Programming, disciplines and methods adopted at Liverpool Hope University

Whitfield A K, Blakeway S, Herterich G E, Beaumont C.

School of Computer Science
Liverpool Hope University
Hope Park, Liverpool L16 9JD
Email: whitfia@hope.ac.uk, blakews@hope.ac.uk, herterg@hope.ac.uk, beaumoc@hope.ac.uk

Abstract: The ability to problem solve is one of the key skills that a computing student requires in order to learn to program. Although students recruited to the more technically challenging Computer Science degree will have higher mathematical skills at secondary level, students recruited to the Information Technology degree usually have considerably less mathematical ability. This paper reflects upon the fundamental techniques that Liverpool Hope University has adopted to assist in the learning of programming for entry level students on an undergraduate level degree in Computer Science and Information Technology. It examines the key principles of design, development and practice and how the teaching incorporates these principles into the first year courses. It presents a set of strategies developed at Liverpool Hope University over several years which have been successful in promoting programming as an interesting and worthwhile discipline. The development of bespoke software and customised text books to guide teaching and learning is discussed, as is students' response, via a questionnaire, to these methods of learning programming.

Keywords: Problem solving skills, teaching programming, programming text books.

Acknowledgements: The authors would like to acknowledge the contribution of Levels C and I students (Academic Year 2006-07) in response to the questionnaire on the learning and teaching of programming. Student comments remain anonymous and verbatim, and permissions have been given to publish.

1. Introduction

1.1 Difficulties of Learning to Program

According to Jenkins and Davy (2001) students now entering computing and IT degree courses are a much more diverse group than twenty years ago. In addition Beaumont (2003) points out that many IT degree students no longer view programming as central to the course.

Beaumont (2003) develops the idea of Jenkins (2002) analysing a number of factors that could contribute to the difficulty of learning to program, as follows:

1.1.1 Multiple skills:

- Problem-Solving which is complex in itself requires a number of skills including identification of the central issues in the problem, recognition of relationships, familiar situations and patterns, development of an algorithm for solution and in the case of computing, it requires the translation of the algorithm into executable code.
- Use of IT such as editing, compiling, saving, loading, combining files and dealing with the IDE interface.
- Hierarchy of skills and knowledge. Detailed knowledge of the syntax is required, and experience is the only answer to interpretation of the obscure error messages that compilers often generate. Extreme accuracy is required. While these are certainly at the lower end of Bloom's taxonomy (Bloom, 1965) they do not mean programming is a low-level skill. In order to write programs, students need to analyse problem statements, synthesise solutions and evaluate whether they meet the specification. These are the higher levels of Bloom's taxonomy.

1.1.2 Educational Novelty and Pace

Constructivists argue that learning involves the creation of meaning by students as they interact with their learning environment, relating new knowledge to existing knowledge and integrating it into their conceptual framework. Programming has little to relate it easily to the familiar; it is largely abstract and thus difficult to relate to existing knowledge. Dijkstra (1989) refers to this as "radical novelty".

The object oriented approach claims to have a naturalness that would help learners relate more easily to the concepts, and hence learn it faster. However, research by Scholtz et al (1993), suggests that this claim is unfounded, and that procedural programming is easier to learn.

Programming is a subject that builds continuously. If a student fails to grasp a particular concept, then it becomes difficult to catch up and the pace of the teaching is often driven by the curriculum, not the learning of the student. (Beaumont 2003)

Thus, programming course designers face considerable challenges in helping students overcome the demands of conceptual complexity, educational novelty and multiple skills required to learn programming. This is compounded by the diversity of the student population.

1.2 Mathematics and Computing

When teaching programming to first year students there are various issues to be considered. At Liverpool Hope University the undergraduate degree course in Information Technology does not require any prerequisite skills. Some students have experience of programming languages prior to degree study, whilst others have none. The more technically oriented Computer Science undergraduate course recruits students with higher mathematical skills at secondary education level but the IT undergraduate course may have students with considerably less mathematical prowess. The standard of mathematics education in schools appears to be at an all time low; this coupled with the lower mathematics entry requirements for incoming students, means that they begin programming courses without the problem solving basics which are associated with a good mathematics education. (Herterich 2004)

The decline in mathematical ability, specifically in analytical and logical reasoning, problem solving skills and algebraic manipulation has had adverse implications not only in Higher Education Mathematics, but also in Computing Disciplines. This decline has lead to problems with formal manipulation of symbols (Programming) and multi-step processes (Algorithmic Design and Data Structures). It was also commented that there is a general lack of “fluency” in mathematical communication skills, both verbal and aural. This has knock-on effects in the

teaching and learning of such skills as programming and analysis and design methods. (Herterich 2004)

The six steps, described by Burton and Bruhn (2003) to write a simple procedural program, are:

