Abstract

This paper describes an approach to the teaching of introductory programming to students with no previous experience of programming. The approach is based on the key idea of making students active participants in classes rather than passive recipients as might be the case when they are exposed to more traditional lecture-based teaching. In these classes the students are called upon to adopt the rôles of various components in a program and to “act out” relevant processes and procedures.

The paper considers why such an approach might be effective and why it might be expected to be particularly effective in the area of introductory programming.

The techniques described were developed and tested during the 1997/98 academic session in the School of Computer Studies at the University of Leeds. At the end of the presentation a brief evaluation was carried out.

In conclusion the effectiveness of the approach after a semester’s experience is considered and planned future developments are described.

Keywords

Programming, participation, interaction, induction, reflection

1. Introduction

1.1. The Problem

The School of Computer Studies at the University of Leeds teaches introductory computer programming to about four hundred students every year. Of these, about two hundred and fifty are enrolled on mainstream computing degree courses; approximately half of these start the course with no previous experience of programming. The teaching of programming to this particular group is a perennial problem; most students have great difficulty in achieving a reasonable standard and every session a significant proportion fail. In many cases failure in the programming module is sufficient for a student to be required to withdraw from the course and leave the University. This is clearly not a satisfactory situation for either the students or the School.

Efforts in recent sessions have focussed on using technology to implement effective support for this group [11]. These efforts have not been successful in addressing the underlying problems, but our experiences have given us a valuable insight into the ways in which these students attempt to learn. This insight provided the motivation for a further attempt at innovation. Moreover, other efforts at Leeds had focussed on the issues surrounding motivation in new students [1]; while this work had been in a different subject area (although still within computing) there were some ideas, notably relating to the style and format of presentation that seemed to show promise for application to programming.

This paper describes attempts made in the 1997/98 session to change the manner in which basic programming concepts are delivered. The key concept in these changes was participation. The aim was to change the students from passive recipients of the teaching into active participants in a learning process. This required dramatic changes
in the way in which the material was presented, and also changes in the attitudes of
the staff presenting the material and, crucially, in the attitudes of the students
themselves.

1.2. Participation

There is evidence that a participative approach may be effective in higher education.
For example, in a computing subject area, Fleury [6] shows how complicated
recursive algorithms can be presented effectively to students using an approach
whereby the students themselves enact the processes involved. Gibbs et al [8]
describe a class in chemistry where “members of an audience [are] assigned the
roles of different atoms, and a DNA molecule [is] constructed by linking their arms
around the room”. Other work in the field of social psychology [16] has shown that
participation and the resulting novelty had meant that similar techniques were
“likely to help [the students] remember the material”.

The basis of the proposed change in the teaching of programming was to take these
ideas and attempt to devise ways of demonstrating basic programming concepts
through having the students act them out. It seemed that such an approach might be
effective in many very basic areas of programming such as loops, parameter passing
to sub-programs as well as in more complicated applications such as pointers and
complex data structures.

The following sections begin by considering why such a change might be expected
to work, or at least might be expected to be an improvement on a more traditional
lecture-based presentation. Then, some sample activities are described. One of these
was the subject of an evaluation to determine whether the students themselves saw
any improvement in their knowledge of a subject area, and whether they found the
classes preferable to more traditional lectures. Finally, some conclusions may be
drawn as to the effectiveness of these changes, and to possible further developments.

2. Learning Through Doing

2.1. The Status Quo

At Leeds, programming is taught in a traditional lecture module. There are three
lectures a week, and the students are given a practical assignment to complete every
week. The lectures of necessity focus on the syntax and semantics of the
programming language, and can pay little attention to the actual process of applying
knowledge of these in the development of a program. It is expected that the students
will acquire this knowledge through the practical work.

As long ago as 1938 Dewey was writing that “all genuine education comes about
through experience” [5]. He argued that education should be based around activities
that are “lively, vivid, and interesting” rather than the then usual “automatic drill”.
It was only too easy to see how the traditional lectures previously used to teach
programming fell more into the latter category. More recent work by Friere [7] and
Schön [22] has focussed on the advantages of “learning through doing” and
becoming what Schön calls a reflective practitioner basing learning on reflection on
experience. More recently, Laurillard writes that “action as an aspect of learning is not in dispute” [14].

Obviously, the traditional lectures used at Leeds were not promoting this approach. They were providing the basis on which the students were expected to reflect. The actual learning would come from the practical exercises. Results over a number of years indicated that this was not happening as well as would have been desirable.

