2-1 clearly identifies Syntactic knowledge as a separate, independent knowledge level that relates to a specific programming language. Conceptual knowledge makes reference to the student's understanding of operation of program statements and complete solutions to a problem. Strategic/ Conditional knowledge is the ability to recognise how a problem can be solved using the existing building blocks available from Syntactic and Conceptual knowledge. McGill and Volet conclude that students should be explicitly guided through the development of Conceptual and Strategic/ Conditional knowledge through teaching materials and add that this is particularly useful for low-ability students.

Mayer (1997) presents a similar framework (Table 2-2) exploring the types of knowledge involved in learning to program. The framework uses four stages to describe programming expertise and programming knowledge. This table arose from analysis and synthesis of earlier work that made clear a distinction between Syntactic and Semantic levels of knowledge (Shneiderman and Mayer, 1979) and the further distinction between knowledge of text (program) structure and development plans (Pennington, 1987). Mayer explores these knowledge levels in the novice programmers and concludes differences exist between

novices and experts across all knowledge levels and that the transition to expert arises only from extended experience. Regardless of the specific levels, Mayer, like McGill and Volet, goes on to highlight that teaching approaches should target directly each level to promote that transition.

Table 2-2 Mayer (1997): Four kinds of programming knowledge (examples are omitted here).

Knowledge	Definition	Common Tests
Syntactic	Language units and rules for combining language units	Recognise whether or not a line of code is correct
Semantic	Mental model of the major locations, objects and actions in the system	Rating pairs of terms for relatedness; providing thinking aloud protocol
Schematic	Categories of routines based on function	Recalling program code or keywords; sorting routines or problems, recognising or naming routines
Strategic	Techniques for devising and monitoring plans	Providing thinking aloud protocols; answering comprehension questions

The upper and lower knowledge levels are clearly defined and consistent for each of Mayer (1997) and McGill and Volet (1997). However, the middle layer of McGill and Volet blurs at the upper extent of the knowledge level. At the lower end, the distinction between syntax (rules for writing program statements) and semantics (operation of those statements) is clear. At the upper end, the Procedural-Conceptual level describes the exploitation of semantic knowledge to write programs, and the Strategic/ Conditional knowledge level considers the development of programs, i.e. designing, coding and testing. The distinction between writing programs – a Conceptual level activity – and designing, coding and testing programs – a Strategic/ Conditional level activity – is not clear since programs of any consequence cannot be written successfully unless they are designed, implemented and tested. Mayer's two central layers of Semantic and Schematic make this distinction explicit, where the generic aspects of semantics – i.e. how the program statements determine program operation – are

clearly distinguished from the higher level activity of how combinations of those statements may be exploited to produce a solution to a problem.

Lund (2002) recognises the utility of the work by Mayer (1997) in particular and has proposed a unified framework of programming expertise to underpin the development of a teaching support tool, which clearly defines the categories of knowledge, required to program. These categories are identified, as per Mayer (1997), as Syntactic, Semantic, Schematic and Strategic. Here, these levels form the foundations of the learning support offered to students and can be mapped explicitly on to the two challenges presented in 2.2, program formulation and problem formulation. The existing description of each challenge is further refined by the use of Mayer's learning model.

2.3.1 Program Formulation

Program formulation encapsulates both the Syntactic and Semantic knowledge levels of Mayer (1997). According to Lund (2002), *Syntactic* knowledge is specifically concerned with the syntax, or form, of a programming language and the ability to apply that knowledge. Correct syntactic expression of constructs within a particular programming language allows development of *Semantic* knowledge, i.e. the operation of those individual constructs, which will enable them to develop a mental model of program execution or "what goes on inside the computer as a result of a line of code" (Lund, 2002). This mental model, combined with knowledge of the underlying syntax, represents the skill set needed for program formulation.

2.3.2 Problem Formulation

Lund also defines *Schematic* knowledge as the ability to recognise patterns in code developed for previous problems, also known as programming plans, and apply these plans to form a solution to the current problem. Novice programmers have difficulty recognising these patterns and, even when the patterns are made explicit in one context, they are unable to transport these patterns to a given problem, i.e. another context. The final layer in the model, *Strategic*, may be defined as the techniques employed to create and monitor plans, so following a software development process (Lund, 2002). Therefore, knowledge at the

syntactic and semantic levels maps directly on to program formulation and knowledge at the schematic and strategic levels maps directly on to problem formulation.

2.3.3 Knowledge Levels and the novice programmer

While it is assumed that the expert programmer will be confident in all areas of the framework, the abilities of the novice programmer cannot be so easily described. Lund (2002) has identified three categories of novice programmer based on a series of experiments. Novices in the first of these categories are typically unsuccessful in completing a working program without assistance from peers or teaching staff and are therefore unable to do either program or problem formulation. Their solutions show no patterns or evidence of planning. The novices are therefore lacking in the relevant knowledge to solve a given problem and their solution development is usually impeded by a lack of knowledge across all four levels of the model. Novices in the second category are able to produce a working solution but their programs lack sophistication, therefore novices in this category have sufficient program formulation but are unable to undertake successful problem formulation. These students tend to write the complete program and then attempt to debug it in stages, rather than think about the program development stages before writing These students typically possess sufficient knowledge at the syntactic and semantic stage but lack schematic and strategic knowledge. Novice students in the third category write their programs in stages, similar to the expert programmer. Novices in this category however tend to use fewer stages than an expert but are still able to produce a successful program with evidence of a planned approach. These students possess sufficient knowledge across all four levels of the model.

Clearly, within the context of an introductory programming class, novice programmers in the first category are the most demanding. Lund (2002) notes that teaching staff spend most of their time with these students, assisting them in the following areas:

- Helping the student understand the given problem they have been asked to solve, i.e Strategic;
- Helping students identify an approach to solve the problem, i.e. Schematic;
- Helping the students solve semantic (execution) errors, i.e. Semantic;
- Helping the students solve syntax (compiler) errors, i.e. Syntactic.

Therefore novices require support at all four levels of programming across program and problem formulation.

2.4 Existing Tools for Supporting Learning to Program

Boufaida (1996) indicates that the activity of programming incorporates multiple tasks including problem comprehension, development of a plan or formula to address the problem, program development including code production and (syntax) error correction, and "memorization of the designed programs", i.e. identification of underlying patterns across problem sets. From this description it is clear that program development extends over all phases of the software development lifecycle from analysis through to testing.

To support this programming activity, and in recognition of the problems faced by novice programmers, introductory programming tutors have developed a wide range of support tools. The majority of development effort has gone into supporting the design and implementation phases. Sections 2.4.1 to 2.4.4 review some support tools from across the lifecycle, and so include other phases of software development, to capture both the common, underlying set of goals and the diversity of the approaches taken. This review abstracts above specific language and methodologies, and categorises these tools based on their provision of support at lifecycle phases. The review informs the establishment of a set of requirements and a conceptual framework to integrate those requirements into a single framework that supports both program and problem formulation in Chapter 3. Consideration is also given to those tools that take a broader view of the program development process (Section 2.4.5).

2.4.1 Support for Analysis and Design

SOLVEIT (Deek and McHugh, 2000) is a problem solving and program development environment. Designed to support the novice programmer, SOLVEIT facilitates problem formulation, planning, design, testing and solution delivery. Students can enter a description of the programming exercise they are trying to solve in a text editor. The student is then prompted by questions on the exercise description, which force them to think more deeply about the problem and extract the necessary goals, conditions and constraints from the

question. Using these extractions, SOLVEIT then assists the student in developing a data model of the program requirements and subsequent structure chart. Students are further prompted to extract the functional requirements of their program. Students are then required to produce appropriate syntax to solve the problem in a programming environment. SOLVEIT allows students to test their solution by helping them to create a set of test cases based on their program design. After a successful program has been created, SOLVEIT then supports the student in producing accompanying documentation for their program. SOLVEIT operates independently of the programming environment and does not support the student in program formulation, i.e. writing the syntax or solving syntactic or semantic errors. SOLVEIT has no external knowledge of the programming exercise other than that specified by the student. Therefore if the student has made some incorrect assumptions about the requirements of the program, SOLVEIT will blindly support the student in problem formulation based on these assumptions.

Ziegler and Crews (1999) try to address problem formulation independent of program formulation. They propose an instructional program development environment called FLINT to promote problem-solving and critical thinking skills. This environment is designed to help students with design by using visual, flowchart notation to describe program operation. Flowcharts are used to reduce the focus on syntax and allow tutors to concentrate their teaching on problem analysis and solution design. The flowchart interpreter system is an environment, which provides students with an “iconic interface for developing flowcharts”. The interface hides low-level details from the user. Students must first create a program structure chart before designing the flow chart. Crews and Ziegler (1998) conducted an experiment to prove that flowcharts are easier to understand than structured code. During this experiment, students were divided into groups and presented with a series of ‘if statements’ represented as either a flowchart or as structured code. The students were given sample data and asked to predict the output from the ‘if statements’ and comment on how confident they were with the accuracy of their answer. Those students given the flowchart representation did significantly better than their peers who had been given the structured code.

Ziegler and Crews demonstrate that problem formulation may be decoupled from program formulation, and the authors recognise that program formulation can impede problem formulation and so remove the details of programming to allow students to concentrate on solving the problem. It is not clear from their work, if concentrating on problem formulation eases the difficulties with program formulation as the work assesses only the reading of programs. Indeed the authors note that their aim is not to develop programming skills per se in their students but to develop generic problem-solving skills for which programming-centric exercises are a useful domain. Consequently, the benefits of separating out problem formulation entirely from program formulation remain uncertain.

2.4.2 Support for Implementation

Schorsch (1995) has developed an automated self-assessment tool, CAP, to check Pascal programs for syntax, logic (i.e. generic, syntactically correct but improperly formulated program parts) and style errors. CAP provides the programmer with more user-friendly messages than the Pascal Compiler. These messages are designed to inform students about what is wrong with their code, why it is wrong and to suggest how they should attempt to fix the problem. The message may, for example, include a sample of correct code for comparison. CAP operates by analysing the entire source code for errors; it is in essence a compiler with simplified error messages. To determine the diagnostic checks that CAP would need to make, data was collected during lab time and from students' assessments to identify common problems. CAP analyses a Pascal program and displays the total number of syntax, logic and style errors. CAP then displays the source code along with embedded error annotations. CAP reports only the first error of a cascading sequence of errors. To evaluate the effectiveness of CAP, 520 students were issued with a survey after their first assignment. Overall, students rated CAP highly, although students with previous programming experience found the support less valuable. Instructors teaching the module were also asked to complete a survey. "All instructors noted a marked increase in the quality of student programs" (Schorsch, 1995), and generally observed that trivial syntax and logic errors were reduced during class time.

CAP recognises the expert nature of the compiler and repackages the compiler errors in a more user-friendly language. CAP also provides support for simple (semantic) program formulation problems such as failing to update a loop counter within the scope of the loop. CAP was also shown to contribute to a reduction in marking time by performing many routine checks including style analyses on indentation and commenting, to give an example. An underlying problem with CAP is that some students chose not to read the error messages fully, and Schorsch (1995) states that this reduces their learning experience. With regards to programming style, it was perceived that some students used CAP as a "crutch": rather than learning from the CAP feedback. Some did not trouble to follow any style rules as they knew that CAP would fix it for them. These students became dependent on CAP and could not program to the (style) standards without it.

Datlab (MacNish, 2000) is a system to provide automated progress monitoring and feedback of students' Java programs. The tool was developed in recognition of the time spent by tutors and supporting staff undertaking routine checks of student work. This checking time was a consequence of a weekly marking scheme, which itself was introduced in recognition of the need to structure student submissions over the term. The system aims to provide immediate feedback, and to support students working on their own in completing assignments by allowing multiple submissions of the work – and feedback each time – without prejudice. MacNish (2000) notes that this increases confidence in programming skills. Datlab is automatically invoked when a student emails their program to the tutor. This program compiles the file and creates an instance of the submitted program. Java's reflection capability is used to assess the content of the program in terms of method signatures and instance and class variable declarations. The profile of the student program is compared to a model solution and feedback is given based on omissions of constructs. Additionally, the operation of the program is assessed in case of run-time exceptions and against known test data.

Through the use of questionnaires and anecdotally, MacNish (2000) demonstrates that the problem formulation support offered by the tool makes a valuable contribution to the student experience. The author notes that some less advanced students wanted more help than was

provided, and that this raises a pedagogical issue underlying all support tools, i.e. what degree of help is beneficial to the student learning process and what degree compromises that process. NacNish (2000) offers no clear view on this but does note that this support tool is not aiming to replace tutor feedback. Indeed the feedback given should complement staff activity in the laboratory sessions. As a result, the degree of help offered to an individual student is a mix of fixed tool support and staff judgement.

Odekirk-Hash and Zachary (2001) have developed an online tutoring system called InStep which is designed to help students in an introductory programming class. InStep is a web-based application, which presents template solutions to specific problems. The template includes partial program code and comments indicating what the student has to do to complete the program. InStep compiles the completed program and reports on any compiler errors. If the program is syntactically correct, InStep runs it against a set of pre-defined test cases. If the program fails the test cases, InStep attempts to examine the code inserted by the student for common errors and reports these discrepancies to the student. If InStep cannot identify the source of the error it attempts to highlight the line of code causing the error. InStep also displays the program output to the user. An experiment to measure the effectiveness of InStep showed a significant difference in the amount of time tutors had to spend helping those students who used InStep against those students who did not.

The approach tests student program output against model answer output and is able to provide informative and context sensitive feedback. However, InStep is limited to a very small subset of programs and helps only with very basic programming concepts. The students are unable to alter the templates and there is little conceptual gap between each template comment and the actual code required. Consequently, this approach removes the problem formulation aspect of programming and simply focuses on program formulation. However, if students make a mistake in program formulation, InStep offers no support beyond the basic compiler messages. Thus program formulation is not as well supported as other approaches, simply more constrained.

Lang (2002) has developed a less ambitious tool suite to support novice programmers in some of the areas provided by CAP. The tool suite consists of three applications: Pre-Scanner, Instant Scanner and Pretty Printer. The Pre-Scanner provides revised error messages and identifies some simple logic (as per CAP) errors. Instant Feedback "reports any problems with coding style, layout and comment coverage" (Lang, 2002). Finally, the Pretty Printer reformats and indents student source code. A distinguishing feature of this work from CAP is that the tool support is integrated into the development environment, whereas CAP invocation is initiated through a separate environment to program development. Lang (2002) also notes "as students gain experience … they will find the tool's rigidity too constraining and so will use it less frequently". This will result in a reduction in usage as the academic programme progresses.

Hristova et al. (2003) have developed an educational tool called Expresso that operates as a pre-compiler to provide the student programmer with detailed error messages relating to Java programs. The tool has been designed to deal with only a small sub-set of Java syntax errors. In order to develop the tool, a survey was conducted among college professors and students in the U.S. to identify relevant Java programming mistakes. This survey involved distributing a questionnaire, which asked people to list the error messages they found difficult to solve.

The benefits of replacing syntax errors and providing logic (as per CAP) problem support have been noted. A clear shortfall in the work is the process of gathering the information. Only a small, un-stated percentage returned the survey. It is likely that, from the students' perspective, the error messages that they best remember are the ones that they best understand. Therefore the survey may not show an accurate distribution of the common error messages which students find difficult. The Expresso tool has not been evaluated in a classroom environment so it is difficult to comment on its effectiveness. The authors note that the tool is aimed at the early part of the academic programme and they would expect to see a decline in usage over time. Further, the need for it to be invoked separately from a compiler and its inability to operate over a wide range of errors may deter many students from using it.

CMeRun (Etheredge, 2004) is a tool developed specifically for novice programmers. It attempts to help them develop coding and debugging skills by allowing the student to "see each statement in a program as it executes … to show the flow of control and the current status of all variables … during execution". CMeRun works by taking a syntactically correct program and produces a new program containing the original code together with interleaved statements to display each line and variable value during execution. CMeRun has been evaluated informally with the academic staff and teaching assistants who rated it favourably, especially as a demonstration aid in teaching program operation.

2.4.3 Support for Design and Implementation

PROUST (Johnson and Soloway, 1985) is one of the earliest support tools for novice programmers, and supports the design and implementation of programs in Pascal. PROUST characterises a problem in terms of a number of programming plans needed to effect a solution. Johnson and Soloway define a programming plan as "a strategy for realising intentions in code where the key elements have been abstracted and represented explicitly". Plans represent key components of a programmed solution such as a running total established within a loop. The specific problem is characterised in terms of goals, i.e. requirements for a correct program. Rules in the knowledge base of the expert system exist to translate each goal to one or more (correct) programming plans via a goal decomposition process. The student program is then analysed and abstracted into the intended plans, i.e. key components, of that program, where the intended plans represent the closest match to the implemented code. The intended plans are then compared heuristically with the correct programming plans arising from the goal decomposition process and one correct programming plan is identified as the direction in which to guide the student. Inconsistencies between the intended plan and the selected correct plan form the basis of student feedback. The system was tested for a single, trivial problem with a large number of student attempts. Where PROUST was able to analyse with confidence the intended (student) plan the system determined where the program did and did not meet the specified goals with 95% accuracy. This was possible in 79% of cases. For other cases, the plan could only be analysed partially or not at all since

parts or all of the program could not be interpreted as being part of a possible plan to solve the problem.

Three clear strengths in the PROUST approach are: i) the system may accommodate variant solutions to a problem; ii) the feedback is sensitive to the difference between the intended plan and the correct (variant) plan closest to that intended plan; and iii) the analysis and feedback may contribute to the tutoring process. However, to be of benefit the approach requires that the student is sufficiently advanced in their program and problem formulation. For program formulation, the system requires syntactically correct code. With regard to problem formulation, a fundamental assumption in both the methodology and the feedback is that the results of the analysis of the intended plans relate directly to the student's understanding of the problem. Therefore the student must understand the problem to the extent that they are able to articulate a plan comparable with the set of plans in the knowledge base.

The Lisp Tutor (Anderson and Skwarecki, 1986) is another early example of an intelligent tutoring system to help novice programmers produce functionally correct programs. The Lisp Tutor is a fully integrated development environment using the Lisp programming language and so the students program through the tutor itself. The intelligent tutor operates on a symbol-by-symbol basis, where the completion of one symbol (delimited by white space) invokes immediate feedback to the student as to the expected syntactic structure of the remainder of the line. For example, if the student types "(defun " a template will appear indicating to the student that they need to include a name, parameters and body for that function. The Lisp Tutor can also provide the student with help on the problem solution and planning through menus included in the interface. To effect this, the tutor prompts students with suggested templates which they need to complete. Additionally the tutor will attempt to guide the student towards producing a syntactically correct solution based on premeditated rules. The Lisp Tutor contains hundreds of ideal rules and examples of "buggy" rules. The tutor contains specific feedback which will be generated if the student produces a particular "bug'".

Anderson and Skwarecki (1986) note that the Lisp Tutor does improve assessed student performance. However, they do also recognise that the Lisp Tutor is designed to support students who write their programs in a top-down, left-to-right process. For example, a student cannot create the body of a ‘for loop’ before the initialisation. Similarly, the tutor provides feedback for each possible symbol as and when it is typed, meaning that a student cannot write a block of code before receiving feedback. Anderson and Skwarecki further point out that the two extremes of feedback – delayed until the program is completed, as in PROUST above, and immediate feedback as here – are not ideal for supporting the learning process. They suggest development of a system able to choose more strategically when to give feedback, based on the context, or extent, of the developed solution.

EXPLAINER has been developed by Redmiles (1993) to assist programmers in performing new tasks using previously completed exercises based on the cognitive theory of learning by example. Redmiles recognises that learners with good mental models of example programs can successfully apply that knowledge to new programming tasks. EXPLAINER is designed to assist students in developing that mental model. EXPLAINER uses code examples, sample execution, component diagrams and text to help demonstrate new concepts to the user. This software operates independently of the programming IDE. EXPLAINER was tested with 23 (conversion) masters-level computer science students and results indicate that those students using EXPLAINER did perform better than those using an on-line manual to explain programming concepts. The study was undertaken with post-graduate students who already have an established skill set in knowledge acquisition, although the benefits of interactive support over a static (on-line) manual most likely applies to first year students enrolled on introductory programming courses.

SWANN (Brna & Mathieson 1993) is an environment for novices to debug Pascal programs. SWANN has been designed to work with a small number of programs. It is able to parse a faulty Pascal program, i.e. a program that fails to meet specified operational criteria, and attempts to provide feedback to the novice in the form of non-programming plans, i.e. non-code based representations of the algorithm to be implemented. The novice is presented with a series of suggested repairs in terms of this higher-level representation of program operation

that they can make to solve their programming error. It detects discrepancies between the actual and expected program behaviour and offers the novice a set of operators which may be used to fix the error. Specifically, SWANN suggests a range of alternate ways of fixing the error for a known problem.

Brna and Mathieson (1993) recognise that supporting novices by providing feedback related to errors during the development of programs may reduce the complexity of programming. They do this by encapsulating knowledge of the particular programming task within the analysis and feedback. This feedback is not in specific code but in suggestions on the operation of the program. They, like PROUST (Johnson and Soloway, 1985), acknowledge variance in programming implementations and so provide a range of suggested solutions. However, students must develop a program of sufficient substance for SWANN to construct an abstract representation, which may be compared meaningfully with existing non-programming plans. Moreover, it is possible that this error support can contain suggested fixes, which are not plausible. The authors recognise that the programs in SWANN's knowledge domain are very small and do not allow the novice to work with more challenging problems. Further, the support extends only to those programs in the knowledge base so students cannot benefit from the SWANN system while working on independent problems.

The Intelligent Verilog Compiler (IVC) has been designed by Moore and Taylor (2005). IVC provides support for students who have already programmed in the Java and ML languages. However, the Verilog environment introduces the additional challenge of working in the restricted and inherently parallel domain of circuit board programming, and so seeks to offer support in this specific domain. Further, the Verilog language is less sophisticated than Java, offering less support in typing and initialisation of variables, and so students may be viewed as novices to some extent. IVC is an on-line tutor that presents a series of programming exercises to the student and is intended to replace lectures. Broadly, the IVC provides context sensitive error messages and assistance by providing links to tutorial pages. A combination of guidance from the module deliverer, an online survey of student views of undertaking the work within the module, and contributions from staff who supervise students in the laboratory sessions informed tool development. The tool was designed to address three

distinct but related areas of learning: conceptual, syntactic and semantic. The conceptual support is introduced to provide support for the transition from serial programming within Java to the parallel environment of Verilog. The Syntactic Support extends from rule-based difficulties, i.e. syntax errors, to variable declarations. This is a simplified version of program formulation. Semantic Support relates to the overall operation of the program in terms of addressing the requirements of the problem, and so is analogous to problem formulation.

Syntactic Support is effected by context sensitive help messages and by program checks. For help messages, the IVC extends the existing compiler error with a more informal explanation, as in CAP (Schorsch, 1995), aiming to represent interactions with a more experienced programmer. Program checks are undertaken to ensure that variables are declared properly, i.e. initialised, and that array ranges are appropriate. This is possible because of the small-world domain in which students develop programs, e.g. the sizes of arrays used are linked to the underlying hardware. Additionally, hyperlinks to the established set of notes are provided in the error feedback. The reduced remit of the programming exercises and environment provided by Verilog allow exploitation of two abstractions of the programming process to support Semantic Support: state transition diagrams and trace tables. For each exercise, correct versions of each may be established and semantic error support is driven by the differences between the correct 'model' solution and the (automatically generated) student version. This way, specific feedback is provided on where the student solution differs from the desired answer. An initial evaluation of the use of IVC to replace attendance at four lectures has demonstrated that students are able to complete laboratory exercises faster. A questionnaire proved that of the students on the course, almost half read the text generated by the hyperlinks, and most used the abstracted state transition diagrams, to help them solve the problem. Only 6% of students did not use the features available within the IVC to debug their programs. Further evaluations on the impact IVC has on students' learning will be demonstrated with a comparison of examination results after June 2006 (Taylor, pers. comm. 28th April 2006).

2.4.4 Support for Program Operation and Testing

VINCE (Rowe and Thorburn, 2000) is a tutorial tool, which graphically displays the execution of a program to help students develop a mental model of control flow and data structures. VINCE can be used with a set of pre-written programs or with a (syntactically correct) student program. The tool provides “a simulated map of the computer’s memory, showing where pointers and variables are stored”. Although students’ perceptions of their programming abilities did not change after using VINCE, a study did demonstrate that the students performed better on programming questions which indicates that the visualisation of program execution did facilitate the learning process.

CourseMarker (Higgins et al. 2003), (formally known as Ceilidh) was originally designed to support module administration through the presentation of materials and the assessment of students. The most relevant of CourseMarker’s features is its ability to assess and provide feedback on students programs. CourseMarker presents the student with a programming question and can also show solution templates to the student. The students can develop their own solution and then submit it for assessment. CourseMarker can return a grade and related comments. An administrator can control how many times the student can submit their work, therefore CourseMarker can be used for formative or summative assessment. Comments provided by CourseMarker can include details on how to improve the solution and links to recommended reading material. Students may be reluctant to use CourseMarker until they are confident their program is complete since all attempts are logged for the tutor and students are sent a receipt file. It is not clear whether CourseMarker can accept programs with syntax errors and how much assistance it can give those students who need considerable help with developing a solution. CourseMarker appears to use one model solution and marking file. CourseMarker has been designed primarily to reduce the time spent marking programs. In addition, the authors recognise the value of providing feedback that relates to any required components within the solution.

2.4.5 Broader Support

LearnOOP is an agent-based approach to providing educational support. Wang (1997) describes an agent as a computer system that provides support to the user by reacting to patterns in the student activity via text-based dialogue. LearnOOP provides a student agent and a teacher agent. The Student Agent builds a history of a particular student and allows the student to “navigate sections and topics”. This operates in terms of monitoring those sections of notes that have been read and those exercises that have been completed. LearnOOP continuously checks a student’s progress and provides guidance and exercises where necessary. The Teacher Agent facilitates creating and editing teaching material, marking assignments and reviewing individual student learning histories. The Teacher Agent also allows interactions between the student and the teacher by providing question and answer forms which can be passed between the two. While LearnOOP is not able to provide constructive feedback for the programming development in itself, it does recognise the importance of facilitating discussions between student and tutors, prompting individuals to seek feedback from peers and human tutors.

Paine (2001) has developed a system called Coach that is integrated into an existing tool set to assist novice programmers at the Open University. Coach has been developed explicitly to deal with the problem of providing immediate feedback on programming errors to students who do not have “immediate access to peers or tutors”. The existing tool set (AESOP) is able to record, analyse and replay the mistakes that students make when programming and their attempts to resolve these errors. Coach utilises this information to predict possible solutions to the common problems that students make. The replay facility is aimed directly at tutors to provide some representation of the student interaction with the environment in a distance-learning environment. Coach is also integrated with the Open University teaching material to allow students to re-visit relevant material when they need help with their understanding of the programming concepts. Paine recognises the importance of recording actual student errors to recognise the common mistakes that students make. Coach has been fully integrated with the teaching material and, combined with the knowledge of common syntax errors, is able to provide context sensitive messages to the student. These may then help directly with

the current problem being addressed. Additionally, Paine recognises that a student may need additional support in understanding programming concepts and Coach is able to link to specific teaching materials. These links are context sensitive to the current activity, although Coach is dependent on the student initiating the support.

Shah and Kumar (2002) have developed a tutoring system specifically aimed at supporting the teaching of parameter passing. The approach asks students to match pre-written program code with program output of key variables. The system is able to generate at random a complete program, according to a prescribed template, consisting of functions, variables and arrays. Students are asked to predict, by typing in values, the expected result of running the program. The system can provide minimal feedback, i.e. whether the student is right or wrong, or detailed feedback, which includes an explanation or the correct answer by describing the program behaviour. The authors attempt to evaluate their system to determine i) that learning of parameter passing had improved and ii) that detailed feedback supported learning better than minimal feedback. The design of the evaluation undertaken by Shah and Kumar (2002) is questionable. Students were given a set of pen and paper tests to undertake in an eight-minute period, termed pre-test, then allowed to use the tutor for twelve minutes, and then given another set of pen and paper tests to carry out for a further eight minutes, termed post-test. Shah and Kumar state that the improved performance in post-test over pre-test is a systematic improvement in their learning as a result of their tool. The authors take no account of the time spent prior to each post-test undertaking related activities, a difference of 20 minutes. Further, because of a small sample size they are unable to identify whether the detailed feedback is (statistically) significantly better than minimal feedback. However, two things are made clear from this work. First, repeated engagement with tasks relating to a specific programming aspect, together with pertinent feedback, will promote learning and confidence in the task. Second, in the informal feedback commentary students responded positively to the detailed feedback and noted that the minimal feedback offered no indicator as to why their answer may have been wrong.

JKarelRobot has been developed by Buck and Stucki (2001) as an extension of the original concept behind Karel the Robot (Pattis 1995). They use Bloom’s Taxonomy of Educational

Objectives (Bloom, 1956) as a guide to structure different styles of interaction according to different levels of cognitive development. JKarelRobot utilises an ‘Inside/Out’ pedagogy that gradually exposes students to more complex programming structures. JKarelRobot can support Java, Pascal and Lisp environments. The system offers various levels of control to the user from minimal command button control for orientation to the world of the robot to program writing in terms of method calls. Additionally, flowcharting is used to present the design of the implemented solution. Karel and its descendents are many and varied, ranging from direct robot analogues in alternate languages such as JKarelRobot to variant contexts including kangaroos (Sanders and Dorn, 2003), pigs (Lister, 2004) and submarines (Culwin et al., 2005). They all promote a reduced programming world, or microworld (Garner, 2003), which offers students the opportunity to explore problem formulation with limited program formulation development. The approaches all successively release details to support exposure to new concepts in a progressive manner. The microworld approach has been recognised to add value to existing teaching methods during the problem analysis stage (Garner, 2003).

BlueJ (Kölling and Rosenberg, 1996, Kölling et al, 2003) is a fully integrated programming environment designed to teach object-oriented programming to novices using a graphical representation of the relevant classes and objects. BlueJ is a reduced development environment, which removes the complexity of testing and syntax. The main goal of BlueJ is to encourage novices to “think in terms of objects” and not code. BlueJ has been fully evaluated and results show that students believe BlueJ has helped them learn object-oriented programming (Haaster and Hagan, 2004). Further, one of the new BlueJ plug-ins possible under an Extensions API (Utting, 2006) allows checks for semantic errors in the use of variables. Variables are allocated roles (Sajaniemie, 2005), e.g. steppers for use in for loops, and the program is checked to ensure the variable usage is consistent with the allocated roles (Johnson, 2006). Kölling (2003) acknowledges that students need to learn to use professional programming tools before leaving university and recognises the problems that students encounter when moving away from the BlueJ environment.

Storey et al. (2003) have described, as part of an ongoing development process, a set of plug-ins, called GILD, for the Eclipse Java Development interface, to be used by novice

programmers. This development recognises two issues with Eclipse. First, it is an industry standard IDE and so houses many functions not needed by novices. Second, it is important that novices learn to use a real programming environment. By using a combination of student questionnaires and both focus groups and interviews for tutors the GILD project has been developed with a specific list of features that are required by novice programmers. Included in these features is access to a file editor with syntax highlighting and line numbering abilities. Support for program operation, in terms of a (semantic) debugger that focuses on the current method call, was also recognised as important. Storey et al. (2003) acknowledge the developmental power of Eclipse yet state that many of its features need to be simplified for the novice user, for example integrating the debugger into the GILD perspective. Some features of Eclipse have been removed to simplify its operation, for example wizards and refactoring, or enhanced, such as the resource view. To further support the novice programmer an HTML browser has been incorporated into the tool set allowing students to navigate course notes within their programming environment.

Gomez-Martin et al. (2003) are currently developing a case-based reasoning system called Javy that is designed to provide an engaging learning environment for students. Javy is an animated pedagogical agent, which monitors the student in order to offer guidance when an error is made and produce a working solution to a given problem with step-by-step instructions. It is assumed that students already have programming experience. Hence Javy is not used with novices. Javy has not yet been implemented in the classroom environment.

OGRE (Milne and Rowe, 2004) is a 3-D software visualisation system for novice programmers learning C++. OGRE is not designed to support the student in producing a valid program. Instead it provides visualisations to assist students in their understanding of the programming concepts that have been previously recognised as the most difficult. OGRE also allows the students to watch a visualisation of a program’s execution. OGRE can be used as a stand-alone application, in which the students can work through a set of built-in tutorials. OGRE can also be used by lecturers and tutors as a demonstration aid. OGRE has been fully evaluated using a set of formative and informative methods, i.e. experiments and interviews. This evaluation has demonstrated OGRE’s effectiveness as a successful teaching

tool to assist students in their understanding of complex programming concepts through visualisations.

2.5 Summary

Existing literature demonstrates the problems encountered by novices learning to program. These difficulties can be categorised into problems in either program formulation or problem formulation. To support the development of an integrative teaching tool, program formulation support should encompass syntactic and semantic knowledge and problem formulation support should encompass schematic and strategic knowledge. The review of existing teaching tools informs the establishment of a set of requirements and a conceptual framework to integrate those requirements into a single framework in Chapter 3.

Chapter 3 Towards a Supportive Framework

3.1 Analysis of Teaching Tools

Program development spans all phases of the software lifecycle and tools exist to support each of these phases. However, most conventional, introductory courses focus on implementation (Robins, 2003). Further, the problems studied are so small that structured problem analysis in particular is not needed. Additionally, the practice of building programs is an appropriate starting point since it involves a clear process of engagement to motivate students through small-scale software development. Consequently, the majority of tools exist to support the design or implementation phases, with the most sophisticated tools addressing both, and here the focus is on these.

FLINT, by Ziegler and Crews (1999) helps students develop problem-solving and critical thinking skills. Through the use of flowcharts students work in a reduced syntax environment. It is clear that the focus of FLINT is to help students with problem formulation as opposed to program formulation. Importantly, FLINT has been developed within a dedicated programming module aimed at non-computer science students. The purpose of the module is to teach problem-solving skills and programming is chosen as a vehicle to teach these skills. However, the aim of the majority of programming modules is to teach programming, and primarily to computer science students. Although problem formulation is critical these students still need to be taught program formulation, which cannot be done in a reduced syntax environment.

For support at the implementation stage, there are many tools and here we discuss in chronological order CAP, Datlab, InStep, the toolkit from Lang, Expresso and CmeRun. CAP (Schorsch, 1995) is a self-assessment tool that allows students to check their code for syntax (for example a missing semi-colon), logic (for example failing to update a loop variable) and style errors (for example failure to indent nested statements). CAP recognises the need to replace compiler error messages with more user-friendly ones for the novice programmer. CAP also recognises the common semantic mistakes that novices make, for

example missing loop counters and the benefit in providing support for these errors. Schorsch (1995) acknowledges that students could become dependent on this level of support and recognises that a tool should give less directed support as the module progresses.

Datlab (MacNish, 2000) was developed in recognition of the students' dependency on class tutors to solve trivial mistakes. Datlab aims to provide feedback to the students and support them in resolving their own mistakes. MacNish (2000) raises the issue of providing the student with too much help and acknowledges that a support tool could be used to complement rather than replace existing teaching methods. InStep (2001) takes advantage of the common errors that students make to predict the likely cause of a problem in a novice program. However, InStep is limited to a small problem base with 'fill-in-the-blank' questions that do not fully support students in program formulation. Lang (2002) also recognised the need to provide students with simplified error messages. Lang suggests that as students gain confidence in their programming abilities they will become less dependent on a support tool.

Expresso (Hristova et al., 2003) has also been developed through recognition of the advantage in replacing compiler errors with more simplified messages for the novice programmer. Similar to CAP, Expresso also detects some common logic errors that novice programmers are known to make. CMeRun (Etheredge, 2004) is recognised as a useful demonstration aid in the classroom, specifically to support the need for one-to-one teaching. CMeRun allows the student to see each statement and variable value during execution. On execution CMeRun indiscriminately interleaves the code with print statements to indicate program state. This is not needed for all program parts and not suitable for all problems, e.g. small world teaching as in Karel the Robot, and is something that the students could do themselves using their own print statements.

For support at the design & implementation stage, PROUST, the Lisp Tutor, SWANN and IVC are discussed. PROUST (Johnson and Soloway, 1985) characterises a problem in terms of a number of programming plans needed to effect a solution. An expert system links goals in the stated problem to one or more (correct) plans and derives the intended plans of the

student. Correct and intended plans are compared and the student is directed towards the closest correct plan. This approach allows variant solutions to a problem and this in turn promotes independent learning in students since they are not constrained by a single ‘correct’ solution and are free to explore approaches with a variety of constructs. Further, the associated feedback is linked to both the closest plan and the intended plan. Johnson and Soloway also indicate that PROUST contributes to the tutoring process. However, PROUST requires that the student is sufficiently capable of constructing a plan in code of adequate sophistication to derive the goal set from the program for comparison and only (necessarily) operates on complete or nearly complete implemented plans since it is, in essence, pattern matching.

The Lisp Tutor (Anderson and Skwarecki, 1986) is a fully integrated learning environment which provides immediate feedback to the student to help them complete their program whenever they produce an error. The Lisp Tutor also predicts the construct that the student is attempting to create and provides them with a template to complete the rest of the code. Although the Lisp Tutor has been proved to help the students, it is also recognised that the immediate feedback can be too intrusive and does not support those students who would rather request assistance when they need it. Indeed, the immediate feedback presented exploits the line-by-line runtime interpretation afforded by Lisp and this is not common.

SWANN (Brna and Mathieson, 1993) is an environment that supports the debugging of a small number of predefined programs written in Pascal. SWANN provides feedback to the novice in the form of (non-programming) plans or outlines. SWANN is able to provide a range of suggested solutions to the novice. The support is limited to a small subset of programs and requires a substantial attempt on the part of the student for the analysis to be useful. In addition to this, some of the suggested fixes are not sufficient to correct the program and may actually direct the student further away from the desired program operation.

IVC (Moore and Taylor, 2005) is an online tutor that presents the student with a series of programming exercises and provides context sensitive error messages and links to relevant

tutorial pages. The error message support provided includes syntax and recognised common semantic errors. IVC also allows students to ask questions in English. Allowing students to have a simple conversation with the support tool helps to guide the students towards a solution whilst supporting deep learning.

3.2 Emergent Requirements

It is evident that a support tool may be used to facilitate the learning process of novices who are learning to programming. This process involves a number of interleaved concepts and there are many methods and approaches to teach these concepts. Therefore, a support tool that is sufficiently flexible to accommodate many different instructional and delivery designs is of the most value. Ideally, the support tool should be decoupled from any specific IDE and further operate as an add-on to the selected programming IDE.

No support tool that aids program formulation with respect to syntax should promote dependence on that tool to such an extent that the student is not able to progress beyond the remit of that tool. To remove this dependence, and to benefit from support that takes account of the taught context, it is necessary to present the student with both the standard compiler messages and the supplementary error messages enhanced with context-relevant support. To further reduce the risk of dependency the level of support should be reduced over time. In addition to these steps, the use of a support tool should be voluntary as this affords students the opportunity to direct their own learning and allows those students with existing experience to opt out of extended support provision. Again at the program formulation level, and in recognition of the frequent semantic errors (or logic errors as defined in CAP) which students make, a support tool should detect these common mistakes and provide relevant warnings to the novice.

To support a student more fully in the development of a solution to a given problem, i.e. problem formulation, a support tool needs to have knowledge of both the key constructs and the relationship among those constructs necessary for a working solution. A support tool should be able to offer assistance to the student in identifying which constructs are needed to solve a particular problem and make sure that their solution meets the requirements of the

exercise. This in itself raises two issues. First, it is important that a support tool does not direct a student towards a single model solution, but allows the student to develop a correct solution of their own design, i.e. the support tool must recognise that variant solutions exist and these should be supported. Second, the tool must provide support in stages, guiding the student through the development process, rather than directing the student towards a complete solution from the outset or only operating with a near solution as in PROUST.

Finally, to reinforce concepts already taught to the students, a support tool should be able to direct novices toward relevant, context sensitive (i.e. the general taught context such as terminology and bespoke teaching packages) and context aware (i.e. the specific teaching activity currently undertaken) teaching materials. The feedback provided, including links to teaching materials, should provide an effective platform upon which to base student-tutor dialogue.

In summary, Table 3-1 lists the requirements of a support tool:

Table 3-1 Core Requirements of an Effective Support Tool

1	All forms of support may be progressively reduced over the teaching period
2	Present both standard compiler and enhanced support concurrently
3	Identify and advise on commonly observed semantic errors
4	Embody knowledge of key constructs needed to solve a given problem
5	Embody knowledge of the relationships between the constructs needed to solve a problem
6	Where appropriate, ensure that this knowledge accommodates variant solution forms
7	The knowledge should be disseminated to students in successive stages
8	Link to teaching resources as a means of information delivery and student-tutor dialogue
9	Use of the tool must be voluntary on the part of the student.

The requirements identified in Table 3-1 can be mapped against the existing reviewed support tools, based on the information available in the public domain, as shown in Table 3-2. Note that none of the existing support tools meet all nine of the requirements. IVC and CAP meets most of the requirements, althought neither supports problem formulation (requirements 4 and 5). In contrast Proust, List and DatLab include some provision for problem formulation but none for program formulation. Thus no existing tool provides the holistic support identified as necessary in Chapter 2.

Table 3-2 Requirements met by Existing Tools

Tool	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
Flint (Zeigler and Crews, 1999)									✓
Cap (Schorsch, 1995)	✓	✓	✓				✓		✓
Datlab (MacNish, 2000)				✓	✓				✓
InStep (,2001)			✓						✓
Toolkit (Lang, 2002)									✓
Expresso (Hristova et al, 2003)			✓						✓
CmeRun (Etheredge, 2004)		✓	✓						✓
Proust (Johnson and Soloway, 1985)				✓	✓	✓			
Lisp Tutor (Andrerson and Skwarecki, 1986)				✓	✓				
SWANN (Brna and Mathieson, 1993)				✓		✓			
IVC (Moore and Taylor, 2005)	✓	✓	✓					✓	✓

3.3 A Conceptual Framework

3.3.1 Overview

An framework is presented that addresses all of the requirements emergent from the literature review. The framework identifies the components needed in a system to develop a

technology-based tool that supports novices learning to program. The following diagram illustrates those components and the interrelationships among them. Central to the framework is a software implementation that draws on established resources and integrates the different aspects of support provided into a single interface. These aspects are outlined here to provide an overall context and then explored more fully in subsequent sections.

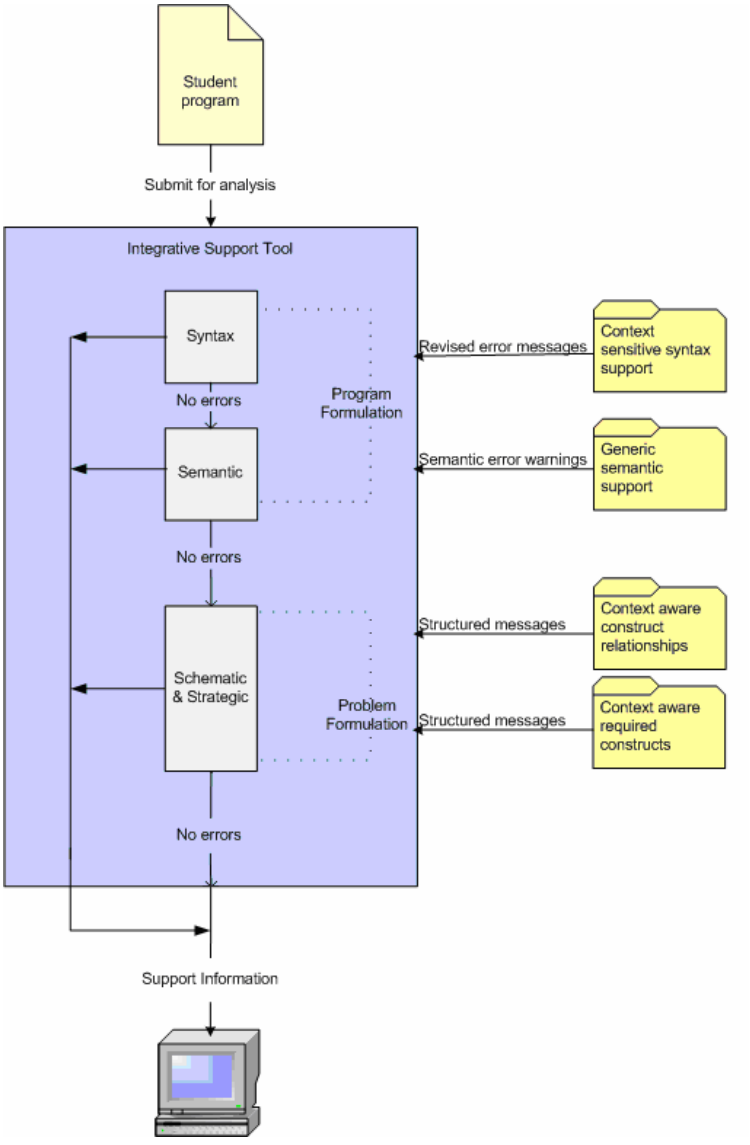


Figure 3-1 Conceptual Framework

The framework comprises five main dimensions: syntactic, semantic, schematic and strategic structure support together with interaction, i.e. invocation and feedback, with the support tool itself. To provide Syntactic Support for program formulation it is necessary to supplement the (original) error messages arising from the compilation process with extended text, where this text is sensitive to the taught context. To provide Semantic Support for program formulation it is necessary to perform checks on the program code for common (logic) errors and provide warnings to students, which indicate these logic errors. Knowledge of the problem domain (a specific practical exercise) may be used to inform the development of strategic and schematic checks for solution formulation, i.e. checks for key constructs and the relations among those key constructs respectively. Through this mix of syntax, semantic, schematic and strategic checks, the support tool may provide feedback to the student. This support must be integrated into the current development environment to provide a seamless interface that is available whenever students elect to use it.

3.3.2 Program Formulation Support

3.3.2.1 Syntax Support

Supporting the student in writing syntactically correct code is imperative for program formulation. As demonstrated in Chapter 2, difficulties with syntax can affect student learning of both the language and the underlying concepts. Many universities use industry-standard programming languages as a tool for teaching introductory programming. The compiler error messages produced by these programming languages are designed for expert programmers, and so their successful interpretation may depend on a degree of knowledge not possessed by a novice. Rather than *replace* these expert level error messages, these messages should be *extended* with content more suitable for novices (Requirement 2). Since the student should always be presented with the original error message, they should soon recognise with the aid of the extended messages why these errors have occurred and be able to resolve compiler-only errors in due course. These extensions should use dialogue that the student is familiar with and, where possible, relate to the content of the actual module (Requirement 8).

3.3.2.2 Semantic Support

Like syntax, Semantic Support is also very important for program formulation. Although problems at this level do not occur with the same frequency as syntax errors, their presence is less obvious to the novice, and so they are harder to detect, since they are manifest only at run-time. As there is no compiler support for these errors students may spend a long time trying to identify and then rectify the problem. Indeed, based on observations (see Chapter 4) students often make structural changes to their program in an attempt to correct its behaviour rather than identifying a (more trivial) semantic error. Some other support tools, for example BlueJ (Kölling et al., 2003), recognise the value of providing support for semantic problems and have incorporated a series of semantic checks into the software support. To address Requirement 3 a support tool should perform checks for commonly observed semantic errors. Again, the dialogue used in feedback should be something that the student is familiar with and, where possible, relate to the content of the actual module (Requirement 8).