1. Read and understand the problem
2. Devise a solution to the problem
3. Formalise the solution to the problem
4. Write a program
5. Test and debug the program
6. Document the program

Students entering a degree course with a strong mathematical background, such as Computer Science students, should already understand some of the six steps. However, we see a reduction in entry requirements in order to attract students. In particular IT courses attract students with lower mathematics skills. The falling numbers of computing applicants over recent years accounts, in part, for this trend. Smith (2004) in his report on The Inquiry into Post-14 Mathematics Education recognised that it is essential to overcome the present situation whereby Sixth Form and FE Colleges cannot attract sufficiently and appropriately qualified mathematics teachers. This can have a direct affect on the lack of problem solving skills in first year students and whether they can successfully engage with the programming course.

1.3 The Post–14 Mathematics Inquiry

The Post–14 Mathematics Inquiry identifies three key issues of major concern:

- the shortage of specialist mathematics teachers, particularly in England and Wales;
- the failure of the current curriculum, assessment and qualifications framework in England, Wales and Northern Ireland to meet the needs of many learners and to satisfy the requirements and expectations of employers and higher education institutions;

- the lack of resources, infrastructure and a sustained continuing professional development culture to support and nurture all teachers of mathematics.

McCartney (2004) considers that given a reduction in entry requirements from A level maths to GCSE grade C for entry onto a Computer Science Degree (University of Ulster) it is possible for students to confidently embark on a course for which they are totally unequipped.

According to Herterich (2004) it seems to be common practice for subjects with a significant mathematical content (Computer Science, Gaming) to:

- set diagnostic maths Tests on entry
- have Maths Support Centres permanently manned and resourced;
- have separate maths modules as opposed to maths in context.

The impact of the present situation can be seen in the following statistics reported from University of Ulster and National University of Ireland (NUI), Galway:

- The Maths for Computing module at the University of Ulster had an 80% failure rate in 2003.
- A 75% failure rate in a first year Maths Course at NUI, Galway.

  Flynn (2004)

Where a decision is made to reduce the maths entry requirement to below A level, then serious consideration needs to be given to the affect this will have on students learning to program.

2. Liverpool Hope Students

The teaching of introductory programming to an increasingly diverse student population is problematic. Computing students at Liverpool Hope University may be on a Combined Subjects degree course, taking computing with another subject; this could be in a non-scientific area. Other students may be on a Qualified Teacher Status degree course, whereby the IT element is a minor part of their degree although their mathematical skills are sufficient for entrance onto a Qualified Teacher degree. The amount of work required of them from their Education studies often overshadows the requirements of their minor subject.

Similarities to this situation are described by Anderson et al (2003) in relation to Liberal Arts Students. Where students do not see programming as an essential core of their studies they will not be motivated enough to devote sufficient time to the study.

2.1 Problem Solving Skills

According to Jenkins (2002) problem solving is a complex process which requires various skills. The problem has to be identified and understood. It can be related to the familiar, to relationships and patterns. In computing, students need to translate the algorithm for the solution into code. Dijkstra (1989) discusses programming with its abstract nature as not being easily related to the familiar or existing knowledge: “radical novelty”. Oldehoeft and Roman (1977) consider that students need to understand problem solving methodologies at the basic level if they are to progress as programmers.

Before an attempt is made to teach programming it is important for students to be able to identify a problem and to understand the steps to be taken to solve that problem. The preliminary steps necessary to enable the student to reach a solution are often ignored in many texts that focus on programming languages, as they are difficult to conceptualise. By taking students through the steps of structured problem solving, flow charts and trace tables, they begin to work towards identifying sequence, selection and repetition.

The steps involved in this process are to understand the problem and devise a solution which can be formalised as an algorithm. This algorithm, or sequence of steps, is translated into a program as described by Burton and Bruhn, (2003). If these steps are not followed logically then the student will find errors in the program which require debugging and retesting.

2.2 Teaching Materials

2.2.1 Programming Text Books

Materials used for delivery of information to students who are beginning to learn programming need to be specific for the purpose and aligned to the particular course offered. Bennedsen et al (2005) describe the usefulness of text books as limited to the delivery of the finished product but not for the development process. Programming text books tend to start off easy to understand, developing very quickly into a complexity that demands a much higher level of

understanding from students. From being able to understand the work students can quickly become confused and demoralised because of this swift build of complexity.

2.2.2 Bespoke Text Books

Advantages

At Liverpool Hope various text books had been used in the past and found to be insufficient for the needs of the students and for the delivery of course materials. They were costly to purchase and students felt they had spent money on something that did not assist in the learning process. A decision was taken to write text books for the delivery of the course. Lectures, exercises and assessment are now appropriate to the delivery of the course through customised texts. This approach of providing tailored texts helps by providing material at appropriate pace and complexity for the recruited student population. An added advantage to students is that they pay for materials at cost price as they are printed in-house, or can be printed from the appropriate web pages.