2.2. How Students Learn to Program

To any experienced programmer, it is obvious that the best (and perhaps the only) way to learn programming is through practice. Race [18] lists four key ways in which people learn effectively:

→ practice;
→ by doing it;
→ by trial and error;
→ by getting it wrong at first and learning from mistakes.

The relevance of these to programming is clear. However, experience at Leeds has shown that many of the students were not acquiring sufficient basic knowledge (from the lectures) to allow them to get to the stage of “making mistakes” in a practical setting. The purpose of change was to get them to this stage at least.

Ripples on a Pond

More recently [19] Race has proposed a model of the learning process based on four key activities interacting as “ripples on a pond”, as shown in the diagram. This model refines the idea of learning as a cyclical process (learning cycles are described in [13]) and suggests that learning takes place when four inter-related processes take place simultaneously. It is an especially appealing idea for a practical topic such as programming.

In the model, the desire to learn (“wanting”) is the driving force to the process. This desire starts the “ripple” moving across the pond, through some “doing” and “digesting” of the outcomes. “Feedback” is provided, and sends the ripple back in to the pond.

To make the analogy specific to the learning of programming, we must assume that the students “want” to learn to program. We would move on with the “doing” task as being the undertaking of one or more practical programming tasks. It is to be hoped that this task would promote some reflection (“digesting”) and then feedback (the most obvious
form of feedback is the result of any assessment). The resulting “ripple” would then produce more “wanting” and “doing”. This in turn would hopefully result in the practical experience needed for effective learning of programming.

It was clear that a lecture-based approach was not promoting this process within the students. There was a clear need for something novel that would “get the ripple moving” through the pond.

2.3. Deep and Surface Learning

Traditional lectures are not an effective way to teach the process of programming to novices [11]. They are reasonably effective for conveying basic ideas of syntax of a programming language to these students, but our experience had shown that knowledge of the syntax (and indeed the semantics) was quite useless without a knowledge of the process for using the language.

With most students traditional lectures promote a surface approach to learning [9] which might be effective in some disciplines or subjects but which is inadequate in such a practical subject where a deep learning approach is essential. (For a description of deep, surface and other learning styles see [15] and [21]). This problem is exacerbated by the rise of the tactical or strategic student [12], whose sole aim is to pass the course. Such students are not interested in learning to program; they are interested only in obtaining the grades available for completing the coursework assignments. Experience has shown that this attitude is common among novice programmers at Leeds.

2.4. Teaching and Learning Approaches

It is clear that a deep learning approach is needed when a student is learning to program. The instructor must therefore devise some way of ensuring that this approach is the only one that can be profitably adopted, even by the most strategic of students. In [17] Postman and Weingartner quote Marshall McLuhan’s aphorism, “The medium is the message”. They go on to state that “the most important impressions made on a human nervous system come from the character and structure of the environment within which the nervous system functions”. In our present situation we might take this to mean that the factors which have the most influence on how well a student learns to program are those relating to the environment in which the student is taught. The content exists independently of this environment, and it is the environment that is important.

The idea that students form a view of the nature of knowledge from the instructor’s approach to teaching it is confirmed in [4] and [23]. If the teacher sees the course content as a set of facts that must be conveyed, memorised, and repeated the students will treat it as such. Of course, if the teacher takes this approach, the most profitable learning style in any assessment is likely to be a surface one, focusing on the memorising of the facts. On the other hand, if the teacher views the material as an overall set of concepts, and places emphasis on this rather than the bare facts, the students will be coerced into a more deep approach.

Relating this specifically to programming, we can see that an emphasis on the basics of a programming language will promote a surface learning approach. Students will
be able to describe the syntax and semantics of a given programming language construction, but will be completely at a loss when faced with a practical task using the same construct. On the other hand, if programming is presented as a “skill” which requires certain tools (the programming language), a deeper approach may well be fostered.

In summary, traditional lectures will at best be useful in conveying basic syntax and semantics; they cannot present the process whereby experienced programmers apply them. This means that the novice student will have notes on the syntax and semantics of the programming language but they will not have the necessary “deep” knowledge to apply them to problems. The required knowledge of process would be acquired through learning by doing [22], an approach that might be encouraged by a more participative presentation style.

3. A Participative Approach

This section considers the design of a set of classes that were intended to present programming concepts in a participative manner. The subsequent sections describe some topics where a participative approach has been used. These topics are those which previous feedback had shown many students find especially difficult. The section concludes with some general observations about the running of the classes.

