Introductory Problem Solving in Computer Science

David J. Barnes, Sally Fincher, Simon Thompson

Computing Laboratory, University of Kent at Canterbury, Kent, CT2 7NF, England

E-mail: D.J.Barnes@ukc.ac.uk
Fax: (+44) 1227 762811

***ABSTRACT**: This paper describes our experiences in devising a lightweight, informal methodology for problem solving in introductory, university level, computer science. We first describe the original context of the experiment and the background to the methodology. We then give the details of the steps of the Problem Solving Cycle—Understanding, Designing, Writing and Reviewing—and the lessons we learned about our teaching from devising the material. We also present practical examples of how it has been applied in a variety of units in our programme.*
***KEYWORDS**: problem solving, programming, functional, imperative, cross-curricular, software engineering.*

1. Introduction

The original motivation behind this work was to develop material to assist a group of students in the first year of our Computer Science degree who were having difficulties across the course as a whole. We wanted to enable them to take a fresh look at the programming tasks they faced, but without burdening them with a lot of new material to learn. In particular we wanted to help students who said "I just don't know how to start with this task...".

Our approach is based on the seminal *How To Solve It*, (Polya 1957), which has been highly influential in mathematics education. We have adapted Polya's idea to a programming context and also investigated how it fits the imperative and functional programming paradigms. Feedback from the students involved in the programme led us to believe that we had an approach that could be made an integral part of our first-year programme rather than being additional (remedial) material. We have now successfully introduced this approach into a broad range of units in the first year of our degree course.

Our lightweight approach is simple to teach, broadly applicable, and easily related to other everyday tasks that students are asked to perform. We use the approach in a number of courses including Structured (Imperative) Programming, Functional Programming, Software Engineering, and Information Systems. This use across the curriculum also has the benefit of reinforcing links between parts of the programme; such links are often the weakest part of a broad curriculum.

Most mainstream Computer Science courses in the United Kingdom have a strong emphasis on Software Engineering running through them and ours is no exception. Unfortunately, traditional software engineering approaches tend to be rather too heavyweight for an individual student to see how to apply them to a programming exercise requiring fewer than perhaps 100 lines of code. Nevertheless, the fundamental themes we develop here form a foundation for further work in software engineering.

In developing the material we worked with the University Study Centre, and it was interesting to note that the problem solving skills we chose to emphasise were close to those taught in programmes about essay writing by the Study Centre. We therefore feel that the skills we aim to impart are of value beyond the confines of the Computer Science curriculum.

In this paper we describe the original context of the experiment; the background to the methodology; the details of the steps of the Problem Solving Cycle—Understanding, Designing, Writing and Reviewing—and the lessons we learned about our teaching from devising the material. We present practical examples of how it has been applied in a variety of units in our programme.

2. Background

Around the middle of the 1995–96 academic year, we were aware that there were a significant number of our first-year mainstream Computer Science undergraduate students who were struggling with basic programming assignments. They were committed to the programme, and were achieving reasonable grades in other courses, but were lacking some of the basic problem solving skills that would be necessary for progression into the second year. A remark that characterized many in this position would be, "I don't know where to start"; they had teaching materials and course texts but each new assignment left them floundering over how to begin to solve it.

To help these students we put together a package directly aimed at enhancing their confidence and competence in problem solving skills*. At the outset, as we were involving a specifically remedial section of the cohort, we took care to present them with both cognitive and meta-cognitive material. That is to say, in the cognitive domain, we provided tools and methodologies which they could immediately apply to the immediate assignments/revisions they were facing: additionally we supplied meta-cognitive material which described in a more general way what the activity of problem solving might be and what it might feel like to have it. This section of the material was constructed from appropriate quotations appended with a series of reflective questions, (Fincher 1996). By presenting the two sorts of material in concert we hoped to increase the effectiveness of the learning. The remainder of the paper describes the construction and use of the cognitive material.

3. The Problem Solving Cycle

Polya's *How To Solve It*, contains a wealth of material about how to approach mathematical problems of various kinds. This ranges from specific hints which can be used in particular circumstances to general methodological ideas. The latter are summarised in a table which gives a four-stage process (or more strictly a cycle) for solving problems. In helping students to program, we have specified a similar summary of method —*How To Program It,*

* At the same time the University Study Centre, who are responsible for generic learning support in the form of workshops on study skills, were seeking to develop materials which were both more subject-specific yet which could be transferred from one discipline to another. Problem solving seemed an appropriate area in which to work, and as a consequence we were able to work with staff from the University Study Centre in developing our support materials. In particular, we are very grateful to Jan Sellers of the University Study Centre at UKC for her assistance and co-operation in developing our problem solving materials.

(Thompson 1996a). The table is largely self-explanatory, and is structured in four phases: Understand, Design, Write and Review.

In **Understanding**, we encourage the students to really shake each problem description up and pull it apart; to ask questions about it and be sure that they fully comprehend its scope. We get them to ask, "What if?" questions and to challenge aspects that are unclear, (Barnes 1997). The collection of sample inputs and expected outputs not only helps to clarify their thinking, but provides the beginnings of a test data set that can be used later in the process.

In **Design**, the pedagogic emphasis is on looking around to find related problems that they have already solved, or similar problems with which they might already be familiar. This fits well with our style of teaching which uses assignments to reinforce current course material through practice, and forces staff to think carefully about the design of examples in notes and course materials, in order to ensure that the assignments achieve their maximum learning impact. Here they can use their input/output samples to check for completeness and consistency in the design.