Disadvantages

The greatest disadvantage of bespoke text books is the amount of staff time need to research, develop and produce the books. The team needs good co-ordination in order to create something which fits the needs of the students on a particular course. Although the greater part of the work is completed the first time of writing and delivery, the team undertake some revision of materials each year in light of course evaluation.

2.2.3 Web Site

Each module delivered by the Computing Department has its own supporting web pages, which contain materials such as portfolio exercises and model solutions to help the novice programmer.

2.2.4 In-House Software

In addition to the development of text books consideration of the appropriate programming language to teach beginners was given much thought. Commercial Integrated Development Environments (IDEs) were considered too complex for first year students and a decision was made to create in-house applications to provide an appropriate simplified environment in

which to teach programming constructs. Initially this was a Pascal Trainer and this academic year, 2006/07, students have had the benefit of a Java Trainer as described later in this paper.

3. Student Experience at Liverpool Hope

Careful composition of courses is essential in order not to confuse students as they begin programming. The three stages involved in teaching programming are:

Stage 1 The tutor models the process of problem solving and provides the solution.

Stage 2 The student imitates the tutor, going through the steps of problem solving until a solution is reached.

Stage 3 The student is able to undertake independent development.

Students who find it difficult to program are often students with limited ability to problem solve. They find it difficult to break a problem down into component parts and apply logical thinking in order to solve it. Students are often keen to write code but fail to assess the initial problem before attempting to code. To overcome these hurdles Liverpool Hope University has adopted and made use of many techniques already available for the teaching of programming. This paper intends to reflect on the success of the techniques offered to students and focus on the in-house software that is used initially before the students use more complex coding environments such as BlueJ™, or a commercial IDE. It should be noted that the undergraduate degree course that the student undertakes is split into two sections; Computer Systems and Structured Problem Solving. The decision was taken a couple of years ago to split the course so that students covered Structured Problem Solving in one seminar and Computer Systems in another. The rationale behind this is that a student may not be strong in one area of the course and may feel that they are failing. By splitting the content a student, weak in one area, has the benefit of achieving success in the other. This paper will focus on the Structured Problem Solving strand of the course which constitutes 40% of the assessment.

3.1 Why Procedural Approach before Object Oriented

A decision was made to teach procedural programming before introducing students to the object oriented approach. In studies that compare the three design methodologies of object oriented, process and data, the novice programmers produced better results when using process methodology and worst using object oriented methodology (Vessey and Conger, cited in Scholtz 1993). Although the object oriented approach is considered by some to be more natural, the team felt that the procedural or top down approach makes more sense to most people when getting to grips with problem solving. The object oriented approach is introduced later in the module when students have grasped the three problem solving constructs and are not as likely to be confused by objects sending messages.

3.2 Methods for Engaging Students

At Liverpool Hope University several approaches have been adopted to teach the students programming skills. Initially, following the constructivist view of learning, problem solving techniques are applied to every day problems. These include examples such as baking a cake, charging cars to park for periods of time, knitting a jumper and so forth. Although these examples are very basic they engage the student in logical reasoning and algorithmic expression. Illustrations are given for the solutions in the form of conventional flowcharts. The flowchart clearly shows the logical flow and the sequence of instructions. These skills are taught from the start and students are encouraged to present their solutions via structured English. They are taught how the problem consists of many components and how to break the problem down and further refine their solution. This enables them to practise problem solving techniques and express the solution in a fashion that could easily be adapted to any high level programming language. Students are taught the three problem solving constructs of sequence, selection and repetition: they assess the problem and apply the appropriate construct in order to provide a solution. At this stage the students are not coding, they are strictly problem solving and presenting their answer in structured English. Once the students have grasped the understanding of problem solving and are able to express their answer in both structured English and flowchart form, they are then taught how to trace through an algorithm for testing and debugging. They are taught how to record the value of conditional statements and the values of any variables within their solution. At Liverpool Hope University

the students are encouraged to solve problems on paper first, before coding. This attends to the issue of complexity and multiple skills by separating the differing problems associated with finding a workable design from those associated with syntax errors. The students are taught how to convert the structured English into the given programming language that is being taught. Currently this is Java. Students are initially taught how to program in a specially designed environment (Java Trainer) which is easy to learn and use, and which offers the students a visual representation of what their program is doing whilst avoiding the complexity of IDEs.