3.3.3 Problem Formulation

3.3.3.1 Schematic

To successfully formulate any program a novice must have sufficient knowledge of the syntactic and semantic aspects of the language. However, the formulation of a solution to a given programming problem requires identification of the required structure of the program, i.e. the interrelationships among the component parts of the program. Many students need assistance with identifying such a program schematic (Lahtinen et al. 2005). A support tool that has knowledge of the appropriate constructs (Requirement 4) and these interrelationships (Requirement 5) required for a specific programming exercise can be used to fully guide the student towards creating a solution. Rather than present the student with a list of all the necessary constructs and interrelationships, students should be guided in a stepwise manner through a solution development (Requirement 7). This guide may then exploit the schematic patterns inherent in introductory exercises to promote a systematic approach to software development.

3.3.3.2 Strategic

Perhaps the most fundamental and yet conceptually challenging level of learning occurs at the strategic level. Here, a support tool should check that the basic components required to solve the problem are in place (Requirement 4). Problems in identifying this basic set of components are driven by a failure to recognise the (often implied) functional requirements stated in a question and how those requirements are translated into the program development process. A support tool should make clear, again in a stepwise manner (Requirement 7), that translation by making transparent the links between program requirements and the components needed to meet those requirements. Further, both here and in the Schematic Support offered, the support tool must recognise that there may be more than one possible solution for a given problem and provide support for a range of solutions appropriate to the problem posed (Requirement 6).

3.3.4 Feedback

To enable students to become more confident in their abilities, and develop a skill set independent of any additional support, it is necessary to reduce the amount of feedback provided as teaching progresses (Requirement 1). The progressive reduction in support over the taught period will ultimately eliminate dependence on a support tool. To ensure that the feedback is relevant and contains terminology that the student is familiar with, the feedback should make direct relation to core and/or supplementary teaching material where possible (Requirement 8). To direct their own learning students must have the opportunity to work with just the basic toolset, for example the compiler messages, and so invocation of any software support must be at the individual student's discretion (Requirement 9).

3.4 Summary

The above sections describe a conceptual framework for meeting the requirements emergent from the literature review. To effect a software implementation of this framework, the process of learning, which has already been established as both continuous and concurrently spanning both program and problem formulation, has been discretised into the levels of

Syntax, Semantic, Schematic, and Strategic. The framework thus suggests that learning support may be provided in terms of those distinct levels for both program analysis and feedback. For analysis, there exist different stages imposed by the nature of the tasks required. First, program formulation and problem formulation may be separated out, as evidenced by previous work addressing one or other aspect. Second, semantic analyses must be preceded by syntactic analyses since semantic behaviour depends on a syntactically valid program. Finally, the assumption that program formulation precedes problem formulation accounts for the fact that structural analyses are made easier on syntactically valid programs, but this ordering is not crucial: checks for key components and relationships among them may in principle be performed on syntactically invalid code.

While the analysis of the program for errors is broken down into the four distinct levels of knowledge there is no reason to constrain the feedback given to a single level. That the activity of programming requires concurrent engagement with those different levels of knowledge suggests that useful feedback must incorporate multiple knowledge levels where appropriate. To explore the relationship between the level of analysis, i.e. where an error is detected, and the (knowledge) level of feedback necessary to aid the student, a series of observations have taken place at two universities. Chapter 4 outlines these observations, undertaken at the University of St Andrews and the University of Abertay Dundee, and presents the results. The observations allow an extended understanding of the problems encountered by students and identify the nature and extent of feedback required to help a student resolve a problem. The impact of this study on feedback design is considered. Based on the framework and emergent requirements of Section 3.2, together with the observations detailed in Chapter 4, Chapter 5 presents an implementation of this framework - a support tool called SNOOPIE that encompasses the necessary elements to support the novice programmer. Chapter 6 details an evaluation of SNOOPIE at the University of St Andrews and the University of Abertay Dundee.

Chapter 4 Exploring Student Feedback

4.1 Introduction

In order to understand the nature of the problems that students encounter when they are learning to program, a series of observations was conducted during the academic sessions 2002-2003 and 2003-2004. The observation process consisted of the recording of problems which students encountered with their practical exercises that they were not able to resolve themselves. Problems ranged from simple syntactic errors to more complex issues regarding their approach to solving the given problem. In all classes used during this study, the observer was able to record the actual problems as they occurred. The recording of errors was further supported by extended, recorded dialogue with individual students about specific errors to explore their understanding of the error. The chosen collection strategy of student observations, coupled with extended dialogue, immediately identified the problems encountered. The strategy employed accounts for two important aspects of collecting observational data from students. First, this immediacy addresses the limitations of the use of post-module follow-up questionnaires to elicit knowledge relating to the provision of error support. For example, Hristova et al. (2003) uses a limited number of such questionnaires from staff and students. The 'expert level' of staff causes a problem in the use of questionnaires because staff are more likely to remember interesting, and so conceptually difficult, errors than more trivial syntax errors and this clearly introduces potential for bias in the study. Likewise students are more likely to remember problems which occurred later in the module, and so again higher-level problems, rather than the barriers encountered in the earlier stages of the learning process. More fundamentally, differences in the understanding of the causes of any errors between staff and students mean that it is imperative that any data collection for aiding students is student-centred.

Secondly, the observational process captures only those errors, at all knowledge levels, which students are unable to solve themselves and this highlights the purpose of the observations. Many previous studies have considered the syntax errors encountered by

students. A good example of exploration of syntax errors only is provided by Jadud (2005), who offers a comprehensive study of the syntax errors generated by novice programmers and uses automated archiving. Jadud considers time between compilations and the extent of the change to the code, which is associated with compiler invocations with a view to understanding how novices use the compiler as a tool to develop programs. Here, the study is not limited to syntax errors. The observation process captures all types of errors encountered by novices since the goal is to identify the underlying reason for the error and to explore the feedback needed to address the error. The observation process employed here focuses directly on the problems that students encounter, explores the student perception of the problem and captures the staff-led resolution of the problem. These are the aspects which are necessary in the formulation of the feedback required to support programming difficulties as experienced in the laboratory.

The universities involved in the study are the University of St Andrews and the University of Abertay Dundee. During the academic sessions 2002-2003 and 2003-2004 at the University of St Andrews the practical class for the first year programming module was repeated three times a week. The students at the University of St Andrews were randomly assigned to one of three practical groups. The data was collected from observations of one of these groups during each academic session, and in each session this group consisted of approximately 18 students. In addition, at the University of St Andrews during the academic session 2003-2004, observations were also made of two groups of students attending their weekly hour-long tutorial session for the duration of the semester.

During the academic session 2002-2003 at the University of Abertay Dundee students were assigned to a practical class based on their performance in an aptitude test that was conducted at the start of the semester. For an initial orientation to the range of difficulties experienced by students, the data was collected from a group of approximately 20 students who performed poorly in the class test and had no previous programming experience and were therefore considered the least experienced and least confident group of students. During the academic session 2003-2004 at the University of Abertay Dundee, the students were randomly assigned to a practical class. The data was collected from two of these groups

during one of two practical sessions in a week where each group consisted of approximately 18 students. In the first session of observations, the author shadowed a class tutor at both universities to provide an orientation to the problems encountered. In the second session, the author was a tutor at both universities.

During the observations at the University of St Andrews, three tutors supervised each laboratory class. It was noted during the observations that students in these laboratory classes rarely waited for assistance; there was usually at least one tutor available. It is likely that this high staff-to-student ratio affected the types of questions that students asked. If a student saw that a tutor was available and not busy with another student he or she might have been less likely to spend time trying to resolve a problem before asking for assistance. It was also observed that tutors typically spent more time with each student than those at the University of Abertay Dundee, explaining what had caused their problem and guiding the student towards a solution. At the University of Abertay Dundee, one tutor supervised each laboratory class. This lower staff-to-student ratio was observed to have an impact on the length of time those students had to wait to receive assistance; frequently, four or five students had their hand up waiting for assistance at any one time. The tutor had less time to spend guiding the student towards a solution and students were frustrated at the length of time they had to wait to receive help (See Chapter 6, Section 6.7).

The purpose of these laboratory observations was to note the problems and record the discussions that the students encountered with their understanding of the programming constructs and theories. An important consideration in any observation process, which is necessarily a sampling from all possible observations, is that the subjects and their activities observed represent a sufficiently broad sub-sample. That the data generated is derived from introductory programming modules at both the University of St Andrews and the University of Abertay Dundee ensures a broader view of student problems. Here, four different cohorts are considered over two successive sessions at two different institutes. For context, the mode of delivery of the modules studied, the entrance qualifications required for the course, the proportion of the first term of study assumed by the programming module and the nature of the exercises studied are all documented. Of note is that there exists a significant difference

in the number and scale of the practical exercise questions being addressed (few and large at St Andrews; many and small at Abertay Dundee). The difference is highlighted by inclusion of part of a St Andrews practical exercise and several exercises used at Abertay Dundee. The impact of these differences on the observation results is discussed.

Conducting a series of formalised observations is not the only route to gaining an understanding of the problems encountered by students. There exists literature that cites common errors made, and other literature that describes the use of questionnaires and compiler error logging tools to identify common errors encountered. This combined with existing, experiential knowledge of the difficulties associated with learning to program may be sufficient to develop an understanding of the nature of the problems requiring support and associated feedback. Here, the scheme of extended observations was selected to ensure that the author gained a genuine, 'hands-on' understanding of both the problems encountered and the learning support provided to resolve those problems.

4.2 Methodology

At both universities, the observer attended one of the weekly laboratory sessions for the duration of the academic session. For each problem that the observer helped a student resolve, information was recorded on the manifest error (e.g. a syntax error description or question posed by the student regarding their understanding of the practical exercise) and the solution that was required to solve the problem. Additionally, for each problem, the solution recorded included the information that was required to explain to the student why they had encountered this problem. Forms were used each week to record the date, university, lab group and tutorial number. The form contained headings for time and date, nature of problem/error messages, cause, solution and comments. Specifically, to complete the form, the observer was required to make an initial recording of the problem as noted by the student, i.e. the manifest error. The observer then established and recorded the actual, i.e. underlying, cause of the problem, for example a missing variable declaration or poor construct use. Errors were then categorised, based on the underlying cause (see discussion, Section 4.7) into one of five categories: Environmental, and a category for each of the four established

knowledge levels (see Section 4.5) of Syntax, Semantic, Schematic and Strategic errors. Environmental problems are limited to problems encountered that relate to the integrative development environment (IDE) used and any other IT related problem, e.g. accessing online resources. The observer then engaged in dialogue with the student to help them recognise why the problem had occurred and what they needed to do to resolve it. This dialogue was also recorded in note form. To provide a context to the dialogue and recorded errors, the programming modules and practical exercises are outlined in sections 4.3 and 4.4 at the Universities of St Andrews and Abertay Dundee respectively.

4.3 Object Oriented Programming at the University of St Andrews

At the University of St Andrews the programming module, Computer Science (CS1002), is mandatory for all first year students on the BSc Computer Science courses and is offered as an elective to other students. The majority of the students studying the module have no previous programming experience. The entrance requirements for the BSc Computer Science course was BBBB at Scottish 'Higher' Grade (at least one 'Higher' pass is required in a science subject). A pre-requisite of the CS1002 module is a credit pass in Standard Grade mathematics or equivalent. The module is worth 20 credits and comprises one-third of the students' studies in Semester 1. Students cover object-orientation, basic constructs and data in preparation for a more advanced programming module, Internet Programming, in Semester 2. The module is taught with continuous assessment throughout the semester contributing to 40% of the final grade. An examination set at the end of the semester constitutes the remaining 60% of the grade.

The Java programming language is used to teach programming concepts, through the Together IDE (www.borland.com). Students receive four one-hour lectures, a one-hour tutorial session and a three-hour laboratory session each week. The aim of the module is to develop students' skills in the use of modelling tools and object-oriented programming whilst introducing them to computer science and the concepts of software design and programming. The objectives of the module are to:

- Be able to design simple object-oriented (OO) models using an OO design notation and supporting software tools.
- Be able to implement an OO model in a high-level OO language using objects, classes, inheritance, arrays, conditionals and iteration.
- Be conversant with effective documentation, layout, debugging and testing.
- Be aware of a selection of current areas of interest in Computer Science.

To contextualise the observation process, problems encountered and feedback notes, the following section outlines the practical exercises undertaken

4.3.1 Practical exercises

Weekly exercises cover expression, conditionals, methods, classes, code design, iteration, arrays and inheritance. Each weekly practical typically comprises one or two substantial exercises. Each exercise is divided into sections, with the accompanying handout detailing the stages required for each section. The handouts towards the beginning of the semester are more detailed than those towards the end of the semester, when students need less guidance. In addition to writing a program, the student is required to write a document of their program design and include textual answers to pertinent questions. For example, in week 5 when the students are introduced to Classes they are given the following problem statement:

> In this practical you have been elected as the Treasurer of SAM, the *Society for Accumulation of Members*. You decide to write a simple accounting system to keep the society's books.
>
> The Society for Accumulation of Members is a new society and starts off with no money. It receives income from an SRC grant; from members' subscriptions, and from selling tickets for a dance. It spends money to produce posters advertising the dance, to hire a dance hall and to hire a band. It keeps its money in a bank account and a cash box, and has no other assets or liabilities.
>
> During the year, SAM has 12 financial transactions. (Transactions are listed with descriptions, amounts, and account). These 12 transactions are the essential *input* to SAM's accounting system. The *output* is a statement of SAM's financial position at the end of the year, like this (required output is displayed). The **Income & expenses** section shows where money has come from and where it has gone. It ends with a surplus, which is the excess of income over expenses. The **Balance sheet** shows where any remaining money is now. Notice that the

year's surplus is copied from the SURPLUS section to the LIABILITIES section. This is not a mistake; it is part of the double-entry book-keeping system, and ensures that *Total assets* and *Total liabilities* are equal. In other words, that the books balance. The accounting system consists of the Account class. Each line of **Income & Expenses** or **Balance Sheet** which has an amount is an Account object. For example, The *stationery* account records stationery expenses. Some of the accounts are totalling accounts. For example, *Total income* is the total of the three separate income accounts *Subscriptions*, *Dance tickets* and *Grant*. The accounting system is to do this totalling automatically. Each account is also *normally in credit* or *normally in debit*. The convention is that income accounts are normally in credit, balance and expense accounts are normally in debit. This simplifies the appearance of the financial statement, since otherwise (for example) income and expense accounts would have to have opposite signs, as would assets and liabilities. By following the convention, a minus sign only appears in the statement if an amount is the opposite sign of what is normally expected for that account.

Credit and debit are easily represented as positive and negative — but which is which? If you think of money as flowing from income (normally credit) accounts to balance (normally debit) accounts, and then from balance accounts to expense (normally debit) accounts, then it follows that credit is best represented as negative and debit as positive.

To implement this, each account records its balance as a signed number, with the convention that negative is credit, and positive is debit. Each account also records (with a boolean variable) whether it is normally in credit or normally in debit. When the account balance is printed, a minus is used only if the amount is not of the normal sign.

Each of the 12 transactions is balanced: it moves an amount of money from one account to another. This is implemented by subtracting (ie crediting) the amount from one account and adding (ie debiting) it to another. No money is created or lost. The total of all the account balances is always zero.

This extended problem statement is part of a larger documentation set for this practical exercise. Students are expected to write a constructor, create objects, write and use instance methods and follow object references for this practical exercise. In addition to this, students are required to write a document describing how their program works. A statement of the learning objectives precedes this description and guidance on solution design and implementation together with sample test cases follows. In all, the document spans 15 pages.

4.4 Object Oriented Programming at the University of Abertay Dundee

At the University of Abertay Dundee the programming module, Object Oriented Programming 1 (OOP1), is mandatory for all first year students on four different computing courses, and the majority have no previous programming experience. The entrance requirements for these courses are widely varied, ranging from BBBC at Scottish 'Higher' Grade on degree programmes to BC at Scottish Higher for DipHE programmes and there is no prerequisite Computing or Mathematics qualification at Higher (or A-Level). The module is taught with continuous assessment throughout the semester. The module is worth 12 credits and so constitutes one fifth of the programme of study. The module is divided into two teaching blocks, each block is five weeks in length. Students begin with basic constructs and data in preparation for a more advanced programming module covering modular abstraction and objects in Semester 2.

The Java programming language is used to teach the programming concepts, through the JCreator IDE (www.jcreator.com). Students receive one one-hour lecture and one two-hour laboratory sessions per week. The aim of the module is to enable students to develop simple programs that illustrate fundamental programming concepts. The objectives of the module are:

- Create and run programs using an integrated development environment.
- Use appropriate data types and control structures.
- Design, implement and extend objects in terms of interface, function, and data.
- Incorporate predefined objects into new programs.

Again, to put the observation process, problems encountered and feedback notes in context, the following section outlines the practical exercises undertaken.

4.4.1 Block 1

The first five weeks of Semester 1 constitute Block 1 and are used to introduce fundamental programming constructs. These constructs are introduced in a stepwise manner and cover sequence, selection (if else) and iteration (for, while and do while). With each concept that is introduced the student is given a tutorial sheet, which contains a set of exercises to enable the student to put a particular concept into practice. There are around seven short exercises in a

week. These exercises exploit a software package that simulates a robot moving around a room. The robot, robby, has a set of methods that the student can use, for example move() to move one square forwards, left() and right() to rotate on the spot, and obstacle_ahead() – a Boolean method allowing interaction with the room. To illustrate by example programming exercises using the Robot class, in week 4 a pair of the exercises presented to the students after their lecture on selection are:

- Write a program, Tut4_1.java, to make Robby follow the trail of yellow tiles present in room 4 (see Figure 4.1). This trail is a clockwise spiral and so only has right turns. Robby should stop moving on the green tile at the end of the trail.
- Write a program, Tut4_4.java, to make Robby follow the trail of yellow tiles present in room 5 (see Figure 4.2). This trail weaves from side to side, and so has both left and right turns. Robby should stop moving on the green tile at the end of the trail.

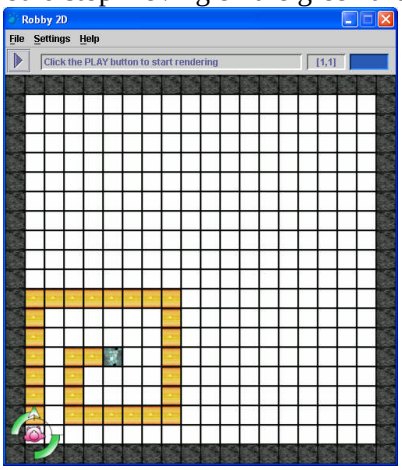


Figure 4-1 Room 4

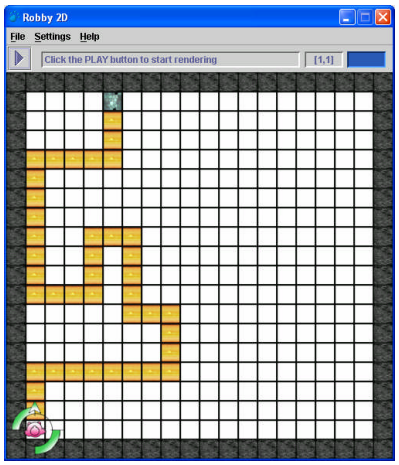


Figure 4-2 Room 5

The first example is formative and is the first question on selection. The solution requires minor modification to a program given in the lecture. The assessed exercise, Tut4_4 in turn needs a simple extension to the formative exercise. To encourage students to apply the new programming constructs learned via the robot paradigm to more general problems they are also given non-robot exercises each week. These non-robot programs are also short, clearly prescribed exercises; for example:

- Write a program Tut3_3.java where the user can input a module number and coursework marks for multiple students. The program will terminate when the user enters '-1' and display the average coursework mark for that module.

4.4.2 Block 2

Weeks 6 to 10 cover Block 2 and focus on data storage through the use of a software package that simulates food moving through a cow's stomachs. The students also make use of a GUI software package to take input from the keyboard at run-time and provide textual output. At the very end of the block, simple (void) methods are introduced in preparation for Semester 2. Examples of the questions which students are expected to complete during this block are:

- Write a program, Tut6_4.java, to make daisy the cow behave randomly:
    1. if she is hungry eat;
    2. else if she is bloated flush;
    3. else randomly choose eat or flush with equal probability.

- Write a program, Tut7_5.java, to read in 10 numbers into an array. The program should sort those numbers into ascending order, regardless of the order of input. Look through some books (!) to figure out how to do this. Note, the solution should work for any length of array (not just 10).

The first question is one of the first formative cow exercises and is highly structured in its requirements, and those requirements offer a structural guide to the program. In contrast, the second question represents one of the less detailed questions offered to students, and many students are not able to address this question without substantial tutor assistance.

4.5 Categorisation of Recorded Errors

The initial observation process, undertaken in 2002–2003, was performed with no prior assumptions of either the types of errors recorded or the underlying reasons for encountering the error. Instead, as complete a record as was feasible was recorded for each error. To devise the categories used, other schemes used for software errors and the general literature review in Chapter 2 were considered. A number of existing categorisation schemes have been developed for software errors. The most widely recognised include the IEEE Standard Classification for Software Anomalies (IEEE, 1993), Software Testing Techniques (Beizer, 1990) and the Preliminary Taxonomy of Object-Oriented Software Faults (Huffman-Hayes, 1994). A review of these schemes quickly demonstrated their limitations with respect to the categorisation of errors and problems encountered by the novice programmer. All of the above schemes have been developed for industry use, primarily to categorise the errors in

software systems. This is necessary for software development process management, allowing identification of those errors caused by poor design on the programmer's part and those arising from inconsistencies in the requirements specification agreed by the user. Categorisation of these errors is used to determine whether the user can be held accountable for changes to the software or if the responsibility for correcting the software system lies with the contracted company. Thus the fundamental goal of taking the designed metrics is inconsistent with the context considered here. Further, there is no capacity in these schemes to record the fine-scale detail of novice programmer problems nor the misconceptions that led to the error. It was therefore decided that these categorisations were not suitable for the problems recorded by the novice programmer.

A limited number of categorisation schemes exist for problems encountered by the novice programmer. Hristova et al. (2003) categorised novice programmer errors as syntactic, semantic and logic. A syntax error is defined as a mistake in the spelling, punctuation or order of words in a program. A semantic error relates to the meaning of the code, particularly a mistaken idea of how the compiler interprets the code. A logic error arises from "fallacious thinking by the programmer". Wertz (1982) categorised novice programmer errors as lexical, syntactic, semantic, teleological and conceptual. Lexical errors are defined as mainly spelling or typographical mistakes. Syntactic errors relate to deviations from the language rules. Semantic errors are typically syntactically valid but are inconsistent in their meaning or are invalid, for example attempting to divide by zero. A teleological error occurs when the program performs something that was not intended by the programmer. Finally a conceptual error arises when the source of the problem is caused by the approach or method selected by the programmer. Bental (1993) developed a categorisation scheme for novice programmer errors based on Wertz's categorisation. Bental grouped lexical and syntactic errors together into a syntactic category. Bental has also grouped semantic and teleological errors together into a semantic category. Bental retains Wertz's conceptual category and adds three new categories; style, type signatures and general incomprehension of the programming language SML. These categorisations are consistent with the type of errors recorded during the observations, and support the differentiation of syntax and semantics for program

formulation, but none provided the granularity of the model developed by Mayer (1997). Therefore it was decided that the errors would be categorised according to those knowledge levels introduced in section 2.3 derived from Mayer (1997), as described in section 4.2, together with the Environmental error category as above. The categorisation of observed errors in accordance with these knowledge levels is described in more detail in the remainder of this chapter.

4.6 Results

Figure 4-3 and Figure 4-4 show the data collected from the students at the University of St Andrews during the academic sessions 2002-2003 and 2003-2004 respectively. Likewise, Figure 4-5 and Figure 4-6 show the data collected from the students at the University of Abertay Dundee during the academic sessions 2002-2003 and 2003-2004. All data relate to problems that the students were unable to resolve without assistance from teaching staff. At the University of St Andrews there is no data for laboratory week 6 as it is Reading Week, with no taught classes, and the week 10 practical concentrated on UML diagrams so these are omitted from the graphs. The errors are presented here as a relative value, i.e. the percentage of the total errors collected in that particular week. Use of an absolute value for the number of errors would more strongly highlight differing numbers of observations made and different class sizes rather than the generic trends across both weeks and session of observation. The following figures depict those relative values over time. The categories are then discussed with reference to the observed errors.

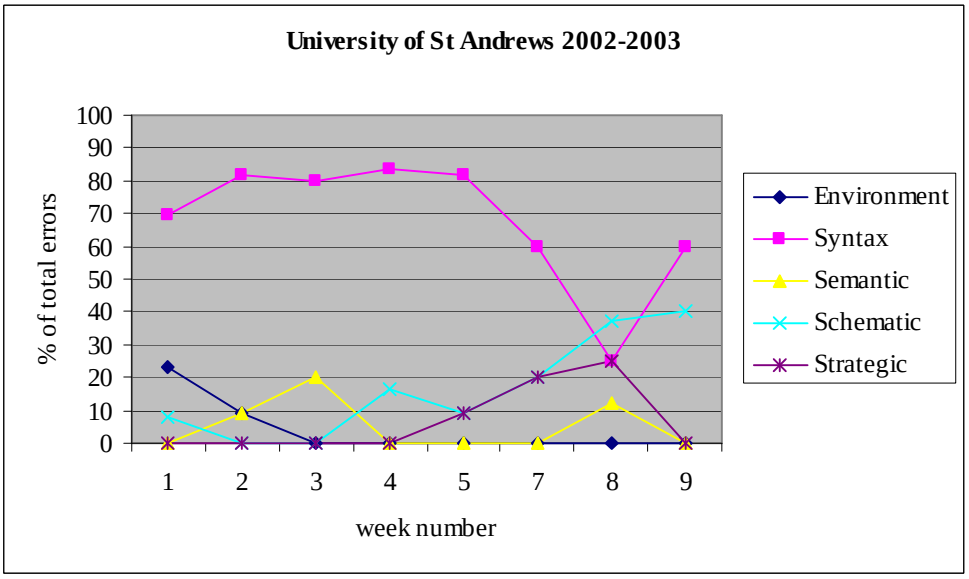


Figure 4-3 Observed errors at the University of St Andrews in session 2002-2003

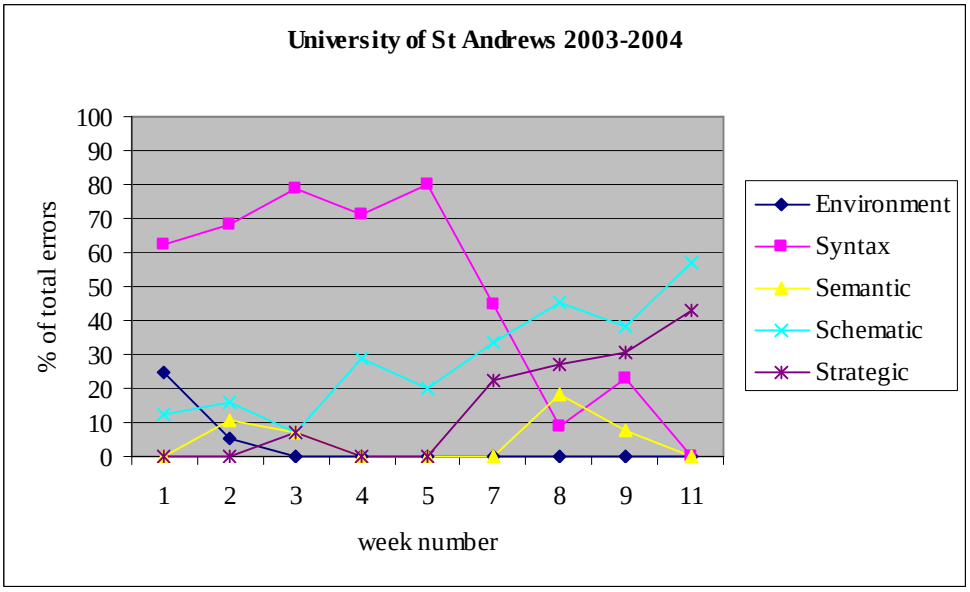


Figure 4-4 Observed errors at the University of St Andrews in session 2003-2004

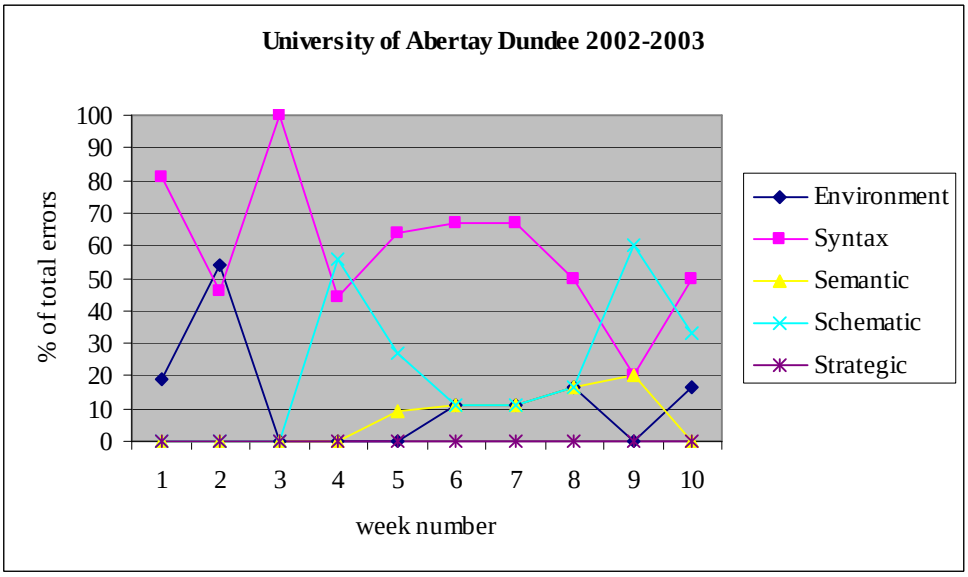


Figure 4-5 Observed errors at the University of Abertay Dundee in session 2002-2003

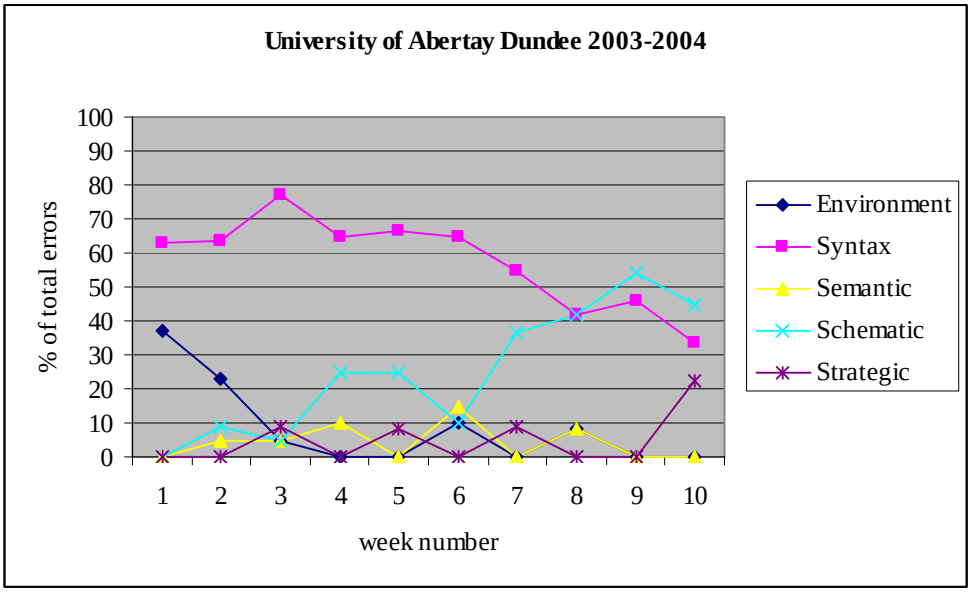


Figure 4-6 Observed errors at the University of Abertay Dundee in session 2003-2004

4.6.1 Environmental Problems

Environmental problems are generally limited to the beginning of the module, when the students are becoming familiar with a new piece of software (here the IDE). These errors

reappear in week 6 at the University of Abertay Dundee (Figures 4.3 and 4.4) because students were introduced to new teaching material at that stage in the module and this required configuration changes in the IDE. In comparison with other categorised errors, the number of environmental errors tends to be less and errors do not persist for the duration of the teaching period. Examples of common environmental problems were:

- Student requiring assistance to set up their project workspace properly
- Student failing to configure links to essential library files

Environmental errors do not impact on the overall laboratory activity: these problems are normally eliminated within two to three weeks of teaching and do not cause substantial problems with the learning process.

4.6.2 Syntactic Problems

Syntactic errors are significantly higher at the beginning of the module when the students are trying to learn the new programming language. Examples of common syntactic errors were:

- cannot resolve symbol
- possible loss of precision
- 'class' or 'interface' expected
- illegal start of expression

Syntax errors rates peak when students are introduced to the different syntactic constructs, where each new construct extends the underlying rule set. These errors tend to reduce towards the end of the module where exercises focus on exploitation of existing constructs in new ways, i.e. new program structures, rather than on new constructs per se. Notably, syntax errors persist throughout the observation period.

4.6.3 Semantic Problems

Semantic errors tend to exist throughout the module. While these errors are not common, they are observed (in the laboratory) to prove difficult for the student to identify because the student program appears to be syntactically correct and yet fails to run as expected. Further, dialogue with the student reveals that they assume the problem lies in their program structure

rather than malformed expressions on a smaller scale. Examples of common semantic errors are:

- Loop counter not being updated causing an infinite loop
- If Boolean value = Boolean value
- Adding a semi-colon to the end of an if condition or loop construct

4.6.4 Schematic Problems

Schematic errors tend to increase during the course of the module as the teaching materials explore the linking together of constructs and, as a result of the wider range of available combinations of constructs, the assessment instruments tend to require more sophisticated solutions. This requires students to make a selection from among constructs to address the question sets provided. Examples of common schematic errors are:

- Selection of the wrong construct: e.g. the use of 'for loop' instead of while loop
- Misplaced components: e.g. code that must be repeated being positioned outwith the scope of any loop

4.6.5 Strategic Problems

Strategic errors are not frequent in first year programming because most of the exercises are relatively simple and contain detailed explanations as to how a student should proceed with a solution. However, some students find the wording of the questions difficult to understand and need further help in decomposing the question into manageable chunks. Where this does occur, as in the case of semantic errors, this poses a significant problem, as the student is unable to formulate a useful interpretation of the question and subsequent outline solution. Examples of strategic problems are:

- Missing program components: e.g. the absence of a 'for loop' when this is a clear requirement of the question
- Failure to recognise key words in question statement, e.g. Write a method to return an average of …, and no method (other than the main method) is present in the program.

4.7 Discussion

The results of the observations reveal similar patterns across the different student groups. At each university and over each academic session the graphs illustrate:

i. a rapid decline in Environmental errors;

ii. a general reduction in Syntax errors over time;

iii. a general increase in Schematic errors over time;

iv. infrequent Semantic and Strategic errors that pose significant problems.

The graphs also show large differences among Syntactic, Semantic, Schematic and Strategic errors towards the end of the observational period. The differences among the five categories for the University of Abertay Dundee in the session 2002-2003 are still evident but are less pronounced, although the streaming used to create a group of students with limited programming aptitude could account for this. Importantly, in spite of differences in teaching approaches at the two universities, significant differences in the size of the exercises posed, and year-to-year variance in cohorts, two patterns are evident. First, the relative trends of each of the categories are conserved across the study groups. This indicates that there is a consistent dynamic to the learning of programming independent of the teaching model used, as described in i) to iv) above. To further support that the observations reveal at least program formulation problems that are general to many programming exercises, results of the observations for program formulation are also consistent with other studies. As noted, Jadud (2005) undertakes a comprehensive study of syntax errors in Java and the range and type of observed syntax errors are consistent with those identified here. Further, the semantic errors identified at the two host institutes have clear overlap with those errors identified by Schorsch (1995) in Pascal, where for example misplaced ‘;’ and failure to update loop counters were recognised logic errors.

Secondly, more detailed analysis of the recorded dialogue for encountered errors reveals evidence that the root cause of a given error is located in one or more levels of knowledge. For example, the reduction in the number of syntax errors that the students are unable to solve towards the end of the module indicates that the students gain competence in the application of syntactic knowledge to formulate a program and are able to interpret compiler error messages without assistance. However syntax errors persist throughout the observation period, even when no new syntax is being taught. Further analysis of the observation records

reveals that some of those (persisting) syntax errors have root causes in more sophisticated levels of knowledge, and are recorded as syntax errors based on their manifestation and not the underlying level of the lack of knowledge.

A simple example of this is in the error ‘else without if’. This is observed to occur because of a failure to include any scoping braces in a program that by indentation reflects a multiple line if (true) block, i.e. a syntax error. The same error may occur when a semi-colon is placed at the end of the ‘if statement’, i.e. ‘if (condition) **;’** - a valid line of code in itself but a semantic error regarding the role of ; in a program. Finally the error may occur when an ‘else’ exists where no ‘if’ is present in the program, and this is a structural (schematic) level problem when the student does not use the complete selection construct in the solution to the problem, i.e. the conditional element of the selection is not present.

The collective review of the observed errors indicated that manifest problems can have root causes in multiple levels of knowledge. Here, four case studies drawn directly from the observation logs are described to highlight this issue.

4.7.1 Case 1

During week four of the observations at the University of St Andrews in the academic session 2002-2003 a student asked for assistance with the following problem:

“I’m not sure how to write the code for the header of my constructor.”

In this particular case, it was clear that the student understood that they had to write a constructor and they appeared confident that they understood the purpose of a constructor. The tutor asked the student what information they were passing to the constructor and the student listed a name, credit, total and balance. The tutor then asked the student what type of things they were and the student explained that they were a string, Boolean, int and int. The tutor then explained how the header of the constructor should be written, i.e. public Account (String name, Boolean credit, int totalling, int balance). The student had not asked for help with a compilation error, nor were they confused by the running of their program. The student understood exactly what they wanted to do but needed assistance with the syntax

formulation. This is an example of a problem in the student's syntactic knowledge that has manifested itself as a strategic error (it could appear that the missing constructor meant the student hadn't understood that this was a requirement of the question).

4.7.2 Case 2

During week three of the observations at the University of Abertay Dundee in the academic session 2003-2004 a student asked for assistance with the following compiler error:

'cannot resolve symbol' on a line of code that read robby.move();

The program required the instantiation of an existing class, Robot, used for teaching that was typically named 'robby'. The tutor reviewed the student's program and identified that they had failed to declare an instance of the Robot class. This mistake could have happened for two reasons: the student had simply forgotten to declare the object, or the student did not realise that object declaration was necessary and therefore did not understand its purpose. The tutor had to establish why the error had occurred before helping the student resolve the problem. The tutor told the student that the computer did not understand what 'robby' was, whilst watching the students face to gauge their reaction. The tutor then asked the student if they knew what robby was, for example was robby a dog. The student replied that robby was a Robot. The tutor further explained that the student had to tell the computer that robby was a Robot, and that the computer knew the commands that Robots could do, such as move but could only allow robby to do that command if it knew that robby was a Robot. The tutor then showed the student the relevant lecture slide from that week's lecture, which demonstrated the objects instantiation necessary to use the robot class. Although this was a simple and common syntax error, it required reference to the lecture notes and an explanation of object declaration for the student to understand why the error had occurred and what was necessary to resolve the error. This is an example of a problem in the student's semantic (operation of memory) knowledge, which has manifested itself as a syntax error.

4.7.3 Case 3

During week four of the observations at the University of Abertay Dundee during the academic session 2002-2003 a student asked for assistance with the following compiler error:

‘else without if’

This error commonly occurs if the student has included too many ‘}’ in their ‘if statement’ and has consequently closed it too early. The tutor reviewed the student’s code and established that this error had occurred because the student had written the else statement as part of a while loop. It was clear to the tutor that this had occurred because the student had no knowledge of the semantics of either construct and did not understand which construct was necessary to solve the problem. The tutor explained to the student the difference between a ‘while loop’ and an ‘if statement’, with reference to the lecture notes. The tutor then discussed the practical exercise with the student and asked which construct (an ‘if statement’ or a while loop) was required for the program. The student was now able to recognise that an ‘if statement’ was necessary. The tutor then helped the student write the appropriate syntax for the ‘if statement’. This is an example of a problem that manifested itself as a syntax error but required an explanation of the semantics of programming constructs. The student also needed assistance at the schematic level to select which construct was appropriate for the problem and then at the syntactic level of assistance to write the required code.

4.7.4 Case 4

During week eight at the University of St Andrews in the academic session 2003-2004 a student was observed to request assistance when their program was not displaying the expected output. The tutor asked the student to run their program and show the output. The tutor then reviewed the code and identified the source of the problem: the student had written number=min instead of min=number. The tutor explained to the student that in an assignment expression, the value on the right hand side of the equals sign was assigned to the left-hand side. The student was satisfied with this and did not need any further support. This was an example of a problem that had its root causes in syntax (the student not knowing the rules of the language) although it appeared as a semantic error in the program behaviour.

4.8 Implications for Feedback

The case studies demonstrate how an error can have its root cause in a number of different knowledge levels. Additionally, they highlight the importance of the personal nature of the teaching process, where implicit signals such as body language and intonation often reveal the level of knowledge evident in the student. The aim of this work is not to replace that interpersonal facet of learning support. Here, as noted in the introductory sections of the thesis, we seek to provide learning support that is usefully informed by the context of the learner to assist students where tutor support is unavailable and to provide support which complements tutor guidance where appropriate. Therefore, since it is not possible to identify where the gap in student knowledge exists from automated program and error analysis alone, automated feedback provided by a teaching tool should provide context sensitive support for a range of possible root causes. It is important that any feedback provided on a syntax error is able to give support on errors caused by simple syntax mistakes *and* errors caused by more fundamental gaps in a student's knowledge, for example failure to recognise the difference between programming constructs. Similarly, students often ask for assistance from the class tutor with writing syntactically correct code, i.e. they know exactly what construct they need to use, and how they want to use it, but they need reminding as to the correct syntax. Therefore a teaching support tool has to be able to provide feedback to students who are missing essential constructs that must include not only an explanation as to why that construct is needed but also the required syntax.

These observations have been used to inform the supportive feedback required by a teaching tool at the program formulation stage. Common syntax errors, not only throughout the module but also at specific stages in the module, for example the introduction of a new programming concept, are used to identify which errors students need most assistance to resolve. The recording of the dialogue with students also informs the necessary level of feedback and appropriate language (content) that students need to solve particular problems. Similarly, the identification of common semantic errors provides a clear contribution to program formulation support, since provision to identify those errors may then be

incorporated into the analysis of a teaching support tool and, again, the recorded dialogue used as a foundation to develop useful content to use in feedback.

Observations of the underlying cause of schematic and strategic problems have helped to inform the problem formulation support of a teaching support tool. Identification of the common misconceptions that students have with particular concepts has revealed the need to include reference to syntactic form in the case of Schematic Support. For Strategic Support, the guidance provided to individual students at key program development stages for specific programming exercises may be used to help guide students in planning a solution. The required text to assist a student in formulating a plan to solve a specific problem may be posed in terms derived from dialogue with students generally and from that particular problem.

It may be seen from these observations that students encounter problems across all four levels of learning: Syntactic; Semantic; Schematic and Strategic. These errors can manifest themselves in many different ways and a support tool should provide support for these different levels however they appear. The observations at the two different universities demonstrate that novice programmers encounter similar problems regardless of the learning framework. Feedback on syntax errors should include guidance on schematic and syntax rules. Students encounter common semantic errors that can be detected through program analysis. Feedback on missing key constructs should include guidance on syntax, identifying appropriate constructs. Finally, students require guidance on problem decomposition to form manageable chunks at key program development points, i.e. strategic help.

Chapter 5 presents an implementation of the conceptual framework devised in Chapter 3, which incorporates both the required analyses, and feedback as informed by these observational studies. This implementation, termed SNOOPIE, is evaluated in Chapter 6.

Chapter 5 SNOOPIE

5.1 Overview

SNOOPIE is a model instantiation of the framework proposed in Chapter 3, and exploits the observational study described in Chapter 4. This chapter describes the technological implementation of SNOOPIE, which is detailed in three sections, and phased over two versions. Following on directly from the observational study of the academic year 2003 to 2004, support for program formulation is implemented for the academic year 2004 to 2005, hereafter referred to as Version 1 of SNOOPIE as described in Section 5.2 and Section 5.3. To ease the description of the implementation, the detail of the operation is divided into Syntactic and Semantic Support. Version 2 of SNOOPIE, implemented for the academic year 2005 to 2006, builds on Version 1 to include support for both program and problem formulation. Version 2 implementation is detailed in Section 5.4.

A fundamental consideration with any computer-based learning support tool is the form of interaction between tool and student. To aid in the identification of the most appropriate form of interaction for this thesis the literature review of existing support tools in Chapter 2 was again exploited. Of the many tools reviewed, two were observed to meet most of the requirements (Table 3-2): IVC and CAP, and each of these tools has a fundamentally different form of dialogue with the student. IVC is freeform allowing students to seek guidance by asking questions on problems related to their programming problems in their own words and the tutoring system, through a bespoke and powerful natural language processing application, is able to answer those questions. In contrast, CAP undertakes analyses on the existing code and then provides instructive feedback on any recognised problems with the code. In principle, IVC could offer an extremely rich mode of interaction. However, the dialogue is limited to general questions relating to programming concepts rather than specific exercises. This limitation is down to the difficulties inherent in natural processing rather than any decision in what is needed from an educational perspective (K Taylor, pers. comm. 28th April 2006).

Here, the aim is to explore the contribution of both program and problem formulation to the learning process and the challenges associated with any natural language processing scheme were recognised at an early stage as likely to impede this exploration within the timeframe available. Consequently, the implementation follows that of CAP in terms of analyses and instructive feedback. This feedback presented in SNOOPIE can, and indeed does, contain a mix of direct instruction on code and/ or constructs to use together with hints to encourage reflection on that feedback as appropriate to the question and problem encountered.

Program formulation support extends to all those errors where textual provision by SNOOPIE was feasible. Only three syntax errors were not addressed properly by SNOOPIE. The first two relate to links to the external environment, i.e. the saved filename and package naming. In the implementation described here, these were addressed in part by the inclusion of specific error texts that extend only to a subset of all possible error messages, i.e. bespoke error trapping of filenames and packages. Note that this limitation has been addressed in the existing implementation of the academic year 2006 to 2007. The only error that is not dealt with is ‘null pointer exception’. This arises from memory mismanagement and students lack the vocabulary to interpret any relevant error message. SNOOPIE advises students to seek tutor advice to resolve this error. For semantic errors, all observed errors are supported by SNOOPIE with one exception: the underlying and flawed assumption that assignment operates from left to right was observed occasionally. However, without an account of variable roles (Johnson, 2006) it is not possible to contribute meaning generally to variables in themselves. Of course, where context defines the role, e.g. a ‘for loop’ counter variable, the role of that variable is factored into the semantic analysis (see examples below). Problem formulation is implemented on a question-by-question basis. For each supported question, the module tutor, based on essential components and their inter-relationships, identifies a defined list of program requirements. This list is structured to promote a stepwise strategy in program development, where students are provided with feedback to guide them from one program development step to the next. Feedback is tailored to relate to the specific tutorial question, again drawing on the previous observational study, which helps those students who

are unable to interpret the question successfully and translate that interpretation into the required solution. Note that variant paths in solution development may be accommodated where appropriate.