3.1. Background

The introductory programming language currently used at Leeds is Pascal, and the classes described here were devised with this language in mind. There seems no reason, however, why classes such as these could not be presented in connection with any other similar procedural language. Indeed, from 1998/99 the introductory language will be C++ and it is anticipated that these classes will be presented with minimal changes.

The classes were presented by two people in a traditional raked lecture theatre to groups of about fifty students. This setting is perhaps not ideal; it would have been preferable to be able to use a more flexible space that might better promote “active” learning. Nevertheless, the theatre proved to be adequate and not overly constraining.

Initially, the classes were offered as additional sessions for students who had had serious problems with programming in the previous semester. It seems sensible to assume, therefore, that the audience consisted of students who, while they had had difficulties with programming were at least highly motivated to learn.

3.2. Participation is Difficult

Even when there seems to be reason to believe that a participative approach will be effective, it is no easy matter to make an audience participate in a lecture. Most lecturers will be familiar with the rows of blank faces that generally greet questions such as “Are there any questions?” or “Do you all understand that?” Gibbs et al [8] suggest a number of reasons for this including a fear on the part of the students of “being considered stupid, attention seekers or creeps”. Race also observes that
“students can feel surprisingly lonely in large group lectures”, and are reluctant to contribute.

For the teacher of introductory programming this silence can be highly exasperating. Experience shows that the majority of the students have little understanding of the material just presented, but that they are unwilling to expose their ignorance. A simple experiment had proved this, but the result was even more exasperating. After the usual (and, to be honest, expected) lack of reaction to the question “Does anyone not understand that?”, the question was repeated, but only after all the students were told to close their eyes. When they knew that their peers could not see them, over three quarters of the class suddenly decided that they did not understand at all!

3.3. Designing the Class Environment

With these issues in mind, it was clear that the classes devised should be, as far as possible:

- **Informal** – there should be as few barriers (physical or mental) between the instructors and the students;
- **Dynamic** – the classes would adapt as they progressed and as more was understood about the students’ level of knowledge of a particular topic;
- **Personal** – there would be no “them” and “us” from either point of view. This would involve, for example, making efforts to learn as many of the students’ names as possible, and attempting to eliminate any feeling of a “front” of the lecture room;
- **Entertaining** – the students had to want to come to the classes;
- **Memorable** – a key idea of the classes was to encourage the students to reflect. To this end, the classes were designed to be memorable so that, even if the students did not understand the content of the class, they remembered the class itself. At a later time, hopefully, they would appreciate the relevance;
- **Fun** – education and learning is supposed to be fun, as is teaching. If the classes were not enjoyable for the students or the instructors there would be no point in either attending.

Of course, taken together, these could be seen as a recipe for anarchy. To quote Bruner [2] “somewhere between apathy and wild excitement there is an optimum level of aroused attention that is ideal for classroom activity”. It was important that the participative element of a class was suitably debriefed to allow for appropriate reflection.

It was decided that each class would follow the same basic outline. It would start with a brief but reasonably formal explanation of the topic to be covered to ensure that at least some basics were generally understood. Then there would be an activity that would seek to illustrate the concepts of the topic in an “interesting” participative way. This activity would be recorded using reasonably formal structured English. To conclude, this activity would be debriefed using pseudo-code (derived from the structured English) and finally actual working Pascal code.
3.4. Cooperation and Openness

It was decided to explain the reasons for running the new classes to the students from the outset. The aim was to use a model of learning based on a cooperative or conversational approach similar to that described by Laurillard [14]; the content of the class (and the plans for future classes) would be changed dynamically as the instructors learned more about the students’ understanding. It seems sensible to suppose that an approach such as this will work better when the students are aware of it, and, importantly, the reasons for adopting it.

There is debate [3] as to whether explicitly teaching students about learning styles (“deep” and “surface” learning) is a good idea. For these classes, it seemed sensible to cover this at first so that, hopefully, the students would see that they would need to alter their approaches for this particular subject area, and might even start to reflect on their own learning style.

Some studies from the 1970s have shown that students tend to prefer lectures where the emphasis is firmly on the process of “information, storage, and retrieval” [25]. While other work reported in [10] suggests that this might not be the case with more recent generations of students, (and Race suggests that students are more likely to attend participative sessions since “time passes more quickly” and “it is harder to catch up” [20]) other experience at Leeds indicated that it certainly was a dominant view among our present students. Final year students had described attempts by a colleague to introduce interaction into lectures as “unfair” and akin to “terrorism”.