3.3 Java Trainer

Java Trainer (Appendix A) is an in-house application that was written specifically to teach the students the three problem solving constructs of sequence, selection and repetition, and Java syntax. The trainer is an application implemented in Visual Basic that has limited functionality but has been designed to teach the students how to use loops, selection statements and sequences of instructions to serve the given purpose. Experience has demonstrated that students react better to visual representations of programming than text based solutions. Java Trainer is similar to Pooples which was developed for teaching in London South Bank University (Culwin 2005) but has more flexibility. In light of this, Java Trainer incorporates functions that enable the creation of a scenario for a creature called PieEater. The PieEater will follow basic commands such as walk, turnleft, turnright, eatpie, penup, pendown. The visual representation will show the creature acting out the instructions that have been set by the programmer. The students are given problem solving exercises based on manipulating the behaviour of the creature. An example of an exercise might be to make the creature walk in a 4x4 square tracing a line as it goes. This teaches the students how to use loops effectively rather than the repeated sequence of instructions walk four times, turnright, walk four times, turnright, walk four times, turnright, walk four times and turnright. Predefined variables are also introduced at this stage, one example being clearahead. The variable is a Boolean and demonstrates to the student how variables can store values that relate to the characteristics of the position of the creature and how the variable can change given a certain circumstance. The example variable given above will hold the value, True, whilst the creature is not immediately facing the boundary of the scenario environment. The problems presented

progress incrementally in difficulty, leading to non-trivial problems such as maze navigation (while eating and counting pies).

3.4 The 3-Bit simulator

It is worth noting here that in the parallel strand of the course, namely Computer Systems, the students are introduced to processor operation via an in-house developed interactive environment called 3-Bit. This presents the students with the fundamental features of computer hardware and reinforces the concepts of Problem Solving at low level coding. They learn how the processor fetches and executes machine code instructions. The intention is that in addition to learning about machine architecture, the students can relate the high level programming constructs to actual assembly language routines, and vice versa. The level of success of this is commented on later.

3.5 Object Oriented Concepts

At this stage the students are introduced to object oriented concepts; classes, objects, signatures, parameters, message passing, attributes and protocols. These concepts are investigated further within another environment developed for learning programming, BlueJ™.

Initially the students explore the world of geometric shapes provided by BlueJ™. This builds on the experience of the Java Trainer using familiar code and extending the experience of IDEs and Objects. The students experiment with message passing with parameters of various types in a completely visual context. As with PieEater, feedback regarding success or failure is instantaneous, making any required corrections much easier to make.

The final three weeks of the module focus on programming activities undertaken in pairs using one workstation per pair. The advantage of pairs is that it forces both participants to contribute to the programming activity. The use of one work station encourage coding away from the machine rather than both parties working independently.

As with the PieEater exercises, the tasks set progress incrementally in complexity. The students are "given" the code which creates a blue circle and moves it across the screen. They are encouraged in the "change and see what happens" approach. The tasks start by

requiring students to amend the given code to change the colour and size of the circle, progressing to changing the direction of travel and culminating in simulations of projectiles incorporating the vertical effect of gravity.

When the students, working in pairs, have completed the exercises to create the code, they use presentation techniques to explain the code which underpins their most complex animation. The completed exercises vary in complexity from pair to pair, but the need to explain the code is still the driving force behind the presentation. Not only do the students respond very well to the open-ended nature of the task, they also gain a great deal by having to explain the Java code using object oriented terminology.

A surprising observation at this stage of the course is the positive response by the majority of students, despite the fact that it carries no assessment mark. An atmosphere of positive competition is generated by the students themselves. Another contributing factor may be that the students can "exit" from the exercises at any point that they feel is appropriate. At this point they are allowed to embellish their current animation without adding to the complexity of the underlying code. For example, the ability to create static, geometric shapes and the ability to create a moving circle are the only prerequisites to creating animated scenes, such as a scene with house and apple tree, moving clouds and apple-eating sheep. (Appendix B)

4. Assessment Methods

Part of the assessment used at Liverpool Hope University is based on continuous portfolio work. Assessment is split into two main sections. The first section, Problem Solving, consists of structured English, flowcharts, trace tables and variables. The second section focuses on the coding for a given problem. The portfolios begin at a simple level and progressively become more complex. An example of early portfolio assessment includes adding the contents of variable A to the contents of variable B and putting the answer in variable C, at this stage the students are also expected to convert structured English to flowchart form and vice versa. The students are also assessed on their ability to trace through a program that uses all three programming constructs and recording variable values and the values of conditional statements. The students are not programming at this stage but examining code

in order to determine how it is structured. To assess the first section students are given an in-class test at the end of the first semester, which constitutes 15% of their overall mark. Students who complete the weekly portfolios appear to do well in the test. The portfolios run continuously from the first semester of the course to the second. At the beginning of the second section the portfolios again begin at a simple level in order to introduce the students to syntax: an example might be to make PieEater walk a few steps, turnaround and walk back. The exercises progress to complex maze navigation. This assessment continues to more complex problems, including the creation of programs that simulates a projectile or an animation which incorporates gravitational effects. The second section is also assessed by an in-class test which also contributes 15% of their overall mark. In addition the portfolios are assessed at 10% of their overall mark. The remaining 60% is obtained through assessment in the Computer Systems strand of the course.