This chapter does not attempt to evaluate the effectiveness of SNOOPIE in the class context. Chapter 6 details this evaluation, including the metrics used and the key findings.

5.2 Version 1: Syntax Support

The first stage in the conceptual framework of Chapter 3 is Syntax Support. SNOOPIE captures the standard error output from the Java Compiler and converts each standard message into a revised structure allowing the integration of supplementary information to enhance the clarity of that message. To illustrate the aim of this stage of program formulation support, three common errors are described here. First, the missing ';' and second, the error 'class or interface expected' are simple errors which are not well explained by the Java Compiler but are readily explained with additional text. Third, a typical error arising from a mismatch between method header and method invocation is provided to demonstrate the more complex error support offered by SNOOPIE.

The Java Compiler error message shown in **Figure 5-1** is replaced with the extended text shown in **Figure 5-2:**

```
C:\deployment\testProg.java:4: ';' expected
              for (i=0;i<2;i++)
                           ^
```

Figure 5-1 Original Java Compiler Error Example 1

```
C:\deployment\testProg.java:4: ';' expected
You may have forgotten to end a line with a ;  (semicolon).
If the line indicated by the error has a semicolon check the line above.
                    for (i=0;i<2;i++)
                                 ^
```

Figure 5-2 Extended Java Compiler Error Example 1

Notably, the enhanced message indicates the (almost certain) location of the error, i.e. the line above that reported.

Similarly, the Java Compiler message shown in Figure 5-3 is extended with the additional text shown in Figure 5-4.

```
C:\deployment\testProg.java:6: 'class' or 'interface' expected
           }}}
              ^
```

Figure 5-3 Original Java Compiler Error Example 2

```
C:\deployment\testProg.java:6: 'class' or 'interface' expected
You may have inserted an extra closing brace }
This is most likely at the end of the program
Make sure all { have a matching } and the } is in the right place.
       }}}
          ^
```

Figure 5-4 Extended Java Compiler Error Example 2

Here, the extension of the information provided makes clear an otherwise obtuse message. Again, through the observation process, it was possible to identify the most likely cause of this error. Finally, the Java Compiler message shown in Figure 5-5 is extended with the text shown in Figure 5-6.

```
C:\deployment\testProg.java:5: add(int,int) in testProg cannot be applied to
(int,int,int)
               add(2,3,4);
                  ^
```

Figure 5-5 Original Java Compiler Error Example 3

```
C:\deployment\testProg.java:5: add(int,int) in testProg cannot be applied to
(int,int,int)
The number of types and/ or arguments must match the number and types of the
parameters. Your parameter list in the method header for add is (int, int). Your
argument list in the method call is (int, int, int).
The numbers and/ or types do not match.
                  add(2,3,4);
                    ^
```

Figure 5-6 Extended Java Compiler Error Example 3

Here, the message is made more explicit and attention is drawn to the underpinning concepts of type, number of parameters and arguments matching and the (initially difficult) terminology of arguments and parameters.

To effect this error message enhancement, the program operation is divided into three distinct phases. First, the error messages emerging from the Java Compiler are captured as an unstructured list of lines of text. Second, that unstructured list is converted into a data structure where each data element represents an individual error, including all related output fields. Finally, each error (data element) is converted into its enhanced form using an external database of extended descriptions of each error. These revised messages are displayed for the novice programmer in place of the standard Java Compiler messages. To facilitate description and highlight the coupling between each phase, phases are described in terms of required input, outline of processing undertaken and defined output, i.e. input for the next stage. Where necessary, these are supported by additional narrative.

5.2.1 Error message capture

This phase compiles the currently active Java file and captures any generated errors. These errors are stored as an unstructured list of lines of text, implemented as a Java ArrayList named OutputList, derived directly from the output of the Java Compiler. An existing software package, errout.exe (URL: http://www.horstmann.com/corejava), designed to both execute a DOS command and capture the output as a Java Stream, is exploited. Here the

DOS command is the Java Compiler and the Stream contains the compiler output, stored in OutputList. Table 5-1 summarises these steps.

Table 5-1 Summary operation of Error message capture phase

Input	Current filename
Process	Invoke compiler using errout.exe with filename While not end of compiler output Add line to OutputList
Output	OutputList

5.2.2 Message pre-processing

The messages are extracted from the unstructured list, OutputList, into an ArrayList of ErrorMessage elements that clearly define the line number, error code, error message and '^' location, i.e. the position of the error in the line of code, for each error. These lines of text are structured into the data structure as shown in Table 5-2.

Table 5-2 Data Structure Elements of Error Message

ErrorMessage	
String File	Name of file containing error
int LineNo	Line in file on which error occurs
String Line	Line of code containing error
int Position	Position of ^ indicator in line of code
String Message	Compiler error message
String Found	Type and name of symbol found, in format type, name
String Required	Type and name of symbol required, in format type, name
String Symbol	Symbol causing error, in format type, name
String Location	Class location of symbol

This is facilitated by iterating through each line in the OutputList data structure, analysing the line content, and allocating the different component parts to the fields in the ErrorMessage. Note that different error messages have different content, e.g. only ‘cannot find symbol’ errors have ‘found’ and ‘required’ fields, and so each error is distributed over different numbers of lines of original output, and these are accommodated in the ErrorMessage elements ‘Found’ and ‘Required’ (see Table 5-2). There exists a small number of variations on the composite fields of error messages, and each variant is accommodated here. The algorithm is outlined in Table 5-3. The line ‘total errors’ indicates that the loop is at the end of the data structure. The program builds up each line into the ArrayList. If the data structure size is 0, no errors were found.

Table 5-3 Summary operation of Error message pre-processing phase

Input	OutputList
Process	If OutputList size is 0, no errors - return null Create an error data element, e For each line of the OutputList If the line does not contain the text ‘total errors’ must be start of an error description. Break (first) line into filename, line number and error message Set e.File to filename, e.LineNo to line number and e.Message to the error message Get next line, ln If ln contains “symbol” Set e.Symbol to symbol error, type and name If next line contains “location” Set e.Location to symbol error location, type and name If ln contains “found” Set e.Found to found type and name Get next line Set e.Required to required type and name Get next line, ln Set e.Line to ln Identify position of error ^, p Set e.Position to p Add data element e to OutputList.
Output	ArrayList of structured errors detailing filename, line number, error message and ^ position.

5.2.3 Message Repackaging

Each individual error in the ArrayList created in Section 5.2.2 is extracted and pattern matched based on the original Java Compiler error message against a list of pre-written extended errors stored in an external data file, Error_mess.dat, to allow easy updating (see **Table 5-4**). The records in the file are structured into an ID used for data logging, the original error message used for pattern matching, and the extended error message. If a match is found between the original error message and the original error elements in the records within the file, the Java Compiler error is extended with the new, enhanced details. The repackaged list (see **Table 5-5**) of extended errors is then displayed on the screen

Table 5-4 ArrayList Elements of Generic Extended Error Messages

Error_mess.dat Data Structure	
Error ID	Error code for logging purposes that identifies the type of error.
Original error	Original error message as returned by the Compiler
New extended error message	Extended error message text
Delimiter	Dashed line to separate each error

Table 5-5 describes the elements of the repackaged error messages, contained in a data structure called NewErr. This is the data structure that is used to store the information that is presented to the novice, i.e. the original and extended syntax errors.

Table 5-5 ArrayList Elements of Repackaged Error Messages

NewErr (objects of type RevisedError)	
String File	Name of file containing error
int code	Unique identifier for each error message
String Line	Line of code containing error
int LineNo	Line in file on which error occurs
int Position	Position of ^ indicator in line of code
String old message	Original Compiler error message
String new message	Extended error message text
String Found	Type and name of symbol found, in format type, name

String Required	Type and name of symbol required, in format type, name
String Symbol	Symbol causing error, in format type, name
String delimiter	Line break to separate each error message

Table 5-6 describes the repackaging of the ArrayList elements to include the extended error message.

Table 5-6 Summary Operation of Error Message Repackaging Phase

Input	oldErr: list of structured errors as per 5.2.2 errorMess.dat: file containing revised errors indexed by Compiler error message

Process	Read in errorMess.dat contents into data structure messageMap.
	Create a new list for new errors, 'newerr', containing objects of class RevisedError
	For each element e in oldErr
	Search through messageMap seeking match on Original Error field
	If (e.Message equals messageMap.originalError)
	Construct a new error, n
	n.File set to o.File
	n.code, set to messageMap.errorID
	n.Line, set to o.Line
	n.LineNo, set to o.LineNo
	n.Position, set to o.Position
	n.oldMessage, set to o.Message
	n.newMessage, set to messageMap.extendedErrorMessage
	n.found, set to o.found
	n.required, set to o.required
	n.symbol, set to o.symbol
	n.delimiter, set to messageMap.delimiter
	Else
	Again, step through this field by field
	Add the original structured error to the new ArrayList with null entries in place of extended error description
	Add n to list of new errors, newerr
	Loop to next error.

Output	Extended error messages, NewErr

5.3 Version 1: Semantic analysis

The second stage in the architectural framework of Chapter 3 is Semantic Support. SNOOPIE progresses seamlessly to this stage if there are no syntax errors. Using the code parsed by stage 1, SNOOPIE checks the code for pre-defined constructs with which students are known to make semantic mistakes and checks each construct for the range of observed errors. If an error is found, this is returned to the student as a warning with an explanation. To illustrate the aim of this stage of program formulation support, four common semantic errors are described here. First, the superfluous semicolon at the end of a 'while loop' and second, a single equals (assignment) used in the 'if condition' of two Boolean values are two simple semantic errors which are not picked up by the Compiler but readily prevented with a warning (Figure 5-7 and Figure 5-8 respectively). Semantic analysis also exploits specific comments associated with 'for loops': if a student has used comments indicating how many times they want their code to repeat, the analysis ensures that the existing implementation does indeed repeat the intended number of times (Figure 5-9). In the final example, the failure to update a variable associated with a 'while loop' condition is checked for (Figure 5-10).

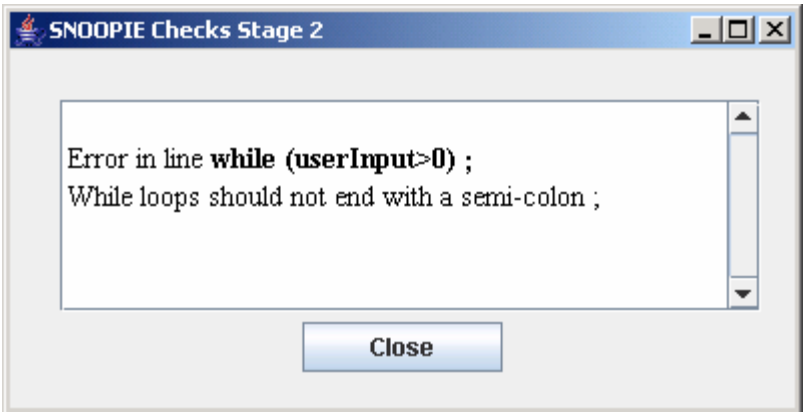


Figure 5-7 Message Warning Of Misplaced Semi-Colon

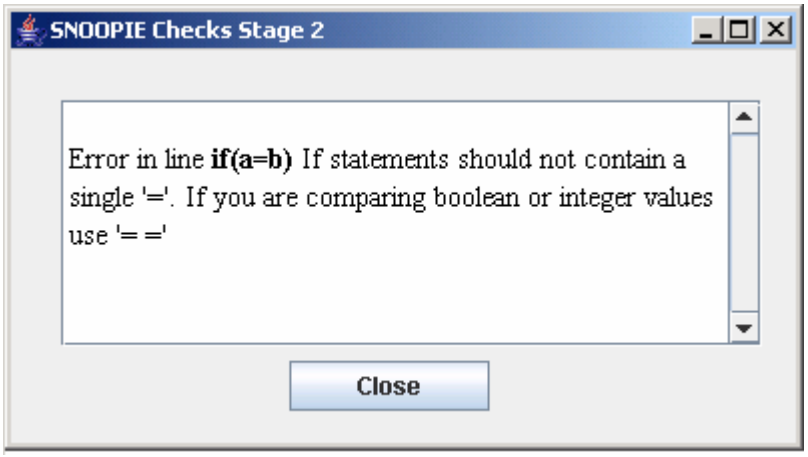


Figure 5-8 Message Warning of Single Equals in Boolean Expression

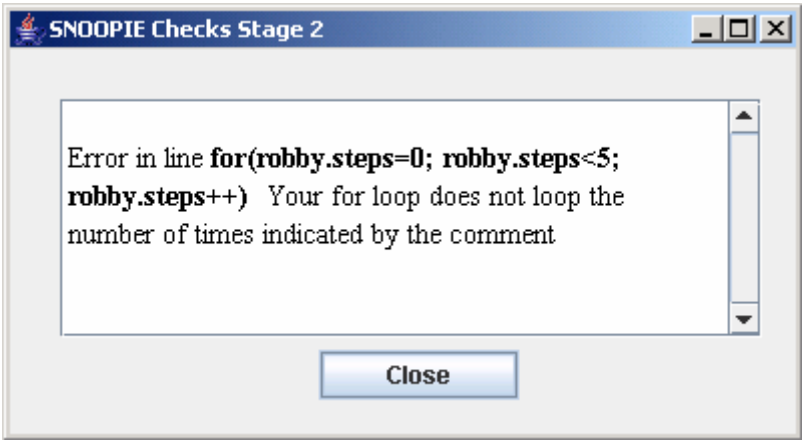


Figure 5-9 Message Warning of Incorrect 'for loop' Iterations

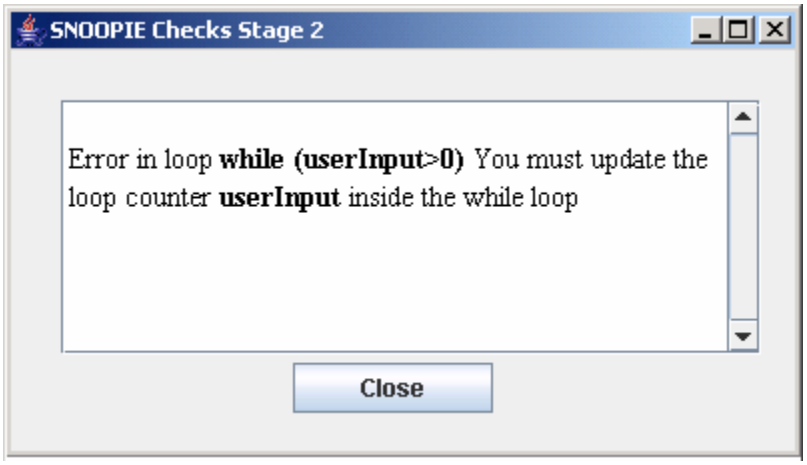


Figure 5-10 Message Warning of 'while loop' Counter Not Updating

The Semantic Support is implemented in two stages. First, the student code is restructured so that each logical line of code is on a single line. Specifically, where multiple program statements exist on one line, delimited by ';', these are placed on individual lines. Where opening and closing braces exist on lines with other code statements these are also separated. Where comments are interspersed with code, these are separated out so that code and comments exist on separate lines. Finally, superfluous white space is removed to collapse the lines of code to only those containing code and comments. This process greatly eases the actual semantic analyses since the beginning and end of constructs may be more readily identified. Second, the restructured code is analysed for the presence of concepts for which semantic error checks exist. Where such constructs exist in the code, the relevant tests are carried out. Tests exist for 'if statements', 'for loops', 'while' and 'do while' loops and 'switch statements'. The overall operation of the Semantic Support stage is outlined in Table 5-7 to present detail on each check would be onerous and repetitive. Much of the semantic analysis undertaken relates to the identification of the scope of a construct, variable updates and positions of semicolons. For explanatory purposes, detail on the operational analysis of the 'for loop' is provided in Table 5-8, as this encapsulates both the fundamental aspects of the construct analysis generally and the majority of the forms of analyses implemented.

Table 5-7 Summary Operation of Semantic Error Checks

Input	Student code in an ArrayList, studcode
Process	Create list for logical lines of code, logicCode For each line of studcode Decompose line into one or more logical lines Logical lines are delimited by statement-ending semicolons and scoping braces, { }. Comments are placed on lines separate to code Trim superfluous whitespace Add each logical line to logicCode Loop to the next line of studcode For each line of logicCode Check for construct usage that can be checked for semantic error If a construct is found Perform analysis on the construct If construct failed analysis Return warning message Loop to the next line of code
Output	Warning message

Table 5-8 describes the range of support in place for the use of the ‘for loop’ construct. The three tests implemented are as follows. First, a misplaced semicolon existing at the end of the ‘for loop’ statement is checked for. Second, the updating of the counter variable in the loop body, while valid Java and a useful facility generally, is not part of the set of exercises to which SNOOPIE is linked and so students are advised against this practice. Finally, where a comment exists on the line preceding a ‘for loop’ that reveals the number of intended iterations, the ‘for loop’ is checked for the actual iterations and any mismatch is highlighted.

Table 5-8 Example of 'for loop' Semantic Analysis

Input:	logicCode
Process:	get current line of logicCode for loop Get previous line of code (it might include a potential comment on intended looping) If end of current line is ';' Return warning: 'for loop' should not end in ';' Else Identify counter variable (lhs of first=) If next line is { 'for loop' extends over multiple lines Identify end of for loop, i.e. matching closing } Else 'for loop' extends over next line only For each line within beginning and end of for loop If counter is updated Return warning: do not update loop counter End if else If previous line is comment defining number of intended iterations Identify number of intended iterations, x Evaluate number of actual iterations If intended!= actual Return warning: 'for loop' does not repeat x times

5.4 Version 2

Stage 3 of the conceptual framework, i.e. support with problem formulation, is a much larger software framework to implement as it requires a more complex level of support than program formulation. There exists a suite of program analyses for each supported tutorial question and a set of corresponding feedback text that can be returned to the user depending on the results of that analysis. For each supported question, the analyses are performed on a priority basis, i.e. they account for the stepwise manner(s) that would be most logical to proceed through the exercise; e.g. it would be logical to assume that after writing the class definition a student would write the main method header. The next logical step would then be to declare any objects to be used as specified in the question, and so on. It is recognised that there may be many 'correct' approaches to solve a given problem and the priority analyses are structured to allow multiple approaches towards a solution to be supported. Moreover if a

student uses support to develop a solution with no clear intent on their part as to the specific approach, i.e. they have not been able to properly formulate a solution of their own, they will be guided towards one particular solution. Support for multiple approaches to a question is best explained by example.

For a given question, SNOOPIE might recognise that either a ‘for loop’ or a ‘while loop’ are appropriate for a given exercise and if a student submits a program to SNOOPIE with a ‘for loop’, SNOOPIE would help them develop a solution to that problem using ‘for loops’. However if a student submitted a program to SNOOPIE that already had a ‘while loop’ in it, SNOOPIE would provide further guidance based on a solution that used a ‘while loop’. If a student submitted a program that did not contain a loop, SNOOPIE would guide the student towards selecting the most appropriate construct for that particular exercise as informed by the pedagogic intent of that exercise. Incorporated into the feedback are links to pertinent slides to provide further guidance allowing links to any appropriate taught material. SNOOPIE might suggest that the ‘for loop’ should iterate 5 times as indicated by the question and direct the student towards some slides that demonstrate a generic example that explains iteration in for loops to help the student identify appropriate syntax.

Java2XML (Badros, 2000) is an existing, third party tool that has been utilised to enable detailed analyses of the student’s program. Java2XML is a Java archive utility that parses a syntactically correct program and creates an Extensible Markup Language(XML) file. A sample of the XML is provided in Figure 5-11. The utility decomposes the code provided into a tagged structure, representing the program components. Methods are factored into name, type, parameters and body. Method bodies comprise statements where each statement is factored into its constituent parts. For example, an ‘if statement’ is decomposed into the ‘if condition’, ‘true block and ‘false block’. This detailed program representation may support a powerful range of analyses to implement a wide range of priority checks. For housekeeping, the XML file is deleted by SNOOPIE after it has completed the analyses.

```
import SE111aClasses.*;
public class Tut1_1{
  public static void main (String[] args)
  {
       Robot robby =new Robot();
```

```
- <!--

 Generated by Java2XML http://java2xml.dev.java.net/

  --> 

- <java-source-program>

  - <java-class-file name="Tut1_1.java">

      <import module="SE111aClasses.*" />

    - <class name="Tut1_1" visibility="public">

      - <method name="main" visibility="public" static="true">

          <type name="void" primitive="true" />

        - <formal-arguments>

          - <formal-argument name="args">

              <type name="String" dimensions="1" />

            </formal-argument>

          </formal-arguments>

        - <block>

          - <local-variable name="robby">

              <type name="Robot" />

            - <new>

                <type name="Robot" />

                <arguments />

              </new>

            </local-variable>
```

Figure 5-11 Fragment of XML generated by Java2XML

Using the Java Document Object Model (DOM) the XML representation of the student's code is parsed for priority failures. These XML priority-based checks are question-specific, although there is an overall structure to the operation of the checking procedure, reflected in the summary Table 5-9.

Table 5-9 Summary of Priority Checks of XML Represented Student Program

Input	XML representation of syntactically correct student program, filename
Process	Identify filename (ignoring case and seeking key identifiers in that name) If the filename is supported Check each tutorial exercise-specific priority in turn to see if it has been met. If program fails a priority Return feedback for the appropriate priority.
Output	Guidance and link to relevant PowerPoint slides

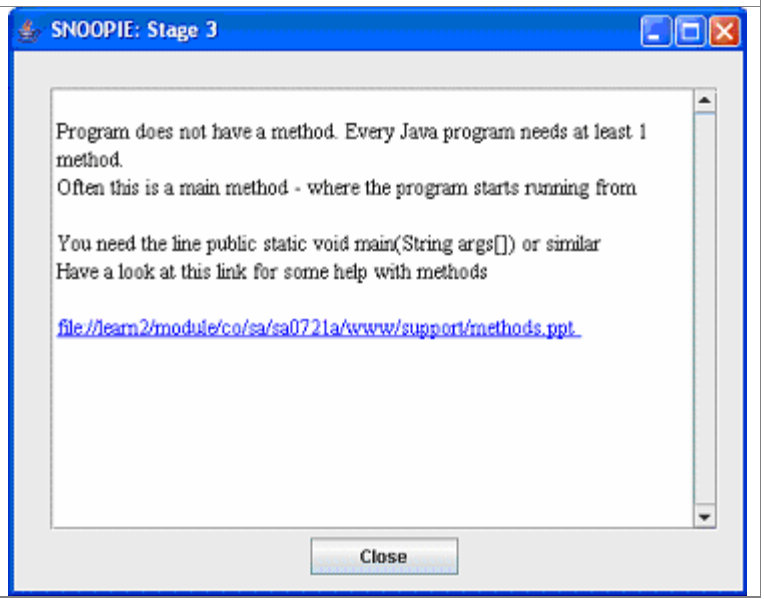
Three different but interrelated groups of analyses support problem formulation:

1) The presence of key components, for example that a 'for loop' is present in the program;
2) The correct use of those key components, for example that the 'for loop' repeats five times;
3) The correct interrelations of components, for example that the 'for loop' contains an 'if … else' statement.

As with Syntactic and Semantic Support, the description of the form of support offered is best explained by example. Here three fully worked through solutions to identified questions within the range of tutorial questions supported are offered to highlight the relationship between student code, priority-based analyses and feedback provided. For each case study, the question is detailed together with the list of priority checks that relate to that question. The first, Table 5-10, shows the level of problem formulation support that SNOOPIE can provide in its fullest, and in this case the example is an early, formative exercise on for loops. The second case study, Table 5-11, shows provision of support for a larger, challenging question, which includes variant paths of development. In both cases, the level of support provided can vary by removing or including priority checks. The first case study is highly prescriptive whereas the second case study, even though there are more priorities stated, offers only a skeleton of the required solution. This flexibility allows for summative exercises where the intent is to provide limited assistance but whilst offering some key structural guidance. The final case study, Table 5-12, reveals the process whereby a given

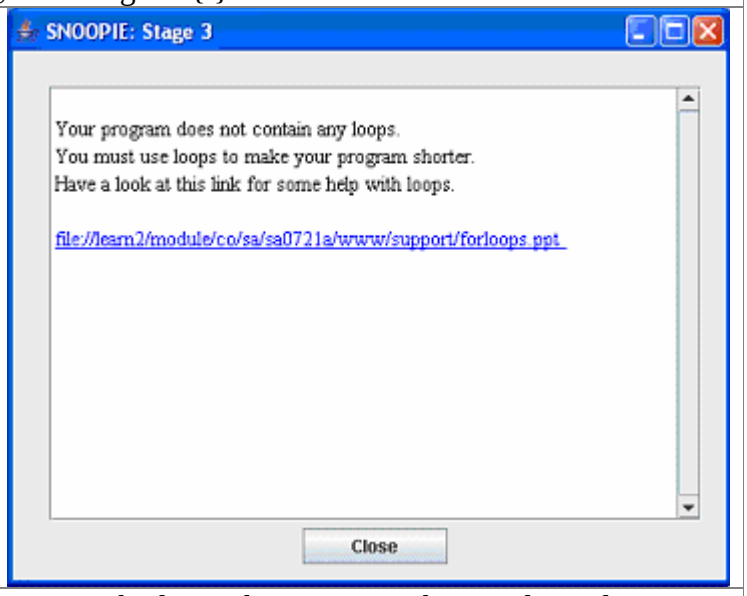
question is translated into a set of priorities. Key to this process, and given explicit treatment here, is the consideration by the module tutor setting the question of the desired stepwise development. It is from this consideration that the specific priorities and the associated feedback for each are derived. Examples are given whereby the priority-feedback mechanism provides both explicit code structures and hints towards inclusions of code structures, as dependent on both existing code and stage of development. This example has been selected to again highlight the capacity of the approach to support multiple problem solution styles. Within the full list of priorities, these are highlighted and the branching of development paths shown.

Table 5-10 Case 1. Example of Feedback from Stage 3 Analyses

<table>
<tr><td colspan="2">Question: Write a program, TimesTables.java, using nested for loops to display the 1,2,3 and 4 times tables.</td></tr>
<tr><td colspan="2">Priority 1. Main method must exist
Priority 2. Need 2 for loops
Priority 3. Loops should be nested
Priority 4. 1 ‘for loop’ should execute 4 times, the other either 10 or 12 times
Priority 5. Inner ‘for loop’ should contain some output statement
Priority 6. Inner ‘for loop’ should contain * for calculation</td></tr>
<tr><td><pre>public class TimesTables{
}</pre></td><td> 
</td></tr>
</table>

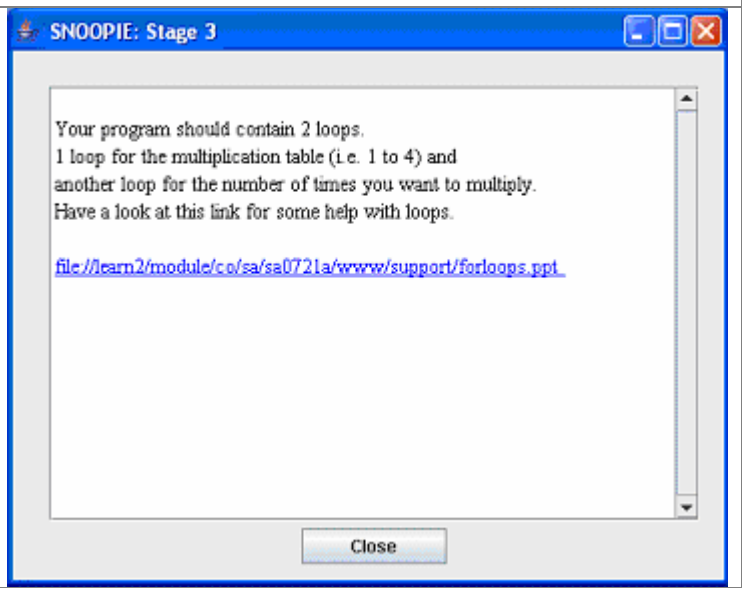
In the above example it is evident that the student has either forgotten the syntax for a main method or has no idea how to progress with their program beyond a class definition. Feedback provided reminds the student that they need to include a main method in their program with the syntax of the main method. The feedback includes a link to a PowerPoint file that explains the scope and purpose of the main method, including the { }.

```
public class TimesTables{

public static void main
(String args[]){
}
}
```



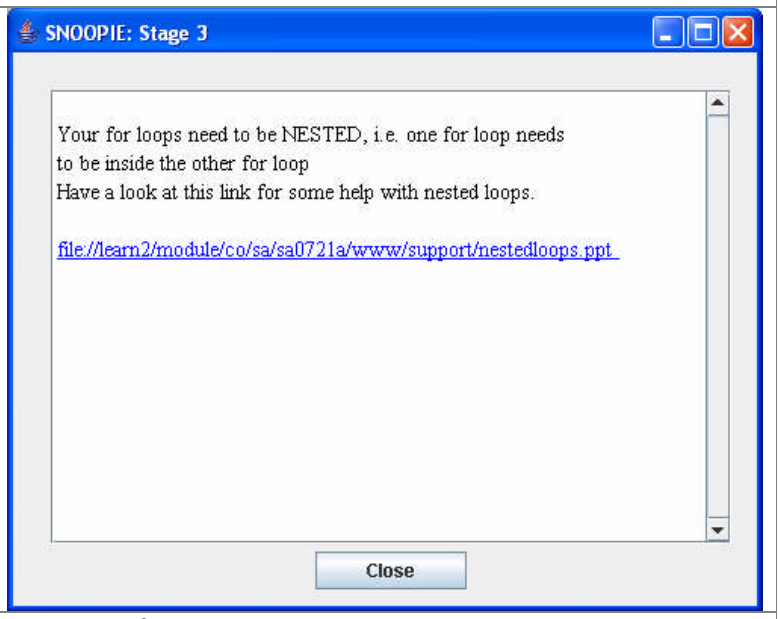
The student has successfully written a main method. For this question, the next logical step in the development of a solution would be to write a ‘for loop’. The feedback reminds the student (as the tutorial question specifies) that this solution requires for loops. If the student needs help with understanding ‘for loop’ or writing the syntax, the link to a PowerPoint file will help them.

```
public class TimesTables{
public static void main
(String args[]){

for(int i=0; i<=4; i++)
{

}

}
}
```



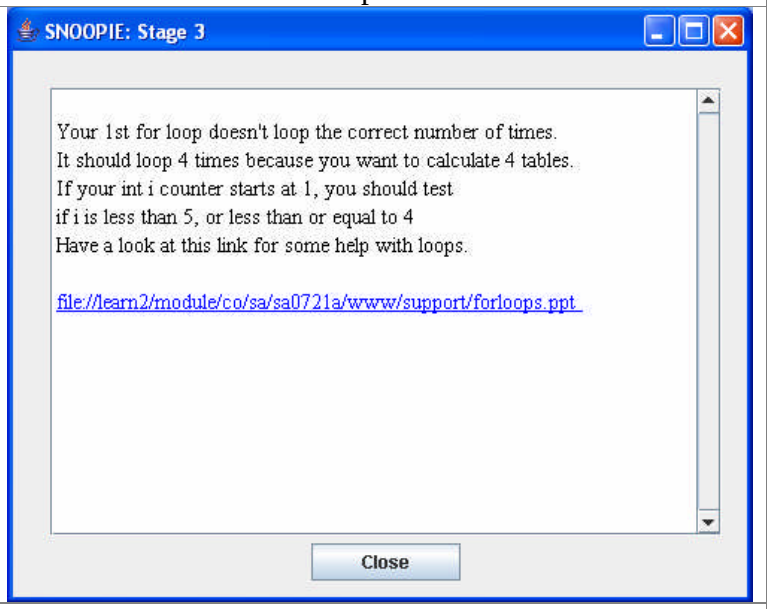
Here, the student has successfully written the skeleton of a ‘for loop’. The feedback now indicates that the solution requires two for loops and explains why two ‘for loops’ are necessary.

```
public class TimesTables{
public static void main
(String args[]){

for(int i=0; i<=4; i++)
{

}
for (int j = 1; j<=10;
j++)
{

}

}
}
```



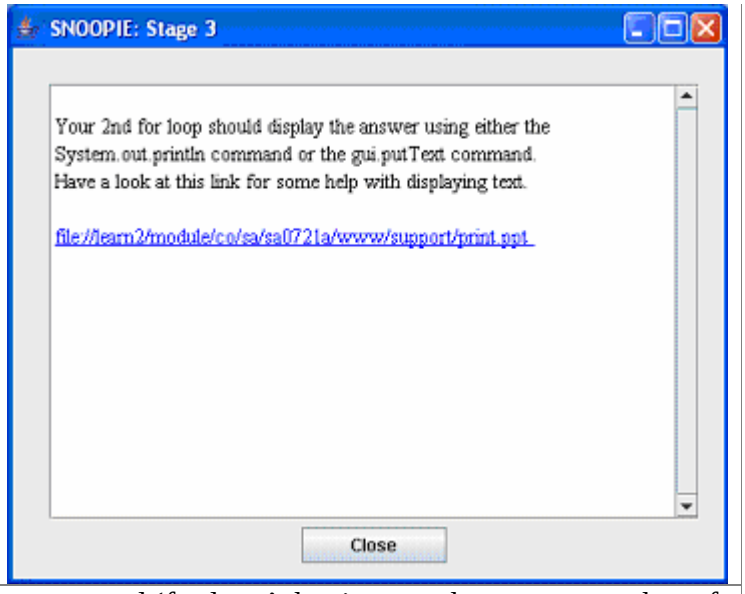
Here the student has successfully written two ‘for loops’, but the loops are not nested. SNOOPIE provides a link to more information on nested ‘for loops’.

```
public class TimesTables{
public static void main
(String args[]){

for(int i=0; i<=4; i++)
{
      for (int j = 1;
      j<=10; j++)
      {

      }
}

}
}
```



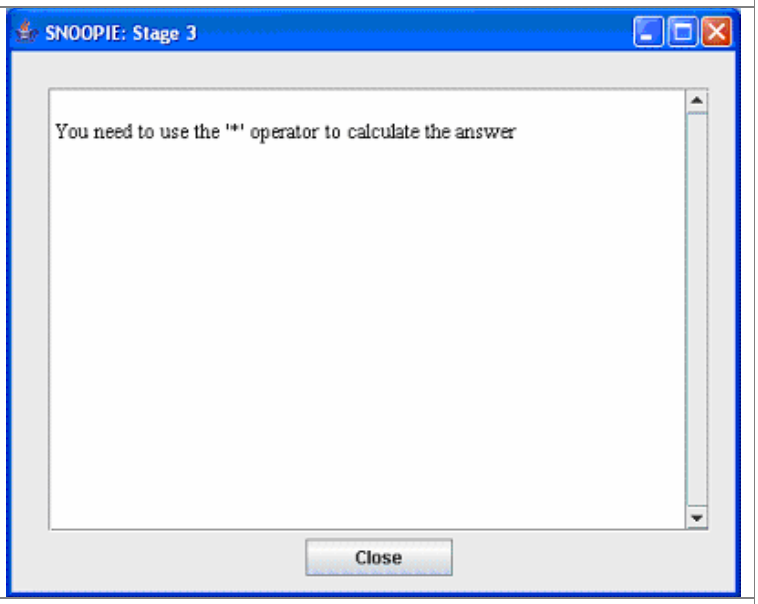
Now that the student has successfully nested the ‘for loops’, SNOOPIE identifies that the outer ‘for loop’ does not iterate the correct number of times. The feedback informs the student how many times the outer loop should repeat and how they can do this with the initial and conditional statements of their ‘for loops’. Again, SNOOPIE provides a link to a slide that explains the components of the ‘for loop’ to those students who need more assistance. Note, the other loop is also checked.

```
public class TimesTables{
public static void main
(String args[]){

for(int i=1; i<=4; i++)
{
      for (int j = 1;
      j<=10; j++)
      {
      }
}
}
}
```



Here, the student has successfully written a nested ‘for loop’ that iterates the correct number of times but the inner ‘for loop’ does not contain a print statement. The feedback informs the student that they need to include a print statement inside their second loop. Notice that the feedback does not include the full syntax required. If the student is confused with the syntax, they can view the PowerPoint file suggested.

```
public class TimesTables{
public static void main
(String args[]){

for(int i=1; i<=4; i++)
{
      for (int j = 1;
j<=10; j++)
      {
      System.out.println(
i +" times " + j + "=");
      }

}
}}
```



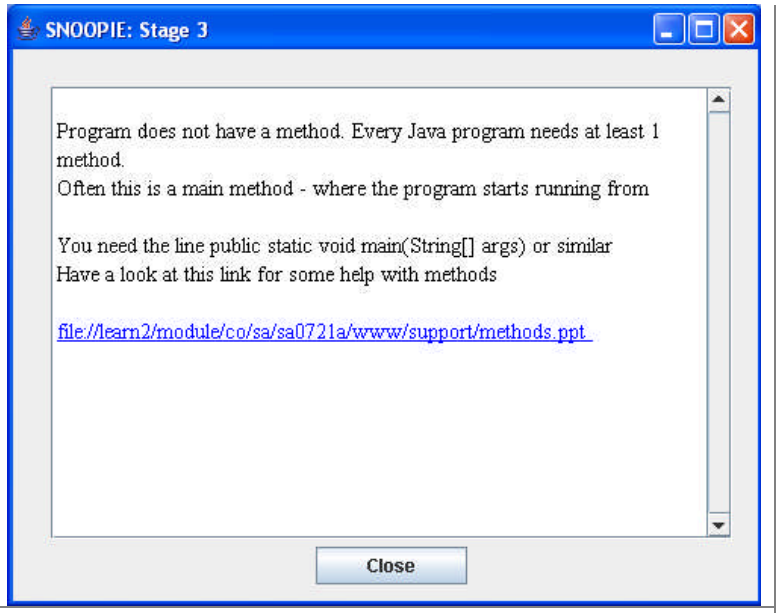
Here, the student has successfully written a print statement inside the inner ‘’for loop’. SNOOPIE has not found the multiplication operator inside the inner ‘for loop’ necessary to perform the required calculation. SNOOPIE simply reminds the student that they need this operator to calculate the answer.

The above table demonstrates the guidance that SNOOPIE could offer through such an exercise with the code provided. As noted, this is both a short exercise and one for which substantial assistance has been developed. The second example shows the use of SNOOPIE to support the production of a relatively complex skeleton to a solution, focusing on the 'if … else' structures required, the objective of the exercise. However, there is no support given for the content of that skeleton.

Table 5-11 Case 2. Example of Feedback from Stage 3 Analyses

Write a program, Swimming.java, for the local swimming pool that displays the admission cost for a group of people based on their age. The program should continue to prompt the user to enter an age until –1 is entered then display the total number of people in the group and the total cost for that group. Admission fees are as follows: ▪ under 16's - £2.50 ▪ over 65 -£3 and ▪ all other swimmers - £5. A 20% discount should be applied to groups of more than 6 people.	
Priority 1.	Main method must exist
Priority 2.	GUI must be used (a standard object to support keyboard input in Java)
Priority 3.	2 ints must be used
Priority 4.	1 double must be used
Priority 5.	1 'while loop' must be used
Priority 6.	'while' condition should test if variable not equal to -1
Priority 7.	Block of 'while loop' should contain 1 getInt (a GUI method for keyboard entry)
Priority 8.	'while' loop should contain an 'if statement'
Priority 9.	an 'if' statement should have 2 false cases
Priority 10.	Outside 'while loop' should contain 1 'if statement' to check if group is >6
Priority 11.	Outside 'while loop' should be an output statement

```
public class Swimming{

}
```

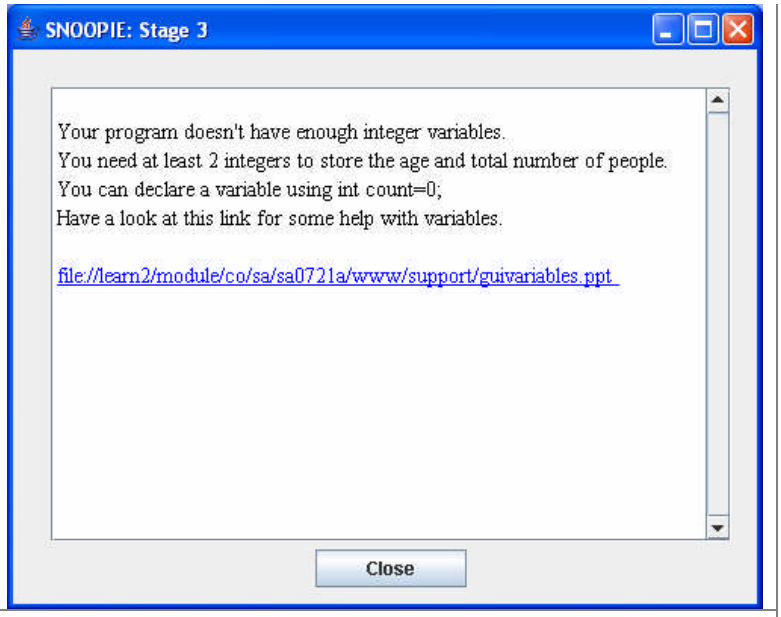


In the above example it is evident that the student has either forgotten the syntax for a main method or has no idea how to progress with their program beyond a class definition. Feedback provided reminds the student that they need to include a main method in their program with the syntax of the main method. The feedback includes a link to a PowerPoint file that explains the scope and purpose of the main method, including the { }.

```
public class Swimming{
public static void main
(String[] args)
{
}

}
```

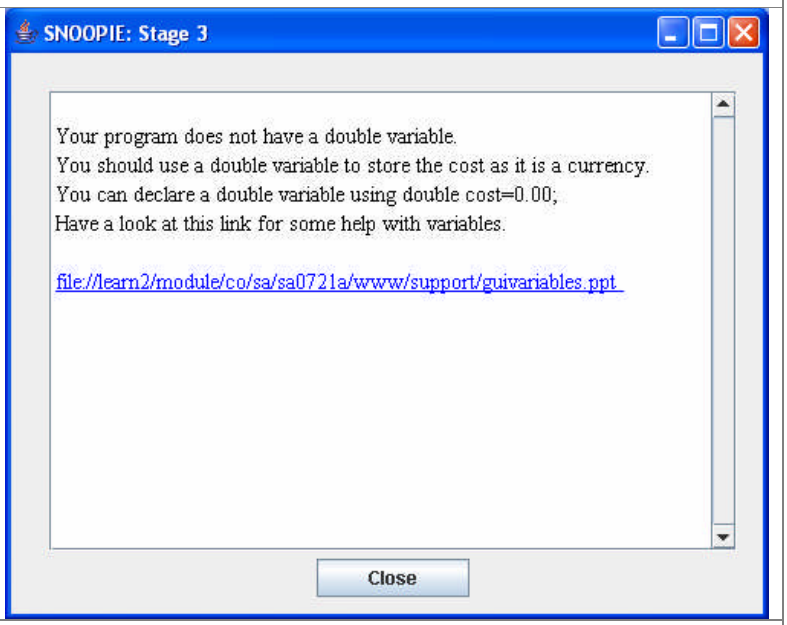
The student has successfully written a main method. For this question, the next logical step in the development of a solution would be to declare any variables. The feedback reminds the student (as the tutorial question specifies) that this solution requires variables and a GUI object declaration. If the student needs help with understanding variables or how to use the GUI they can access PowerPoint files for more details.

```
public class Swimming{
public static void
main(String[] args)
{
GUI gui = new GUI();
int age=0;
}

}
```



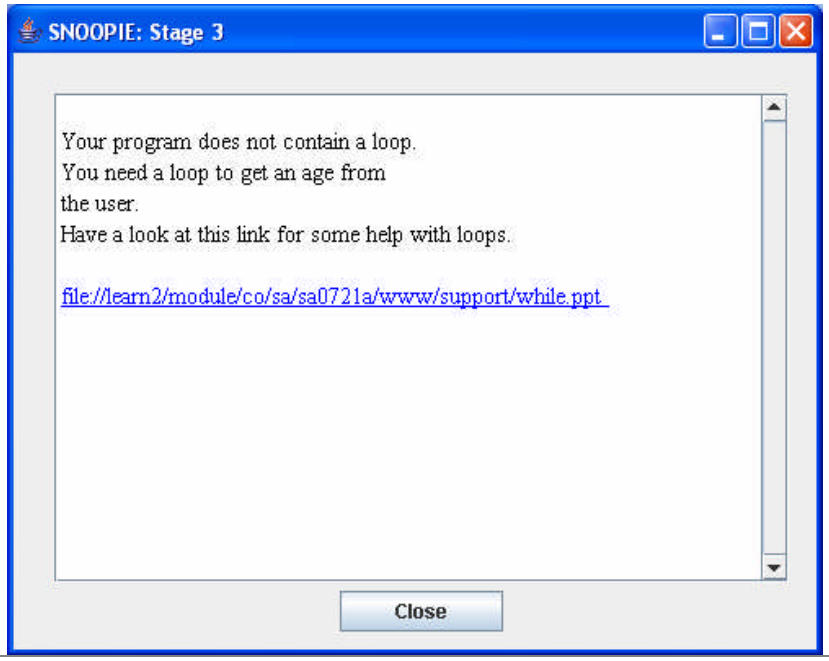
The student has now declared at least one variable and a GUI object but SNOOPIE has recognised that the student has still not declared enough variables to satisfy the requirements of the question.

```
public class Swimming{
public static void
main(String[] args)
{
GUI gui = new GUI();
int age=0;
int total=0;
}

}
```



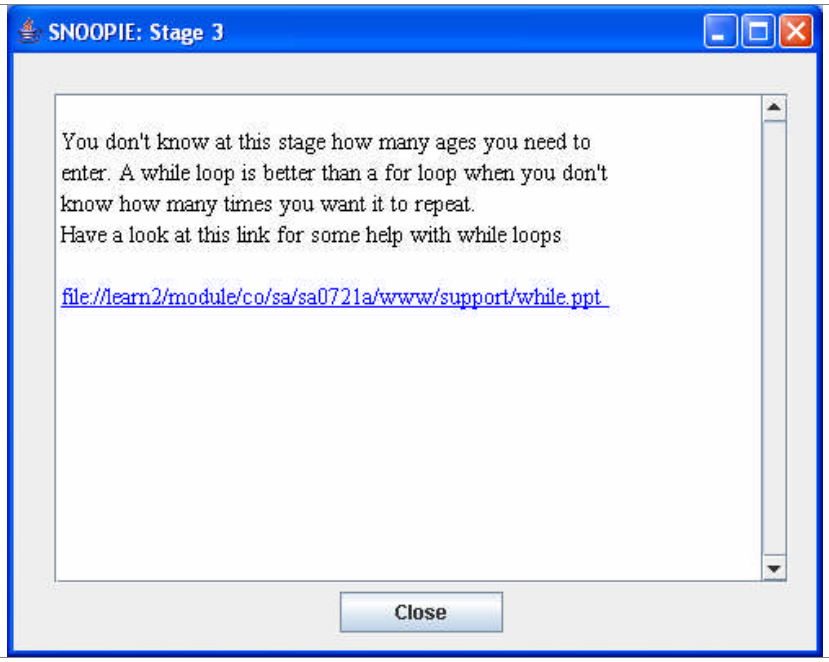
The student has now declared sufficient integer variables. SNOOPIE prompts the student to declare a double variable that will be used to store the cost.

```
public class Swimming{
public static void
main(String[] args)
{
GUI gui = new GUI();
int age=0;
int total=0;
double cost=0.00;
}

}
```