The **Writing** stage involves turning the design into a working program. It is here that there is necessarily a focus on the target implementation language, whereas earlier stages can be largely language independent. Indeed, in the pilot, we were teaching the approach in the context of courses on both functional and imperative programming languages. If simplified versions of the problem have been the focus at the design stage, these can be carried through to implementation and serve as building blocks, or outline solutions, for a larger problem the value of gaining confidence through achieving at least some result should not be minimized. It could be argued that such an approach to implementation seems rather piecemeal or even hacky, and flies in the face of something like the discipline of top down stepwise refinement, (Wirth 1971), that we ought to be using. In practice, however, techniques such as stepwise refinement are quite difficult to perform if you don't really know what the solution should be, and students find them hard to apply to small scale exercises.

The final stage is **Review** a time of reflection and looking back on the finished product once the deadline is passed. The aim here is to consolidate the learning process and appreciate lessons learned. It should also provide a means to build up the store of experience that is applied in future Design stages, as described above. As we prepared this course, it became clear to us that this was an area which staff typically make difficult for students. Students are assessment driven moving from one assignment to the next and staff leave them little time to reflect on the work just completed. In addition, by the time the work has been marked and returned, its detail has been forgotten (by them) and some of its learning value lost.

4. An Example in Haskell

As we saw in the previous section, it is more difficult to give useful language independent advice about how to write programs than it is about how to design them. It is also easier to understand the generalities of *How To Program It* in the context of particular examples. We therefore provide students with particular language specific advice in tabular form. These tables allow us to

- give examples to illustrate the design and programming stages of the process, and
- discuss the programming process in a much more specific way.

The full text of *Programming it in Haskell* is available on the World Wide Web, (Thompson

1996b). In the rest of this section we look at one of the smaller examples and the points in the process which they illustrate.

Problem: find the maximum of three integers.

A first example is to find the maximum of three integers. Even in a problem of this simplicity there can be some discussion of the specification: what is to be done in the case when two (or three) of the integers are maximal? This is usually resolved by saying that the common value should be returned, but the important learning point here is that the discussion takes place. Also one can state the name and type, beginning the solution:

```
maxThree :: Int -> Int -> Int -> Int
```

More interesting points can be made in the design stage. Looking for related problems, we might think of the function `max` to find the maximum of two integers,

```
max :: Int -> Int -> Int
max a b
  | a>=b          = a
  | otherwise     = b
```

this can be used in two ways. It can form a model for the solution of the problem:

```
maxThree a b c
  | a>=b && a>=c      = a
  |     ....
```

or it can itself be used in a solution

```
maxThree a b c = max (max a b) c
```

It is almost universally the case that novices produce the first solution rather than the second, so this provides a useful first lesson in the existence of design choices, guided by the resources available (in this case the function `max`). Although it is difficult to interpret exactly why this is the case, it can be taken as an indication that novice students find it more natural to tackle a problem in a single step, rather than stepping back from the problem and looking at it more strategically. This lends support to introducing these problem solving ideas explicitly, rather than hoping that they will be absorbed by osmosis. More details of using the problem solving approach for teaching functional programming can be found in (Thompson 1997).

5. Impact of the Course

The Introductory Problem Solving course was given in 3 sessions in the Summer term of 1996, with approximately 20 students invited to attend one of two classes each week. Printed course materials covering the cycle and the meta-cognitive aspects of problem solving were distributed and discussed, but the main emphasis was on applying the cognitive methodology to a number of small problems. Consequently, in subsequent iterations, the material supplementary to this aim (whilst remaining available) is not specifically included in the teaching.

The original course was well received by those who chose to attend and its effectiveness was indicated by the fact that a number of those who had been struggling succeeding in passing their examinations at the end of June. However, the impact on our own thinking was that we felt their was a need to teach such skills as introductory, rather than remedial. As a result, we made significant changes to four of our core first year courses in the following academic year, introducing the cycle into the units *Imperative Programming*, *Functional Programming*, *Software Engineering* and *Information Systems*. As a particular example, the lack of opportunity for Review led us to bring the discipline of keeping an informal, but assessed, personal log book into these courses; requiring the students to recognise (and, it is to be hoped, learn from) their own reflective processes in this domain. This has been very successful to date with considerable positive feedback from students on their value.

References

Barnes, David J. (1997) 'Students Asking Questions: Facilitating Questioning Aids Understanding and Enhances Software Engineering Skills',
http://www.cs.ukc.ac.uk/people/staff/djb/papers/asking.html

Fincher, Sally (1996) 'Problem Solving in General',
http://www.ukc.ac.uk/computer_science/Html/Courses/PSinGeneral.html

Polya, George (1957) 'How To Solve It', Princeton University Press, 2nd Edition

Thompson, Simon (1997) 'Where do I begin? A problem solving approach in teaching functional programming', in First International Conference on Declarative Programming Languages in Education, Krzysztof Apt, Pieter Hartel, Paul Klint (editors) Springer-Verlag

Thompson, Simon (1996a) 'How To Program It',
http://www.ukc.ac.uk/computer_science/Html/Courses/HowToProgIt.html

Thompson, Simon (1996b) 'Programming it in Haskell',
http://www.ukc.ac.uk/computer_science/Html/Courses/ProgInHaskell.html

Wirth, Niklaus (1971) 'Program development by stepwise refinement', Comm. ACM 14:4, p221-227