An example of portfolio questions can be seen in Appendix E.

4.1 Questionnaire

In order to determine the student perception of this method of teaching, and of how effective they considered it to be, a survey was conducted at the end of academic year 2006-07 of all the first year computing students. The questions asked were a mixture of multiple choice and response directed, and played a key part in evaluating the success of this teaching method. The purpose of the questionnaire was to determine what experience students had of programming on entry, and whether the teaching materials and exercises were considered helpful. They were also asked what they felt had enabled them to understand the course. Finally, they were asked about 3-Bit Processor which is part of the Computer Systems strand. (See Appendix D for questions)

4.2 Analysis

Previous experience of programming

Only a small percentage (15%) of the students surveyed had experience of programming, the most common of which was Visual Basic.

Mathematical qualifications on entry

The majority (85%) of students who responded had GCSE mathematics as their highest qualification over a range of grades down to D. Ten percent had AS level qualifications.

Course booklets and programming exercises

All students who took part in the survey agreed that the in-house text book, and weekly lectures were easy to follow. No difficulties were identified by any student in respect to either the course books or the lectures.

All students agreed that the weekly programming exercises contributed to their understanding of the fundamental problem solving constructs: sequence, selection and repetition. Ninety percent of the students appreciated the relationship to everyday activities and felt that the exercises helped develop a much better understanding of the lectures and of programming. By using the knowledge gained from the lectures and applying it to previously un-encountered situations and problems the learning process was challenging and effective.

Ninety percent of the students agreed that structured English for problem solving proved very useful, whilst 95% considered that the programming exercises developed understanding of how problems were broken down into components.

Java Trainer and PieEater

Seventy-five percent agreed that Java Trainer and the PieEater feature of the program contributed to the learning of the three programming constructs: sequence, selection and repetition. General comments indicated that students appreciated the immediate, graphical feedback from their programming and considered the PieEater environment to be fun.

General comments from students

"Found it very easy to pick up once I had been shown how to use it. Also found it useful the way it picked up on any errors which then enabled us to fix them by ourselves which aided in my learning."

"It was fun and exciting."

“Really enjoyable; I liked just messing around with the software in order to get to know how it worked and thought that when you made an error the information needed and reasons why were provided.”

“It was good how entering Java code would make the PieEater move.”

“Trying to get the Pie (*PieEater)* around the grid to make various shapes or follow certain paths was enjoyable.”

The majority of students (75%) agreed that the experience with Java Trainer helped when introduced to a commercially available programming environment such as BlueJ™.

Only 35% of students felt that there were any benefits in using 3-BIT in relation to understanding programming, however 50% of students said they could easily relate sequence, selection and repetition to 3-BIT operation.

Eighty-five percent of students said that the structured problem solving exercises allowed them to reach a level commensurate with their own ability and 90% considered that problem solving skills taught at Level C would be of benefit in future situations and study.

BlueJ™

Students were very positive in their response to the final BlueJ™ animation exercise and comments included:

“challenging, made the module more interesting and gave me more chance to choose what to base my work on.”

“It was good to use my own imagination and create something that I was interested in rather than following the exercises.”

“The task set was quite challenging so when we completed the final exercise we had to make BlueJ™ do something a bit different, in pairs. The best was then chosen by the class. Although ours was not the best we made the ball change colour a number of times and also move around the screen in a number of directions. After we had achieved this it felt quite satisfying as it was tricky to complete.”

“Building Teamwork skills”

“Given the freedom to choose, design and develop our own project”

“Knowing how to manipulate the code and how to troubleshoot the code. How to solve coding problems ourselves”

“It was a fun program to use, and encourageing”

“It progressed in difficulty at a nice pace, and was all related which meant you could apply what you learnt in the previous exercise to the current exercise”

“You can see the code working, or not, in some cases.”

5. Conclusion

Experience of teaching novice programmers at Liverpool Hope would suggest that although students have low mathematical skills, they can be successfully taught programming after they have had sufficient exposure to structured problem solving. It is imperative that students can identify a problem, break it down into component parts and logically apply a solution to each element before attempting to program.

The use of in-house books has proved very popular with the students and has been a good source of additional reading that reinforces the lectures given. Past experience of the Computing team has shown that many current programming text books are not suitable for an undergraduate degree student because of the assumed knowledge of problem solving. Students speak very positively about the bespoke textbooks provided for their course.