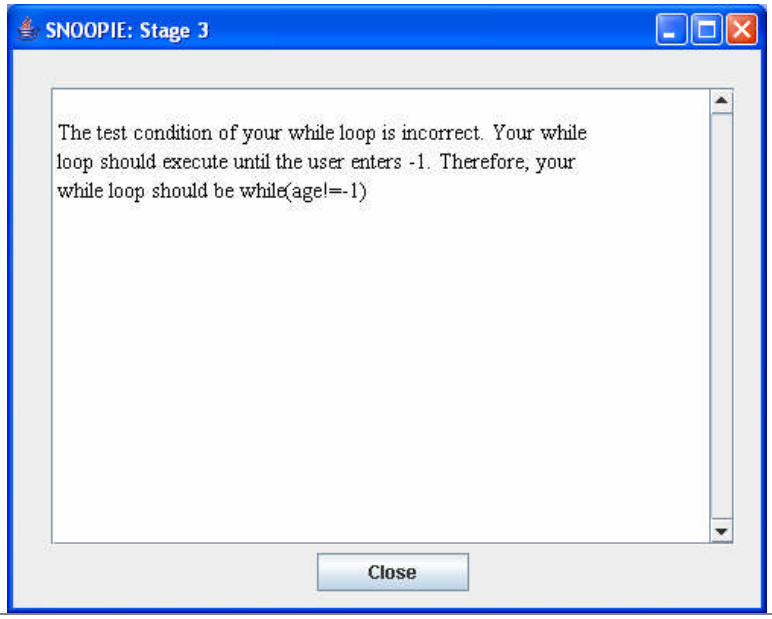
The student has now declared all the necessary objects and variables. SNOOPIE advises the student of the need to include a loop in their program. Notice that SNOOPIE does not suggest a specific loop.

```
public class Swimming{
public static void
main(String[] args)
{
GUI gui = new GUI();
int age=0;
int total=0;
double cost=0.00;

for (age=0; age<16;
age++)
{
}
}

}
```



In this example the student has introduced a ‘for loop’ into their program. SNOOPIE explains why a ‘while loop’ would be more appropriate for this exercise. Those students who need assistance with the syntax of a ‘while loop’ can follow the hyperlink for more help.

```
public class Swimming{
public static void
main(String[] args)
{
GUI gui = new GUI();
int age=0;
int total=0;
double cost=0.00;

while (cost<10)
{
      cost = cost +1;
}
}
}
```

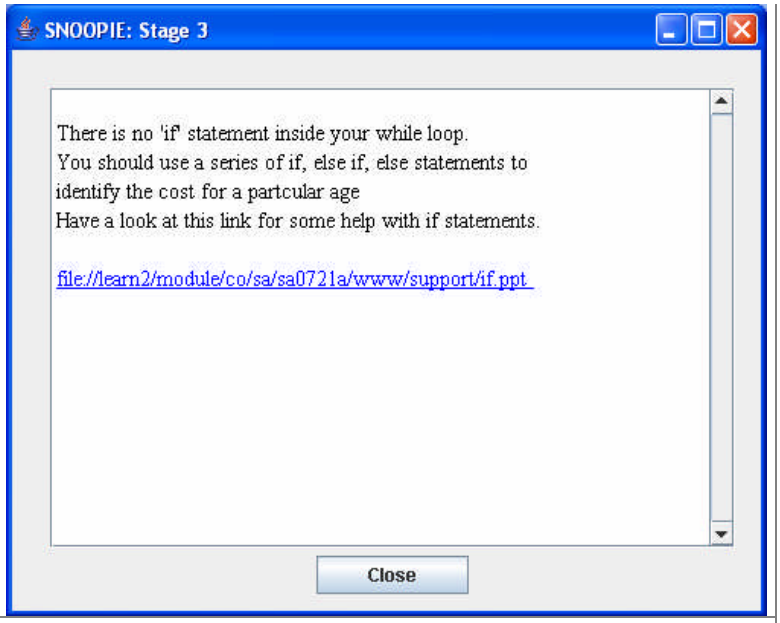


The student has now replaced the 'for loop' with a 'while loop'. SNOOPIE checks the condition of the 'while loop' and reminds the student that the loop should execute when the user enters –1, as indicated by the tutorial question.

```
public class Swimming{
public static void
main(String[] args)
{
GUI gui = new GUI();
int age=0;
int total=0;
double cost=0.00;
while (age!=-1)
{
}
}
}
```

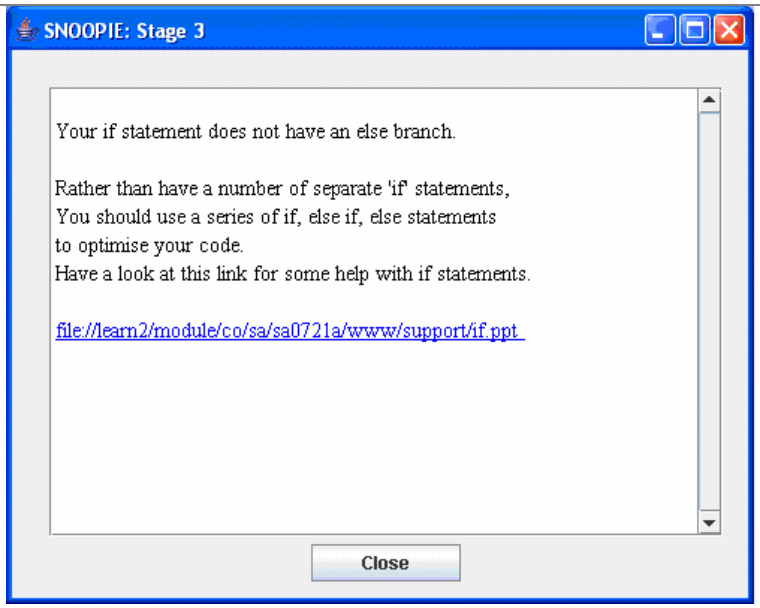
Here, the student has an empty while loop. SNOOPIE reminds the student of the requirements of the question, i.e. prompt the user to enter an age. SNOOPIE suggests that this can be done with the GUI. If the student needs help with the syntax of the GUI, they can follow the hyperlink.

```
public class Swimming{
public static void
main(String[] args)
{
GUI gui = new GUI();
int age=0;
int total=0;
double cost=0.00;
while (age!=-1)
{
      age=gui.getInt("Enter
the age ");
}
}

}
```



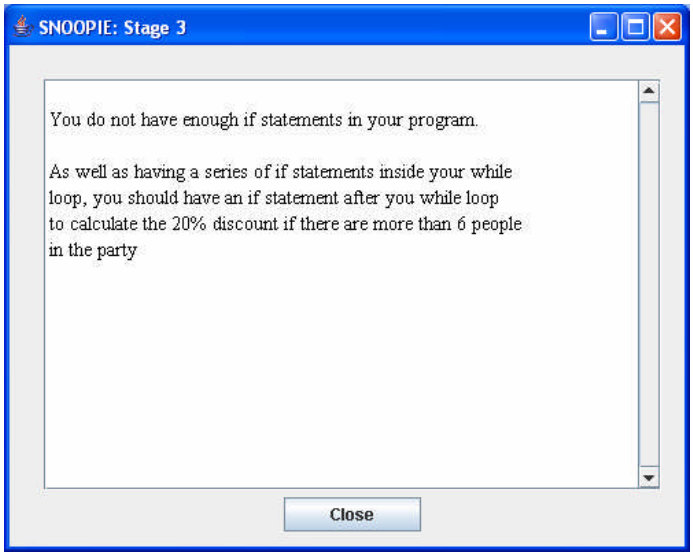
SNOOPIE recognises that the student's program does not contain any 'if statements' inside the body of the while loop. SNOOPIE informs the student that they need to include a series of 'if statements' inside the 'while loop' to help identify the cost based on a particular age. SNOOPIE also provides a link that offers more guidance on 'if statements'.

```
public class Swimming{
public static void
main(String[] args)
{
GUI gui = new GUI();
int age=0; int total=0;
double cost=0.00;
while (age!=-1)
{
      age=gui.getInt("Enter
the age ");
      if (age==0)
      {
      }
      if (age>60)
      {
      }
      if (age<60)
      {
      }
}}}
```



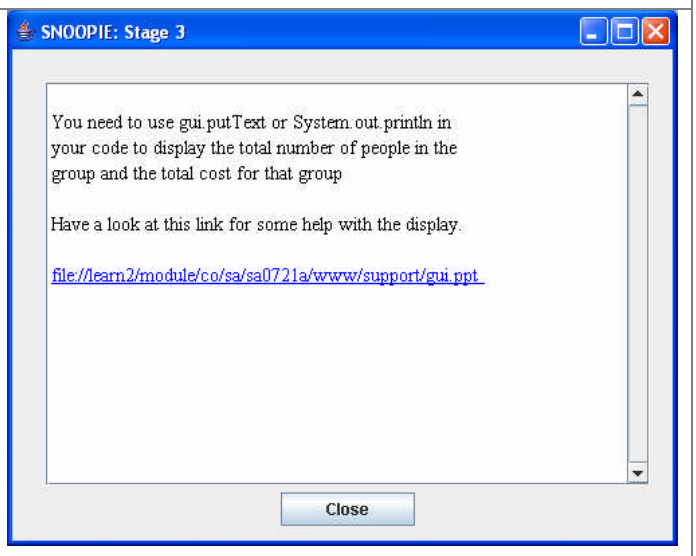
Here, SNOOPIE guides the student towards creating an 'if, else if, else' statement rather than a series of 'if' statements. Notice that the feedback only guides the student towards a skeleton solution, for example the first if condition (if age==0) is not actually correct but SNOOPIE has not been configured to offer support with the 'if conditions' for this assessment.

```
public class Swimming{
public static void main(String[]
args)
{
GUI gui = new GUI();
int age=0;
int total=0;
double cost=0.00;

while (age!=-1)
{
      age=gui.getInt("Enter the
age ");
      if (age==0){}
      else if(age>60)
      {
      }
      else{}
}}}
```



SNOOPIE prompts the student to include an 'if statement' after the body of their while loop. SNOOPIE makes reference to the tutorial question, hinting that the 'if statement' condition should check if the total number of people in the group is greater than 6.

```
public class Swimming{
public static void main(String[]
args)
{
GUI gui = new GUI();
int age=0;
int total=0;
double cost=0.00;

while (age!=-1)
{
      age=gui.getInt("Enter the
age ");
      if (age==0){}
      else{
            if(age>60){}
            else{}
      }
}
if (total<=6)
{}}}
```



SNOOPIE reminds the student that they need to include a print statement to display the total cost for the party. Again, a hyperlink is provided for those students that need more help.

Table 5-12 Case 3. Example of Feedback from Stage 3 Analyses

Question: Write a program, called Tutorial9_4 that imitates a guessing game. The program picks a random number from 1 to 10. The user tries to guess the random number and the program should write "cold" when the guess is 3 or more away from the correct answer, "warm" when the guess is 2 away, and "hot" when the guess is 1 away. For each random number the user gets three guesses. When the user enters the correct number the program writes a winning message. If the user fails to enter the correct number in three guesses, the program writes a failure message. The game should consist of 10 "rounds", where each round uses a different random number. After the 10 rounds, the program prints out how many of the 10 rounds were won and how many were lost.

- Players who win 7 or fewer rounds are rated as "nooblets"
- Players who win 8 or 9 rounds are rated as "uber," and
- Players who win all 10 rounds are rated as "leet"

Preamble to tutor dialogue

Generally, questions are written such that the ordering of the text represents a good strategy for solving the problem in terms of both logical stages in general and the coverage of instructional material to support this particular exercise. Here, the requirements of the module tutor for SNOOPIE implementation are posed in terms of the program checks desired in an order that steers students towards a working solution. In this requirement there is no account of the feedback given to students. That part of the development is well represented within the description of the resultant priority operations that follow the priority list. Note that only a subset of the requirements are expanded into detail here. Those subsets that are expanded have been selected because they demonstrate the multiple solution paths that may be supported. Those requirements shown in grey text have not been selected for expansion.

Tutor dialogue on required support

The module has not covered programs with multiple methods to date, and so the only method that should be in the program is main.

The first specific requirement is the generation of random numbers. Students are encouraged from the outset to consider the libraries required and the consequent need here to import the Java utilities library. This library allows instantiation of an object of the Random class. Here, one such random number generating object is required. Further, per iteration of the guessing game only one random number is required and so only one invocation of the number generating method ‘nextInt’ is needed. That single call is to generate a number in the range 1 to 10. To effect this, the ‘nextInt’ method requires a parameter of 10 (to return a random number uniformly distributed in the range 0...9) and the resulting value requires an increment of 1;

the result of this call requires storage in an integer. Note, in the past this task has caused significant confusion, since random numbers have not been used often, so is to be well supported by SNOOPIE.

User input is required once for the guessing game. The GUI object is the only object that is required for this and it contains several methods including one (getInt) for integer input. This is very familiar to the students and so requires less attention than the preceding task.

Students are typically encouraged to get the core aspects of a program operating and then wrap iterative constructs around these to promote easy testing of their program in stages. Here, the next stage in program development, prior to any iteration in terms of guesses or rounds, is to check the difference between the random number and the guess. Some students will recognise the potential to use Math.abs here (shown once previously) to reduce the ranges (>3, >2, >1, >-1, >-2, >-3) to categories (>3, >2, >1). In the case of the latter, the switch construct lends itself to categorical analysis; in the former, a more complex if statement is required as switch cannot deal with ranges. Note that students are much more familiar with if statement conditions than using the Math library and so are not particularly steered toward the abs method. Consequently, if Math.abs is used in the program, students are encouraged to use switch, otherwise if.

An approach using the switch construct (combined with Math.abs) requires a case for each of 1, 2 and 3, and depending on structure possibly a correct case (0). An if statement requires 3 or 4 alternate branches (for 1, 2 and 3, and depending on structure possibly a correct case) and the OR operator to deal with positive and negative values for the difference between guess and random number. In each construct, each alternate needs an output statement to feed back to the user the difference between guess and random number.

Next, the student must deal with the iteration for guesses. Two options present themselves here: a while loop that will stop when the guess is correct and a for loop to repeat 3 times. Clearly, the while loop is a better choice but some students are much more comfortable with a for loop and the associated break required when the guess is correct. The scope and details of whichever loop used are, in this case, to be checked so that the integer input and the selection construct are contained within the loop to allow the guesses.

A second loop representing the number of rounds is required. This is fixed by the program specification and so should be a for loop that repeats 10 times. The rounds loop should wrap around the guesses loop. Of observed confusion in previous years is that the random number generation should occur each round and not once only in the program and so this generation (i.e. nextInt) should occur within the rounds loop but not the guesses loop.

Finally, at the end of the program overall performance in the game should be related based on the number of

correct guesses across all rounds. Thus a selection construct to determine user rating is needed.

Resultant priorities

Priority 1. check that a method exists

Priority 2. check that a method called main exists

Priority 3 check that import java.util exists

Priority 4 check that an object of class Random exists

Priority 5 check that a method call to nextInt exists

Priority 6 check that only 1 method call to nextInt exists

Priority 7 check that nextInt method call contains the number 10

Priority 8 check that nextInt method value is incremented by 1.

Priority 9 check that nextInt method returns a value to an int variable

Priority 10 check that an object of class GUI exists

Priority 11 check that a method call to getInt exists

Priority 12 check that only 1 method call to getInt exists

Priority 13 Check that either selection construct exists. If neither exists, check if Math.abs is used

a. if Math.abs is used, guide towards switch
b. else guide towards if

Priority 14a if 'switch' exists

1. check that method call to Math.abs exists
2. check that 3 or 4 (hot, warm, cold and optionally correct) cases exist (semantic analysis ensures presence of default case)
3. check that each case contains a method call to print, println or putText

Priority 14b if 'if' exists

1. check that if statement has 3 or 4 branches (hot, warm, cold and optionally correct)
2. check that 3 of these if conditions have a logical operator if Math.abs is not used
3. check that logical operator is OR in each condition
4. check that each if branch contains a method call to print, println or putText

Priority 15 check that a loop exists

a. If no loop exists, guide towards a do…while

Priority 16a if loop is a while or do while,

1. check that condition contains a logical operator

Priority 16b if loop is a for

1. check that it repeats 3 times
2. check that a break statement exists

Priority 17 check that getInt method call is inside a loop

Priority 18 check that a selection statement is inside a loop

Priority 19 Check that two loops exist

1. If only one loop exists, guide second loop towards a for
2. If loop is a for, check that for repeats 10 times

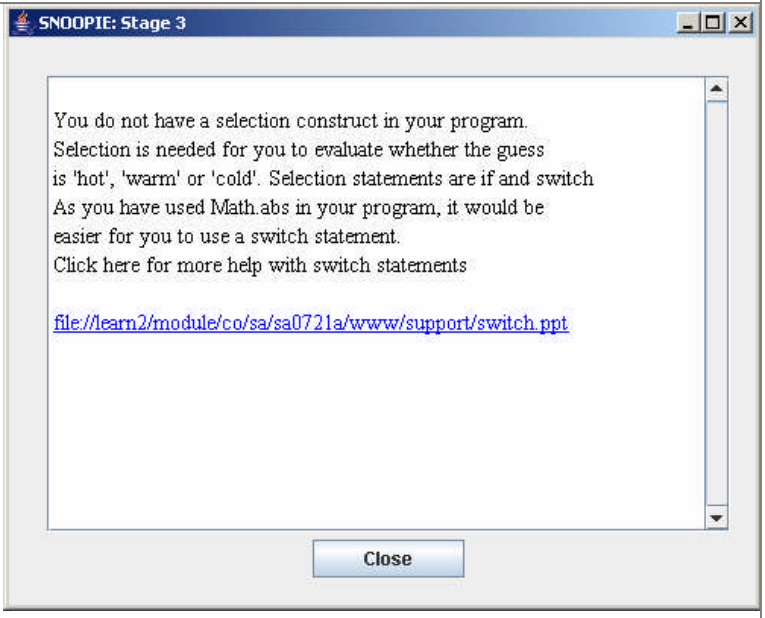
Priority 20 Check that only two loops exist

Priority 21 Check that loop contains a loop

Priority 22 Check that nextInt is inside outer loop but not inner loop

Priority 23 A selection construct should exist outside of all loops

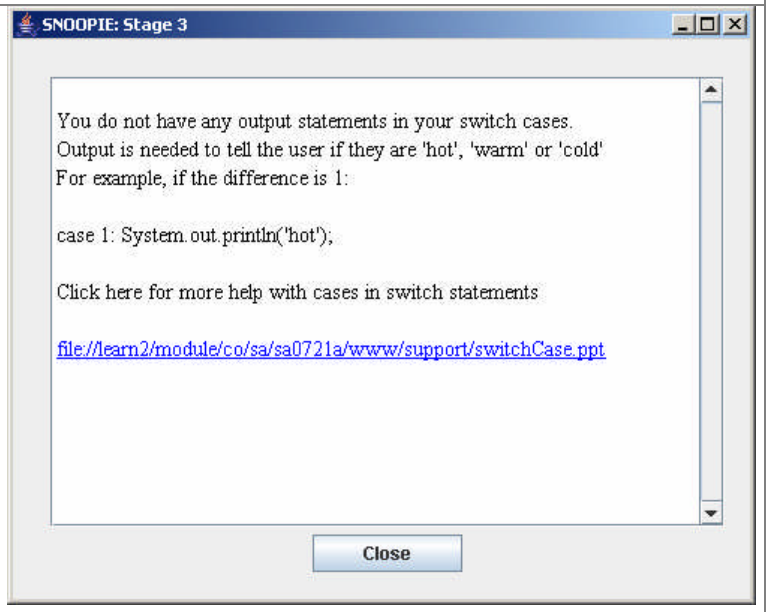
```
import java.util.*;
public class Tut9_4 {
    public static void
main(String[] args) {
        Random r = new
Random();
        GUI gui = new GUI();
        int maxrounds = 10;
        int max_guesses = 3;
        int guesses = 0;
        int thenum = 0;
        int theguess = 0;
        int correct = 0;
        int rounds=0;
        int diff=0;
        thenum =
r.nextInt(10)+1;
        theguess =
gui.getInt("...");
        diff =
Math.abs(theguess-thenum);
```



```
System.out.println("diff
is"+diff);
}
}
```

In the above example it is evident that the student has either forgotten the syntax for selection construct or has no idea how to progress with their program beyond evaluating the difference between the user guess and the random number. Feedback provided reminds the student that they need to include a selection statement in their program to assess how far the guess is from the random number. Both if and switch constructs are indicated but, since the student has used Math.abs (see instructor's notes) switch use is encouraged for the resulting categories. The feedback includes a link to a PowerPoint file that explains the switch statement.

```
import java.util.*;
public class Tut9_4 {
    public static void
main(String[] args) {
        Random r = new
Random();
        GUI gui = new GUI();
        int thenum = 0;
        int theguess = 0;
        int diff=0;
        thenum =
r.nextInt(10)+1;
        theguess =
gui.getInt("...");
        diff =
Math.abs(theguess-thenum);
System.out.println("diff
is"+diff);
switch (diff)
{
case 0: break;
case 1: break;
case 2: break;
default: break;
```

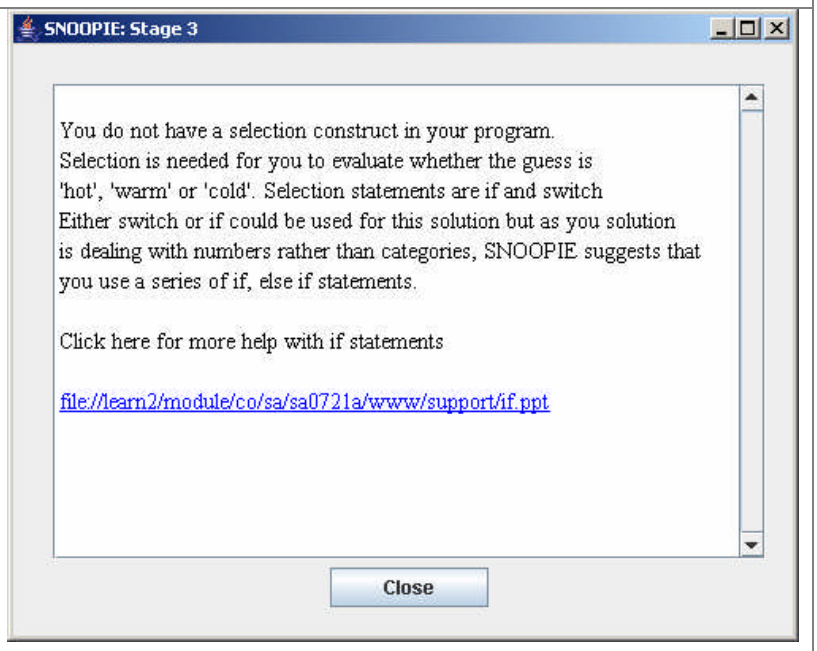


```
}
}}
```

The student has successfully written a switch statement. For this question, the next logical step in the development of a solution would be to display output to the user depending on the case of the switch statement. The feedback reminds the student that they need some output statements in the switch cases and provides an example to demonstrate what this means. If the student needs help with understanding how switch cases are structured, they can access a specific PowerPoint file for more detail.

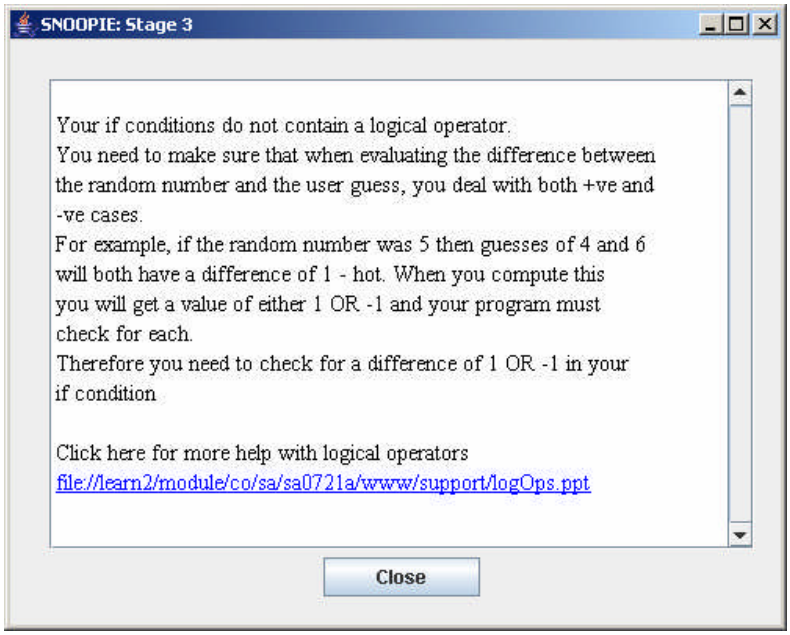
The above examples of priorities demonstrate one variant of program development using a switch statement. The following examples demonstrate a second variant of program development using n if statement to meet the same program requirement of selecting among hot, warm and cold for the guesses. The branch occurs at Priority 13 into Priority 14a OR Priority 14b. The program analysis converges again at Priority 15.

```
import java.util.*;
public class Tut9_2b {
    public static void
main(String[] args){
        Random r = new
Random();
        GUI g = new GUI();
        int random;
        int number=0;
  }}
```



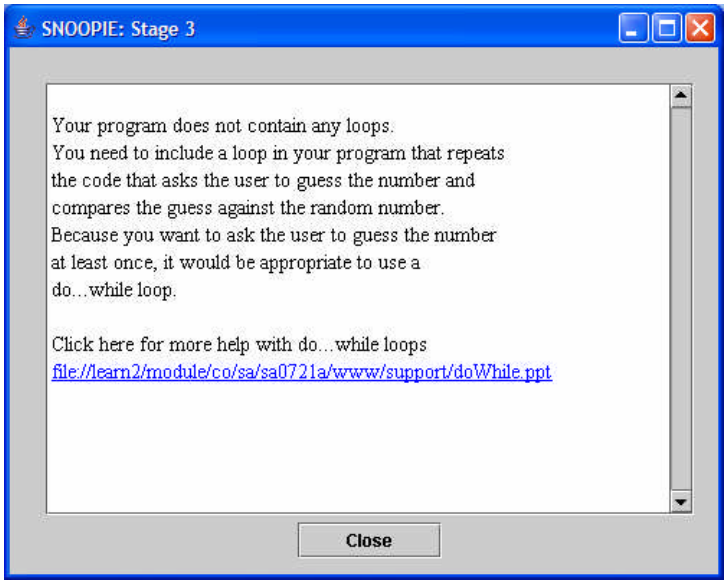
In the above example it is evident that the student has either forgotten the syntax for selection construct or has no idea how to progress with their program beyond evaluating the difference between the user guess and the random number. Feedback provided reminds the student that they need to include a selection statement in their program to assess how far the guess is from the random number. Both if and switch constructs are indicated but, since the student has not used Math.abs (see instructor's notes) if statement use is encouraged for the resulting values. The feedback includes a link to a PowerPoint file that explains if statements.

```
import java.util.*;
public class Tut9_2b {
public static void
main(String[] args){
Random r = new Random();
GUI g = new GUI();
int random;
int number=0;

if  (number-random>2)
{System.out.println("cold");
}
else if (number-random>1)
{
System.out.println("warm");
}
else if  (number-random>0)
{
System.out.println("hot");
}
}}
```



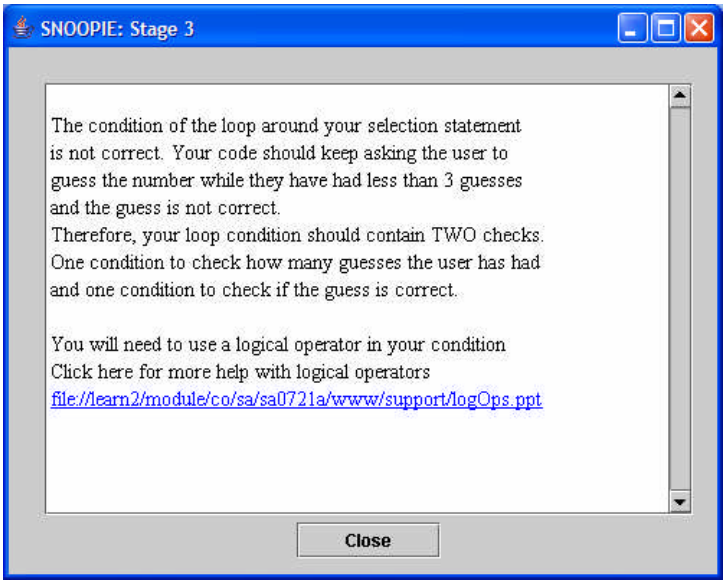
The student has written an if statement to check how close the user's guess is to the random number. The student has not used Math.abs and has not included the logical operator OR in their if conditions to accommodate for the potential negative result. This issue is explained to the student and they are guided towards a Powerpoint slide explaining the use and syntax of logical operators.

```
import java.util.*;
public class Tut9_2 {
public static void main(String[]
args) {
Random r = new Random();
GUI gui = new GUI();
int thenum = 0;
int theguess = 0;
int diff=0;
theguess = gui.getInt("...");
diff = Math.abs(theguess-
thenum);
switch (diff)
{
case 0:
System.out.println("correct");
break;
case 1:
System.out.println("hot");
break;
case 2:
System.out.println("warm");
break;
default:
System.out.println("cold");
break;
}
}
}
```



The student has now written a correct selection statement to display ‘hot, ‘warm’ or ‘cold’. SNOOPIE now checks the program to identify if the student has written a loop in their program. As the above program contains no loop, SNOOPIE guides the student towards writing a do…while loop as this would be the most appropriate construct for this part of the questions. The student can view a PowerPoint file for help with the syntax of a do…while loop.

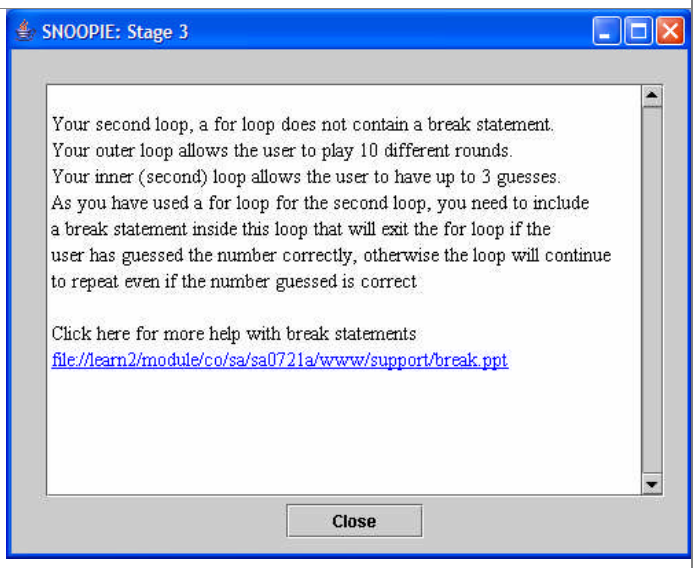
```
import java.util.*;
public class Tut9_2 {
public static void main(String[]
args) {
Random r = new Random();
GUI gui = new GUI();
int thenum = 0;
int theguess = 0;
int diff=0;
int guesses=0;
do
{
theguess = gui.getInt("...");
guesses++;
diff = Math.abs(theguess-
thenum);
switch (diff)
{
case 0:
System.out.println("correct");
break;
case 1:
System.out.println("hot");
break;
case 2:
System.out.println("warm");
break;
default:
System.out.println("cold");
break;
}
while(guesses<3);
}
}
```



In the above example, the student has included a do…while loop around the select statement but the ‘do while’ loop condition does not contain a logical operator. SNOOPIE has advised the

student that they should write the condition so that the loop repeats until the user has had 3 guesses or until the number guessed is correct. SNOOPIE provides a link to a PowerPoint file explaining the logical operator.

```
import java.util.*;
public class Tut9_2b {
public static void main(String[]
args){
Random r = new Random();
GUI g = new GUI();
int random;
int number=0;
int correct=0;

for (int outer =0; outer<10;
outer++)
{
random=r.nextInt(10)+1;
for(int inner=0; inner<3; inner++)
{
number=g.getInt("Enter guess");
if ( (number-random>2) || (number-
random<-2) )
{
System.out.println("cold");
}
else if ( (number-random>1) ||
(number-random<-1) )
{
System.out.println("warm");
}
else if ( (number-random>0) ||
(number-random<0) )
{
System.out.println("hot");
}
}
```



```
if (number==random)
{
System.out.println("correct");
correct++;
}
else
{
System.out.println("not correct,
out of guesses");
}
}
if (correct<8)
{
System.out.println("nooblet");
}
else if (correct<10)
{
System.out.println("uber");
}
else
{
System.out.println("leet");
}
}
}
```

In the above example, the student presents a program that already contains two loops. As both loops are ‘for’ loops, SNOOPIE identifies that the inner loop does not contain a break statement, which is necessary to stop it repeating when the user has guessed the correct number in less than 3 guesses. SNOOPIE advises the student that they need to include a break statement inside their inner loop and includes a link to a PowerPoint file explaining break statements.

The examples above serve to highlight the capacity of SNOOPIE to support problem formulation. It should be noted that these examples do not reflect any particular student activity. In actuality, most students did not use the tool for support at all identified stages (see Chapter 6).

While the examples show tool operation they do not provide any detail on tool implementation. As with the description of Version 1, a complete description of all analyses employed here is prohibitive. For breadth in explanation, another case study, Table 5-13, with a diverse set of priority checks is used to highlight the operational aspects of SNOOPIE. Here, code is checked for the following priorities:

- Priority 1 main method exists
- Priority 2 two methods exist
- Priority 3 non-main method has return type void
- Priority 4 non-main method has string parameter
- Priority 5 - while condition is –1
- Priority 6 – between 1 and 2 getInt exists
- Priority 7 - only 1 if within while

The analyses performed for each priority are described in Table 5-13

Table 5-13 Examples Of Priority Checks

Priority 1 main method exists Construct a list of all elements that are tagged “method” If the list is empty, fail priority Else For every method in the program If method name equals main Set foundFlag to true Next method If not foundFlag, fail priority Else proceed to next priority
Priority 2 two methods exist If length of list of all methods != 2, fail priority Else proceed to next priority
Priority 3 non-main method has return type void For every method in the list Find method with name not equal to “main” and store it in an element, meth Next method For each “return type” element in method meth If “return type” name equals “void” Set FoundFlag to true Next type element If FoundFlag is false, fail priority Else proceed to next priority

Priority 4 non-main method has string parameter Create list requiredArgs with one item “String” in it With element meth (as in priority 3) For each “formal argument” element in the method Add to list of actualArgs If list of requiredArgs is not the same length as actualArgs, fail priority Else if contents of requiredArgs does not equal actualArgs, fail priority Else proceed to next priority
Priority 5 - while condition is –1 Establish that only one ‘while loop’ exists in the program For the ‘while loop’ element Get the ‘test’ element of the loop If the ‘test’ does not contain a ‘binary-expression’ Fail priority Else if operator of binary expression is not != Fail priority Else if ‘unary-expression is not ‘-‘ Fail priority Else if literal value is not ‘1’ Fail priority Else proceed to next priority
Priority 6 – between 1 and 2 getInt exists Gather all method calls in a list If the length of the list is 0, fail priority Else For each method call If the method call message element is ‘getInt’, add one to count of getInts Next method call If count of getInts is less than 1 or greater than 2, fail priority Else proceed to next priority

Priority 7 - only 1 if within while

(Previous priority has established presence of only one 'while loop' in the program)

With the while loop

Create list of all 'if' blocks inside the while loop

If length of list is 0, fail priority

If length of list is greater than 1, fail priority

Else pass priority

These example priorities shown in Table 5-13 demonstrate the level of analysis that is possible and in the implementation of SNOOPIE there is a wide range of analyses undertaken. Priority checks and associated feedback have been developed for all required aspects of more than sixty questions. In all cases, the XML representation has supported implementation of the necessary program checks.

Appendix A presents the complete list of priority checks in terms of the program code modules required to implement the current SNOOPIE support. Note that the list provided is not constrained by the implementation technologies, it is simply sufficient to provide all needed support for the exercises linked to SNOOPIE. The relationship between this generic list of priority checks and the specific tutorial questions set within the taught material is shown. For convenience in reporting, each taught module block (1-3) is presented in a separate section. To allow comparisons across blocks, the full range of priority checks are presented, even if not used in a given block. To ease reading, the checks are structured into the conceptual areas of:

- 'General', meaning those relating to the inclusion of a main method and import statements.
- 'Use of pre-written code', relating to use of the module specific libraries (i.e. Robot, GUI and Cow) together with generic Java Standard Edition libraries such as Random.
- 'Constructs', i.e. all priority checks focused on the inclusion of specific constructs and interrelationships between specific constructs and other program components.

- ‘Data’, i.e. all priority checks supporting analysis of inclusion of Java primitives, e.g. int, double, etc.
- ‘Student-written methods’, supporting the set of tutorials associated with method implementation.

Two observations are noteworthy from the interrelationships between priority checks and exercise set. First, there are clear patterns in the types of checks used in each of the blocks, and this represents the progression of material within the module. Block 1 priority checks are largely in the areas of using prewritten objects and in constructs, and notably loops as these are taught first. In contrast, block 2 has a clear focus on selection and data, with significantly less use of prewritten objects. Both block 2 and 3 draw on the student written method sections, and indeed block 3 focuses almost entirely on this.

This shift in the nature of provision is consistent with the second observation, that the support for exercises is generally reduced over the teaching period. This is evidenced clearly here by comparison of block 1 and block 3. Block 1 programs are typically short, simple exercises with the solution limited to the use of prewritten objects and a single construct. However, these exercises are fully supported in terms of the number of priority checks for such small questions. In contrast, block 3 questions are larger, require multiple construct use and multiple method implementations. However, the support is very focused on method implementation, with only limited attention to the range of constructs and data types needed. The result of this difference in focus between block 1 and 3 is that early exercises are given support that extends towards across all programming constructs and interrelationships that would be required for a solution, whereas later exercises are given only a subset of those constructs and interrelations. This variation in provision is also well evidenced in Tables 5-10 and 5-11, where in 5-10 support towards a complete solution is offered and in the later 5-11 exercise support only extends to a skeleton solution.

This, of course, represents a significant investment of time and the benefit of that investment is considered in Chapter 6. This requirement for implementation is more critically appraised in Chapter 7.

Chapter 6 Evaluation

6.1 Introduction

Testing the hypothesis that SNOOPIE helps students learn to program is, of course, not possible since exact measurement of what a student has learned over the course of an academic session is precluded. An evaluation of SNOOPIE v1 and v2 is necessary to demonstrate its effectiveness in terms of supporting the learning process through the testing of hypotheses associated with that learning process. Here, existing evaluation techniques for teaching support tools are identified and appraised. This review is used to determine the range of evaluation instruments for SNOOPIE. These measures in turn drive the formulation of testable hypotheses that reveal whether or not SNOOPIE makes a positive contribution to the learning process. SNOOPIE is then evaluated using a range of formal and informal mechanisms, and the evaluation is derived from an integrated view of those evaluation mechanisms. Such an integrated perspective provides a more sophisticated impact measure in relation to any teaching tool as no single measurement approach can provide a complete perspective on the multi-faceted contribution made by learning support tools (Koile and Singer, 2006).

6.2 Problems with Evaluating Teaching Support Tools

As discussed in chapter 2, the processes involved in learning to program are complex and cover a wide range of cognitive levels, and this makes measuring a student's understanding of any complex material difficult. In a typical programming module a mixture of practical exercises and exams may be used to award a grade based on student performance. This grade may be heavily influenced by a student's determination or assessment technique and is not just a measure of their ability or comprehension.

In addition to problems in measuring understanding in students, other factors make the measuring process difficult. The heterogeneity in UK teaching approaches and content, in contrast to the US for example, means that any study to measure understanding is typically

limited to a single module. The number of students participating in a study is typically small in statistical terms. Even within a single module it is difficult to test the effectiveness of a learning support tool. Two options present themselves: integrating the tool into the teaching activity; and extending the student activity out with the module, i.e. bespoke experiments. Tool integration into a module requires strong links with and commitment from the module leader in advising on the module content and delivery. Assessing the effectiveness of a tool external to module activity requires students to first volunteer and then participate in extra-curricular activates, and this both lowers numbers and introduces bias in the type of (student) participant. The extent of integration in the module may also constrain the way in which evaluation may be carried out. For example, if the tool is evaluated only through student activities external to the module, then no long-term patterns of support or opinions may be derived. On the other hand, integrated tools cannot be compared or contrasted with alternatives if this is all a student has used, and so no comparative data can be generated.

Many factors influence the types of evaluations that are feasible and these constrain the formulation of hypotheses. In scientific study, it is common practice to design replicated experiments where one factor is varied in a controlled manner across those replicates. Clearly in educational research it is neither possible nor ethical to construct such replicates. For ethical reasons a class of students cannot be offered different teaching materials for the duration of the module. Different, e.g. successive, cohorts cannot be compared fully, simply because they comprise unique individuals who cannot repeat the learning process and it is difficult to preserve consistency in delivery across multiple cohorts. Finally the duration of the measurement process must be consistent with the duration of the process that is being measured – a long-term process like learning to program cannot be assessed by only short-term measures yet long-term measures are difficult to design and interpret. It is not possible under these circumstances to identify a single measurement to determine if SNOOPIE has had a positive impact on the learning process. The range of available approaches is reviewed in the next section to determine an evaluation scheme for SNOOPIE.

6.3 Evaluation Methods

Chapter 2 introduced a range of learning support tools that seek to enhance student understanding of programming. The majority of these tools have been evaluated through different instruments and here those instruments are identified and appraised. The evaluation mechanisms extend over a range of timescales and are a mix of quantitative and qualitative data from formalised data gathering techniques together with informal and typically anecdotal approaches.

6.3.1 Experiments

The learning support tools of Explainer (Redmiles, 1993), VINCE (Rowe and Thorburn, 2000), Shah & Kumar (2002) and OGRE (Milne and Rowe, 2002), described in Chapter 2, have all been evaluated using experiments.

Redmiles (1993) conducted an experiment on twenty-three students completing a programming task. Eight students completed the programming task using the full Explainer support tool support. A further eight students completed the task using a modified version of Explainer (the interactive menu was switched off) and seven students completed the task using an on-line manual. The experiment measured the number of different variant solutions that an individual would attempt in solving the task, assuming that the more support an individual had received, the lower the number of variations would be. The experiment also recorded the time taken to complete the task and the number of runs conducted by the student. An ANOVA test did not indicate a significant difference in means for the measurements although a Square-ranked test demonstrated that those students using Explainer had significantly fewer variations and runs than those students using the on-line manual.

Rowe and Thorburn (2000) conducted an experiment of VINCE using 16 students who had completed a nine-week introductory programming course. The students were assigned to one of two groups depending on their programming ability. Their programming ability was determined by their performance on exercises and exams during the nine-week course. The two groups were supposed to “mirror each other”, i.e. Rowe and Thorburn tried to ensure that

for one student of a particular ability in group A, there would be another student with a similar ability in group B. Students were initially interviewed to determine their view of their programming skills and were asked questions concerning sample C programs. Students in one group were then asked to view the on-line VINCE tutorials over a three-week period. The students in the other group were instructed not to use VINCE over that period. After the three-week period, the students were interviewed again with similar questions to the initial interview. The results from the study demonstrate that the students who used VINCE during the three-week period performed significantly better in the questions that tested their understanding of programming than those students who did not use VINCE.

Shah and Kumar (2002) conducted two 28-minute experiments using the tutoring system. 13 students participated in the two experiments and were divided into two groups. All students participated in an 8-minute pre-test. Students in the control group then spent 12 minutes practising with the tutoring system receiving minimal feedback, while students in the other group spent 12 minutes practising with the tutoring system receiving detailed feedback. Student groups were changed for the second 28-minute experiment. 11 of the 13 students performed better in the post-test after using the tutoring system but no significant difference was noted between the results of those students receiving detailed versus minimal feedback.

OGRE (Milne and Rowe, 2002) has been evaluated using an experiment. The experiment involved 25 students divided into two groups, a control group and a test group. The experiment required both groups of students to do a pre-test and a post-test. Between the tests, the students in the test group worked through eight tutorials, using OGRE to graphically explain the topics in the tutorial. The students in the control group did not complete any tutorials during that time but were allowed to work on their programming coursework. The results demonstrated that those students who completed the tutorials, with the OGRE assistance, performed significantly better in the post-test than those students who had not completed any tutorials. This clearly shows that students exposed to OGRE have assimilated the assistance offered and exploited it in their post-test activity.

Experiments can be employed to determine if a support tool impacts on test performance, which is at least a measure of short-term engagement and understanding. All of the above methodological designs allow for the establishment of test and control groups to allow direct evaluation of specific, testable hypotheses. Thus experiments, when designed scientifically, can be employed to test hypotheses framed around at least short-term learning goals.

6.3.2 Questionnaires

A common mechanism for evaluation of existing programming support tools is questionnaires. For example, CAP (Schorsh, 1995), DatLab (MacNish, 2000) and the tutoring system by Shah and Kumar (2002) have all been evaluated using this approach.

Schorsh (1995) distributed surveys to the 520 students enrolled on the CS110 module, 'Introduction to Computer Science', using CAP. The survey included questions asking the students to rate CAP as learning tool and describe how CAP "helped them learn to program better". Schorsh does not indicate how many students completed the survey. As the module is an introductory class it is not clear what the students are comparing CAP with, or if they have learned to program in another environment. The survey indicated that those students who expected to do well on their assignment rated CAP more favourably than those students who expected to do poorly. This would suggest that students' perceptions of CAP were influenced by their opinion of how competent they were as programmers. Schorsh also distributed a survey to the instructors who taught on the CS110 module. Again this survey asked for opinions and views on the success of CAP to evaluate its effectiveness.

Datlab (MacNish, 2000) was evaluated using a survey distributed to the students at the end of the module. The students were asked to rate how well they believed the system achieved specified goals and their views on their experience and level of feedback provided by the system. The study does not indicate how many students participated in the survey. The results are used to demonstrate its effectiveness as a learning aid to the module. Usefully, this questionnaire design recognised the goals of the system for evaluation. For example, the immediate feedback provided and encouraged students to spread their work throughout the

semester by achieving weekly milestones. These goals are used to determine some of the questions posed, together with more general experiential questions.

Shah and Kumar (2000) asked 13 students to complete a feedback form after using a teaching tool during the 28-minute experiment described above. One of the questions on the form asked students if the teaching tool had helped them to learn new material, yet the study reports that all the material covered by the tutor had already been covered in the class.

Questionnaires can be used to gather quantitative data on staff and students perceptions of a support tool. MacNish (2000) demonstrated the importance of incorporating the goals of the system into the questionnaire design to establish whether these goals had been met. Schorsh recognised the importance of asking students to rate their programming experience in the questionnaire to establish if this had any impact on their opinion of the tool.