To introduce a programming environment that has been stripped of complicated libraries and other additional features that enable the students to learn the three programming constructs has proven to be a very successful aid in the learning and teaching of students. Many students agree that the introduction to a ‘light’ programming environment enabled them to focus on learning the techniques associated with programming. They like the Java Trainer environment, with its visual feedback in the form of a creature moving around a grid eating pies. This enables the students to learn each of the three programming constructs from very simple problems set, for example, an exercise might be “walk around the grid until you have eaten 3 pies, then return home.” This simple problem teaches them how to apply logical thinking in programming.

It is also apparent from the survey that the students find Java Trainer very useful before being introduced to a more complex programming environment. A programming environment which appears very complicated can act as a deterrent to students who are being introduced to programming for the first time.

With respect to the teaching of 3-Bit only half of the students who took part in the survey feel that previous knowledge of Java Trainer helped them recognise the three problem solving constructs with regard to programming at the machine level. This area of the course can be developed further in order to link the two strands of the module more closely for the student.

The success of the students completing the first year of undergraduate study is partly due to the structured programming exercises set each week. The exercises set become progressively harder. It is apparent from the research conducted that although the exercises are extremely useful the students also need to be encouraged to think creatively. By the end of the course they are demonstrating their ability to solve problems according to the procedures that they have been taught to apply. Their ability to do this is evidenced in the presentation of the final set of exercises where students create an animation using Java and basic shapes (squares, rectangles, circles and so forth). They comment that this was especially enjoyable and that they put more effort into the creation of their own projects than the exercises set. As a result of this research discussion of additional 'free range' exercises will be a consideration when revising the module content and structure.

The questionnaire was sent to students who had completed their first year of University and were due to progress to their second year and approximately half (30) responded. This research has not considered the students who failed, for various reasons, to complete the course. Further research into why students withdraw or stop attending part of the way through their first year would be useful.

We acknowledge all registered trade marks within this paper.

6. References

Anderson P B, Bennedsen J, Brandorff S, Caspersen M E, Mosegaard J, (2003) *Teaching Programming to Liberal Arts Students – a Narrative Media Approach*, ITiCSE'03, June 30 – July 2003, Thessaloniki, Greece Copyright 2003 ACM 1-58113-672-2/03/0006

Beaumont C, & Fox C. (2003) *Learning Programming: Enhancing Quality through Problem-based Learning* LTSN-ICS conference paper, August

Bloom, B S, (1965) *Taxonomy of Educational Objectives* London, Longman

Bennedsen J, Caspersen M E, (2003) *Revealing the Programming Process* SIGCSE'05, February 23–27, 2005, St. Louis, Missouri, USA. Copyright 2005 ACM 1-58113-997-7/05/0002

Burton P J, Bruhn R E (2003) *Teaching Programming in the OOP Era,* ACM SIGCSE Bulletin, Reviewed Paper, Volume 35, Number 2 (June 2003) ACM Press.

Culwin F, Adeboye K, Campbell P. (2005) *POOPLE (Pre-Object Oriented Programming Learning Environment) prototypes.* (unpublished)

Dijkstra, E. W. (1989). *On the Cruelty of Really Teaching Computing Science*, *Comm. ACM* **32**: 1398-1404.

Flynn S (2004) *The NUI, Galway experience* (Workshop: Mathematics for Computing: University of Birmingham, 17-11-2004)

Herterich G E, (2004) One Day Workshop: Mathematics for Computing: 17th November 2004 University of Birmingham

Jenkins T (2002). *On the Difficulty of Learning to Program*. 3rd Annual Conference of the LTSN Centre for Information & Computer Sciences, Loughborough, LTSN-ICS.

Jenkins T and Davy J (2001) *Diversity and Motivation in Introductory Programming, ITALICS* **1**(1).

McCartney (2004) *Computer Scientists and Mathematics* (Workshop: Mathematics for Computing: University of Birmingham, 17-11-2004)

Oldehoeft R R, Roman R V (1977*) Methodology for Teaching Introductory Computer Science,* ACM SIGCSE Bulletin, Proceedings of the seventh SIGCSE technical symposium on Computer Science Education, Volume 9, Issue 1, ACM Press.