6.3.3 Interviews

Interviews are recognised as a sound mechanism for assessing understanding and deriving opinions from individuals in a structured manner. They are less commonly used for evaluation of programming support tools, most likely because of the time required to undertake individual interviews and the consequent small sample size. Notably, OGRE (Milne and Rowe, 2002) has been evaluated using a series of individual interviews with staff and students. These interviews were recorded formally and transcribed, and were used to assess the effectiveness of the OGRE interface, the impact OGRE had on a student's confidence in their abilities and the appropriate environment for using OGRE. It is not clear how many interviews were conducted and whether they consisted of set questions or were more conversation-led. Milne and Rowe acknowledge the issues impeding a thorough evaluation of a support tool used in a real-time teaching environment. They do, however, state that their conclusions, which are drawn from a combination of interviews, experiments and anecdotal evidence (see below), demonstrate OGRE's effectiveness as a valid aid to learning.

6.3.4 Module Performance Comparison

For those teaching tools which are fully embedded in a taught module, it would appear obvious to look to module performance as a means to evaluate the impact of that tool on student performance. For example, IVC (Moore and Taylor, 2005) state their intention to compare module performance results across two different cohorts to identify if IVC has impacted on the learning process. To date, these results have not yet been made available. The use of module performance as an evaluation mechanism is rare for a number of reasons. First, most support tools are not fully embedded within the teaching of a module, and so the support tool may not significantly affect the performance of students at that resolution. Secondly, many modules comprise a mix of practical (program development) and more theoretical (written) examination activities, and while programming support tools may contribute to practical activities within the laboratory context the impact of learning support tools on performance of more theoretical concepts is at best indirect. Thirdly, it is difficult to replicate experimental, controlled conditions at the scale of the module and so the drivers that determine any change in module performance are difficult to identify. Finally, the module grade is typically aggregated across a mix of assessments and assessment types and so any affect on learning will be diluted in the summarised results.

6.3.5 Anecdotal Evidence

DatLab (MacNish, 2000), CAP (Schorsh, 1995) and CmeRun (Etheredge, 2004) all used anecdotal evidence to evaluate the effectiveness of the support tool. The anecdotal evidence included comments made in passing by staff and students and observations noted during use of the tools.

MacNish (2000) states that anecdotal evidence from both staff and students “has shown a very positive response to the system”. MacNish observes that the students appeared to enjoy submitting their programs for analysis in order to improve their results and concludes that this must have educational benefits.

Schorsh (1995) acknowledges some of the problems and limitations of CAP that appear to have been highlighted through anecdotal evidence. For example some students not reading

the error messages fully and making incorrect changes to their code as a result and , further, students becoming reliant on the CAP annotations to ensure their programs meet the style rules. Problems such as these are difficult to establish using questionnaires or experiments but can be observed during the timetabled programming session if the support tool has been incorporated into the module.

Etheredge (2004) had not been fully tested prior to publication of the paper due to the time constraints of the CS1 course (students were still being introduced to the C++ syntax therefore most of their errors would not have been suitable for CmeRun). Some staff and students had tested CMeRun, and their opinions were used anecdotally to evaluate the system. Etheredge reported that many of the people who tested the tool found it to be effective for debugging simple programs. Staff observed that it would be particularly useful as a demonstration aid.

Anecdotal evidence can be used to gauge general opinion of a support tool, particularly in relation to its usefulness. It can also be used to gauge whether students are using the tool as intended or are becoming too dependent on it. If the support tool has been incorporated into the module, anecdotal evidence can be used informally its long-term patterns of usage, benefits – actual and perceived – and problems encountered by both staff and students.

6.3.6 SNOOPIE Evaluation Methods

The review undertaken above describes a range of mechanisms used for evaluation of programming support tools. The next section draws on this review and defines the set of methods used to evaluate SNOOPIE, i.e. experiments, questionnaires, students and staff interviews and module performance, since SNOOPIE is embedded into the teaching of one module. Anecdotal evidence is incorporated into the discussion that considers a more integrated view than is offered by these separate methods. SNOOPIE is evaluated through the following hypotheses:

1. The program formulation support provided by SNOOPIE (v1 and v2) improves short-term student performance.
2. The problem formulation support provided by SNOOPIE (v2) improves short-term student performance.

3. The problem formulation support provided by SNOOPIE (v2) reduces the time taken to complete an ideal solution.
4. A combination of problem and program formulation (SNOOPIE v2) improves long-term student performance when compared with program formulation alone (SNOOPIE v1).
5. Students perceive that SNOOPIE (v2) has a positive impact on the learning process.

Hypotheses 1,2 and 3, which relate to short-term performance, are tested using experiments (6.4). Hypothesis 4 is tested at the scale of the module, using a combination of module performance (6.5), interviews with students (6.7) and written statements from module tutors (6.8). Hypothesis 5 considers the perception of the support by students and is tested using a combination of experiments (6.4), questionnaires (6.6) and interviews with students (6.7). In each section the specific evaluation instrument is described and the results particular to that instrument discussed. An integrated evaluation is provided in section 6.9. Evaluations of SNOOPIE have been performed at both the University of St Andrews and the University of Abertay Dundee. The majority of these evaluations have been performed at the University of Abertay Dundee because the software used (JCreator) utilised the standard Java Compiler. At the University of St Andrews, shortly before the start of the first term in the academic year 2004 to 2005, the software changed from Borland Together to Eclipse. The Eclipse environment does not use the standard Java Compiler and so SNOOPIE could not be modified in the short time frame to link into the new environment. This did provide an opportunity for a different form of experiment with the students at the University of St Andrews, as described in Section 6.4.1.

6.4 Experiments

Experiments have been undertaken at each of the Universities of St Andrews and Abertay Dundee. These experiments are fundamentally different in structure, purpose and participation, and so must be described independently. The experiment at St Andrews was in the form of short programming exercises undertaken in a practical laboratory environment, and involved students who were already familiar with another environment (Eclipse, http://www.eclipse.org/). This experiment was designed to explore student perception of SNOOPIE when students had already had long-term exposure to another development

environment, so contributing to the evaluation of hypothesis 5. As this experiment was peripheral to the taught module there was only limited participation. The experiment at Abertay Dundee, although not formally part of the taught module, was conducted during the timetabled session and used one of the catch-up weeks. The majority of students participated and all had long-term access to SNOOPIE. To control the experiment, and limit student interactions, a class test styled environment was imposed on the participants. The purpose of the Abertay Dundee experiment was to identify the impact of SNOOPIE on short-term student performance in both program and problem formulation (Hypotheses 1, 2 and 3).

6.4.1 University of St Andrews

At the University of St Andrews, all students participating in the CS1002 first year programming module were invited to participate in a 2-hour experiment. 62 students were registered for the module but only 8 students volunteered to take part in the study. The impact of the self-selection by participants is noted as appropriate in the results, Sections 6.4.1.1 and 6.4.1.2. The experiment at St Andrews, undertaken outside of the module contact time, was conducted in practical laboratory conditions. The students were divided into two groups, one group using the SNOOPIE software and one group using the standard Compiler. This was the first time that students at the University of St Andrews had used the SNOOPIE (v2) software. Both student groups were asked to use JCreator, a different IDE from that used in their programming module (Eclipse), to ensure that all students were using unfamiliar software. Note that the only difference between using JCreator with and without SNOOPIE is that students press one button (SNOOPIE) or another (Compiler) to compile their program. For the first 20 minutes of the experiment, the students were guided through the IDE and the software used for the practical exercises. The practical exercises were based on programming constructs with which the students were already familiar. The students were instructed to treat the experiment as a normal programming laboratory and were allowed to ask for tutor help if needed. Due to the small number of students taking part in the experiment and the fact that the students did not know each other it was observed that students did not work together

or discuss the exercises during the experiment. Therefore all assistance provided came from SNOOPIE or from the tutor. The experiment practical exercises are in Appendix A B.

6.4.1.1 Results

All students were asked to complete a questionnaire based on their experience of using both JCreator and SNOOPIE, although only 5 of the 8 students completed the questionnaire. Of those 5, only 1 student who did not use SNOOPIE completed the questionnaire.

Students were asked to relate their experiences in using both IDEs. All students indicated that Eclipse was not difficult to use at the beginning of the module and all students now find Eclipse easy to use. Most students found that JCreator was no more or less difficult to use than Eclipse and those that expressed a preference would prefer to continue with Eclipse. Students were also asked to comment on the difficulty of the questions posed to provide some calibration as to the level of support they would have liked to complete those questions. No student found the questions posed difficult. Further, most rated their own programming skills highly.

With regard to program formulation, the syntax errors provided by Eclipse and SNOOPIE, most students preferred Eclipse error messages, although anecdotal comments from students indicated that the expanded syntax error messages provided by SNOOPIE were clear. The only student who was negative about the JCreator support was the student who did not have SNOOPIE, i.e. the student who received only standard Compiler errors.

For problem formulation, most students felt that SNOOPIE helped them to produce a solution to the questions, although no student found SNOOPIE more useful than Eclipse. Most students recognised that the problem formulation support provided by SNOOPIE is useful. In particular, one student commented on the good use of links to support program formulation.

6.4.1.2 Discussion of Results

It is evident from the results that the participating students favoured Eclipse, largely because of the fact that this is a familiar development environment and that Eclipse provides

Syntactic Support as code is typed. In spite of the unfamiliar environment of SNOOPIE, most students were positive about the overall support provided.

The experiment was voluntary and out with class time and this introduces a self-selection effect into the experiment. All students were strong programmers, as evidenced by their self-evaluation and the recognition that for these students the questions posed were not difficult. The questions were designed to ensure that the only new concept was the introduction of the support tool itself, and so those questions required only already familiar programming concepts. This meant that, for these competent students, no support was really required for the questions posed. Further, in this experiment SNOOPIE is not presented as alternative to the standard Compiler and so students may receive additional help when it is not yet required. In this case, support for those particular concepts and questions were in the main not required at that stage of development of the student skill set in program and problem formulation. As a result, the assistance offered by SNOOPIE was not needed. Indeed one student makes a clear statement relating to the potential benefits of problem formulation but notes that such provision should be optional.

6.4.2 University of Abertay Dundee

At the University of Abertay Dundee an experiment was conducted during Term 2 to evaluate the effectiveness of SNOOPIE v1 and SNOOPIE v2. The experiment was conducted in the style of a voluntary, formative class test in Term 2 to ascertain knowledge and ability to complete exercises in a formalised environment, i.e. under test conditions. The experiment lasted for 2 hours. As part of the experiment, the IDE had been configured with three separate tool buttons. The first tool button invoked the standard Java Compiler, with no program or problem formulation, and logged details of the student's compilation attempt, i.e. filename, date, time, total number of errors and error messages with corresponding line numbers. The second tool button invoked SNOOPIE v1, i.e. program formulation, and logged details of the student's compilation attempt as per the first button. The third tool button invoked SNOOPIE v2, i.e. program and problem formulation; the logged details of the student's compilation attempt were extended to include the failed priority identifiers. All

students were asked if they wished to take part in the experiment and forty-seven out of sixty-three registered students participated. Those students in the experiment were randomly assigned a group number (one of 1, 2, or 3) and asked to use that particular tool button for all compilation attempts for the duration of the test. Fourteen students were assigned to Group 1 (standard Compiler), sixteen students were assigned to Group 2 (SNOOPIE v1) and seventeen students were assigned to Group 3 (SNOOPIE v2).

At the end of the class test, the related Java files for all participating students were voluntarily submitted for analysis and the error logs of those students were also collected. A class test type of environment was chosen as a mechanism for conducting the experiment to ensure that students did not receive ad hoc assistance from classmates or tutors to complete the exercises. Further, analysis of individuals' performance in a separate assessment exercise that occurred at a time close to the experiment ensured that no one group was significantly better or worse at related activities than the others (see Table 6-1). The table presents the performance of students in terms of the number of correct programs out of a maximum of 12 in that related assessment. Therefore any observed difference in performance among the groups is a result of SNOOPIE assistance only. For the purpose of evaluation, the Java files from all forty-seven students are compared for completeness. The class test questions can be found in Appendix C. The class test was based on programming concepts with which the students were already familiar. Note that problem formulation feedback was applicable only to questions 1,4 and 8 (see Appendix C).

Table 6-1 Results Of Assessment Coincident to Experiment

	Group 1	**Group 2**	**Group 3**
Mean	10.2	10.6	9
Standard deviation	1.5	1.7	2.5

An ANOVA single factor test was performed on the above results in Table 6-1 to determine whether there was a significant difference between the abilities of the groups with p-value

>0.05, indicating a high probability that the three groups are drawn from the same distribution.

The exercises themselves were structured into eight questions. Questions 1, 4 and 8 required the student to write a program to solve a problem. For these questions, every student's Java file was analysed for program completeness and assigned to one of four categories: Syntax Errors; Compiles; Correct Output; and Ideal Solution. The Java files in the first category, Syntax Errors, typically contained a number of syntax errors and incomplete programming constructs. The Java files assigned to the second category, Compiles, contained syntactically correct code but were missing the necessary programming constructs to meet the prescribed functionality of the question. The Java files assigned to the third category, Correct Output, were syntactically correct and able to display correct program output given specified input values but did not meet all the criteria of the question; for example using integers instead of doubles or not using a method to calculate the answer, instead performing the calculations in the main method. The Java files assigned to the fourth category, Ideal Solution, were syntactically correct, able to display a correct answer and met all the criteria of the question.

For questions 2, 3, 5, 6 and 7 the students were provided with existing Java files which contained some common syntax and semantic errors. The students were required to correct the errors. Every student's Java file was inspected to identify how many of the original errors remained. For each exercise, the graphs indicate those students who were able to solve all errors and those students who were unable to resolve each of the original errors, e.g. if a program originally had 3 errors and a student was not able to solve error 1 and error 2, that student would appear in both the error 1 and error 2 categories.

Some students ran out of time and were not able to attempt all exercises. It would be inappropriate to categorise them as not able to solve the error. If a student's log file does not demonstrate that they attempted to compile a Java program, they are excluded from the analysis for that question or the entire class test respectively. One student assigned to group 2 spent only 28 minutes on the experiment as the student was ill on the day and had to leave

early. The performance of this student is not included because of the low numbers of students per group.

An additional complication in the analysis is that a small number of students migrated towards the higher levels of support when they found difficulty addressing the questions without that support. This was evident only on analysis of the log files after the event. The migration occurred as a combination of this being a voluntary exercise, so no strict rules could be put in place, and a desire on the students' part to complete the questions in recognition of the value of formative instruction. This migration does, of course, reveal the underlying view of some students, based on their long-term exposure to the tool during the course of the module that SNOOPIE was of benefit to the process of programming. For analysis of each question, students are allocated to each group on the basis of the tool (1, 2 or 3) used to solve that question, as determined by evidence in individual log files. Therefore, each group reflects the performance of those students using the corresponding tool for each question, and there are small fluctuations in group numbers across the questions.

6.4.2.1 Results

Figure 6-1 to Figure 6-8 show the results of the experiment based on program completeness for all students for each question. The graphs depict percentages across different categories for visual comparison. Some students did not attempt questions 5 to 8 (as evident by the compilation log) and so these students are not included in the graphs relating to those questions. The χ^2 test (Clarke and Cooke, 1982) has been used on the actual numbers of participants to determine if any significant difference between performances of different groupings of students exists. Questions 1 and 4 are presented first as SNOOPIE v2 support was provided for these. Questions 2, 3, 5, 6 and 7 are then presented as only SNOOPIE v1 support was provided for these. Finally question 8 is presented, as the results require a different analysis from the other questions.

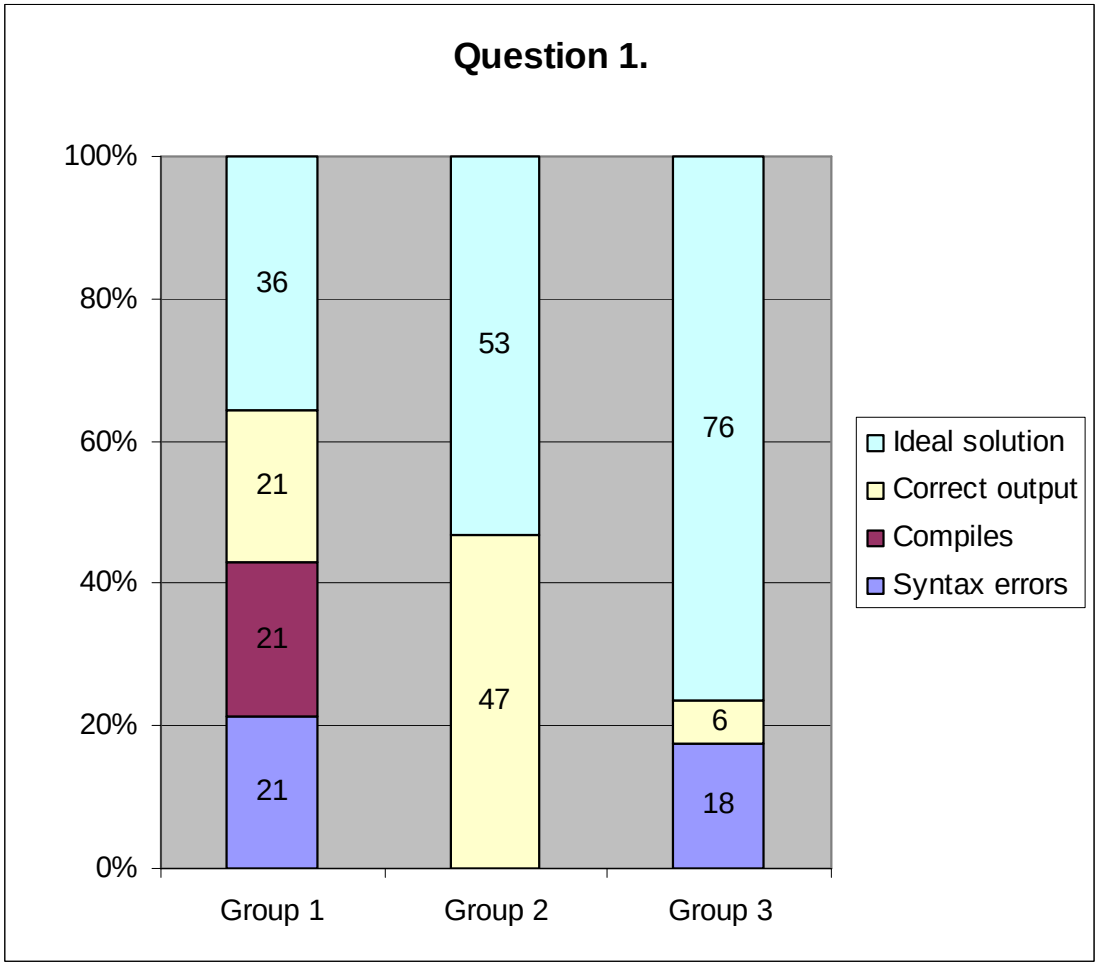


Figure 6-1 Percentage Results Per Category From Question 1 Of The Experiment

Figure 6-1 shows the results from the first question in the class test. The results demonstrate that there was no significant difference in the performance among all the three groups, $\chi^2(2, 46)=5.28$, $p=0.07$. Those students who were in group 3, receiving support from SNOOPIE v2, were significantly more likely to produce an ideal solution that met the criteria of the question than students in Group 1 and 2 combined, $\chi^2(1, 46)=4.37$, $p<0.05$. The difference in performance between Group 1 and Group 3 was also significant, $\chi^2(1, 31)=5.23$, $p<0.025$. The performance differences between Group 2 and Group 3 ($\chi^2(1, 32)=1.89$, $p=0.169$) and Group 2 and Group 1 ($\chi^2(1, 29)=0.91$, $p=0.340$) were not significant. The interpretation of these, and all other questions, is discussed in Section 6.4.2.2.

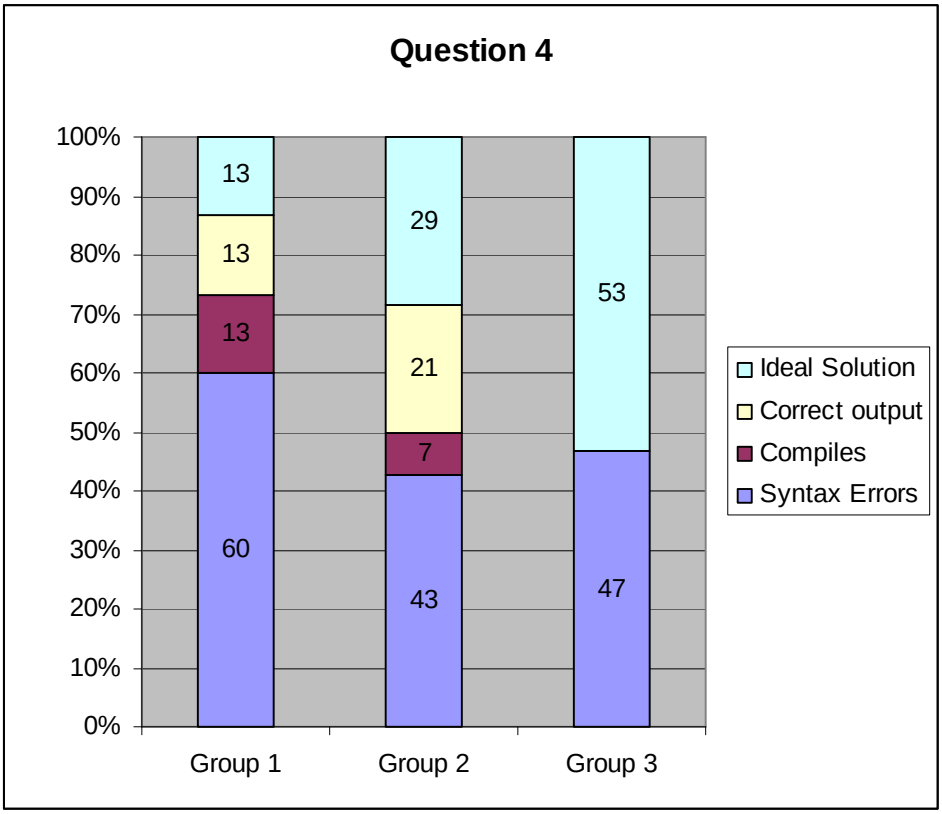


Figure 6-2 Percentage Results Per Category From Question 4 Of The Experiment

Figure 6-2 shows the results from the fourth question in the class test. The results demonstrate that there was no significant difference among all the three groups, $\chi^2(2, 44)=5.63$, $p=0.06$. Those students who were in Group 3, receiving support from SNOOPIE v2, were significantly more likely to produce an ideal solution that met the criteria of the question than students in Group 1 and 2 combined, $\chi^2(1, 44)=4.86$, $p<0.05$. The difference in performance between Group 1 and Group 3 was also significant, $\chi^2(1, 30)=5.4$, $p<0.025$. The performance differences between Group 2 and Group 3 ($\chi^2(1, 29)=1.83$, $p=0.2$) and Group 2 and Group 1 ($\chi^2(1, 29)=1.02$, $p=0.311$) were not significant.

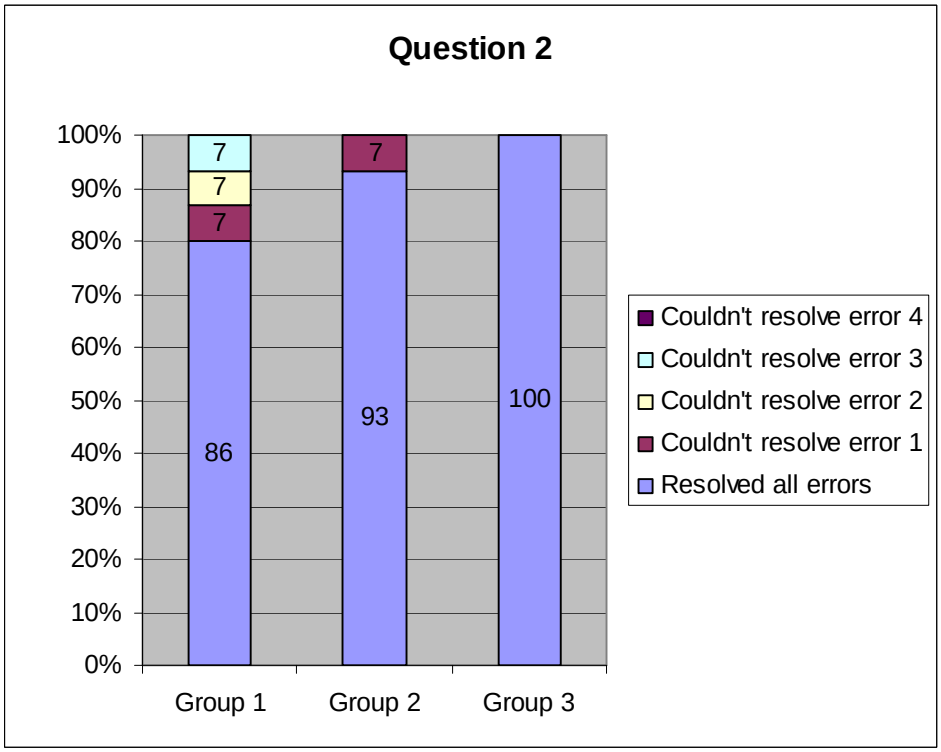


Figure 6-3 Percentage Results Per Category From Question 2, Blink.Java

Figure 6-3 shows the results from the second question in the class test. For this exercise students were given a Java file which contained four common (simple) syntax errors. The graphs demonstrate that all the students receiving SNOOPIE v2 support (Group 3) were able to resolve all the error messages. Error 1, a typing mistake in the class name, caused some problems for students receiving phase 1 and no tool support. Error 2, a missing '(' in a method header also caused some problems for students with no tool support. Error 3, 'an unclosed string literal' caused a problem only for students receiving no tool support. Error 4, and extra closing bracket at the end of a program did not cause a problem for any student. However, the majority of students corrected all errors. The results demonstrate that there was no significant difference among all the three groups together, $\chi^2(2, 46)=2.57$, $p=0.276$, or any pair wise combination.

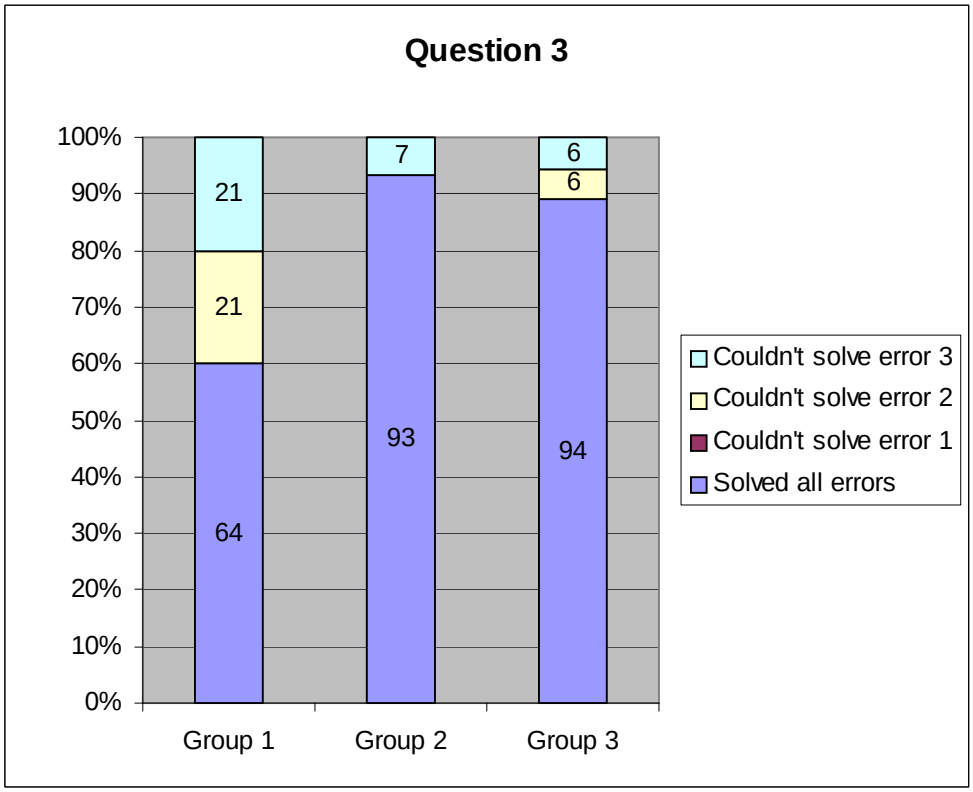


Figure 6-4 Percentage Results Per Category From Question 3, Buzz.Java

Figure 6-4 shows the results from the second question in the class test. For this exercise students were given a Java file that contained three common, and less obvious, syntax errors. Error 1, lower case s in the reserved word System, did not cause a problem for any student. Error 2, a typing mistake in a variable name caused some problems for students receiving version 2 and no tool support. Error 3, a single equals in the conditional of an 'if' statement caused a problem for all students. The results demonstrate that there was a significant difference among all the three groups, $\chi^2(2, 46)=6.55$, $p=0.037$. Those students who were in groups 2 and 3, receiving program formulation support from SNOOPIE (v1 and v2), were significantly more likely to resolve the syntax errors than students in Group 1, $\chi^2(1, 46)=6.55$ $p=0.01$. There was no significant difference between students in Group 2 and students in Group 3, $\chi^2 (1, 32)=0.008$ $p=0.92$.

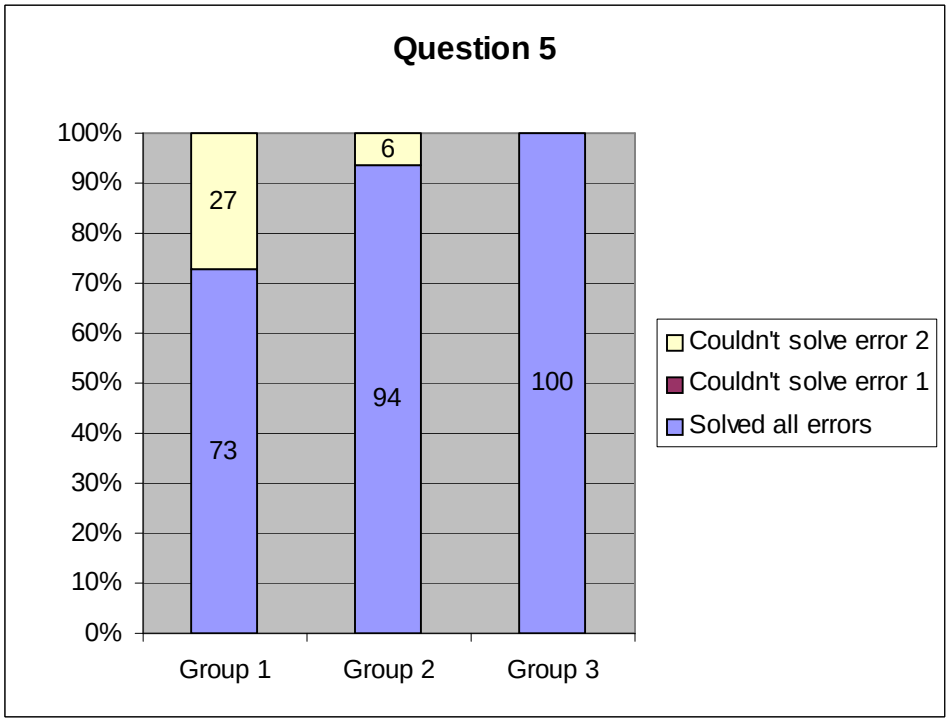


Figure 6-5 Percentage Results Per Category From Question 5, Narf.Java

Figure 6-5 shows the results from the fifth question in the class test. For this exercise students were given a Java file that contained two common syntax errors relating to the placement of the ';'. Error 1, a missing semi-colon after a method call did not cause a problem for any student. Error 2, a misplaced semi-colon after a method header caused a significant problem for students with no tool support and a problem for one student receiving v1 support. All students receiving v2 support were able to resolve both errors. The results demonstrate that there was no significant difference among all the three groups, $\chi^2(2, 39)=5.11$, $p=0.078$. Those students who were in groups 2 and 3, receiving program formulation support from SNOOPIE (v1 and v2), were significantly more likely to resolve the syntax errors than students in Group 1, $\chi^2 (1, 28)=4.82$, $p=0.02$. There was no significant difference between students in Group 2 and students in Group 3, $\chi^2 (1, 32)=0.777$ $p=0.378$.

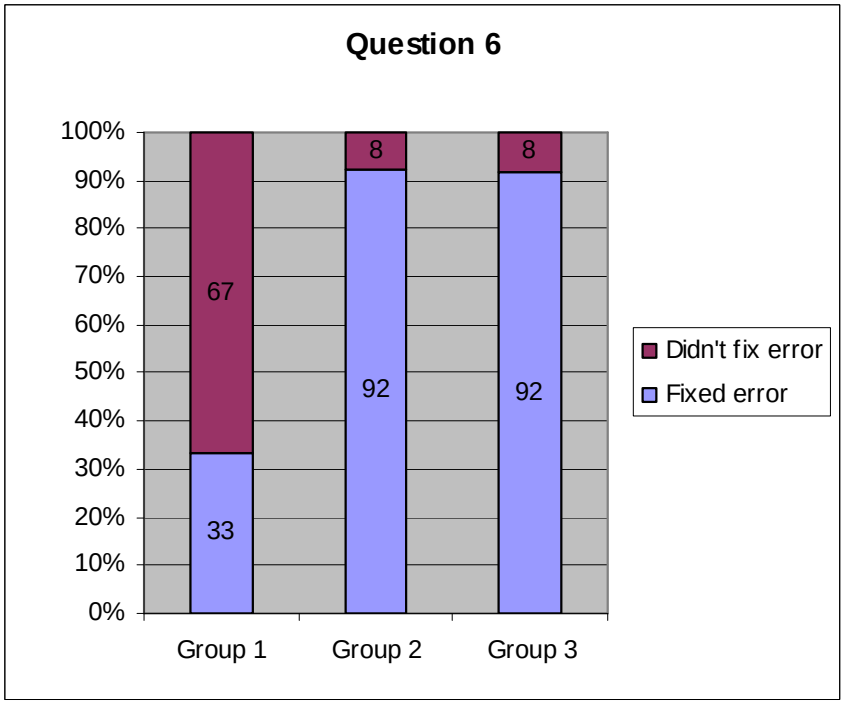


Figure 6-6 Percentage Results Per Category From Question 6, Blimp.Java

Figure 6-6 shows the results from the sixth question in the class test. For this exercise students were given a Java file that contained one common semantic error, a misplaced semi-colon at the end of a 'for loop'. The graph demonstrates a significant difference between those students with no tool support and those students who received a message about the semantic failure in the code. The results demonstrate that there was a significant difference among all the three groups, $\chi^2(2, 34)=12.66$, $p=0.002$. Those students who were in groups 2 and 3, receiving program formulation support from SNOOPIE (v1 and v2), were significantly more likely to resolve the syntax errors than students in Group 1, $\chi^2(1, 34)=12.658$, $p=0.001$. There was no significant difference between students in Group 2 and students in Group 3, $\chi^2(1, 25)=0.003$ $p=0.95$.

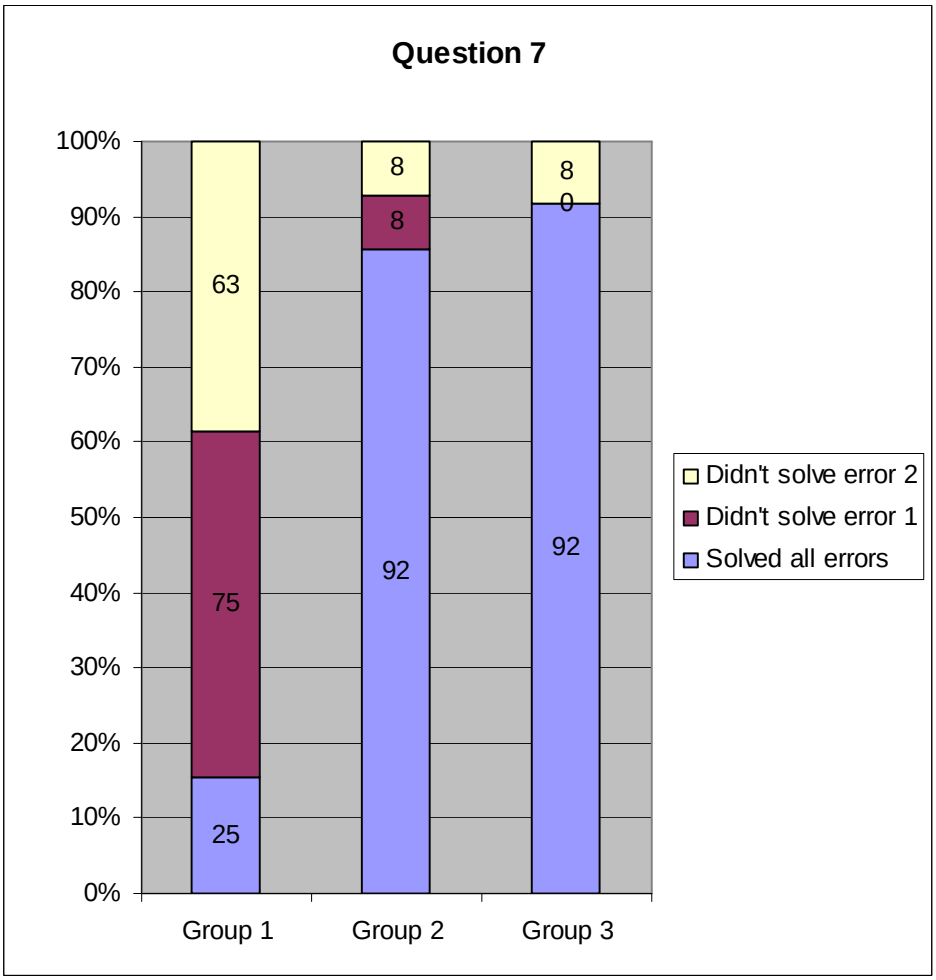


Figure 6-7 Percentage Results Per Category From Question 7, Ping.Java

Figure 6-7 shows the results from the seventh question in the class test. For this exercise students were given a Java file that contained two common semantic errors. Error one was a single equals in an 'if' condition on two Boolean values. Error two was a misplaced semi-colon after an 'if' condition. Of those students attempting this exercise only 22% of students with no tool support were able to resolve both errors, compared with over 90% of those students with tool support. The results demonstrate that there was a significant difference among all the three groups, $\chi^2(2, 33)=14.815$, $p=0.0006$. Those students who were in groups 2 and 3, receiving program formulation support from SNOOPIE (v1 and v2), were significantly more likely to resolve the syntax errors than students in Group 1, $\chi^2(1,$

33)=14.814, p=0.0006. There was no significant difference between students in Group 2 and students in Group 3, $\chi^2(1, 25)=0.003$ p=0.95.

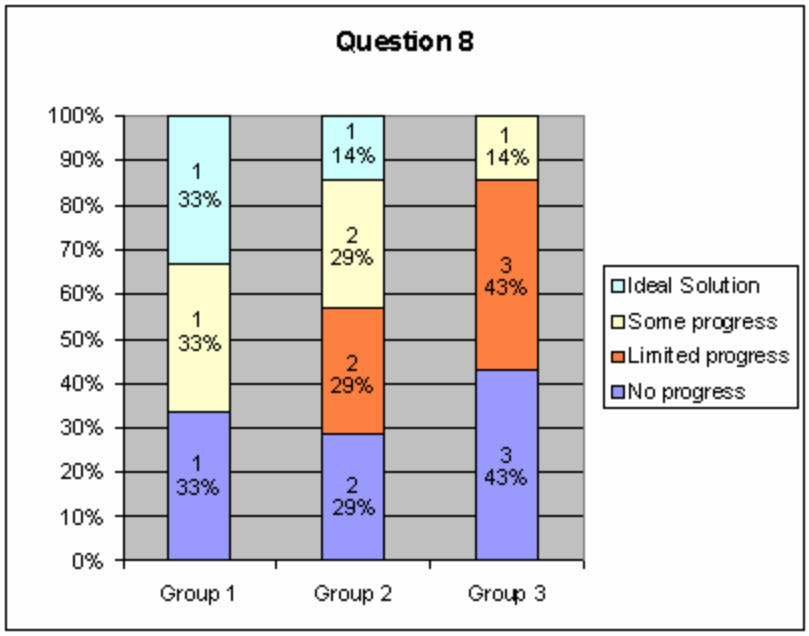


Figure 6-8 Percentage Results Per Category From Question 8, Matrix.Java

The final question in the class test was the most challenging exercise of the experiment, and four categories of student performance are established for analysis. 'Ideal solution' represents those students who produced a solution that met the criteria of the question; 'Some progress' represents those students who developed a partial solution; 'Limited progress' describes students who developed small amounts of code toward the solution but were not able to produce any substantial attempt; 'No progress' is used to denote students who created a file and typed in the existing code provided in the question. To highlight only those who progressed to question 8, the absolute number of students in each category is shown in Figure 6-8. For consistency, the figure also shows the percentage of students in each category in parentheses. As noted, this question is challenging with only a small number of students making any substantial progress: In Group 1, only 2 students made any substantial progress with the question; for Group 2 this is 3; for Group 3, only 1 student. It is apparent here that

individual performance outweighs any contribution made by the learning support tool. Those students who produced an ideal solution spent over half of their total experiment time on question 8 and were the two most able students on the module. These students were able to produce a solution to this and the majority of questions set in the module generally without recourse to SNOOPIE. The 'ideal solution' student in Group 1 had ignored some of errors in questions 6 & 7 after spending only a few minutes on those programs. This student made the decision to proceed to what he probably saw as a more interesting question. For the rest of the students it is useful to consider performance on a group-by-group basis.

Students in Group 1 were less likely to attempt question 8 than students in Groups 2 and 3. Analysis of these students' error logs indicates that they typically spent longer resolving the syntactic and semantic errors found in questions 2, 3, 5, 6 and 7. Java Compiler and run-time feedback indicated to these students that these programs were not correct and the students then spent longer trying to resolve the errors. Students in groups 2 and 3 received SNOOPIE assistance with these errors and were typically able to resolve them faster. These students were then more likely to have time to spend addressing question 8. Analysis of the error logs of students in Group 3 indicates that very few of students were able to proceed past the Java Compiler errors to receive SNOOPIE v2 feedback (note SNOOPIE v2 support can be provided only on syntactically correct code). Therefore these students typically received the same feedback as students in Group 2. Typically those students who made limited or no progress with question 8 had little difference between the time spent on question 4, the next most difficult question, and the time spent on question 8. The students who performed better on question 8 spent far longer on that question than on question 4. The small numbers of students in each category precludes any meaningful statistical analysis.

6.4.2.2 Discussion of Results

It is clear from the results of questions 2, 3, and 5 that Syntactic Support is beneficial. The results of question 2 indicate that trivial syntactic errors can usually be resolved using feedback from the Compiler. However, the results from questions 3 and 5 demonstrate that extended error message support from SNOOPIE was useful in helping students resolve more

complex and less common syntactic errors. The errors present in question 2 were errors that the students had most likely seen on a regular basis during the course of the programming module and it could be assumed that they had already established a strategy for recognising the likely cause of these error messages and then taking corrective actual. Although the students in Group 1 were as likely to solve the errors in question 2 as students in Group 2 and 3, Figure 6-9 demonstrates that students in Group 1 took longer to resolve these errors. The smaller number of students in Group 1 who were able to correct the errors also took longer to correct errors in Questions 3 and 5, although the effect is less pronounced in the latter case.

Questions 6 and 7 demonstrate that Semantic Support is beneficial. With both questions, there was a significant difference between those students in Group 1 who did not receive feedback on the error and students in Groups 2 and 3 who received SNOOPIE support. As the Compiler does not provide feedback with these mistakes, it was difficult for students in Group 1 to establish the location of the problem. These students had to rely on the output of the program at run-time to establish what had gone wrong in the program. Figure 6-9 demonstrates that students in Group 1 who did not receive feedback and assistance with these errors spent longer on questions 6 and 7 than students in Group 2 and 3 who received feedback from SNOOPIE on the semantic errors.

Support with problem formulation (SNOOPIE v2) aids students in producing a solution that meets the criteria of the question. Questions 1 and 4 demonstrate that those students in Group 3, receiving support with problem formulation, were significantly more likely to produce a solution that met the criteria of the questions than students not receiving problem formulation support, i.e. Groups 1 and 2. It is also clear from these results that students using SNOOPIE v2 support, Group 3, are significantly more likely to produce a solution that meets the criteria of the question than students using the standard Compiler, Group 1. While there was no significant difference between Group 1 and Group 2 or between Group 2 and Group 3, Group 2 performance lay between Groups 1 and 3. These results demonstrate that both program formulation and problem formulation support solution development and this support is maximal where both are provided.

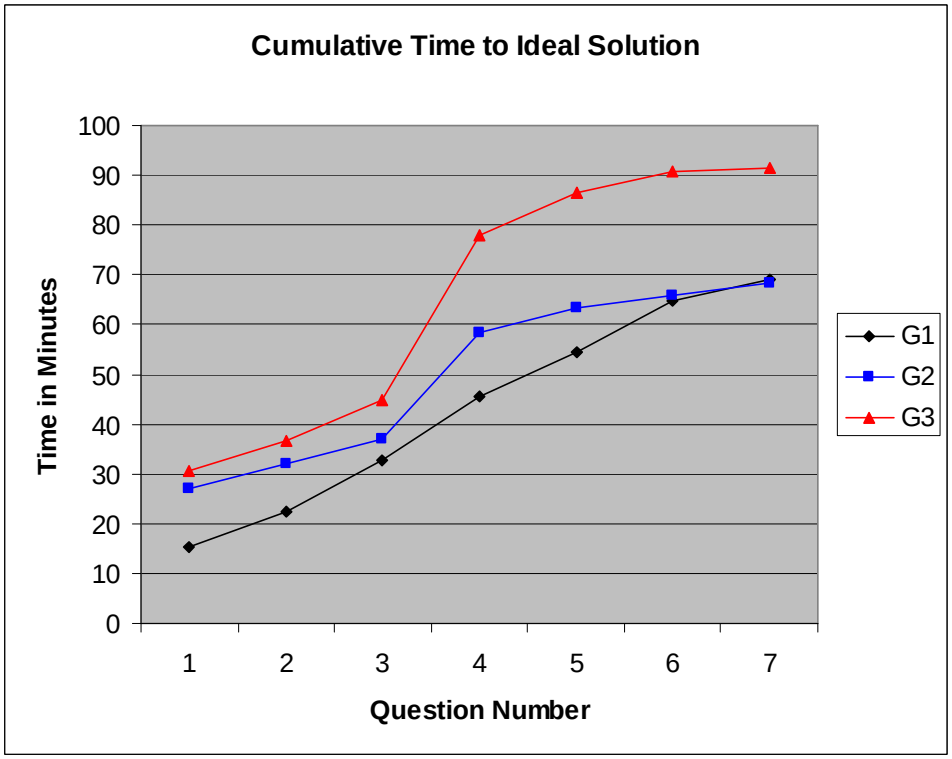


Figure 6-9 Cumulative Average times taken per group to ideal solution

An important observation is that those students in Group 3 took longer to produce a solution that met the criteria of questions 1 and 4 than students in Groups 1 and 2, as shown in Figure 6-9. This graph shows the cumulative average time taken to reach an ideal solution for each group. This demonstrates that SNOOPIE v2 is not simply providing the students with the answer in a series of stages. Instead, SNOOPIE v2 is promoting considered engagement with the problem structured in a stepwise manner, and useful engagement with a problem actively promotes learning (Koile and Singer, 2006). The data used in Figure 6-9 is taken from each student's compilation log, where they produced an ideal solution for a particular question. There were two students in Group 1 whose compilation logs could not be used for this graph as they had not used the Group 1 button consistently throughout the experiment, therefore the time stamps were incomplete and could not be declared as accurate. This reflects a flaw in the design of the experiment, where students were able to access an equivalent functionality (the standard compiler) that was not part of the intended experimental design, and so did not log compilation attempts. Note that, based on the intervals between compilation attempts, the

majority of students in Group 1 did not switch to the standard Compiler for any substantial length of time and so behavioural data is representative. Further, based on logs and observation, students in Groups 2 and 3 persisted in using the designated button in recognition of the additional help provided.

6.5 Module Performance

Figure 6-10 shows the average module grades for first year students at the University of Abertay Dundee in the academic years 2004 to 2005 and 2005 to 2006. The modules shown are only those modules that students undertaking the programming module studied as part of their course. Those students in the academic year 2004 to 2005 had access to SNOOPIE v1 and those students in year 2005 to 2006 had access to SNOOPIE v2. Module performance across the two cohorts may be used to determine if SNOOPIE v2 has had an impact on the performance of students undertaking the programming module (OOP1). Where possible, the module delivery in the session 2004 to 2005 was replicated in the session 2005 to 2006. The teaching materials, delivery schedule and environment were the same. The core teaching staff was also consistent, although there was some variation in support staff. Note that comparison between students in the session 2003 to 2004 and the session 2004 to 2005 is not meaningful because a number of factors changed substantially. Students moved from a 3-hour per week delivery model to a 4-hour per week structure, the module changed from being 12 CATS points to 15 CATS points, a separate lecture and laboratory model in 2003/4 was revised into an integrated lecture/ laboratory style of teaching in 2004 to 2005 (and 2005 to 2006), and the module was taught ‘fat’ in one term (Term 2) in 2003 to 2004 whereas it was taught ‘thin’ (both terms) in 2004 to 2005.

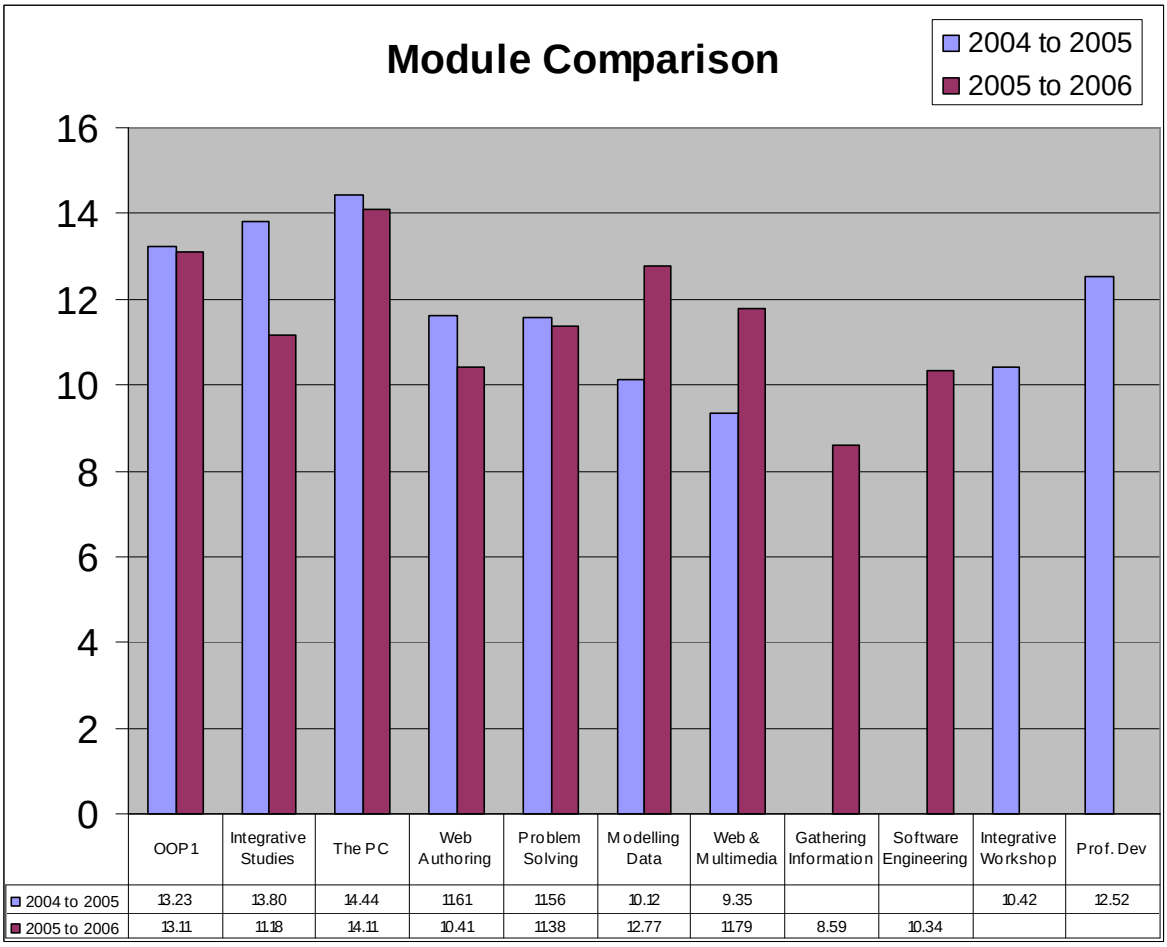


Figure 6-10 Average Module Grades at the University Of Abertay Dundee

Overall, average grades for the module are lower in the year 2005-2006 than in the session 2004-2005 with the exception of Modelling Data and Web and Multimedia Fundamentals. The Modelling Data module was delivered in Semester 2 in the session 2004-5 and in Semester 1 in the session 2005-2006, which most likely accounts for the higher average grade since students are typically more motivated in Semester 1. The Multimedia Fundamentals module was taught by different tutors, which could also account for the difference in module grade. For OOP1 there is no statistically significant difference in module performance between 2004 - 2005 and 2005 – 2006. The absolute average grade is lower in 2005 – 2006, although this lower average is within the content of a generally lower set of average grades for all modules that were largely consistent across academic session

(OOP1, The PC, Integrative Studies, Web Authoring and Problem Solving). In 2004 - 2005 the average grade for these modules was 12.93; in 2005 - 2006 this average was 12.03. In 2004, the average OOP1 grade was only 0.3 above the overall average; in 2005 to 2006 the average OOP1 grade was 1.07 above the average. This demonstrates a lower absolute but better relative performance for OOP1 within a consistent context. Further the relative ranking of OOP1 has changed from being 3rd tope module in 2004-2005 to 2ns tope module in 2005-2006.

6.6 Questionnaire Results from University of Abertay students

In order to find out the patterns of usage that students employed when using SNOOPIE throughout the academic year 2005 to 2006, a questionnaire was distributed towards the end of the module to all students attending the practical laboratory sessions. Thirty questionnaires were completed (see Appendix D for questionnaire and full results). Figure 6-11 shows the distribution of how students used SNOOPIE. Of the 30 students who completed the questionnaire, only 27% indicated that they had not made use of SNOOPIE (defined as Rare Users). A further 27% of the respondents were heavy users of SNOOPIE, typically observed to use it *instead* of the standard Java Compiler (defined as Regular Frequent Users). The remaining 46% of the students used SNOOPIE at various intervals throughout the module, defined as Initial Frequent Users (i.e. they used it a lot at the beginning of the module), Belated Frequent Users (i.e. they used it a lot towards the end of the module) or Regular Sporadic Users (i.e. they used it when they were stuck throughout the module). Students in the Initial Frequent User category used SNOOPIE less towards the end of the module as they became more confident in their programming abilities. Students in the Belated Frequent User category did not initially recognise the value in using SNOOPIE. Those students who used SNOOPIE at various intervals (Regular Sporadic Users) were typically observed to use SNOOPIE when they were stuck with an aspect of the question or the program before seeking assistance from the tutor or from a peer.

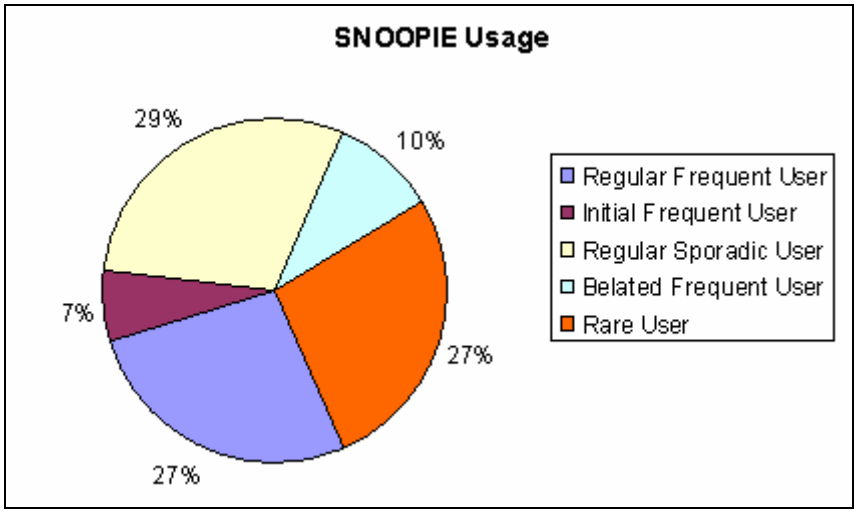


Figure 6-11 SNOOPIE Usage Throughout the Term as Indicated by the Questionnaire Results

Of those students who chose not to use SNOOPIE at all, 88% of these respondents indicated that they preferred to get assistance from a tutor or a friend. Therefore it can be assumed that their lack of engagement with the system was not caused by poor design of SNOOPIE. The majority of the students agreed that SNOOPIE helped them to solve a problem if the tutor was busy helping another student.

88% of the students in the 'regular frequent user' category used SNOOPIE to check that their solutions were correct and all the students agreed that SNOOPIE helped them to resolve mistakes that they had made with their code. 85% of the students who chose to use SNOOPIE at some stage in the term agreed that SNOOPIE helped them to produce a working solution. 75% of the students indicated that sometimes SNOOPIE did not provide enough help although they acknowledged that they did not want the tool simply to provide them with the answer.

6.7 Student Interviews

Three students have been formally interviewed on a one-to-one basis to ascertain their views and opinions of SNOOPIE. The students all participated in the module SA0721a Object Oriented Programming at the University of Abertay Dundee during the academic year 2005

to 2006 where SNOOPIE was integrated into the module delivery. All interviews were recorded and a transcription of the conversations can be found in Appendices E,F and G. Student A (Appendix D) had previous programming experience from school. Student B (Appendix F) had no previous programming experience. Student C (Appendix G) had previous industrial programming experience prior to becoming a student. Student A chose to use SNOOPIE sometimes during the course of the module, in addition to the standard Compiler when they were unable to resolve a problem. Student B typically used SNOOPIE instead of the standard Compiler. Student C chose not to use SNOOPIE.

Selected comments from the interview with Student A are listed in Table 6-2. Comment 1 demonstrates that Student A found that SNOOPIE became more useful as the programming concepts and exercises became more complex, and especially in cases where SNOOPIE made direct reference to the content of the teaching materials. Comments 2, 3 and 6 further indicate the value in providing context sensitive support. Comment 4 demonstrates how the students used SNOOPIE when the tutor was busy and how some students used the feedback from SNOOPIE to help explain to their peers what was wrong with their program. Based on observations, questionnaires (Section 6.6) and informal dialogue with other students, Student A's usage of SNOOPIE was consistent with more than a third of students in the module, i.e. SNOOPIE was used in conjunction with the standard Compiler and not instead of that Compiler (see comment 5). Student A found SNOOPIE particularly useful in developing strategies for resolving common syntax errors so that they were eventually able to resolve these errors without the additional feedback (comment 7).

Table 6-2 Selected Comments from Interview with Student A, full transcript in Appendix E

1	"Semester 1 to begin with I didn't use it at all I mostly got help [from the tutor] but gradually as it started to get harder and harder and the lecturers were busier and busier you know with other people I started using one and it was good you know cause it didn't have all the cryptic messages and stuff it had like it pointed it out and because it was based on the robby and stuff where as their tool wasn't it was just based on general programming it was straight to the point like robby can't do this because you've not got this or you've not got the next thing." (Comment 17)
2	"like I said how it's straight to the point unlike their own [the standard Java Compiler messages] it tells you what line it is what's wrong with it why robby can't do it things like that" (Comment 23)
3	"so it's not like variable string da da da and it's got all this jargon it's not got all the jargon like to begin with" (Comment 27)
4	"it made them much better I think cause like I said…em…the lecturers would be helping other people and there was only like 2 lecturers or 3 at most at one time so you know they were busy at the end of the room and you'd be like 'ch ch ch' for half an hour…waiting but with that you could click on it and it would tell you and you could work through it and you could get it and then if someone else beside you was stuck on it like say I sat beside my friends Fred and Barney and they weren't…they weren't as prolific at programming [laughs] as me… so I would… we would look at it together and then we would do it together we'd go over it and I'd show them or maybe they'd get it" (Comment 45)
5	"see I didn't do that I used…I did both Compilers I would use the main one and your one just to see what they said to compare them but I used your own mostly sometimes" (Comment 61)
6	"the support tool takes your program, finds the errors and the helps you…gives you better messages than the standard Compiler…helps you with the actual program instead of just the general syntax and stuff it helps you with the actual robby not…not with programming in general it's actually specific for robby for the module" (Comment 83)
7	"it taught me how to error trap easier, it showed me… it taught me how to look through the program and find stuff that you wouldn't normally have found with the Compiler, but see if there were semicolons missing that were hard to pinpoint, I would manage to pinpoint them easier because I'd been using the support tool and that'll be like look at this error and it would teach you…it would stay in my brain that that was what I had to look for." (Comment 85)

Selected comments from the interview with Student B are listed in Table 6-3. Comment 1 demonstrates that Student B used SNOOPIE to ensure that they were developing a solution that met the criteria of the question, not so much for the specific code fragments given but more for reassurance that their understanding of the question and strategy for development were correct. Comments 2, 5 and 6 demonstrate the perceived usefulness of the hyperlinks to additional resources in the problem support provided by SNOOPIE. Student B found these

links particularly helpful as they focused on the relevant programming concept when they were doing the practical exercise and it was more relevant to them than during a lecture. Comments 3 and 4 highlights that Student B did not become dependent on the tool and was able to use it less as they developed more confidence in their programming ability.

Table 6-3 Selected Comments from Interview with Student B, full transcript in Appendix F

1	"I used it most of the time and quite a lot I'd use it to back up something that I wasn't, you know I maybe had an idea in my head of that was how to do it but I'd maybe check against the support tool to see if I was going in the right direction." (Comment 8)
2	"It did especially, say… the links, because you could sit through a lecture and maybe not take it in, you know… I thought some of the lectures were quite long and by the end of it the stuff was going over your head so that was definitely helpful to have some sort of reference that you could go back to, it just sort of refreshes your mind a bit." (Comment 18)
3	"Sometimes if I knew that I wouldn't need much help I would try the standard Compiler first before using the tool but I definitely used the support tool more." (Comment 22)
4	"Definitely in the first semester but towards the end of the module I just used it to check that what I'd typed with my code was ok." (Comment 26)
5	"Yes, especially with the PowerPoint things because it pulled out the relevant lecture notes so that definitely helped." (Comment 28)
6	"Yes it helped me to understand because I was working on that bit at that time so it was sinking in more because I was actually putting it into practice rather than just reading it so yes that definitely helped." (Comment 32)

Selected comments from the interview with Student C are listed in Table 6-4. Comment 3 indicates why Student C did not use SNOOPIE: as a competent programmer this students did not need additional support to resolve syntax errors or identify a solution to a given problem. Comment 1 demonstrates that Student C recognised the value of SNOOPIE to other students who were able to use SNOOPIE to resolve some issues in their program rather than waiting for help from a tutor. Comment 2 also supports this view that SNOOPIE helped the programming classes to operate quicker as students were more able to help themselves

Table 6-4 Selected Comments from Interview with Student C, full transcript in Appendix G

1	"it helps you get help quicker than… you know cause there's only you [the author] and another tutor in the class… a lot of people can get help themselves [from SNOOPIE] rather than just waiting for it" (Comment 26)
2	" yeah… helped them run faster, (Comment 28)
3	"um I just didn't … really need to… I would… I never really thought" (Comment 73)

6.8 Staff comments

Two members of staff at the University of Abertay Dundee have provided a commentary on their views of SNOOPIE. Both staff taught on the module SA0721a Object Oriented Programming at the University of Abertay Dundee during the academic year 2005 to 2006 where SNOOPIE was integrated into the module delivery. Staff tutor A was the module leader, staff tutor B was a supporting tutor. The commentary can be found in Appendices H and I respectively.

Selected comments from staff tutor A's commentary are listed in Table 6-5. Comment 1 indicates the ease with which staff tutor A observed students invoking SNOOPIE and provides evidence that it was not a complicated tool requiring an additional learning curve. The high uptake observed by staff tutor A indicated that students recognised the value of SNOOPIE (Comment 3). Staff tutor A also noted that although the (Syntax) support offered by SNOOPIE made assumptions about the most likely cause of an error, this cause was most often correct (Comment 2). Staff tutor A observed that SNOOPIE helped students to resolve errors on their own, helping to promote confidence (Comment 4). Comment 5 demonstrates that SNOOPIE was not used as a crutch by the students but instead they used it as a tool to help them learn. One of the main advantages of SNOOPIE from a tutor's perspective was the consistency in feedback from one staff member to another. SNOOPIE was used by the staff to direct students on the required solution using appropriate terminology (Comment 6).

Staff tutor A was observed twice during the academic year 2005 to 2006 by external academics. Both academics responded very positively to SNOOPIE and one is now

implementing the idea for the teaching language they use with their own first years (Comment 7). Staff tutor A found SNOOPIE to be a substantial contribution to the programming module (Comment 8) both from a tutor and student's perspective.

However, staff tutor A did notice that even when students used SNOOPIE and the feedback provided was exactly what was needed to solve the problem, some students were still observed to request tutor assistance. When the student was then asked by the tutor to read the feedback again, the students then realised what they needed to do to resolve the issue. Staff tutor A has suggested that some changes be made to the appearance of the text of the tool, perhaps with colour additions so that a student can read the text quickly and extract the essential information (Comment 9).

Notably, based on the experiences of the academic year 2005 to 2006, staff tutor A has requested SNOOPIE support to be provided for students on the programming module in the academic year 2006 to 2007 in recognition of its value in the previous academic year (Comment 10).

Table 6-5 Selected Comments from Staff A, Appendix H.

1	SNOOPIE tool was very easy for students to invoke. (Paragraph 2)
2	Of course, the specific text supporting each error message made assumptions about the most common cause(s) of that error. It is worth noting that these assumptions were invariably correct. (Paragraph 2)
3	Students in the 2004/5 cohort were receptive towards SNOOPIE version 1. (Paragraph 4)
4	The tool afforded more opportunity for students to correct errors on their own and I could see that this promoted confidence in their ability to solve problems. (Paragraph 4)
5	Some students, generally those that already had some exposure to programming, did not want to use the tool preferring to work with the basic compiler support only. At the other end of the spectrum, those students that were typically least confident in their programming ability used the tool most of the time. Interestingly, even though (at my request for a small number of formative exercises) SNOOPIE almost wrote the code for the student by guiding them through the program

	required for each requirement specified, at no point did I see a student blindly stringing together the advice provided. All students wanted to learn to program and used SNOOPIE for support and not as a shortcut to exercise completion. (Paragraph 7)
6	A useful side effect of SNOOPIE was consistency in staff feedback to students. Several staff were involved in the module and SNOOPIE feedback provided a useful platform for support staff to advise students in a manner consistent with my intentions for a given question. All support staff that have worked on the module have made this comment. (Paragraph 8)
7	Additionally, my teaching has been peer observed by two other Universities in the 2005/6 session. Both observers were very positive about the Version 2 support, and were particularly impressed by the flexibility in support available. In fact, one of the observers has taken the concept of the support offered and is implementing a similar tool for his functional programming course. (Paragraph 8)
8	The SNOOPIE tool has made a significant contribution to the teaching of programming, including promoting confidence in students who are otherwise hesitant in exploring program development on their own and reducing the time spent by staff solving problems that students can solve with SNOOPIE support. (Paragraph 9)
9	Repeatedly, students did not actually read the text provided by SNOOPIE. In some but not all cases, the text provided was expressed as a contiguous block. Perhaps colour coding of key textual elements would make the text more accessible. That said, the text was written clearly and concisely and those students who chose to read it found it useful. (Paragraph 9)
10	Perhaps the strongest supportive statement I am able to make is that, at my request, I have been able to incorporate the SNOOPIE software in the module delivery for this 2006/7 session. (Paragraph 10)

Selected comments from staff tutor B's commentary are listed in Table 6-6. Comment 1 demonstrates the positive view that staff tutor B has of SNOOPIE. Comment 2 indicates that staff tutor B has observed errors having a negative impact on a student's learning experience in previous years and notes in comment 4 that SNOOPIE helps the student with these errors by providing more useful feedback. Staff tutor B indicates that this enables them to spend

more time explaining the complex issues of programming to a student. Staff tutor B has observed (comment 3) that SNOOPIE encourages self-learning in the student through the links to PowerPoint slides and simplified feedback. Comment 5 demonstrates that staff tutor B observed students using the structured feedback of SNOOPIE (version 2) when they were unsure how to progress with the programming exercise. Staff tutor B notes that SNOOPIE was of particular value to support staff on the module as SNOOPIE helped them to ensure that the feedback that they provided to the student was consistent with that offered by another member of staff (comment 6). Staff tutor B notes in comment 7 that the students give positive feedback about SNOOPIE and that SNOOPIE can teach them how to interpret the normal Java Compiler error messages (comment 8).

Table 6-6 Selected Comments from Staff B, Appendix I.

1.	I find it to be an extremely useful teaching tool. (Paragraph 1)
2.	In previous years, when faced with a list of seemingly complex errors, students new to programming can panic and feel as if they will never resolve them. SNOOPIE encourages the student to deal with one error at a time, and can pick up the syntax errors and provide useful feedback on how to resolve them. This allows me as a tutor to spend more time explaining the more complex problems to the students rather than trouble shooting trivial errors. (Paragraph 2)
3.	I have found that SNOOPIE encourages self-learning, as not only does it provide simplified explanations of the errors, there are links to Power Point slides that give programming examples. (Paragraph 3)
4.	The dialogue employed by SNOOPIE encourages the student to explain and think about the problem they are having in the correct terminology. I have found that students who can resolve the majority of the compiler errors that they encounter have more belief in their own abilities and therefore learn better. (Paragraph 4)
5.	SNOOPIE version 2 not only provides feedback for syntax errors, but it can provide structured feedback to set exercises that are not detected by the compiler. From an instructor's point of view, this guides the student towards a model solution by providing outputs that give guidance as to the structure of the program (e.g. prompting the student to include two for loops). I found this to be useful for students that were not sure where to begin as it enabled them to make a start to the program. (Paragraph 5)
6.	As there are several instructors involved in teaching the module, it enables us to easily give uniform guidance to the students on solutions, which prevents confusion when they discuss the solutions with their peers. (Paragraph 6)
7.	The feedback that I get from the students about SNOOPIE is also very positive. They can get immediate hints and guidance without having to wait for tutor assistance, and because it is optional as to whether they use the compiler or SNOOPIE, those that are capable programmers are not held back from using the normal system. (Paragraph 7)
8.	Additionally, because the compiler errors are also displayed, when the students are using a normal compiler, they recognise and are able to solve errors that they have encountered and have been explained by the SNOOPIE tool. I find it a very useful addition to the learning tools for novice programmers. (Paragraph 8)

6.9 Discussion

SNOOPIE has been evaluated with a number of mechanisms consistent with the evaluation of other teaching tools. The results of these independent mechanisms may be integrated to test the five hypotheses described in Section 6.3.6.

1 The program formulation support provided by SNOOPIE (v1 and v2) improves short-term student performance.

2 The problem formulation support provided by SNOOPIE (v2) improves short-term student performance.
3 The problem formulation support provided by SNOOPIE (v2) reduces the time taken to complete an ideal solution.
4 A combination of problem and program formulation (SNOOPIE v2) improves long-term student performance when compared with program formulation alone (SNOOPIE v1).
5 Students perceive that SNOOPIE (v2) has a positive impact on the learning process.

Hypothesis 1 was tested using experiments, as described in Section 6.4. Student performance was measured as a student's ability to resolve errors relating to program formulation that had been deliberately inserted into programming files, which were given to the students. The results of these experiments demonstrate that those students receiving support with program formulation, i.e. those receiving more detailed feedback on Syntax errors and warnings of Semantic mistakes, were significantly more likely to resolve the errors than those students receiving feedback only from the standard Compiler. Additionally, students receiving support with program formulation, i.e. SNOOPIE v1 or v2, were able to solve these errors faster than those students without SNOOPIE support. Therefore it may be concluded that program formulation support improves short-term student performance.

Hypothesis 2 was tested using the experiments described in Section 6.4. Student performance was measured as a student's ability to produce a programming solution to a given problem, ensuring that it met the criteria of the question. The results of the experiments demonstrated that those students receiving problem formulation support (SNOOPIE v2) were significantly more likely to produce a solution that met the criteria of the question than those students who did not receive problem formulation support. It may be concluded that program formulation support improves short-term student performance.

Hypothesis 3 was tested using the experiment described in Section 6.4. Students receiving problem formulation support (SNOOPIE v2) took longer on average to complete an ideal solution than those students in other groups. Of the students who were able to produce this solution, students receiving problem formulation support typically spent longer on the question than those students receiving other forms of support. This demonstrates that SNOOPIE v2 was not spoon-feeding the students with the answer. If this was the case, students using SNOOPIE v2 should have produced a solution quicker than other students.

Instead, students who would otherwise not have completed the questions, based on the higher percentage of students that completed the question compared with other experiment groups, were able to think in more detail about the requirements of the question and effectively identify the appropriate constructs and their relationships with support from SNOOPIE. Therefore problem formulation support does not reduce the time taken to complete an ideal solution.

Hypothesis 4 was tested using a combination of module performance, interviews with students and written statements from module tutors. Section 6.5 describes the module results from the academic year 2004 to 2005 where students used SNOOPIE v1 (program formulation support) and 2005 to 2006 where students used SNOOPIE v2 (program and problem formulation support). Whilst the results described in the section are not significant, the module has improved from being the third top module with SNOOPIE v1 support to the second top module with SNOOPIE v2 support and the relative average performance of the module has improved in 2005 to 2006. Student interviews, as described in Section 6.7 demonstrate the role SNOOPIE had in learning to program for a sample of students. Those students who used the SNOOPIE support reported that the feedback enabled them to resolve an issue with the program that they could not previously resolve with the standard Compiler. The hyperlinks to relevant PowerPoint slides were particularly useful for students attempting to comprehend the necessary constructs and their required relationships for a given question. Comments from module tutors, as described in Section 6.8, have been particularly useful in comparing the effect of SNOOPIE v2 on long-term student performance against SNOOPIE v1. Staff tutor A was aware of students requesting tutor assistance with different types of problems: In the previous academic year when students had access only to program formulation support they requested more assistance with questions relating to problem formulation. Student confidence was seen to improve through their use of SNOOPIE v2. SNOOPIE v2 does improve long-term student performance compared with program formulation alone (SNOOPIE v1).

Hypothesis 5 was tested using a combination of experiments (Section 6.4), questionnaires (Section 6.6) and interviews with students (Section 6.7). In the experiment, the recorded log

files demonstrated that the students receiving program and problem formulation support (SNOOPIE v2) were least likely to change groups. Students receiving program formulation support were less likely to change groups than students receiving support only from the standard Compiler. It can be assumed that students were aware of the positive impact that using SNOOPIE v2 would have on the performance in the experiment and this influenced their decision over which button to use, regardless of the button they had been requested to use as part of the experiment. The questionnaire results demonstrate that the students recognised the value of the SNOOPIE v2 support, as 73% of those who completed the questionnaire indicated that they used it. Even those students who chose not to use SNOOPIE stated that they thought it was a good idea, but indicated that it did not suit their needs or learning styles. Both students interviewed who used SNOOPIE had perceived it to add value to their learning experience. One student said explicitly that it helped them learn the most likely cause of common errors in a program. Another student found that some of the problem formulation support was of more value than the lectures. Therefore the students generally perceived SNOOPIE v2 to have a positive impact on the learning process.

6.10 Summary

The literature review of Chapter 2 and subsequent support tool analyses of Chapter 3 led to a set of requirements for the design of a support tool for novice programmers. For convenience, these requirements are shown again in Table 6-7 for information. These requirements were aligned with a conceptual framework to inform implementation. Additionally, Chapter 3 provided a summary analysis of the review of existing support tools in tabular format with respect to each of the identified requirements. The extent to which the sample implementation meets the requirements is reviewed below and, by way of summary, the review table from Chapter 3 is repeated here with the addition of SNOOPIE (Table 6-8).

Table 6-7 Core Requirements of an Effective Support Tool

1	All forms of support may be progressively reduced over the teaching period
2	Present both standard Compiler and enhanced support concurrently
3	Identify and advise on commonly observed semantic errors
4	Embody knowledge of key constructs needed to solve a given problem

5	Embody knowledge of the relationships between the constructs needed to solve the problem
6	Where appropriate, ensure that this knowledge accommodates variant solution forms
7	The knowledge should be disseminated to students in successive stages
8	Link to teaching resources as a means of information delivery and student-tutor dialogue
9	Use of the tool must be voluntary on the part of the student.

Table 6-8 Revised summary table of existing support tools with respect to the above requirements.

Tool	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
Flint (Zeigler and Crews, 1999)									✓
Cap (Schorsch, 1995)	✓	✓	✓				✓		✓
Datlab (MacNish, 2000)				✓	✓				✓
InStep (,2001)			✓						✓
Toolkit (Lang, 2002)									✓
Expresso (Hristova et al, 2003)			✓						✓
CmeRun (Etheredge, 2004)		✓	✓						✓
Proust (Johnson and Soloway, 1985)				✓	✓	✓			
Lisp Tutor (Andrerson and Skwarecki, 1986)				✓	✓				
SWANN (Brna and Mathieson, 1993)				✓		✓			
IVC (Moore and Taylor, 2005)	✓	✓	✓					✓	✓
SNOOPIE	✓	✓	✓	✓	✓	✓	✓	✓	✓

The implementation of SNOOPIE allows for all forms of support to be reduced over the teaching period to discourage dependency on a high level of support (Requirement 1). For program formulation, the extended messages for Syntax errors can be changed and particular

semantic checks can be turned off or reworded. Problem formulation support can also be progressively reduced by implementing fewer priority checks for a particular tutorial exercise and providing less guidance within each priority check. For the instantiation of SNOOPIE, it was not felt necessary to reduce the program formulation support. Problem formulation support was reduced throughout the teaching period, and was reduced even within the practical exercises for a given week, i.e. the first few formative exercises of a given week's exercises would receive a higher level of SNOOPIE support that the later summative exercises. Students gradually weaned themselves off the SNOOPIE support and were not dependent on it to complete a practical exercise, as evidenced through student interviews, staff tutor comments and general observations.

SNOOPIE has been implemented to provide both the standard Compiler and the extended error messages concurrently (Requirement 2). During the student interviews, one student in particular commented on how useful this was in teaching them the real (most likely) meaning of the standard errors so that eventually they were able to interpret these errors without the SNOOPIE extensions. The Semantic Support provided by SNOOPIE was consistent with the common mistakes made at this level (Requirement 3). No obvious semantic errors were noticed and not addressed during the academic year 2005 to 2006.

The priority checks implemented in SNOOPIE allow for knowledge of key constructs and their relationships to be embodied within the SNOOPIE feedback (Requirements 4 and 5). SNOOPIE has been implemented to accommodate variant solutions to a given problem, for example if a student has produced a solution that includes either for loops or while loops, and either is acceptable for the given exercise, SNOOPIE will offer guidance on both variants (Requirement 6). Further, the guidance given takes account of the initial intentions of the student and supports them accordingly. The framework of a series of priority checks allows the feedback on various required constructs and relationships to be disseminated in successive stages (Requirement 7). This serialised support was observed to promote a stepwise development practice. Hyperlinks to PowerPoint slides in the feedback allowed students to review the material at their own pace and helped tutors ensure consistency in dialogue (Requirement 8).

Finally, SNOOPIE has been installed as an additional button within the IDE, and implemented as a stand-alone application requiring only the current filename as an argument. Students were able to use the support as and when they wanted it; access to the standard Compiler remained the same. This ensured that SNOOPIE use was entirely voluntary (Requirement 9).

Chapter 7 Summary and Future Work

7.1 Summary

The main aim of this research was to design, implement and evaluate a novel support tool for novice programmers, which addressed the challenges faced when learning to program. A review of the existing literature explored the different levels of knowledge required to undertake the activity of programming and the need to promote deep learning through problem-led engagement in a harsh learning environment. The review revealed that programming is a complex activity requiring concurrent engagement with different levels of knowledge relating to generic programming language rules and operation, together with strategies for analysing programs and structuring a code-based solution. These levels were structured in accordance with other studies into two categories: program formulation and problem formulation. Program formulation, i.e. knowledge of syntax and the semantics of individual lines of code was shown to be characteristically different but fundamentally coupled to the activity of problem formulation, i.e. the process of translating a specific problem into a specific solution. The interrelated knowledge levels of Syntax, Semantic, Schematic and Strategic have been explicitly mapped onto these two areas, and the necessity of supporting novice programmers with these facets is made clear.

Existing support tools that have been used with different approaches to programming were also considered. Support tools exist for many stages in the software development process and a broad cross-section was reviewed to contextualise the design and implementation. In recognition of the problems faced in the activity of program formulation, many support tools had been developed to aid students in developing syntactically developed code, and a smaller number considered the semantics of individual lines of code. A different, and smaller, set of tools targeted the challenges posed by problem formulation. The review of knowledge levels and existing tools led to a clear list of requirements to inform the development of a learning support tool.

A conceptual framework was presented which can be used to inform the design and implementation of a learning support tool that meets these requirements. The framework indicated how Syntactic Support could be provided by extending original (compiler) error messages with additional explanatory text that was relevant to the taught material. Likewise, Semantic Support could be provided through performing simple program checks for (known) common novice mistakes. For problem formulation, Schematic and Strategic levels could be supported using knowledge of the problem domain to guide students on the correct use of programming constructs and their relationships.

In order to verify the evidence in the literature from which the learning model underpinning tool development was derived and to determine the nature of the feedback required, laboratory observations were conducted over a period of two years at two academic institutes. These observations were necessary to explore the relationships between the problems as they manifest themselves at the user interface and the root cause of the problems that students encounter in terms of lack of knowledge. The observations revealed that problems identified at one knowledge level may have their root cause at one or more knowledge levels and it was necessary to develop a learning support tool that recognised multiple, concurrent levels of feedback. These observations were also useful to document the dialogue employed between staff and student when explaining a particular concept or clarifying why an error had occurred. This dialogue was particularly useful in informing the text used in the feedback.

SNOOPIE is presented as an example implementation of the conceptual framework derived from the review and the observations. SNOOPIE provides extensions to the Compiler errors through capture of the original error messages and pattern matching with a database of supplementary information. Semantic level checks associated with programming constructs were undertaken on an abstracted representation of the code. A more substantial framework was presented to support problem formulation activities, based on predefined priorities for a given tutorial exercise. By example, the capacity of this framework to underpin a wide range of priorities was demonstrated.

The evaluation of SNOOPIE was informed by the existing literature review and the mechanisms employed by existing tools are presented within the context of the complexity of evaluating a learning support tool. Based on the capacity of these mechanisms, five testable hypotheses were presented. A series of evaluations was carried out: namely experiments, questionnaires, module performance comparisons, student interviews and staff commentary. The results of each evaluation were discussed independently. The five hypotheses were assessed through integration of the different evaluations to demonstrate that SNOOPIE had an impact on the learning experience. Finally, the SNOOPIE implementation met all initial requirements.

7.2 SNOOPIE Developments

While the sample implementation of the conceptual framework was demonstrated to have a positive impact on the learning process in line with the specified tool requirements, two clear areas for improvement of that implementation arise. First, while the updating of the feedback to take account of specific module content for Syntactic Support is easy and the semantic analyses are already in place, the amendment of priorities for specific exercises is a laborious process and requires a level of knowledge which precludes wide-scale deployment. To address this two proof-of-concept priority development languages, one menu-driven and one scripted, for automated priority creation have been developed, and these are described in Sections 7.2.1 and 7.2.2 respectively. Secondly, staff feedback and observations highlight that the textual display of the SNOOPIE feedback could be improved with the use of more links and a better structural layout. A proposed revision to the feedback interface is presented in Section 7.2.4

7.2.1 Menu Driven Interface For Automated Priority Creation

A student developed a system to automate the marking of first year Java programs as part of a Senior Honours project at the University of St Andrews (Harvey, 2006). The system provides a menu driven interface which allows the tutor to indicate the required structure, style, features and correctness for a given program and indicates how many marks should be

awarded for each section. The characteristics of the system that are relevant to this research are the provisions for selecting structure checks and features (key components) which are required for a solution. The tutor can indicate the minimum and maximum number of classes, methods and variables for a given solution. The tutor can also select from a list box of accommodated key components, those that are required in a program (these are shown in Figure 7-1). For example, if the tutor selects 'while loop' and 'nested for loop' as required features, the system will check the students program for the presence of a while loop and the presence of a nested for loop at some point in the code.

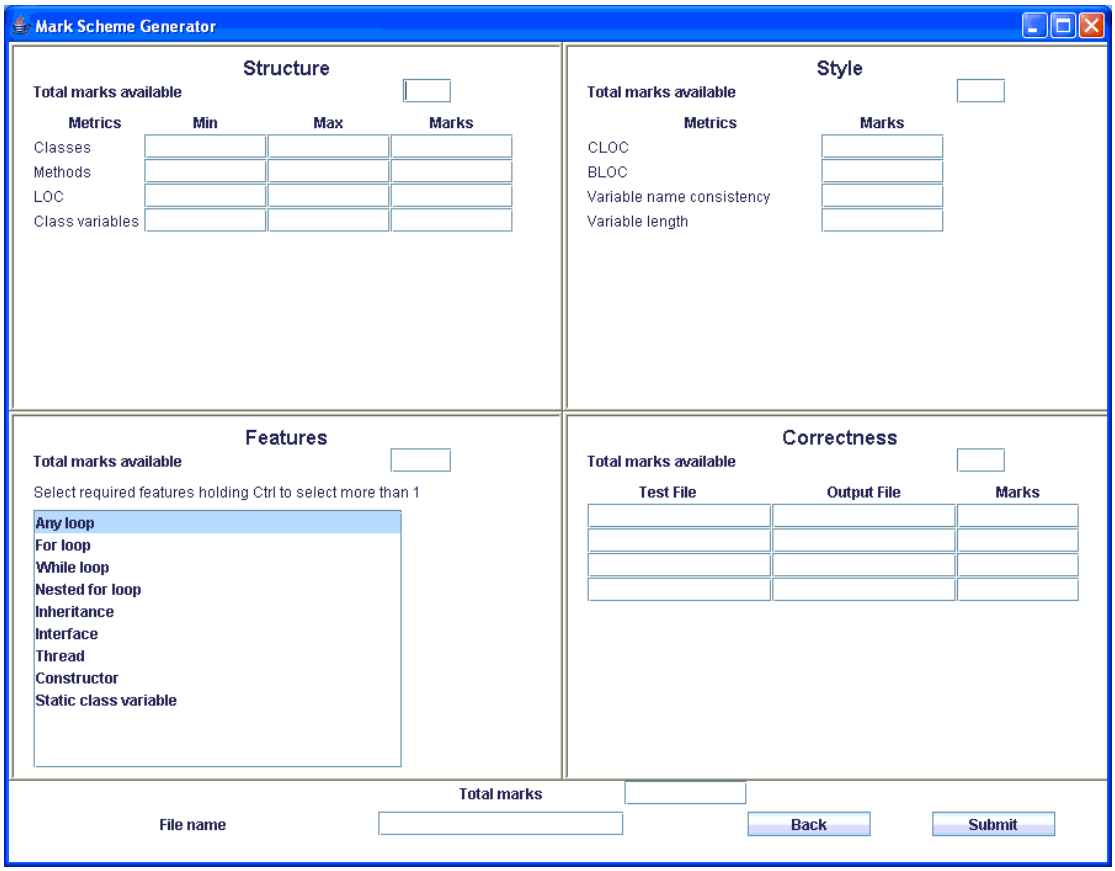


Figure 7-1 Screen shot of Menu Driven interface allowing a tutor to specify the components required for a solution and their associated marks

Following completion of this form, an XML file is created that details the required components and associated marks for each practical exercise. This XML file is then parsed in

order to identify the appropriate checks to perform on the student's code, which is also represented as XML. Although this implementation does not allow the tutor to identify required relationships between key constructs, for example an 'if statement' within the 'while loop' of a method with return type void, or their finer details, for example the number of iterations of a 'for loop' or the condition of an 'if statement', it does demonstrate that a menu-driven interface can be used to perform simple problem formulation checks. In principle, this implementation could be extended to indicate the order in which these checks should be carried out, any programming blocks that they should be contained in and described feedback that could be provided to the student if their program does not include a required feature.

7.2.2 Scripting Language For Automated Priority Creation

A second student developed a scripting language to automate the priority and related feedback creation for a Java program as part of an Honours project at the University of Abertay Dundee (Etter, 2006). The scripting language allows a tutor to specify the required presence of key constructs, the number of instances of a given element and the order of commands, e.g. method calls, in a program. Appropriate feedback is also indicated should a student's program fail a given priority. The scripting language is then parsed and the required Java code generated automatically utilises the Document Object Model, similar to the current problem formulation support of SNOOPIE. For example, the tutor can state that a program must have no more than two methods and one of the methods should be called 'main' as exemplified in Figure 7-2.

```
EXERCISE=tut1_ex2

BEGIN

CHECK methods methCount<3 ON countMatch FEEDBACK (“You have too many methods”) ENDCHECK

CHECK methods methPresent main ON presentFalse FEEDBACK (“You need a main method”) ENDCHECK

END
```

Figure 7-2 Sample Scripting Language

All programs written in the scripting language must start with the keyword EXERCISE. This describes the name of the Java programming exercise that the priority checks relate to. The priority checks are embodied in the keywords BEGIN and END. Each priority begins with the keyword CHECK. The first priority in this example checks that the total count of all methods in the program is not greater than three. If this priority check fails, the feedback “You have too many methods” will be displayed to the student. ENDCHECK is the symbol used to end the checking code.

The Scripting Language allows a tutor to select the key constructs, their relationship and feedback for a given tutorial exercise. In order to use this language, the tutor would be required to learn the syntax and its functionality. This would be almost as time consuming as learning how to use the DOM to create the priorities as described in Chapter 5.

7.2.3 Summary of Automatic Priority Creation

A combination of the approaches described in Sections 7.2.1 and 7.2.2 could be employed to develop a fully automated priority creator. The GUI and menu selection described in Section 7.2.1 could be used as a front-end interface allowing a tutor to select from a list of required constructs. The menu interface could be extended to allow the tutor to indicate any relationships between constructs, order of priorities, programming blocks, to search for key

components, and further details such as number of iterations of a ‘for loop’. The interface could link to the scripting language, allowing the priorities specified by the state of the interface to be created without learning the syntax of the language.

7.2.4 Interface Redesign

The interface of the feedback presented to the student could be improved. It was observed that some students used the tool but still requested tutor assistance, even when the solution to their program was displayed in the SNOOPIE feedback. Figure 7-3 shows the original format that would be used to display example feedback relating to the use of if statements. Figure 7-4 shows a revised, proposed format for displaying this example feedback. The initial statement is highlighted in bold and positioned centrally in the screen. Additional hyperlinks have been added to this statement that will help a student who is confused with the terms ‘if statement’ and ‘else branch’. Figure 7-5 shows the slide displayed on clicking the ‘if statement’ hyperlink. Figure 7-6 shows the slide displayed on clicking on the ‘else branch’ statement. The rest of the feedback is displayed in a block of text, with a hyperlink on ‘if, else if, else statements’. If the student needs clarification on the syntax of an ‘if, else if, else’ statement, they would follow this link, as shown in Figure 7-7. Finally, the student can click on another hyperlink to get more detail on the feedback, as shown in Figure 7-8.