Sholtz J, (1993) Object Oriented Programming: the promise and the reality. *Journal of Systems and Software (*November)

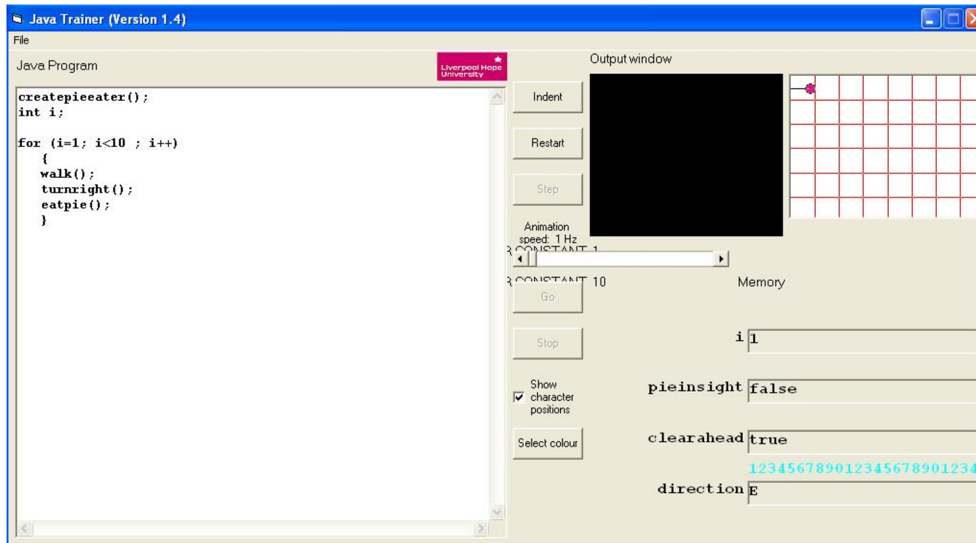
Smith A, (2004) *Making Mathematics Count*, the Report of the Inquiry into Post-14

In-house Software Development (B T Farrimond) available at http://hopelive.hope.ac.uk/imc/Level_c/cc/software.htm Includes Pascal Trainer, Java Trainer and 3-Bit Processor.

7. Appendices

7.1 Appendix A

Java Trainer showing code and grid with PieEater.



7.2 Appendix B

BlueJ™ environment showing graphical image of student work.

7.3 Appendix C

3-Bit Processor

7.4 Appendix D

Student Questionnaire

1. Had you any prior knowledge of programming before you started your undergraduate degree?

If yes, please state which languages.

2. What is your highest qualification in mathematics?

3. Did you find it easy to follow the booklet and the weekly lectures on programming?

If no, can you identify what were the difficulties?

4. Did the programming exercises, set each week, help in your understanding of the fundamentals of problem solving: sequence, selection and repetition?

Any additional comments would be useful

5. Did the way that the programs were related to everyday examples help you to understand programming?

6. Did you find Structured English for problem solving useful?

7. Did the programming exercises develop understanding of how problems are broken down into components?

8. Did Java Trainer and more specifically PieEater software aid in the learning of the three programming constructs: sequence, selection and repetition?

If yes, can you identify what was positive about the programs that enabled you to understand the course?

9. Did you enjoy PieEater?

If yes, could you please list the reasons?

10. Do you feel that having experience with Java Trainer helped when you were introduced to a commercially available programming environment (i.e. BlueJ™)?

11. Do you feel the problem solving skills you learnt at Level C will be of benefit to you in future situations?

12. Were there any benefits in using 3-BIT with regard to understanding programming?

If yes, could you please list the reasons?