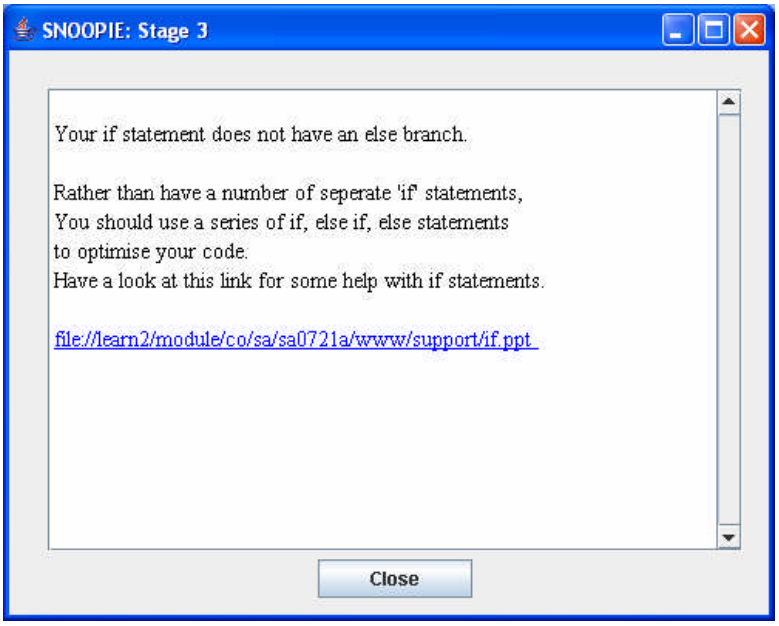


Figure 7-3 Current Interface With Example Feedback

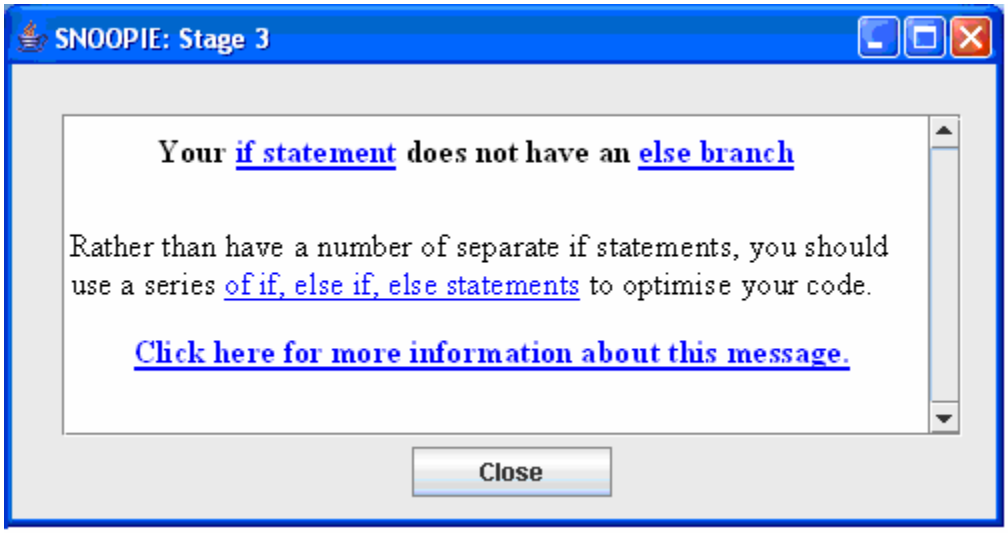


Figure 7-4 Proposed Interface For Example Feedback

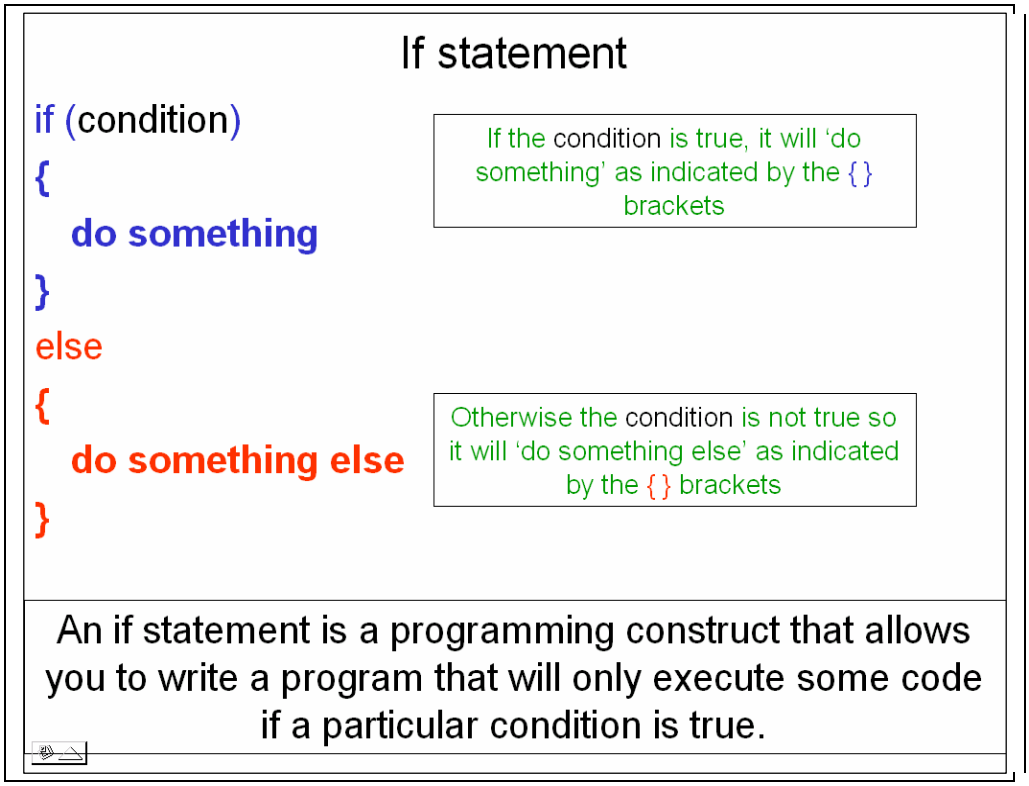


Figure 7-5 PowerPoint Slide, Linked From 'If Statement' Hyperlink

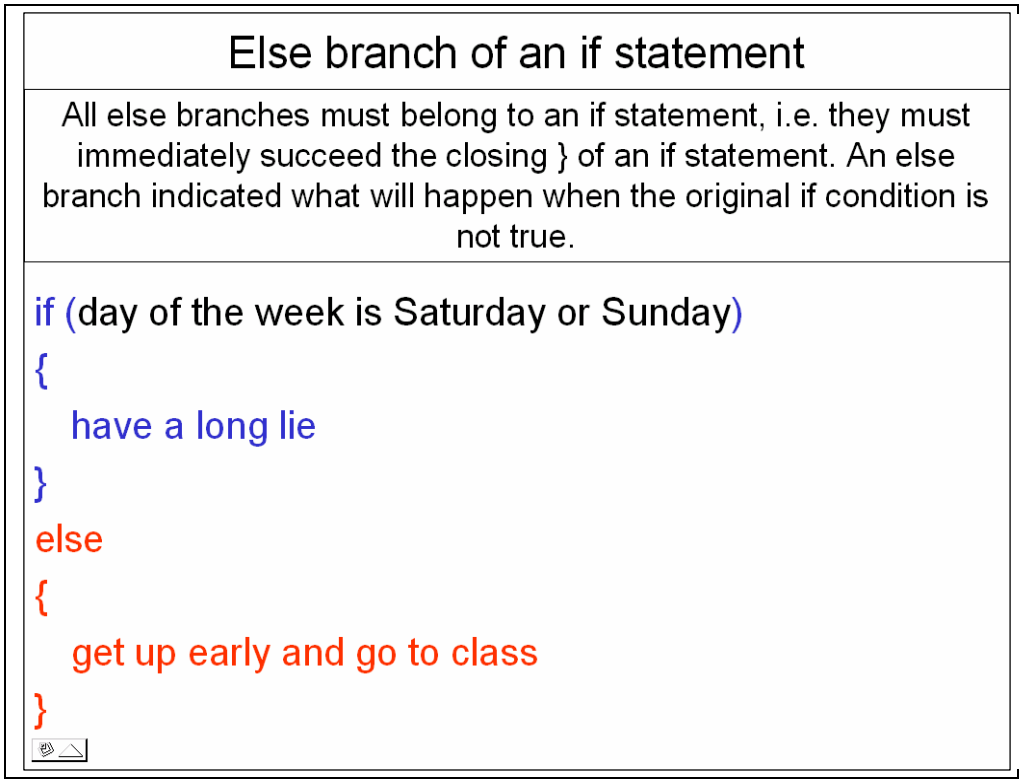


Figure 7-6 PowerPoint Slide, Linked From 'Else branch' Hyperlink

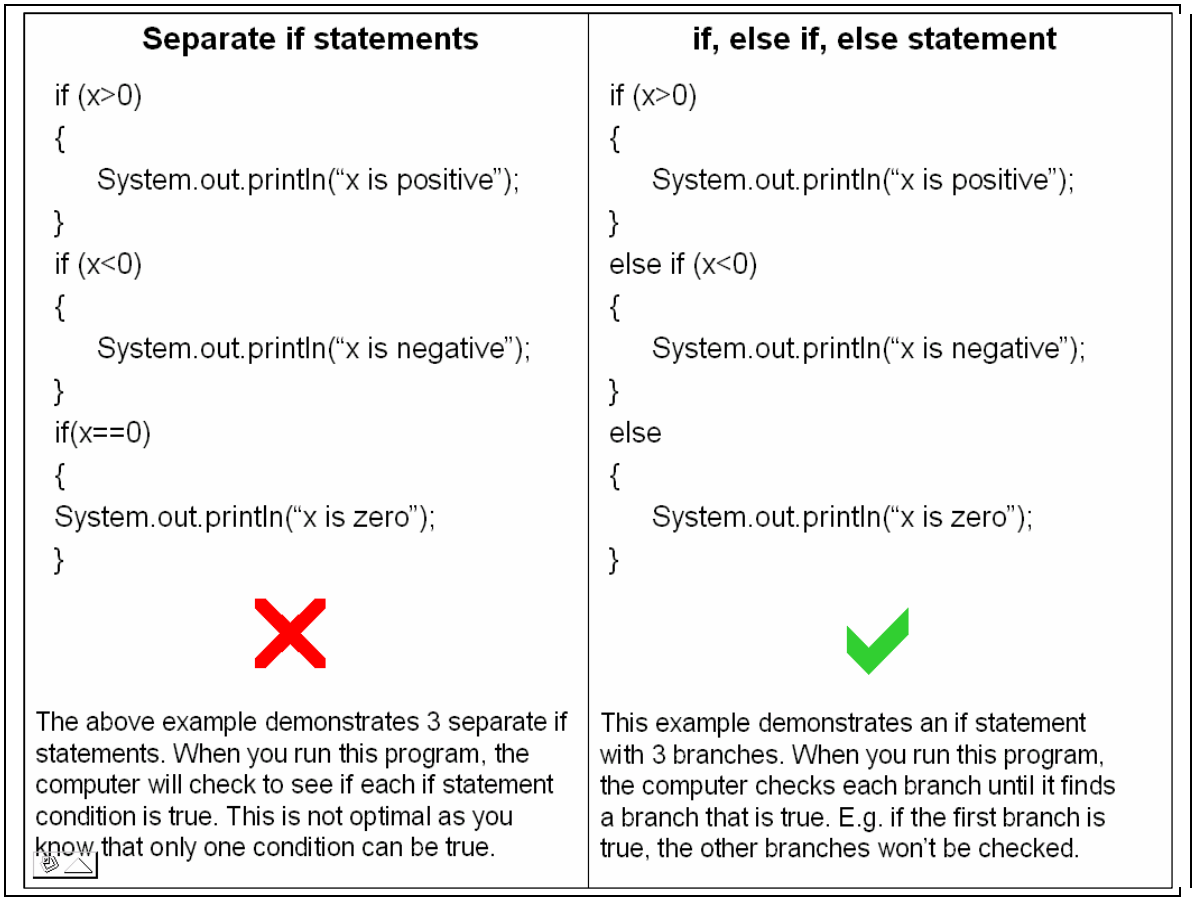


Figure 7-7 PowerPoint Slide, Linked From 'if, else if, else statement' Hyperlink

Your if statement does not have an else branch

An if statement is used when you want to do some code only when a certain condition is true, for example **if a number is less than 0, display a message saying the number is negative**.

For this exercise, you need to indicate that some code will happen when the initial if condition is not true, using an else. As a solution to this question requires you to check more than one condition, you should have an **if else if else** structure similar to the one below.

```
if (x>0)
{
    System.out.println("x is positive");
}
else if (x<0)
{
    System.out.println("x is negative");
}
else
{
    System.out.println("x is zero");
}
```

Figure 7-8 PowerPoint Slide, Linked From 'click here for more information' Hyperlink

The figures above demonstrate how a given error message may be broken down into key words. The inclusion of hyperlinks to describe these key words in more detail allows each student to extract as much or as little information from this message as they need. The initial statement is in bold to summarise the main point of the message. If a student does not understand the key words used in this description, they can follow they hyperlinks for more details. A short description of the problem then follows, again with a hyperlink that assumes students understand the basic key concepts. This hyperlink includes syntactically correct code. Finally, if the student is still struggling to grasp what is wrong with their program, they can follow the hyperlink for a more detailed explanation, again including syntactically correct code. It is anticipated that this progressive and interactive mechanism for providing

information will address, at least in part, the problem of students not reading the information currently provided by SNOOPIE.

7.3 Future Developments

7.3.1 Towards Context-Aware Syntactic Support

The conceptual framework presented in this thesis recognises the context in which the teaching process occurs for both program and problem formulation. In the case of program formulation, this is implemented as a database of extensions that take account of the overall style of teaching, i.e. terminology in the module, together with any bespoke teaching tools. This database is easy to amend and so the details of extensions associated with error messages may be updated as the teaching material changes. As such, the support offered for syntactic errors is context sensitive.

Problem formulation support is more sophisticated and takes account of the specific question and the program code at the time of support invocation. Knowledge of the question is encapsulated in a set of priorities that check for program code key program components and interrelationships among those constructs. This support is thus context aware and is clearly a richer level of support than that offered by context sensitive program formulation.

A new, funded project is underway to assess the feasibility of providing context aware support for program formulation. The messages returned by the compiler are limited in content and relate only to the conflict between the program code and the language rules. For a given compiler error there exists a range of possible causes in malformed code. Expert programmers are able to relate the compiler error to corrective action via the program code. The goal of this new project is to encapsulate and subsequently exploit that expert knowledge in the syntactic feedback given to novices.

To date, the software system Qme has been developed which captures the essential detail to reach this goal. Qme encapsulates SNOOPIE within a classroom management system, allowing students to register a problem, be it a problem detected by SNOOPIE, the standard compiler or a free-form question, with a queue. Students are shown a visual representation of

the queue and their position in it. The system stores pairs of syntax error and program code causing the error (or in the case of a free-form question, it stores that question). The focus, as per BlueJ (Kolling et al., 2003), is on only the first syntax error. Additionally, the system also records a short commentary by the laboratory tutor on the corrective action taken, and where this corrective action results in a change in state of the (student) program code, as opposed to dialogue with the student alone, this code is also recorded. This archiving of transactions with the compiler provides a platform upon which to build context aware support, although the challenges posed by this development are significant.

The intended methodology is to mine the data for a given syntax error message to provide the set of all programs that lead to that error message. This set of programs will be categorised into different root causes of the syntax error through automated analysis of the program code. This distillation will be challenging since the code is, by definition, malformed and so the parsing required to identify common code structures and components is non-trivial. A combination of multi-dimensional data visualisation techniques of program code to elicit patterns and expert judgement on the interpretation of those patterns will be used to identify the different root cause categories. These categories will inform the development of enhanced, context aware syntax error support beyond that of the standard compiler. The feedback given to novice programmers will be a mix of the original error message and supplementary information centre arising from the root cause as per the source program code. Note that in the case of problem formulation the context awareness related to the specific question in addition to the program code structure and components. The opportunity to extend the context awareness of the syntactic support to incorporate question specific feedback will be reviewed.

7.3.2 Virtual Lab

Virtual Lab is the conception of the author and the focus of a longer-term research agenda that integrates this programme of research, the work described in the previous section and the activities of a developing network of contacts. The aim of Virtual Lab is to provide an intelligent, supportive learning environment for the novice programmer through a

combination of existing technologies, including the SNOOPIE system described here. Here, each technology is described in turn and their integration outlined.

The first technology is the BlueJ environment, developed by the University of Kent at Canterbury. BlueJ (Kolling, 2003) is currently deployed in over 500 institutes worldwide and, like SNOOPIE, is designed to support the novice programmer. BlueJ provides both visual and textual interfaces to the Java Compiler. The second technology is IVC, developed by Cambridge University. As noted in the literature review, IVC (Moore and Taylor, 2005) is an intelligent tutor system that provides syntax errors with a more informal message and links to tutorial pages. IVC also exploits a generic query engine able to parse text-based questions, isolate keywords and infer meaning. This query engine is linked to a database comprising keyword definitions and these are used to provide a response to the initial question.

In addition to these interface technologies, two applications that profile user interactions exist. GRUMPS (2001), was developed at the University of Glasgow. GRUMPS is a generalised user interaction profiling application able to log all keys pressed, window focus changes, mouse clicks and mouse movements. This logging may be attributed to one or more specified applications, and is archived in terms of a tagged data structure. For each application, the attributes of the logging may be determined, for example no textual logging of the text in a student's email. BlueJ has a new profiling extension (Utting, 2006) able to log all user interactions with the BlueJ environment. In contrast to GRUMPS, the BlueJ profiling extension is able to attribute contextual meaning to logged data in terms of text typed in specific windows, buttons pressed and menu items selected.

These profiling technologies have the capacity to generate enormous amounts of data relating to the activity of user, ranging from time spent in different applications to the evolution of program code. Analysis of the data is challenging and the final technologies to comprise Virtual Lab will aid this analysis. First, a new HIVE facility at the University of Abertay Dundee will provide an environment to visualise these data to aid in the identification of patterns in usage. The HIVE is a Human Immersive Virtual Environment and is part of the

resource underpinning an interdisciplinary research group, called Whitespace, of psychologists, computer artists, complex system software developers and other research disciplines. Complementing and quantifying the patterns emerging from the visual analyses is an expert in spatial statistics from the University of Aberdeen. These specialist resources may combine to provide the foundations for Virtual Lab, and here one such integrative framework of these independent components is outlined to facilitate explanation of the general concept.

In Virtual Lab, the BlueJ IDE could serve as the primary user interface, allowing code entry, compilation and execution. The visual mechanism for code generation and (standard) text entry could operate as in the standard BlueJ environment. Entered code would be passed on to the SNOOPIE system for program and problem formulation support as per the approach in this thesis. BlueJ is shipped with a standardised set of tutorial exercises and these, together with experience of the BlueJ group, may be used to generate the context-sensitive program formulation and context-aware problem formulation. Note that should the research outlined in 7.3.1 provide an effective mechanism for offering context-aware program formulation then this capacity would be integrated. SNOOPIE feedback would be provided through the BlueJ error console.

To provide an additional form of interaction with the development environment, the BlueJ IDE would be extended to provide a free-form window that supports text-based queries relating to both Java syntax and terminology together with the necessary strategies for solving aspects of the question. For example, students may ask for advice on “What” a particular keyword means or the syntax for usage of that keyword, and this relates to program formulation as per SNOOPIE. Additionally, students may query the system in terms of “How” to approach a particular facet of a question such as the role of a looping construct in a specific question, and this relates to problem formulation. To provide an integrated and consistent supporting technology, both SNOOPIE and IVC would access a single resource of data for feedback.

In addition to coupling BlueJ to the assistive technologies of SNOOPIE and IVC, and aligning those technologies with appropriate data, analyses and feedback relating to the specific exercises, the programme of research would seek to exploit the user activity profiling data to enhance the intelligence and adaptive ability of the Virtual Lab.

The BlueJ profiling tool offers fine-grained data on the activity of a given user. Existing logs from student interactions with the BlueJ environment (Jadud, 2006) demonstrate that individual students may reach a point in their program development process where they are unable to proceed in any useful way. Misunderstandings of the root-cause of syntax error messages are shown to lead to inappropriate courses of action in an attempt to correct the error, leading to further and different errors. In some cases, students are shown to be ‘stuck’, evidenced by a pattern of interaction where sustained activity focusing on one small area of the program fails to resolve the error. Experts who browse through these logs were observed to recognise the problem and that the student clearly lacks a key element of knowledge and will not resolve this problem without further support (Jadud, 2006).

The BlueJ profiler allows auditing of interactions with the code, including code added and removed at specific lines and error messages associated with the current state of the code. Combining profiling logs with the visual and statistical data analysis techniques above, it may be possible to identify when a student is ‘stuck’ and for how long they have been stuck. Assuming it is feasible to identify when and for how long a student has been unable to proceed with developing a solution to a problem, the assistive technology may then react in terms of the support offered. In its simplest form, this knowledge of how stuck a student is could act as a volume control on the level of support provided by SNOOPIE/ IVC. A more intelligent, adaptive support could be provided if it is possible to identify why a student is stuck, based on their current area of activity in the code, from the profiling data, combined with knowledge of the question and potential solutions as per the program formulation details encapsulated within SNOOPIE.

The role of the GRUMPS system would be general user interaction auditing. At its minimal level of auditing, the system can log the length of time users are in different applications.

This alone provides a potentially useful view on what students do when they are unable to proceed meaningfully with a problem. At its maximal level of auditing, the system can log the same level of detail as the BlueJ profiling tool; note the context-rich information offered by BlueJ is not possible in GRUMPS. This could provide a complete record of user interaction with the operating system although this raises ethical issues. It is anticipated that GRUMPS could be used to identify the level of interactivity with the system generally, and this may contribute to the determination of how long a student is stuck with a programming problem. For example, for a given student it is possible to distinguish between leaving the machine, undertaking significant activity in another, unrelated application and occasional interactions with the core (BlueJ) application. The last more strongly relates to time where a student is unable to proceed than the first two.

While the above text reflects only one potential combination of technologies, as recognised by the author, the potential for integrating these currently separate systems is significant. It will allow multi-levelled and multi-modal support for individual novice programmers in a framework that can respond dynamically to the system and application level activity, with the potential to recognise when and how ineffectively a student is unable to proceed and to provide an appropriate level of guidance based on that recognition.

Of course, none of this assistive technology is a replacement for the human teacher, who is able to recognise through informed dialogue and other cues the needs of the individual student. The aim of this form of technology is to support that teaching process by providing as supportive and flexible an environment as possible, to offer a platform for teacher-student interaction and, more importantly, promote opportunities for self-study and ultimately deep learning of the fundamental concepts and problem solving strategies associated with programming.

Appendix A Table of Priority Checks

Block 1

	Week 1				Week2						Week 3							Week 4						Week 5		
Priority	1	2	3	4	1	2	3	4	5	6	1	2	3	4	5	6	7	1	2	3	4	5	6	1	2	3
General																										
A method exists	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
A method called main exists	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Use of existing code																										
A specific library is imported																										
A specific number of objects of a specific class exists																										
Object declaration exists	✓	✓	✓	✓	✓	✓		✓			✓	✓	✓	✓			✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Object of a specific class exists	✓	✓	✓	✓	✓	✓		✓			✓	✓	✓	✓			✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
A specific object declaration has a specific argument	✓	✓	✓	✓	✓			✓			✓	✓		✓					✓		✓			✓	✓	
A method call exists	✓	✓	✓	✓	✓		✓		✓		✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
A specific method is called	✓	✓	✓	✓	✓		✓	✓	✓		✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
A specific number of method calls exist		✓	✓	✓	✓		✓	✓	✓				✓		✓	✓	✓					✓				
A specific method is called in a specified order of method calls		✓	✓	✓	✓	✓		✓	✓		✓	✓		✓	✓	✓		✓	✓		✓	✓	✓	✓	✓	
Method calls exist in one of a number of specific possible orders				✓											✓	✓								✓		
Constructs																										
A loop exists						✓	✓	✓	✓	✓	✓		✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
A specific number of loops exist						✓	✓	✓	✓	✓	✓		✓	✓			✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
A specific number of a specific loop type exists						✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓		✓	✓	✓	✓	✓		✓
‘for’ loop should repeat a specified number of times						✓	✓	✓	✓	✓						✓						✓		✓		

	Week 1				Week2						Week 3							Week 4						Week 5		
Priority	1	2	3	4	1	2	3	4	5	6	1	2	3	4	5	6	7	1	2	3	4	5	6	1	2	3
A series of one or more specific method calls are contained inside a specific construct or method						✓	✓	✓		✓	✓		✓	✓	✓	✓		✓	✓	✓	✓	✓	✓	✓		✓
Loop should be nested inside another loop									✓	✓						✓										
A specific operator is contained inside a specific construct						✓	✓	✓		✓																
A specific loop type condition contains a call to a particular method											✓	✓		✓	✓	✓		✓			✓					
A specific loop type condition contains a specific operator or value											✓		✓	✓	✓	✓	✓	✓			✓		✓	✓	✓	✓
A specific method call is contained outside a specific construct													✓													
An operator exists inside a specific construct													✓													
An operator exists outside a specific construct													✓													
Object of class Random exists																	✓									✓
if statement exists																		✓	✓	✓	✓	✓	✓	✓	✓	✓
A specific number of if statements exist																				✓	✓	✓		✓	✓	
A specific number of else if statements exist																				✓	✓	✓	✓	✓	✓	
If statement has an else branch																					✓					
Operator exists outside a specific construct																							✓			
Loop exists inside a specific construct																								✓	✓	
Program must not contain any loops of a specific type																										
Condition of a specific loop in a series of loops meets specific criteria																										

	Week 1				Week2						Week 3							Week 4						Week 5		
Priority	1	2	3	4	1	2	3	4	5	6	1	2	3	4	5	6	7	1	2	3	4	5	6	1	2	3
If statement is nested inside the else branch of another if statement																										
A switch statement exists																										
A switch statement exists inside a specific construct																										
A specific number of switch cases exist																										
A specific number of switches exist																										
A specific number of switch breaks exist																										
Each switch case contains a method call in a specific order																										
Switch condition meets specific criteria																										
An unnested loop of a specific type exists																										
A specific method call contains a specific number																										
Either an if statement or a switch is used																										
A specific number of conditional branches exist																										
Data																										
A variable declaration exists													✓						✓	✓		✓	✓			✓
A specified number of variables exist													✓						✓	✓		✓	✓			✓
Variables of a certain type exist													✓						✓	✓		✓	✓			✓

	Week 1				Week2						Week 3							Week 4						Week 5		
Priority	1	2	3	4	1	2	3	4	5	6	1	2	3	4	5	6	7	1	2	3	4	5	6	1	2	3
Student written methods																										
A specific number of methods exist																										
A method exists with a specific name																										
A method has a specific return type																										
A method has a specific number of parameters																										
A method has specific parameter types																										
A specific method is called from main																										
A for loop exists inside a specific method																										
An if statement exists inside a specific method																										

Block 2

	Week 6						Week 7					Week 8					Week 9					Week 10			
Priority	1	2	3	4	5	6	1	2	3	4	5	1	2	3	4	5	1	2	3	4	5	1	2	3	4
General																									
A method exists																									
A method called main exists	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
A specific library is imported	✓	✓	✓	✓	✓	✓	✓	✓	✓																
A specific number of objects of a specific class exists																									
Use of pre-written code																									
Object declaration exists		✓																							
Object of a specific class exists	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓									
A specific object declaration has a specific argument																									
A method call exists																									
A specific method is called	✓	✓	✓	✓	✓	✓	✓	✓		✓	✓	✓	✓	✓											
A specific number of a specific method call exists	✓	✓	✓	✓	✓	✓	✓	✓			✓	✓	✓	✓	✓	✓									
A specific method is called in a specified order of method calls	✓	✓	✓	✓	✓						✓														
Method calls exist in one of a number of specific possible orders						✓																			
Constructs																									
A loop exists	✓	✓	✓	✓	✓	✓																			
A specific number of loops exist	✓										✓														
A specific number of a specific loop type exists	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓					✓									
'for' loop should repeat a specified number of times	✓						✓		✓	✓	✓					✓				✓					

	Week 6						Week 7					Week 8					Week 9					Week 10			
Priority	1	2	3	4	5	6	1	2	3	4	5	1	2	3	4	5	1	2	3	4	5	1	2	3	4
A series of one or more specific method calls are contained inside a specific construct or method	✓	✓	✓	✓	✓	✓		✓		✓	✓	✓		✓	✓	✓	✓	✓	✓	✓	✓				
Loop should be nested inside another loop											✓														
A specific operator is contained inside a specific construct										✓															
A specific loop type condition contains a call to a particular method		✓	✓	✓	✓			✓																	
A specific loop type condition contains a specific operator or value		✓	✓	✓	✓	✓		✓																	
A specific method call is contained outside a specific construct																									
An operator exists inside a specific construct																									
An operator exists outside a specific construct																									
if statement exists				✓	✓	✓				✓		✓		✓	✓										
A specific number of if statements exist										✓		✓		✓	✓						✓				
A specific number of else if statements exist				✓	✓	✓				✓		✓		✓	✓						✓				
If statement has an else branch				✓	✓	✓						✓		✓	✓										
If statement is contained inside a specific construct				✓	✓	✓				✓															
If condition contains a specific method call				✓	✓	✓						✓		✓	✓						✓				
If condition contains a specific operator																									
Operator exists inside a specific construct										✓															
loop condition meets specific criteria depending on whether student has used for or while																									
If condition is contained outside a specific construct																									
Operator exists outside a specific construct																									

	Week 6						Week 7					Week 8					Week 9					Week 10			
Priority	1	2	3	4	5	6	1	2	3	4	5	1	2	3	4	5	1	2	3	4	5	1	2	3	4
Loop exists inside a specific construct																									
Program must not contain any loops of a specific type	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓					✓									
Condition of a specific loop in a series of loops meets specific criteria		✓																							
If statement is nested inside the else branch of another if statement				✓																					
A switch statement exists								✓	✓																
A switch statement exists inside a specific construct								✓	✓																
A specific number of switch cases exist								✓	✓																
A specific number of switches exist								✓	✓																
A specific number of switch breaks exist								✓																	
Each switch case contains a method call in a specific order								✓	✓																
An array with a specific number of dimensions exists									✓	✓	✓														
A specific number of arrays exist									✓	✓	✓														
Switch condition meets specific criteria									✓																
An unnested for loop exists											✓														
Object of class String exists												✓	✓	✓	✓	✓									
A specific number of Strings exist												✓	✓	✓	✓	✓									
A specific method call contains a specific number													✓								✓				
Either an if statement or a switch is used																✓									
A specific number of conditional branches exists																✓									

	Week 6						Week 7					Week 8					Week 9					Week 10			
Priority	1	2	3	4	5	6	1	2	3	4	5	1	2	3	4	5	1	2	3	4	5	1	2	3	4
Data																									
A variable declaration exists				✓	✓	✓		✓			✓														
A specified number of variables exist				✓	✓	✓		✓		✓	✓					✓									
Variables of a certain type exist				✓	✓	✓		✓		✓	✓					✓									
Student written methods																									
A specific number of methods exist																	✓	✓	✓	✓	✓				
A method exists with a specific name																	✓	✓	✓	✓					
A method has a specific return type																	✓	✓	✓	✓	✓				
A method has a specific number of parameters																	✓	✓	✓	✓					
A method has specific parameter types																	✓	✓	✓	✓	✓				
A specific method is called from main																	✓	✓	✓	✓	✓				
A for loop exists inside a specific method																				✓					
An if statement exists inside a specific method																					✓				

Block 3

	Week 11				Week 12				Week 13			
Priority	1	2	3	4	1	2	3	4	1	2	3	4
General												
A method exists												
A method called main exists	✓	✓	✓	✓	✓	✓	✓	✓	✓			
A specific library is imported												
A specific number of objects of a specific class exists												
Use of pre-written code												
Object declaration exists												
Object of a specific class exists												
A specific object declaration has a specific argument												
A method call exists												
A specific method is called											✓	✓
A specific number of a specific method call exists											✓	✓
A specific method is called in a specified order of method calls												
Method calls exist in one of a number of specific possible orders												
Constructs												
A loop exists												
A specific number of loops exist												
A specific number of a specific loop type exists				✓					✓		✓	✓
‘for’ loop should repeat a specified number of times									✓			

	Week 11				Week 12				Week 13			
Priority	1	2	3	4	1	2	3	4	1	2	3	4
A series of one or more specific method calls are contained inside a specific construct or method	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓		
Loop should be nested inside another loop												
A specific operator is contained inside a specific construct												
A specific loop type condition contains a call to a particular method												
A specific loop type condition contains a specific operator or value												
A specific method call is contained outside a specific construct												
An operator exists inside a specific construct												
An operator exists outside a specific construct												
if statement exists												
A specific number of if statements exist												✓
A specific number of else if statements exist										✓		
If statement has an else branch												
If statement is contained inside a specific construct									✓			
If condition contains a specific method call												
If condition contains a specific operator												
Operator exists inside a specific construct												
loop condition meets specific criteria depending on whether student has used for or while												
If condition is contained outside a specific construct												
Operator exists outside a specific construct												

	Week 11				Week 12				Week 13			
Priority	1	2	3	4	1	2	3	4	1	2	3	4
Loop exists inside a specific construct												
Program must not contain any loops of a specific type				✓								
Condition of a specific loop in a series of loops meets specific criteria												
If statement is nested inside the else branch of another if statement												
A switch statement exists												
A switch statement exists inside a specific construct												
A specific number of switch cases exist												
A specific number of switches exist												
A specific number of switch breaks exist												
Each switch case contains a method call in a specific order												
Switch condition meets specific criteria												
An unnested loop of a specific type exists												
A specific method call contains a specific number												
Either an if statement or a switch is used												
A specific number of conditional branches exist												
Data												
A variable declaration exists												
A specified number of variables exist												
Variables of a certain type exist												
An array with a specific number of dimensions exists											✓	✓

	Week 11				Week 12				Week 13			
Priority	1	2	3	4	1	2	3	4	1	2	3	4
Student written methods												
A specific number of arrays exist												
A specific number of methods exist	✓	✓	✓	✓	✓	✓			✓	✓		
A method exists with a specific name	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓		✓
A method has a specific return type	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓		
A method has a specific number of parameters	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓		
A method has specific parameter types	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓		
A specific method is called from main	✓	✓	✓	✓								
A specific construct exists inside a specific method				✓	✓	✓	✓	✓	✓	✓		

Appendix B University of St Andrews experiment questions

Java Programming Experiment

The first 30 minutes of the experiment will be spent guiding you through creating solutions to the exercises described in part one. This will enable you to become familiar with the new environment and the robot and cow objects. The additional handout includes information on setting up the JCreator environment and describes the robot and cow methods that you can use.

Part 1 – Orientation.

robby the Robot exercises:

1. In a new program, StA1_1.java make robby traverse the empty room perimeter anti-clockwise. Make sure robby finishes the journey facing upwards (as robby began). Use a 'while loop' for each wall.

2. Write a program, StA1_2.java, to make robby follow the trail of yellow tiles present in room 4. This trail is a clockwise spiral and so only has right turns. robby should stop moving on the green tile at the end of the trail.

daisy the Cow exercises:

3. Write a program, StA1_3.java, to manage daisy's feeds through a series of conditional loops in the following sequence:

- feed daisy until she is content
- flush daisy until she is hungry
- fill her up to bloated
- empty daisy out providing her with suitable thoughts until she is empty (but not dead)

DO NOT TURN OVER THE PAGE UNTIL INSTRUCTED BY THE TUTOR

For this part of the experiment, you are required to try and produce solutions to the following practical exercises. If you are stuck you can ask for help from the tutor. When you are finished completing the exercises or run out of time, please complete the questionnaire handed out at the beginning of the experiment.

Part 2 – Assessment exercises

1. Write a program, StA2_1.java, to make robby follow the trail of yellow tiles present in room 5. This trail weaves from side to side, and so has both left and right turns. robby should stop moving on the green tile at the end of the trail.

2. Write a program, StA2_2.java, to make robby count all the white, blue, yellow and green squares in a column in a room. The column to be counted should be chosen by input of a number from the keyboard (using the getInt method of the GUI). robby is allowed to walk through blue squares (normally robby should walk round them) for this counting exercise. Display the number of white, blue, yellow and green squares found at the end using the putText method of the GUI.

3. Write a program, StA2_3.java, to makes daisy behave randomly: ,
   - if she is hungry eat;
   - else if she is bloated flush;
   - else randomly choose eat or flush with daisy being twice as likely to eat than flush.

   The following code will help you generate a random number:

```
import java.util.*;
Random rndm = new Random( );          // Random number generator
int number = rndm.nextInt(6)+1;       //Generate a random number between 1 and 6
```

4. Write a program, StA2_4.java, to randomise daisy's thoughts and behaviour as follows:
   daisy should randomly chose between eating and flushing, weighted by her mood (so up to three weightings in all - see example). The weights should be entered in via the GUI at the beginning of the program as 3 separate integers - this requires some consideration!

   Sample weightings per mood might include:

- if daisy is hungry there is an 80% chance that she will eat and a 20% chance that she will flush, i.e. int hungryEatChance = 80.
- if daisy is content there is a 60% chance that she will eat and a 40% chance that she will flush.
- if daisy is bloated there is a 20% chance that she will eat and an 80% chance that she will flush.

Additionally, no eating or flushing should kill daisy.

Appendix C University of Abertay Dundee experiment questions

Some questions have been adapted from "How to think like a computer scientist, Java version" (Downey, 2002).

1. Create a new program called Calculate.java. Write a method called 'calculate' that takes 3 doubles (x,y and z) as parameters and prints the result of x*y+z. Write a main method to test 'calculate' by calling it with some simple parameters, for example calling 'calculate' with the numbers (2,3.5 and 4) would results in the number 11.0 being displayed.

2. The program 'blink.java' contains some simple errors. Debug the errors in this program so that it compiles successfully.

3. The program 'buzz.java' contains some simple errors. Debug the errors in this program so that it compiles successfully.

4. Pierre de Fermat is perhaps the most famous number theorist who ever lived. His 'Last Theorem' states that there are no numbers for x,y and z so that $x^n + y^n = z^n$ (except when $n <= 2$)

Create a new program called 'Fermat.java'.Write a method called 'check' that takes four integers as parameters x, y, z and n and that checks to see if Fermat's theorem is correct. If n is greater than 2 and it turns out to be true that $x^n + y^n = z^n$, the method should return "The theory is wrong!" Otherwise the method should return "The theory is right!" You should write a method named 'powerOf' that has two integer parameters and calculates the 1st integer to the power of the second. For example: int x = powerOf (2, 3); would assign the value 8 to x, because $2^3 = 8$.

The following code fragment is a partial implementation of powerOf(a, n)

```
int answer=a;
for (int i =1; i<n; i++)
{
answer=answer*a;
}
return answer;
```

5. The program 'narf.java' contains some errors. Debug the errors in this program so that it compiles and runs successfully.

6. The program 'blimp.java' contains some errors. Debug the errors in this program so that it compiles and runs successfully. The program should display the number 30.

7. The program 'ping.java' contains some errors. Debug the errors in this program so that it compiles and runs successfully. The program should display 'false, true, ping'.

8. A matrix is a set of numbers displayed in rectangular form. To add two matrices together, the number of rows and columns in each matrix have to be the same. The answer is derived from adding an element from matrix A to its matching element in Matrix B. For example, in MatrixA+B, the number in element(1,3) in

row 1, column 3 is derived from adding the number in MatrixA in row 1 column 3 to the number in MatrixB in row 1 column 3. In this case, 2+7 = 9. For example:

Matrix A			+	Matrix B			=	MatrixA+B		
3	4	2		1	9	7		4	13	9
9	10	5		5	2	10		14	12	15

Matrices can be represented in java using 2-dimensional arrays. Create a new program called ‘Matrix.java’. Copy the code below into your new program. Write a method called enterMatrix that asks the user to specify how many rows and columns they want and asks the user to enter numerical values for each element of the matrix. Write a method called printMatrix that displays each element of the specified matrix in an appropriate form. Write a method called AplusB that returns a new matrix containing the sum of matrix A and B. This method must first check that the matrix dimensions are valid for addition.

```
public static void main(String[] args) {
//int[][] a = enterMatrix("Matrix A");
//int[][] b = enterMatrix("Matrix B");
//printMatrix(a, "Matrix A");
//printMatrix(b, "Matrix B");

//int[][] a_plus_b = AplusB (a, b);

//printMatrix(a_plus_b, "Matrix A+B");

}
```

N.B. This code is commented out so that you can build the program in stages. Uncomment the code when you have written the appropriate methods to be used.

Blink.java

```
1  public class blink{ Error 1: wrong case used in class name
2
3  public static void zoop (){
4  baffle ();
5  System.out.print ("You wugga ");
6  baffle ();
7  }
8  public static void main (String[] args) {
9  System.out.print ("No, I ");
10 zoop ();
11 System.out.print ("I ");
12 baffle ();
13 }
14 public static void baffle) { Error 2: missing ‘(‘
15 System.out.print (".); Error 3: unclosed string literal
16 ping ();
17 }
18 public static void ping () {
19 System.out.println ("wug");
20 }
21 }
22 } Error 4: extra closing bracket
```

Buzz.java

```
1  public class Buzz {
2  public static void baffle (String blimp) {
3  system.out.println (blimp); Error 1: lower case ‘s’ used in word System
4  zippo ("ping", 0);
5  }
6  public static void zippo (String qunice, int flag) { Error 2: variable
   misspelled
7  if (flag = 0) { Error 3: single equals in ‘if statement’
8  System.out.println (quince + " zoop");
9  } else {
10 System.out.println ("ik");
11 baffle (quince);
12 System.out.println ("boo-wa-ha-ha");
13 }
14 }
15 public static void main (String[] args) {
16 zippo ("rattle", 13);
17 }
18 }
```

Narf.java

```
1  public class Narf {
2  public static void zoop (String fred, int bob) {
3  System.out.println (fred);
4  if (bob == 5) {
5  ping ("not ");
6  } else {
7  System.out.println ("!");
8  }
9  }
10 public static void main (String[] args) {
11 int bizz = 5;
12 int buzz = 2;
13 zoop ("just for", bizz) Error 1 missing semi-colon
14 clink (2*buzz);
15 }
16 public static void clink (int fork); { Error 2 superfluous semi-colon
17 System.out.print ("It's ");
18 Zoop ("breakfast ", fork) ;
19 }
20 public static void ping (String strangStrung) {
21 System.out.println ("any" +strangStrung+ "more ");
22 }
23 }
```

Blimp.java

```
1  public class Blimp{
2  public static int[] make (int n) {
3  int[] a = new int[n];
4  for (int i=0; i<n; i++) {
5  a[i] = i+1;
6  }
7  return a;
8  }
9  public static void dub (int[] jub) {
10 for (int i=0; i<jub.length; i++) {
11 jub[i] =jub[i]*2;
12 }
13 }
14 public static int mus (int[] zoo) {
15 int fus = 0;
16 int j=0;
17 for (int i=0; i<zoo.length; i++); { Error 1 superfluous semi-colon
18 fus = fus + zoo[j];
19 j++;
20 }
21 return fus;
22 }
23 public static void main (String[] args) {
24 int[] bob = make (5);
25 dub (bob);
26 System.out.println (mus (bob));
27 }
28 }
```

Ping.java

```
1  public class Ping{
2  public static void main (String[] args) {
3  boolean flag1 = isHoopy (202);
4  boolean flag2 = isFrabjuous (202);
5  System.out.println (flag1);
6  System.out.println (flag2);
7  if (flag1=true){ Error 1 single equals in ‘if statement’
8  System.out.println ("ping!");
9  }
10 if (flag1 && flag2); { Error 2 superfluous semi-colon
11 System.out.println ("pong!");
12 }
13 }
14 public static boolean isHoopy (int x) {
15 boolean hoopyFlag;
16 if (x%2 == 0) {
17 hoopyFlag = false;
18 } else {
19 hoopyFlag = true;
20 }
21 return hoopyFlag;
22 }
23 public static boolean isFrabjuous (int x) {
24 boolean frabjuousFlag=true;
25 if (x > 0) {
         a. frabjuousFlag=true;
26 } else {
         a. frabjuousFlag=false;
27 }
28 return frabjuousFlag;
29 }}
```

Appendix D Questionnaire

Please fill in the circle for the statement to describe your use of the support tool during term 1

1	I used it frequently for the duration of the term	27%
2	I used it frequently at the start of term and less towards the end	7%
3	Sometimes I used it when the standard Compiler didn't help me and I was stuck	30%
4	Rarely at the start of term but more frequently towards the end	10%
5	Never, or almost never.	27%

SECTION A Type 1 Results (i.e. those who used the support tool frequently for the duration of the term – option 1 above, and other types are referred to similarly).

Please indicate strength of agreement with following statements (Strongly agree, agree, neutral, disagree, strongly disagree)

1.	The support tool helped me to produce a working solution	13%	50%	25%	13%	0%
2.	The support tool helped me solve mistakes that I had made with my code	25%	75%	0%	0%	0%
3.	The support tool helped me learn to program	0%	38%	38%	25%	0%
4.	Sometimes the support tool gave me too much help	0%	13%	38%	25%	25%
5.	The support tool was a useful addition to the messages provided by the standard Compiler.	0%	63%	38%	0%	0%
6.	The text provided was difficult to read in the size of the window	0%	0%	38%	63%	0%
7.	I used the support tool to find out what programming constructs (e.g for loops and 'if statement' s) I needed to write my programs	0%	38%	38%	25%	0%
8.	I used the support tool to make sure that my programs were correct	63%	25%	13%	0%	0%
9.	Sometimes the support tool did not give enough help	0%	75%	25%	0%	0%
10.	I would have liked to have been able to control how much help it gave me	13%	25%	50%	13%	0%
11.	There was too much text provided by the support tool	0%	13%	38%	50%	0%
12.	I didn't understand the text provided by the support tool	0%	38%	38%	25%	0%
13.	I found it easier to understand the text in the support tool when it was explained to me verbally.	0%	50%	38%	13%	0%
14.	Although I wanted the support tool to give me the answer, I didn't want to have to read all the text to figure it out for myself	0%	75%	13%	13%	0%
15.	When I used the support tool I wanted it to give me the exact code that I needed to use	25%	25%	0%	50%	0%
16.	The support tool helped me to produce a working solution	0%	75%	13%	13%	0%
17.	In my opinion, other students liked the support tool	13%	13%	75%	0	0

SECTION B Type 2 Results

Please indicate strength of agreement with following statements:

Most of the time through the term, at the Beginning of the term, at the End of the term or Not at all. Choose M for Most of the time Choose B for the beginning of the term Choose E for the end of the term Choose N for not at all		m	b	e	n
1.	The support tool helped me to produce a working solution	100%	0%	0%	0%
2.	Sometimes the support tool gave me too much help	0%	0%	0%	100 %
3.	The support tool helped me learn to program	0%	100%	0%	0%
4.	The support tool helped me solve mistakes that I had made with my code	50%	50%	0%	0%
5.	I used the support tool to find out what programming constructs (like for loops and 'if statement' s) I needed to write my programs	50%	50%	0%	0%
6.	Sometimes the support tool did not give enough help	0%	0%	50 %	50%

Please indicate strength of agreement with following statements (Strongly agree, agree, neutral, disagree, strongly disagree)						
7.	Sometimes I needed assistance to produce a working solution but preferred to get that help from a tutor or friend	0%	0%	100 %	0%	0%
8.	The support tool was a useful addition to the messages provided by the standard Compiler	0%	100 %	0%	0%	0%
9.	Sometimes when I was stuck the tool helped me solve a problem when the tutor was busy with another student	0%	100 %	0%	0%	0%
10.	I would have liked to have been able to control how much help it gave me	50%	0%	50 %	0%	0%
11.	There was too much text provided by the support tool	0%	0%	100 %	0%	0%
12.	When I used the support tool, I found it difficult to read all the text and take out the information that I needed	0%	0%	50 %	50 %	0%
13.	I found it easier to understand the text in the support tool when it was explained to me verbally.	0%	50%	50 %	0%	0%
14.	When I used the support tool I want it to give me the exact code that I need to used	0%	50%	0 %	50 %	0%
15.	I would have used the tool more often if it had less bugs	0%	50%	50 %	0%	0%
16.	I like the idea of the support tool	0%	100%	0 %	0%	0%

SECTION B Type 3 Results

Please indicate strength of agreement with following statements:

Most of the time through the term, at the Beginning of the term, at the End of the term or Not at all. Choose M for Most of the time Choose B for the beginning of the term Choose E for the end of the term Choose N for not at all		*m*	*b*	*e*	*n*
1.	The support tool helped me to produce a working solution	63%	38%	0%%	0%%
2.	Sometimes the support tool gave me too much help	0%%	13%	0%%	88%
3.	The support tool helped me learn to program	13%	25%	25%	38%
4.	The support tool helped me solve mistakes that I had made with my code	75%	25%	0%%	0%
5.	I used the support tool to find out what programming constructs (like for loops and 'if statement' s) I needed to write my programs	38%	25%	13%	25%
6.	Sometimes the support tool did not give enough help	25%	0%	50%	25%

Please indicate strength of agreement with following statements (Strongly agree, agree, neutral, disagree, strongly disagree)						
7.	Sometimes I needed assistance to produce a working solution but preferred to get that help from a tutor or friend	25%	25%	38%	13%	0%
8.	The support tool was a useful addition to the messages provided by the standard Compiler	25%	50%	25%	0%	0%
9.	Sometimes when I was stuck the tool helped me solve a problem when the tutor was busy with another student	25%	38%	38%	0%	0%
10.	I would have liked to have been able to control how much help it gave me	0%	38%	50%	0%	13%
11.	There was too much text provided by the support tool	0%	0%	50%	50%	0%
12.	When I used the support tool, I found it difficult to read all the text and take out the information that I needed	0%	13%	13%	63%	13%
13.	I found it easier to understand the text in the support tool when it was explained to me verbally.	13%	13%	50%	13%	13%
14.	When I used the support tool I want it to give me the exact code that I need to used	0%	25%	13%	38%	25%
15.	I would have used the tool more often if it had less bugs	0%	0%	75%	25%	0%
16.	I like the idea of the support tool	25%	25%	38%	13%	0%

SECTION B Type 4 Results

Please indicate strength of agreement with following statements:

Most of the time through the term, at the Beginning of the term, at the End of the term or Not at all. Choose M for Most of the time Choose B for the beginning of the term Choose E for the end of the term Choose N for not at all		*m*	*b*	*e*	*n*
17.	The support tool helped me to produce a working solution	0%%	33%	33%	33%
18.	Sometimes the support tool gave me too much help	0%%	0%%	0%%	10%0%%
19.	The support tool helped me learn to program	0%%	0%%	0%%	10%0%%
20.	The support tool helped me solve mistakes that I had made with my code	0%%	33%	0%%	67%
21.	I used the support tool to find out what programming constructs (like for loops and 'if statement' s) I needed to write my programs	0%%	33%	0%%	67%
22.	Sometimes the support tool did not give enough help	33%	0%%	33%	33%

Please indicate strength of agreement with following statements (Strongly agree, agree, neutral, disagree, strongly disagree)						
23.	Sometimes I needed assistance to produce a working solution but preferred to get that help from a tutor or friend	67%	33%	0%%	0%%	0%%
24.	The support tool was a useful addition to the messages provided by the standard Compiler	0%	0%%	67%	33%	0%%
25.	Sometimes when I was stuck the tool helped me solve a problem when the tutor was busy with another student	0%	33%	33%	33%	0%%
26.	I would have liked to have been able to control how much help it gave me	0%	33%	67%	0%%	0%%
27.	There was too much text provided by the support tool	0%	0%%	33%	67%	0%%
28.	When I used the support tool, I found it difficult to read all the text and take out the information that I needed	0%	0%%	67%	33%	0%%
29.	I found it easier to understand the text in the support tool when it was explained to me verbally.	33%	33%	0%%	33%	0%%
30.	When I used the support tool I want it to give me the exact code that I need to used	33%	0%%	0%%	33%	33%
31.	I would have used the tool more often if it had less bugs	0%	67%	33%	0%%	0%%
32.	I like the idea of the support tool	0%	100%	0%%	0%%	0%%

SECTION C Type 5 Results

Please indicate strength of agreement with following statements (Strongly agree, agree, neutral, disagree, strongly disagree)

1.	I was able to produce a working program with almost no assistance so didn't need to use the support tool	29%	14%	43%	0%%	14%
2.	Sometimes I needed assistance to produce a working solution but preferred to get that help from a tutor or friend	43%	43%	14%	0%%	0%%
3.	The support tool gave me too much help	14%	0%%	71%	14%	0%%
4.	I would have used the tool more often if it had less bugs	14%	14%	43%	14%	14%
5.	I would have used the tool more often if it just helped with my syntax errors	0%%	29%	43%	14%	14%
6.	I like the idea of the support tool but it is not how I wanted to learn	0%%	57%	29%	14%	0%%
7.	In my opinion, other students liked the support tool	14%	29%	57%	0%%	0%%

SECTION D

Please fill in the circle for the statement that best describes your programming abilities

	1	2	3	4	5
I have found the programming module easy	0%%	0%%	13%	0%%	14%
I like programming and although some programs are hard I enjoy working out a solution.	50%%	50%%	88%	67%	57%
I find programming hard and don't really understand it	38%	50%%	0%%	0%%	29%
I hate programming and I never want to do it again	13%	0%%	0%%	33%	0%%

Please fill in the circle for the statement that best describes your module expectations

I am aiming to get a grade A in this module	13%	0%%	25%	67%	43%
If I do a lot of work I think I will get a good pass in this module	50%%	50%%	75%	0%%	29%
If I get a lot of help I think I will get a fair pass in this module	25%	0%%	0%%	0%%	14%
I'm worried that I might fail this module	13%	50%%	0%%	33%	14%

Male	63%	50%%	75%	10%0%%	86%
Female	38%	50%%	25%	0%%	14%

DipHE Computing & Information Technology	25%	0%%	13%	0%%	43%
BA Information Systems	25%	0%%	0%%	0%%	0%%
BSc Computing	38%	50%%	25%	67%	43%
BSc Web Design & Development	13%	50%%	63%	33%	14%

If you took part in the experiment please indicate which group you were in

Group 1	13%	0%%	13%	0%%	14%
Group 2	25%	50%%	13%	0%%	14%
Group 3	50%%	0%%	13%	33%	29%

Appendix E Interview with Student A

Interview recorded on 18th July 2006 at 3pm

1. N: [discussion of orientation to interview, semester 1 support]
2. N: do you like programming?
3. A: I like programming yeah um I've done is since 3rd year [school] and I enjoyed it in 3rd year being a freak child and all you know?
4. N: [laughs]
5. A: liking it in 3rd yeah straight through till 6th year [school]. I didn't do it in 6th year for my project I did web design for my project in 6th year.
6. N: so you've done standard grade…
7. A: yeah. I did um Pascal for a bit and I did Visual Basic
8. N: right ok
9. A: but oh I've forgotten everything [laughs]
10. N: [laughs]
11. A: but programming is easy it's just logic
12. N: so you enjoy it
13. A: I enjoy programming
14. N: So what is it about programming that you like?
15. A: um… I don't know it's just when you finish it when you see something that the computer's actually doing for you it's like oh look wow [laughs] look what I've done [laughs] like a sense of achievement yeah cause like with computers you can click and that's it but with that it's like writing it out and making the computer do things that it can't do normally like…what is it we did again… like the connect 4 and making your own game and stuff like wow I never thought I'd be able to do something like that before.
16. N: ok, can I ask then how much would you say you've been using the tool do you reckon in semester 1
17. A: semester 1 to begin with I didn't use it at all I mostly got help [from the tutor]but gradually as it started to get harder and harder and the lecturers were busier and busier you know with other people I started using one and it was good you know cause it didn't have all the cryptic messages and stuff it had like it pointed it out and because it was based on the robby and stuff where as their tool wasn't it was just based on general programming it was straight to the point like robby can't do this because you've not got this or you've not got the next thing.
18. N: ok so you would say… do you remember… did you fill in the questionnaire?
19. A: yeah, yeah
20. N: one of the categories in the questionnaire one of them was that you would've been an infrequent user at the start of term but towards the end of term you used it more would you agree that you fall into that category?
21. A: I think I said that [in the questionnaire] but um I didn't use it all the time as I got more and more confident I did it myself
22. N: ok so what in your opinion is the best thing about the tool?
23. A: like I said how it's straight to the point unlike their own [the standard Java Compiler messages] it tells you what line it is what's wrong with it why robby can't do it things like that
24. N: so that fact that it actually mentioned robby?
25. A: it pinpoints… it pinpoints
26. N: yeah
27. A: so it's not like variable string da da da and it's got all this jargon it's not got all the jargon like to begin with
28. N: uh huh ok so was there anything else that you liked about it that you thought was…

29. A: em…[pause]… the interface as well like how it popped up and like you know how…I thought that was quite cool [laughs] you know how it's [the standard Java Compiler] got it at the bottom that's like superficial [laughs] but yeah the thing at the bottom yeah
30. N: I interviewed somebody last week and they said exactly the same, that they didn't like it coming up in the standard output window. I don't know if you noticed that I changed it about half way through
31. A: uh huh
32. N: so that instead of it popping up in the window
33. A: yeah
34. N: like you're talking about
35. A: uh huh
36. N: I actually made it come up in just the normal
37. A: box
38. N: and the student last week said I had to go back to the normal window cause he preferred it and I changed it cause I thought
39. A: no I prefer the box [referring to the pop-up window]
40. N: I know that I was wrong to change it because I didn't even speak to any students to say, I was just watching over their shoulders and I saw students minimising the window and then having to flick between the two because you can't have the window open and edit your code and I just thought to myself they don't like that and I went ahead an changed it so it's really interesting that you say that
41. A:I definitely preferred the pop-up maybe I you position it so it's out of the way you know it would probably work better like see if it didn't pop up in the middle but default to the side because I don't know why I just preferred it like that
42. N: so you think it was nice popping up with formatted text rather than just in the standard window
43. A: uh huh, yeah
44. N: em… do you think the support tool made the programming labs better?
45. A: it made them much better I think cause like I said…em…the lecturers would be helping other people and there was only like 2 lecturers or 3 at most at one time so you know they were busy at the end of the room and you'd be like 'ch ch ch' for half an hour…waiting but with that you could click on it and it would tell you and you could work through it and you could get it and then if someone else beside you was stuck on it like say I sat beside my friends Fred and Barney and they weren't…they weren't as prolific at programming [laughs] as me… so I would… we would look at it together and then we would do it together we'd go over it and I'd show them or maybe they'd get it
46. N: that's quite interesting because I was never…obviously if I was helping another student at the time I never actually witnessed that that was going on that a group of students would actually use it together
47. A: yeah I would look at it and then if Fred was stuck I'd show it to him I'd help her of if Barney was stuck I'd help him things like that
48. N: that's good
49. A: we'd help each other
50. N: is there anything about programming that you found hard
51. A: em… the overloading… I never got that
52. N: oh right… ok
53. A: it's not that I don't get it… I got it but… pfff…[laughs]… and remember the matrix program that we had to do for the test… when I did the second one I didn't like that
54. N: yes that was quite hard
55. A: yeah I'm not a very good maths person because like I said I'm not very good with maths I was always an English person so the maths kind of got a bit…eurgh…but em…methods were easy it took us a while to grasp the making your own program and linking it externally like linking to it…like that…but I'm ok with that now…em things like that…the basic syntax I could get
56. N: do you think that the tool helped you with the aspects that you found hard?
57. A: em…I wouldn't say that overloading because there wasn't any support for that cause it was later on but em it would be nice to get some support for that cause it is a hard concept cause one method can have, you can have 3 methods that are the same but for different variables
58. N: so you think it would be good then if the tool has support extended for material in semester 2?

59. A: yeah if it kind of extended into semester two but not hold your hand as much
60. N: yep I didn't want it to be like that cause some students used it all the time
61. A: see I didn't do that I used…I did both Compilers I would use the main one and your one just to see what they said to compare them but I used your own mostly sometimes
62. N: right, until semester 2
63. A: no I still used your one in semester 2 cause it still came up with nice messages although they weren't so related
64. N: so what do you think was the worst thing about the tool?
65. A: em…it's hard…because I generally got on with it…em…it wasn't as cryptic as the Compiler…but it could be cryptic at times but not overly so…I would manage to get through it eventually
66. N: so you found some of the messages
67. A: some of the messages would still be cryptic
68. N: they were not informative enough?
69. A: they could've been like…um…dumbed down [laughs]
70. N: is there anything else that you disliked about the tool?
71. A: no
72. N: did you find it quite buggy?
73. A: em.. I didn't pick up on it. Some of the messages…it would display three messages and I didn't know which one was which if you know what I mean like it would display a couple of things for one line… it was to much
74. N: ok did you ever find it frustrating to use?
75. A: only when I was totally stuck and there was no lecturer about and the tool still wasn't giving me the right thing
76. N: did you ever use the hyperlinks in the messages?
77. A: I didn't even know that there was hyperlinks in there
78. N: [explanation of hyperlinks, being in earlier version of tool]
79. A: no I never saw them, maybe I just missed those messages that's a really good idea…I would use that
80. N: so you used the standard Compiler normally and my software if you were stuck
81. A: yeah when the lecturer was busy
82. N: can you explain to me what you think the tool did? If you had to explain it to a new first year students how would you describe it to them?
83. A: the support tool takes your program, finds the errors and the helps you…gives you better messages than the standard Compiler…helps you with the actual program instead of just the general syntax and stuff it helps you with the actual robby not…not with programming in general it's actually specific for robby for the module
84. N: that's fine. Did it help you learn to program…did it teach you anything?
85. A: it taught me how to error trap easier, it showed me… it taught me how to look through the program and find stuff that you wouldn't normally have found with the Compiler, but see if there were semicolons missing that were hard to pinpoint, I would manage to pinpoint them easier because I'd been using the support tool and that'll be like look at this error and it would teach you…it would stay in my brain that that was what I had to look for.
86. N: so the extended messages helped you become more aware of what the standard messages actually meant
87. A: yeah things like names and spelling things wrong
88. N: did it help you understand the module content at all?
89. A: it helped me understand because at the beginning I'd never done programming this extensively before. I'd done Pascal and Visual Basic like I'd said but em I hadn't done something this full on like actual programming…like the chess…not the chess the connect four…the AI but it would teach you how to get it to work
90. N: do you think we should use it again next year?
91. A: I think it would help them probably but like I said they probably won't use it as much to begin with but if you push it say you know we've built this for you

92. N: yeah we didn't really sell it to you last year
93. A: yeah cause it wasn't working on all the machines on the first day and of course if you're like this is a support tool try and use this with the Compiler tell them it's there for you and it's been built for the specific module and it's much more helpful cause it is cause it's to do with robby
94. N: I'll tell them that I spoke to students from last year and they say you have to use it [laughs]
95. A: [laughs] it's much more helpful
96. N: is there anything else that I should change?
97. A: hyperlinks more prominent…change it back to the pop-up window too…maybe even if it was like a bar along the top
98. N: yeah that might work better
99. A: if it's buggy get rid of that and make some of the error messages less…just you know dumb them down so it's not like you know if you're to get like string variable da da da it just…like overloads…like string variable what what what? Things like that.

Appendix F Interview with Student B

Interview recorded on 26th July 2006 at 5.30pm

1. N: (Orientation to interview.) How much did you enjoy the programming module?
2. B: I started to enjoy it more towards the end when I understood it more… um… first it was just too daunting … programming is not my thing I remember doing it at college and didn't get on very well with it was just like Pascal and stuff but once I started… you know developing programs which would do something… if you could actually see like the calculator one and things like that I kind of started to enjoy it more because we were seeing something at the end… I was… I thought that the questions were quite hard to interpret sometimes.
3. N: how did programming compare with your other modules?
4. B: it was definitely the worst, definitely
5. N: and that was because you found it hard rather than boring?
6. B: yeah, I just didn't get it, it took quite a while to click in, it was frustrating because I was like "I want to do this!" and the more I thought I want to do it the more it wasn't working. That was frustrating.
7. N: how often did you use the support tool?
8. B: I used it most of the time and quite a lot I'd use it to back up something that I wasn't, you know I maybe had an idea in my head of that was how to do it but I'd maybe check against the support tool to see if I was going in the right direction.
9. N: so did you still use the standard Compiler?
10. B: yes, I used it a lot, then the support tool to check.
11. N: did you like the tool?
12. B: I thought it was good. It threw back some strange messages sometimes and it seemed to be that it threw back the same answer for a lot of different problems but you had the web link to the PowerPoint slides that was really useful because then you could actually understand it better, it explained it more
13. N: So you like it?
14. B: Yeah, it was helpful, yeah. It was useful when it went to the PowerPoint thing to explain it.
15. N: Was there any aspect of it that you didn't like?
16. B: Just really when it brought up the same sort of answer for different problems, but that was to do with mistakes that I had in my program [syntax]
17. N: Did the tool enhance your learning experience?
18. B: It did especially, say… the links, because you could sit through a lecture and maybe not take it in, you know… I thought some of the lectures were quite long and by the end of it the stuff was going over you head so that was definitely helpful to have some sort of reference that you could go back to, it just sort of refreshes your mind a bit.
19. N: Did you find the tool frustrating?
20. B: Only when it brought up the same thing and I couldn't work out what the problem was.
21. N: Why did you choose to use the standard Compiler at times?
22. B: Sometimes if I knew that I wouldn't need much help I would try the standard Compiler first before using the tool but I definitely used the support tool more.
23. N: So did you see the tool as a definite 'support' tool?
24. B: Yes but I relied on it a lot.
25. N: Did you use it to work out the right solution?
26. B: Definitely in the first semester but towards the end of the module I just used it to check that what I'd typed with my code was ok.
27. N: Did it help you in the beginning of the module then?
28. B: Yes, especially with the PowerPoint things because it pulled out the relevant lecture notes so that definitely helped.
29. N: Did you ever think that you were getting too much help?
30. B: No! But I was probably turning to it too much rather than trying to think for myself
31. N: Did you find that the PowerPoint links, did they help you to understand?

32. B: Yes it helped me to understand because I was working on that bit at that time so it was sinking in more because I was actually putting it into practice rather than just reading it so yes that definitely helped.
33. N: What do you think the tool did, in your own words?
34. B: It is a support tool and it helps to find faults in the program and give guidance. It doesn't give you the answer but it either confirms what you're thinking or can point you in the right direction.
35. N: Should we use the tool next year?
36. B: Definitely yes
37. N: Did you use the tool if the tutor was unavailable or instead of the tutor?
38. B: I would use try the tool first to see if it could help me before I asked for help from the tutor or from a classmate.

Appendix G Interview with Student C

Interview recorded on Thursday 12th July at 2.30pm

1. N: [orientation to interview] so what is it about programming that you like?
2. C: uh… The challenge… of…getting a problem and trying to work out a solution to it [pause]. I got one at work last year, when I was working as a scripter and they knew that I wanted to be a programmer, so they gave me one called pentominoes, don't know if you've heard of that before…
3. N: no
4. C: you get a pentomino like a polyomino,
5. N: uh huh
6. C: [laughs] I've probably said it wrong, you know like a dominos so it's five squares instead of… uh …two and you have a 12 by 5 grid
7. N: right
8. C: and you have to… there's 12 of the pentominoes, and you have to work out every single combination that they can go in
9. N: so why were they wanting…?
10. C: uh… they didn't need it for anything it was just a thing they gave to people to test them
11. N: ah right… so was this as part of your induction thing?
12. C: eh no… it was like… I'd already been working there like 4 months when they gave it to me… it was after I'd done it they let me do some programming… cause you had to do it… after you'd solved it you had to do it fast as well…
13. N: uh huh
14. C: That was the real thing tough thing… you had to do it in like ten minutes…
15. N: [speak about programming competition for a bit…]
16. N: em.. so you think that you maybe ran the tool a couple of times, but not very much…
17. C: yeah
18. N: ok… so did you think that the tool in itself is a good idea
19. C: Oh yeah, yeah
20. N: what about it do you think is a good idea?
21. C: um… well I think the idea of it is it helps people, say you've got a problem in your tutorial, it's to like help people complete it, you know if it's a loop and they haven't got a loop in there it'll tell them they need a loop, it just sort of helps them along the way without actually telling them the solution.
22. N: uh huh, so you did see that that was…coming through
23. C: yeah…
24. N: that's good
25. N: is there anything else about it that you thought was a good idea?
26. C: um I suppose it helps you get help quicker than… you know cause there's only you and another tutor in the class… a lot of people can get help themselves rather than just waiting for it
27. N: so do you think it improved the programming labs?
28. C: yeah… helped them run faster,
29. N: I know this question doesn't apply but is there any aspect of programming that you found hard, or was there anything in the delivery method that wasn't right?
30. C: no I don't think so
31. N: were you aware of anything that your peers struggled with?
32. C: um yeah a lot of people when they got to… they were sort of following it… sort of plodding along up until when methods got introduced and then people started struggling and then when you went on to writing new classes which totally lost it
33. N: uh huh, [discussion of order of teaching material]
34. C: especially for people that aren't really interested in the course
35. N: yeah… um, it's really difficult for us to find a way of teaching it… um it's so hard but it's kind of got to be dry… [more discussion of Thursday classes]

36. N: what did you think… or… um…what do you think is the worst thing about the support tool.
37. C: uh, [laughs], em..
38. N: [laughs]
39. C: um probably the fact that it dumps the output back into the little console thing in JCreator, it could be more readable really than just …text there
40. N: so did you ever see it when it opened up in a separate window?
41. C: no I didn't
42. N: ah right, cause there was… um… depending on the type of problem it found in the students' program sometimes it delivered the message in the standard output window and what I thought when I was going round the class… I thought I saw… the students didn't like the other window popping up
43. C: yeah
44. N: and I thought… they just couldn't… they just didn't like flicking between the two… they couldn't modify their code and still have the other window visible…
45. C: yeah
46. N: …so, about half way through I think I changed it so it was delivering all the output in the smaller window but that then meant that I wasn't able to include hyperlinks
47. C: yeah
48. N: in the… the code… cause I can't seem to figure out a way of it recognising hyperlinks in the output window,
49. C: I can see that, but it's like… it's probably the fact that it is <u>just</u> [emphasis] text, like if it came and it said task list like it does when you've got a compilation error
50. N: uh huh
51. C: then that's… it's like cause you can select it and it's got all your individual errors all formatted out so you can…
52. N: uh huh
53. C: but that's like a minor thing
54. N: yeah… yeah
55. C: [laughs]
56. N: but you thought… like dumping it in the output window you… you see is like a negative thing
57. C: yeah especially if there is a few things wrong it might be a little bit overwhelming
58. N: uh huh…[pause], right ok… [pause] cause I felt I had to chose between having it in a window with hyperlinks or having it in the output window and it's a decision I made without speaking to any students I just thought that they were… it just seemed to… the window would just pop up and they were having a quick scan through the text then minimising it to look at the code without really reading it properly
59. C: that's probably right actually
60. N: but I do think that's a very valid point cause I don't think… I know I'm very limited with the format
61. C: and there's only so much room as well
62. N: yeah and in the output window and it defaults to the bottom and as you say if there's a lot of text then they have to scroll up the top… em…is there anything else that you thought…
63. C: em… no I don't think so
64. N: just bugs [laughs]… do you imagine that it would have been frustrating at all to use… is there any aspect of it that you consider could have been frustrating, apart from what you've just mentioned
65. C: em… no I don't think I can't think of anything apart from that
66. N: em… ok so if you could just say a bit in tape sort of about how often you used the tool
67. C: em… just a couple of times throughout the year
68. N: ok, just very very rarely
69. C: just to see what it did [laughs]
70. N: ok… em… did you ever… em… I suppose you never got stuck… but did you ever use it if you were stuck
71. C: I think I might of once or twice… you know you get a problem and you can't figure out what the hell it is … and it's probably really simple and I tried to see if it could give me any feedback with the Compiler error… couldn't
72. N: uh huh… so why did you choose not to use it

73. C: um I just didn't … really need to… I would… I never really thought
74. N: yeah so you recognised that you were able to work with the standard
75. C: yeah
76. N: so do you think that you definitely saw it as a support tool
77. C: oh yeah it's definitely…
78. N: rather than something you would use every compilation attempt
79. C: yeah definitely… I think some people just don't use it that way, that's the problem, maybe if you get a Compiler error you can get the support tool as well
80. N: I think there were a couple of students in the class who used it like with every button press they used the support tool and there were a think more students in the group that used it when they were stuck so you know if they didn't understand the Compiler error… em… but when I built it I didn't really think that it would be seen as a support for only when you're stuck only I didn't want to remove the standard compile either… you know
81. C: yeah
82. N: for people like yourself who are quite capable I think then it just makes you feel like you're getting wrapped in cotton tool
83. C: yeah you need to get used to those message at some point anyway
84. N: absolutely…
85. C: cause they are so… crap…[laughs] that you need to
86. N: yeah [laughs] that's why we reduced the amount of support so that the students were forced to become more independent than [can't hear this part]… em I would like it if you could tell me what you think the support tool did
87. C: em it helps
88. N: you know if you had to describe it to another student what would you say it does
89. C: it's a tool that assists you when you've got a problem if you can't figure out why this program's working say if it compiles fine but it's not working correctly then you can use the support tool and it'll tell you what you're doing wrong
90. N: ok. That's fine … so do you think that the tool should be used next year
91. C: yeah,
92. N: so you think it's a good idea
93. C: oh yeah
94. N: if I get rid of the bugs [laughs]
95. C: yeah
96. N: Apart from the text window can you think of anything about the tool that I could change to make it better
97. C: um… well are you still going to have those, like 3 Compilers next year
98. N: no
99. C: cause that's confusing
100. N: no that was for the purpose of the test
101. C: it's just cause they hung around after [laughs]
102. N: [laughs] it's just cause it was hard to get information services to come cause we have to create a profile and then set it up on every single machine
103. C: yeah
104. N: and I suppose I was a bit lazy that I didn't say to information services that they could remove them [laughs]
105. Um… no next year it would just be one button
106. C: I can't really think of anything, I think it does the job
107. N: ok… em… you're obviously familiar with the queuing idea.
108. C: yeah
109. N: what do you think about that concept?
110. C: I think it's a good idea. It'll help… its'… it'll help fairness more than anything else cause I have heard people complain
111. N: yeah, if we don't see their hand up

112. C: that can be frustrating… especially if you're stuck on something… uh for a while and someone gets more attention than you
113. N: yeah… ok so you certainly think it's a good idea… that students would see value in it
114. C: yeah, apart from that it tells you… its not just about… uh being fair and telling you who next… you'll be able to see exactly how many people have got the same problem as you as well and you know it'll log all the data and you'll be able to sift through it as well
115. N: uh huh and I'm certainly hoping that there would be cases with certain exercises where a student would benefit from getting the information that another student with that some problem had and then they might be able to solve it themselves. em…ok… that's us done.

Appendix H Module Leader Comments

Commentary on the programming support tool, SNOOPIE

Module leader, Object Oriented Programming 1, University of Abertay Dundee

1. I have taught OOP1 from 2000 to present. The SNOOPIE learning support tool has been incorporated into the delivery of this module in both 2004/5 (version 1) and 2005/6 (version 2) sessions at the University of Abertay Dundee.

2. Version 1 packaged the standardised syntax errors from the compiler together with a more detailed description of those errors, and the SNOOPIE tool was very easy for students to invoke – a single button is available next to the standard compiler. I understand that the concept of extending error messages is not new, but the ease with which that additional text can be revised to take account of the current stage of teaching was very attractive. For example, when teaching within a particular micro-world I was able to update the errors to reflect that micro-world. Of course, the specific text supporting each error message made assumptions about the most common cause(s) of that error. It is worth noting that these assumptions were invariably correct.

3. Additionally, version 1 provided some support for syntactically valid program code. This focused on both misplaced ';', for example at the end of a while loop, and on ensuring that looping counters are used properly. This type of learning support was not as generally useful as the syntax error extension facility. However, I have observed SNOOPIE advising students on such rogue ';' or the failure to update while loop counters in particular, and students correcting the code in accordance with that advice. Where needed SNOOPIE identifies common problems such as this and students can use the information provided to address those problems.

4. Generally, students in the 2004/5 cohort were receptive towards SNOOPIE version 1. The majority, if not all, of the students recognised the value of the extended syntax error messages in making clear otherwise terse error messages, and appreciated the general level of support offered. The dynamic updating of the text associated with each compiler error as the module progressed gave a sense of union between the teaching notes and the Java programming language. Based on my discussions with students, they saw no real difference between the syntactic support and other support offered by SNOOPIE – it was seen as the same support program. I believe that SNOOPIE version 1 provided students with a sense of being supported throughout the module, and particularly in the first term. The tool afforded more opportunity for students to correct errors on their own and I could see that this promoted confidence in their ability to solve problems.

5. SNOOPIE Version 2 included all of the provision of version 1, and made a more substantial contribution to the program development process. As with version 1, this learning support was available by pressing a single button within the development environment. Each supported programming exercise was characterised, in consultation with me, as a number of ordered program requirements, i.e. key fragments of code that had to be in place. These requirements were then implemented by the author and specific feedback written to explain to the student both why their program code did not (currently) meet the requirements of the question and what actions to undertake next. Generally, the style of the feedback created by

the developer was sensitive to the needs of the student and the content was appropriate for guiding the student through the question in a structured manner.

6. Version 2 affords differentiated support for different questions. As appropriate to my teaching needs, I was able to specify many or few requirements for each question and for each requirement a suitable level of guidance towards the next step. This flexibility allowed SNOOPIE to provide substantial support for new topics or specific areas that have caused difficulties with previous cohorts and simultaneously reducing support for familiar topics and key assessment activities. SNOOPIE's flexibility extends into the types of program requirement checks that were possible and the form of the feedback given. At no point was I constrained by the types of requirements that I wanted to check for. The feedback allowed hyperlinks to both the lecture notes, including very specific reminders on construct usage as provided by the author of the tool, and the web as required.

7. Some students, generally those that already had some exposure to programming, did not want to use the tool preferring to work with the basic compiler support only. At the other end of the spectrum, those students that were typically least confident in their programming ability used the tool most of the time. Interestingly, even though (at my request for a small number of formative exercises) SNOOPIE almost wrote the code for the student by guiding them through the program required for each requirement specified, at no point did I see a student blindly stringing together the advice provided. All students wanted to learn to program and used SNOOPIE for support and not as a shortcut to exercise completion.

8. A useful side effect of SNOOPIE was consistency in staff feedback to students. Several staff were involved in the module and SNOOPIE feedback provided a useful platform for support staff to advise students in a manner consistent with my intentions for a given question. All support staff that have worked on the module have made this comment. Additionally, my teaching has been peer observed by two other Universities in the 2005/6 session. Both observers were very positive about the Version 2 support, and were particularly impressed by the flexibility in support available. In fact, one of the observers has taken the concept of the support offered and is implementing a similar tool for his functional programming course. Based on my experiences with SNOOPIE version 2 in Term 1, I asked the author to extend the support into Term 2, covering the teaching of methods.

9. I believe that the SNOOPIE tool has made a significant contribution to the teaching of programming, including promoting confidence in students who are otherwise hesitant in exploring program development on their own and reducing the time spent by staff solving problems that students can solve with SNOOPIE support. However, there has been an issue with the style of presentation. Repeatedly, students did not actually read the text provided by SNOOPIE. In some but not all cases, the text provided was expressed as a contiguous block. Perhaps colour coding of key textual elements would make the text more accessible. That said, the text was written clearly and concisely and those students who chose to read it found it useful.

10. Perhaps the strongest supportive statement I am able to make is that, at my request, I have been able to incorporate the SNOOPIE software in the module delivery for this 2006/7 session.

Appendix I Lab Demonstrator Comments

Commentary on the programming support tool, SNOOPIE
Lab Demonstrator, Object Oriented Programming 1, University of Abertay Dundee

1. I have taught Object Oriented Programming for the past four years and have used the SNOOPIE tool for the last two of these years. I find it to be an extremely useful teaching tool. SNOOPIE can simplify the compiler errors that students get.
2. In previous years, when faced with a list of seemingly complex errors, students new to programming can panic and feel as if they will never resolve them. SNOOPIE encourages the student to deal with one error at a time, and can pick up the syntax errors and provide useful feedback on how to resolve them. This allows me as a tutor to spend more time explaining the more complex problems to the students rather than trouble shooting trivial errors.
3. I have found that SNOOPIE encourages self-learning, as not only does it provide simplified explanations of the errors, there are links to Power Point slides that give programming examples.
4. The dialogue employed by SNOOPIE encourages the student to explain and think about the problem they are having in the correct terminology. I have found that students who can resolve the majority of the compiler errors that they encounter have more belief in their own abilities and therefore learn better.
5. SNOOPIE version 2 not only provides feedback for syntax errors, but it can provide structured feedback to set exercises that are not detected by the compiler. From an instructor's point of view, this guides the student towards a model solution by providing outputs that give guidance as to the structure of the program (e.g. prompting the student to include two for loops). I found this to be useful for students that were not sure where to begin as it enabled them to make a start to the program.
6. As there are several instructors involved in teaching the module, it enables us to easily give uniform guidance to the students on solutions, which prevents confusion when they discuss the solutions with their peers.
7. The feedback that I get from the students about SNOOPIE is also very positive. They can get immediate hints and guidance without having to wait for tutor assistance, and because it is optional as to whether they use the compiler or SNOOPIE, those that are capable programmers are not held back from using the normal system.
8. Additionally, because the compiler errors are also displayed, when the students are using a normal compiler, they recognise and are able to solve errors that they have encountered and have been explained by the SNOOPIE tool. I find it a very useful addition to the learning tools for novice programmers.

Bibliography

Anderson J. R. and Skwarecki E. 1986. The Automated Tutoring of Introductory Computer programming. *Communications of the ACM*. 29. p842-849

Badros, G. 2000. JavaML: A markup language for java source code. In *Proceedings of the Ninth International Conference on the World Wide Web,* Amsterdam. The Netherlands. May 2000. Elsevier Science B. V.

Bental, D. 1993. Why doesn't my program work? Requirements for automated analysis of novices' computer programs. Workshop on automated program understanding AI&ED, Conference on AI in Education

Beizer, B. 1990. *Software Testing Techniques*. 2nd ed. Van Nostrand Rheinold, New York.

Bloom, B.S. (1956). *Taxonomy of Educational Objectives: Handbook of Cognitive Domain*. New York: McKay. Quoted by Buck, D. and Stucki, D. J. 2001. JKarelRobot: a case study in supporting levels of cognitive development in the computer science curriculum. In *Proceedings of the Thirty-Second SIGCSE Technical Symposium on Computer Science Education.* North Carolina, USA. p16-20.

Boufaida, M. 1996. What courseware dedicated to computer science? *SIGCSE Bulletin.* 28(4). p8-14.

Brna, P. and Mathieson, M. 1993. Support for Novices Learning to Debug: SWANN's Way. *Artificial Intelligence in Education, 1993: Proceddings of World conference on AI in Education. Virginia USA.* p505-512.

Buck, D. and Stucki, D. J. 2001. JKarelRobot: a case study in supporting levels of cognitive development in the computer science curriculum. In *Proceedings of the Thirty-Second SIGCSE Technical Symposium on Computer Science Education.* North Carolina, USA. p16-20.

Chase, W.G., and Simon, H.A. (1973) "Perception in chess," Cognitive Psy. 4. p55-81. Quoted in: Mayer, R. E. 1981. The Psychology of How Novices Learn Computer Programming. *ACM Comput. Surv.* 13(10). p121-141.

Chmiel, R., Loui, M. 2004 Debugging: from novice to expert. In *Proceedings of the 35th SIGCSE Technical Symposium on Computer Science Education*, Norfolk, Virginia, USA. p17 – 21.

Clarke, G.M. and Cooke, D.1982. *A Basic Course in Statistics* 2nd Ed. Edward Arnold, London, UK.

Clark, R., and Harrelson, G. L. 2002. Designing Instruction That Supports Cognitive Learning Processes. *Journal of Athletic Training.* 37(4). S152-S159

Crews, T. and Ziegler, U. 1998. The Flowchart Interpreter for Introductory Programming Courses. *Proceedings of Frontiers in Education Conference.* Tempe, AZ, USA. p307-312.

Culwin, F. Adeboye, K. and Campbell, P. 2005. POOPLE (Pre-Object Oriented Programming Learning Environment) Prototypes. . In *Proceedings of 6th Annual Conference of the ICS HE Academy*. York, UK. August 2005.

du Boulay, J.B.H. 1986. Some Difficulties of Learning to Program. *Journal of Educational Computing Research*. 2(1). p57-73.

Deek, F. P. and McHugh, J.A. 2000. SOLVEIT: An Experimental Environment for Problem Solving and Program Development. *Journal of Applied Systems Studies, Special Issue on Distributed Multimedia Systems with Applications*. 2(2). p376-296

Downey, A. 2002, *How to think like a computer scientist, Java version*. Green Tea Press. viewed 21 November 2005. <http://greenteapress.com/thinkapjava/>.

Ellington, H., Earl, S. 1996 *How Students Learn – A review of some of the main theories.* [online]. Available from: http://apu.gcal.ac.uk/ciced/ Ch02.html [Accessed 2nd May 2006].

Ebrahimi, A. 1994. Novice programmer errors: language constructs and plan composition. *International Journal of Human-Computer Studies.* 41(4) p457-480.

Etheredge, J. 2004. CMeRun: program logic debugging courseware for CS1/CS2 students. In *Proceedings of the 35th SIGCSE technical symposium on Computer science education*. p22-25.

Etter, M. 2006. JeXQL Language and Compiler for the development of a teaching tool. BSc. Hons thesis, University of Abertay Dundee.

Fincher, S. Barnes, D. Bibby, P. Bown, J. Bush, V. Campbell, P. Cutts, Q. Jamieson, S. Jenkins, T and Jones, M. Some good ideas from the disciplinary commons. In *Proceedings of 7th Annual Conference of the ICS HE Academy*. Dublin, Ireland. August 2006. p153-158

Garner, S. 2003. Learning Resources and Tools to Aid Novices Learn Programming. *Joint Conference Informing Science InSITE*. Poland.

Gómez-Martín, P.P., Gómez-Martín, M.A. and González-Calero, P.A. 2003. Javy: Virtual Environment for Case-Based Teaching of Java Virtual Machine. In *Proceedings of the 7th International Conference on Knowledge-Based Intelligent Information & Engineering Systems*. p906-913.

GRUMPS. (2001). *The GRUMPS Research Project*. [online]. Available from: http://grumps.dcs.gla.ac.uk [Accessed: 21st November 2006]

Haaster, K. and Hagan, D. 2004. Teaching and Learning with BlueJ: an Evaluation of a Pedagogical Tool, *Information Science and Information Technology Education Joint Conference*, Rockhampton, QLD, Australia.

Hanks, P. 1998. *The New Oxford Dictionary of English*. Oxford: University Press.

Harvey, N. 2006. JAMS: Java Automated Marking System. BSc (Hons) thesis. University of St Andrews.

Higgins, C., Hegazy, T., Symeonidis, P., and Tsintsifas, A. 2003. The CourseMarker CBA System: Improvements over Ceilidh. *Education and Information Technologies.* 8(3). p287-304

Higher Education Academy, Engineering Subject Centre. 2000. *Deep and Surface Approaches to Learning*. [online] Available from: http://www.engsc.ac.uk/er/theory/learning.asp. [Accessed 5th May 2006].

Hristova, M. Rutter, M.and Mercuri, R. 2003. Identifying and Correcting Java Programming Errors for Introductory Computer Science Students. In: *Proceedings of the 34th SIGCSE technical symposium on Computer Science Education.* Reno, Nevada, USA. 34. p153-156.

Hayes, J. H. 1994. Testing of Object-Oriented Programming Systems (OOPS): A Fault-Based Approach, *International Symposium on Object-Oriented Methodologies and Systems.*Palermo, Italy September 1994.Springer Valley.

IEEE 1993. Standard Classification forSoftware Anomalies. IEEE Std 1044-1993(R2002). The Institute of Electrical and Electonics Enginners, Inc. New York, USA.

Jadud, M. 2005. A first look at novice compilation behavior. *Computer Science Education*, 15(1).p25-40

Jadud, M. C. 2006. Methods and tools for exploring novice compilation behaviour. In *Proceedings of the 2006 international Workshop on Computing Education Research* .Canterbury, United Kingdom, September 09 - 10, 2006. ACM Press, New York, USA. p73-84.

Jenkins, T. 2001. The motivation of students of programming. ITiCSE '01: *Proceedings of the 6th Annual Conference on Innovation and Technology in Computer Science Education.* Canterbury, United Kingdom. ACM Press, New York, USA. p53-56.

Johnson, C. 2006. Roles of Variables. A one-day workshop of the 2nd International Computing Education Research Workshop ICER 2006. 8th September. Kent, UK.

Johnson, W. L. 1990. Understanding and debugging novice programs. W. J. Clancey and E. Soloway, Ed. *Artificial intelligence and Learning Environments*, Bradford Company, Scituate, MA, USA. p51-97.

Johnson, W.L. and Soloway, E.1985. PROUST: Knowledge-Based Program Understanding. *IEEE Trans. Software Eng.* 11(3). p267-275

Kessler, C. M. and Anderson, J. R. 1986. A model of novice debugging in LISP. In E. Soloway and S. Iyengar, eds, *Papers Presented At the First Workshop on Empirical Studies of Programmers on Empirical Studies of Programmers.* Washington, D.C., USA. Ablex Publishing Corp., Norwood, NJ, USA. p198-212

Koile, K. and Singer, D. 2006. Improving learning in CS1 via tablet-PC-based in-class assessment. In *Proceedings of the 2006 international Workshop on Computing Education Research*. Canterbury, United Kingdom, September 09 - 10, 2006. ACM Press, New York, USA. p119-126.

Kölling, M. and Rosenberg, J. 1996. Blue—a language for teaching object-oriented programming. In K. J. Klee, ed: *Proceedings of the Twenty-Seventh SIGCSE Technical*

Symposium on Computer Science Education. Philadelphia, Pennsylvania, United States, February 15 - 17, 1996.ACM Press, New York, USA. p190-194

Kölling, M. Quig, B. Patterson A. and Rosenberg, J. 2003 The BlueJ system and its pedagogy. *Journal of Computer Science Education*. 13(4). p249-268.

Lang, B. 2002. Teaching new programmers: a Java tool set as a student teaching aid. *Principles and Practice of programming in Java*. p 95-100

Lahtinen, E., Ala-Mutka, K., and Järvinen, H. 2005. A study of the difficulties of novice programmers. In ITiCSE '05: *Proceedings of the 10th Annual SIGCSE Conference on innovation and Technology in Computer Science Education.* Caparica, Portugal, June 27 - 29, 2005. ACM Press, New York, USA, p14-18.

Lister, R. 2004. Teaching Java first: experiments with a pigs-early pedagogy. In R. Lister and A. Young, eds: *Proceedings of the Sixth Conference on Australasian Computing Education.* Dunedin, New Zealand. Australian Computer Society, Darlinghurst, Australia.p177-183.

Lund, G. 2002. *Quality Aspects of the Program Development Process used by Learner Programmers*. PhD thesis, University of Abertay, Dundee, UK. (2002)

Mayer, R. E. 1981. The Psychology of How Novices Learn Computer Programming. *ACM Comput. Surv.* 13(10). p121-141.

Mayer, R.E. 1997. From Novice to Expert. In Helander, M. Landauer, T.K. and Prabhu, P., eds. *Handbook of Human Computer Interaction* 2nd.Ed. Elsevier-Science B.V.

McGill, T.J. and Volet, S.E. 1997 A Conceptual Framework for Analysing Students' Knowledge of Programming. *Journal of Research on Computers in Education*. 29(3). p276-297.

MacNish, C. 2000. Java Facilities for Automating Analysis, Feedback and Assessment of Laboratory Work. *Journal of Computer Science Education.* 10(2). p147-163.

Milne, I. and Rowe, G. 2004. OGRE: Three-Dimensional Program Visualization for Novice Programmers. *Education and Information Technologies.* 9(3). p219-237

Moore, S. and Taylor, K. 2005. An Intelligent Interactive Online Tutor for Computer Languages. In: *Proceedings of the 25th Annual International Conference of the British Computer Society's Specialist Group on Artificial Intelligence (SGAI).*

Odekirk-Hash , E. and Zachary, J. 2001 Automated feedback on programs means students need less help from teachers. *ACM SIGCSE Bulletin*. 33(1). p55-59

Paine, C. 2001. The Coach – Supporting student in the area of error reports. In: *Proceedings of the 13th Workshop of the Psychology of Programming Interest Group*. Bournemouth UK.

Pattis , R., Roberts J. and Stehlik, M. 1995 Karel the robot: a gentle introduction to the art of programming. 2nd ed. John Wiley & Sons, Inc., New York, USA. Quoted by Buck, D. and

Stucki, D. J. 2001. JKarelRobot: a case study in supporting levels of cognitive development in the computer science curriculum. In *Proceedings of the Thirty-Second SIGCSE Technical Symposium on Computer Science Education.* North Carolina, USA. p16-20.

Pennington, N. 1987. Comprehension strategies in programming. In G. M. Olson, S. Sheppard, and E. Soloway, eds. *Empirical Studies of Programmers: Second Workshop*,. Ablex Series Of Monographs, Edited Volumes, And Texts. Ablex Publishing Corp., Norwood, NJ, USA. p100-113.

Redmiles, D. F. 1993. Reducing the variability of programmers' performance through explained examples. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. Amsterdam, The Netherlands, April 24 - 29, 1993. p67-73.

Robins, A., Rountree, J. and Rountree, N. 2003. Learning and teaching programming: A review and discussion. *Computer Science Education*, 13(2). p137 - 172.

Rowe, G. and Thorburn, G. 2000. VINCE-an on-line tutorial tool for teaching introductory programming. *British Journal of Educational Technology*. Blackwell Synergy.

Sanders, D. and Dorn, B. 2003. Jeroo: a tool for introducing object-oriented programming. In SIGCSE '03: *Proceedings of the 34th SIGCSE Technical Symposium on Computer Science Education* Reno, Navada, USA, February 19 - 23, 2003. ACM Press, New York, USA. p201-204.

Sajaniemi J. 2005. Roles of Variables and Learning to Program. In: A. Jimoyiannis ed.: Proceedings of the 3rd Panhellenic Conference "Didactics of Informatics", University of Peloponnese, Korinthos, Greece, 7-9 2005

Schorsch, T. 1995. CAP: An Automated Self-Assessment Tool To Check Pascal Programs For Syntax, Logic And Style Errors. ACM SIGCSE Bulletin. 27(1). p168-172.

Shah, H. and Kumar, A. N. 2002. A tutoring system for parameter passing in programming languages. In ITiCSE '02: *Proceedings of the 7th Annual Conference on innovation and Technology in Computer Science Education.* Aarhus, Denmark, June 24 - 28, 2002. ACM Press, New York, USA. p170-174.

Shneiderman, B. and Mayer, J.E. 1979. Syntactic/ semantic interaction in programmer behavior: A model and experimental results. *International Journal of Computer and Information Science*. 8 (3), p219-238.

Shneiderman, B.1980. *Software psychology: Human factors in computer and information systems,* Winthrop, NewYork, USA. Quoted by: Mayer, R. E. 1981. The Psychology of How Novices Learn Computer Programming. *ACM Comput. Surv.* 13(10). p121-141.

Soloway, E. and Ehrlich, K. 1989. Empirical studies of programming knowledge. In T. J. Biggerstaff and A. J. Perlis, eds, *Software Reusability: Vol. 2, Applications and Experience*. ACM Press: New York, USA. p235-267

Spohrer, J.C. and Solway, E. 1986. Analyzing the High Frequency Bugs in Novice Programs. Empirical Studies for Programmers: First Workshop, Ablex Publishing Corp.

Storey, M., Damian, D., Michaud, J., Myers, D., Mindel, M., German, D., Sanseverino, M., and Hargreaves, E. 2003. Improving the usability of Eclipse for novice programmers. In *Proceedings of the 2003 OOPSLA Workshop on Eclipse Technology Exchange* Anaheim, California, October 27 - 27, 2003.ACM Press, New York, USA. p35-39.

Utting, I. 2006. The BlueJ Extensions framework and API. In: Using BlueJ as a Research Tool. A one-day workshop of the 2nd International Computing Education Research Workshop ICER 2006. 8th September. Kent, UK.

Vessey, I. 1985. Expertise In Debugging Computer Programs: A Process Analysis. International Journal Of Man-Machine Studies, 23 (5), p459-494.

Wang, H., LearnOOP: An Active Agent-Based Educational System. *Expert Systems with Applications*. 12(2). p153-162

Watson, M., McSorley, Foxcroft C. and Watson, A. (2004) Exploring the motivation orientation and learning strategies of first year university learners. Tertiary Education and Management 10: 193-207.

Wertz, H. 1982. Stereotyped program Debugging: an aid of Novice Programmers. *International Journal of Man-Machine Studies.* 16. p379-392

Ziegler, U. and Crews, T., 1999. An integrated program development tool for teaching and learning how to program. In: *Proceedings of the thirtieth SIGCSE technical symposium on Computer science education*, March 24-28 1999. New Orleans, Louisiana, USA. p276-280.