13. Could you easily relate sequence, selection and repetition to 3-BIT operation?

14. Did the exercises allow you to reach a level commensurate with your own ability?

15. Identify 3 or more positive features relating to the final BlueJ animation exercise?

7.5 Appendix E

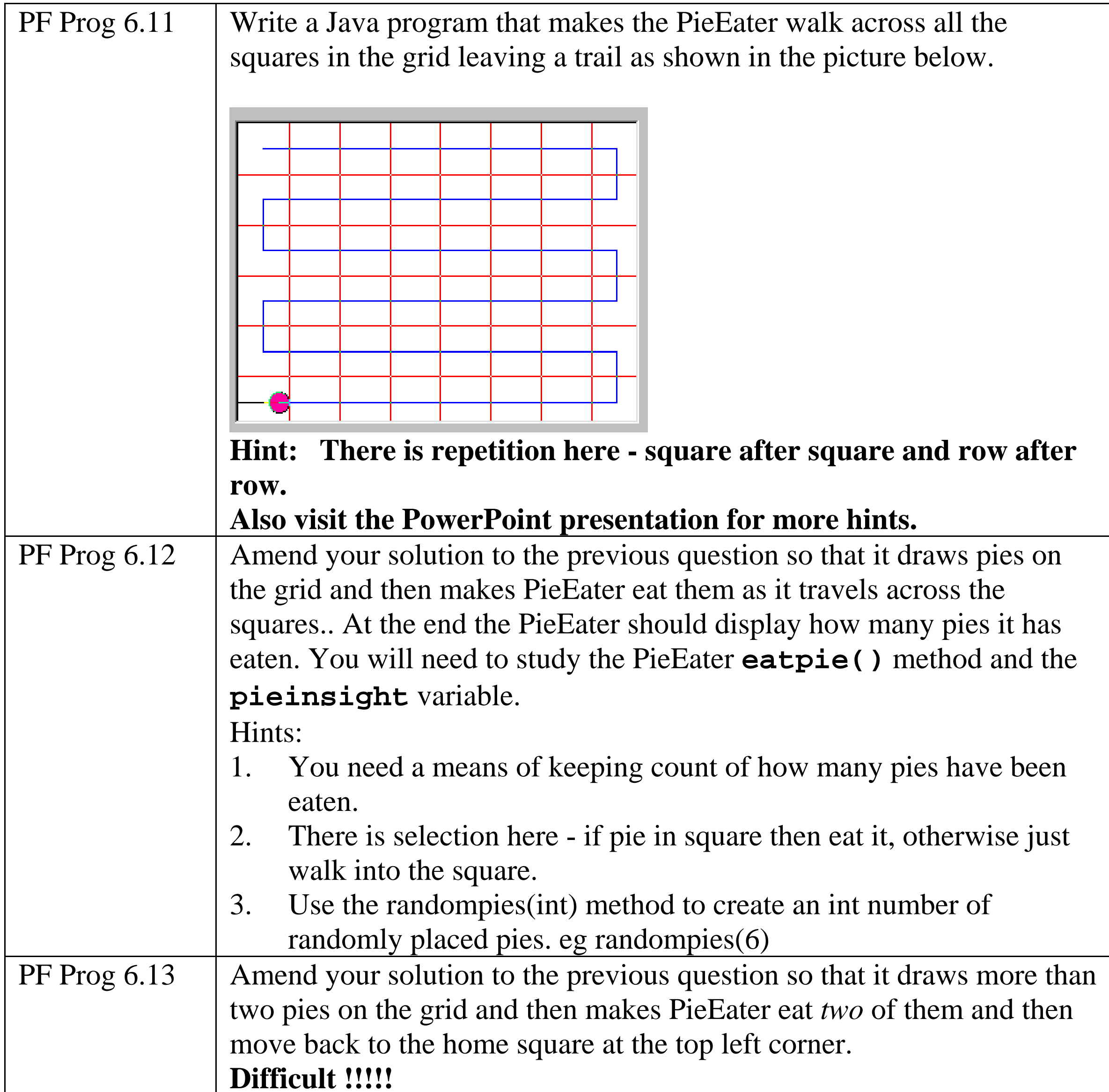
Structured Problem Solving, Java Programming
Portfolio 6, Weeks 10 - 11

PieEater Exercises

In each of the following exercises, the PieEater starts at the home square. For each question:

- write a Java program
- test the program

PF Prog 6.1	Walk across the screen and back
PF Prog 6.2	Learn to Polka: a three quarter turn, walk one step, repeat for 4 steps. (You will need to move to the middle of the "dance floor" first)
PF Prog 6.3	Draw your initials on the grid.
PF Prog 6.4	Draw a square two steps wide ending back on the home square facing east.
PF Prog 6.5	Turn a full circle on the spot at the top left corner of the grid.
PF Prog 6.6	Walk three steps across leaving a trail from the top left corner.
PF Prog 6.7	Walk six steps across leaving a dashed trail.
PF Prog 6.8	Draw a square with side length 4 steps.
PF Prog 6.9	Draw a square with side length entered by the user through a **System.in.read(x);** instruction.
PF Prog 6.10	There is a variable in PieEater world called **clearahead**. Read its description in the table on page 12 of the module booklet. Use it to write a Java program which makes the PieEater walk until he reaches the wall he is currently facing. When he reaches it, have it display the message. *"Got there!"*. The program should work no matter where PieEater starts on the grid.

PF Prog 6.11	Write a Java program that makes the PieEater walk across all the squares in the grid leaving a trail as shown in the picture below.  **Hint: There is repetition here - square after square and row after row.** **Also visit the PowerPoint presentation for more hints.**
PF Prog 6.12	Amend your solution to the previous question so that it draws pies on the grid and then makes PieEater eat them as it travels across the squares.. At the end the PieEater should display how many pies it has eaten. You will need to study the PieEater **`eatpie()`** method and the **`pieinsight`** variable. Hints: 1. You need a means of keeping count of how many pies have been eaten. 2. There is selection here - if pie in square then eat it, otherwise just walk into the square. 3. Use the randompies(int) method to create an int number of randomly placed pies. eg randompies(6)
PF Prog 6.13	Amend your solution to the previous question so that it draws more than two pies on the grid and then makes PieEater eat *two* of them and then move back to the home square at the top left corner. **Difficult !!!!!**