There were also several cases of students dropping modules presented in a participative style in favour of “fact retention” modules. There was clearly a barrier to overcome.

It was hoped that a frank discussion of why the material would be presented in a particular way might win over the students (cynically, of course, it could also be argued that it would keep those unwilling to participate away). Overall, it is also to be hoped that this openness would start to foster the necessary informal atmosphere needed for the classes to succeed.

4. Topics and Classes

4.1. A Simple Example - Variables and Values

Novice programmers often have difficulty with the most basic concepts\(^1\). This presents a particular difficulty as these concepts are often taken for granted by an instructor, and can tend to be glossed over. In any case, many instructors will find them very difficult to explain adequately as they are so basic they are rarely thought about. One such area is that of variables, values and identifiers.

---

\(^1\)Another example is the concept of a “file” on a computer system. How is it possible to explain what a file is without using some analogy involving books in a library or similar? Why does an experienced computer user have an intuitive model of what a file is, but a novice user does not, and seems to “need” to know precisely what a file is?
When teaching Pascal, we describe the following as the “most difficult statement in the language”:

```pascal
    aValue := aValue + 1;
```

Novices have difficult with this statement because it is “obvious” that `aValue` cannot equal `aValue + 1`! Until they have an appreciation of the overall process of programming the statement simply does not make sense. On the other hand, an experienced programmer (or an instructor) would find this statement so natural that they would probably not stop to think about it, and would probably find difficulty in explaining it adequately to a novice.

Similarly, consider the following fragment of Pascal:

```pascal
var
    anInt: integer;
    aChar: char;
begin
    anInt := 0;
    '3' := aChar;
    anInt := aChar
end.
```

To an experienced programmer the errors in this code are obvious⁵; such a programmer would probably never make such errors and will find it difficult to remember a time when they would have done. To a novice programmer, however, the intended meaning of the code is equally “obvious”, and the fact that there are errors is mysterious.

**Declaration**

The first class is designed to illustrate the basic ideas in use here. The underlying idea is to select students to “play the parts” of variables of certain data types. They are assigned values by the instructor, and simple “programs” can be devised.

The class begins with the declaration of three variables. The identifiers of the variables are taken from the names of the “volunteer” students. The declarations are displayed to the class using a form of structured English:

```
    Zoe can remember one integer.
    Phil can remember a single character.
    Ajay can remember TRUE or FALSE.
```

**Initialisation**

With these declarations in place, the immediate first step is to ask each student what value they are currently remembering. Of course, they are remembering none, and

---

¹ The second assignment statement is reversed, and there is a type mis-match in the third statement. Pascal does not allow the assignment of characters to integers.
this neatly illustrates the need to initialise variables. The variables are now assigned suitable initial values, again in structured English:

Zoe is told to remember the number 0.
Phil is told to remember the letter 'a'.
Ajay is told to remember the value TRUE.

Ideally, the students with numeric or character variables display their values on large sheets provided for the purpose. A student with a boolean variable can display their value by wearing a baseball cap, or not, as appropriate.

Application

Data type checking can be illustrated by moving values round the variables. For example, with the present declarations, it is clearly nonsense to ask Ajay to remember an integer (he has no sheet to write it on), or Zoe to remember a boolean value (she has no baseball cap).

The incrementing of a variable appears much more natural in this setting:

Zoe is told to add one to the number she is remembering.

This operation can be illustrated to the class so that they appreciate the concept before the instructor returns to the Pascal, by way of some pseudo-code.

Extensions

If this class goes well, it can be extended to include a brief consideration of conditional statements. These are trivial to express using the variables declared; for example:

if Zoe is remembering a number greater than 4 and
   Ajay has hit hat on then
       Phil should be told to start remembering 'D'

With careful choice of variable names, this can be made to be remarkably close to some actual Pascal code:

if (ZoesNumber > 4) and AjaysHatOn then
   Phil := 'D'

These ideas can also be used to present classes on loops or more complex control structures. The techniques, though, can also be used to illustrate data structures that the students will meet later in the course.

4.2. Arrays

Students always seem to find arrays difficult. This is perhaps not surprising as many texts describe them as “variables that can hold multiple values”, a rather difficult concept to grasp to say the least!

The key idea when presenting this topic is to use a group of the students to perform the functions of an array with each student taking the part of a single element. Once the array has been set up the instructor can demonstrate some common applications and operations using arrays.
Ideally, the “array” would consist of the entire class, but this is seldom feasible because of numbers. In order to maximise participation, therefore, the array is established over as wide an area of the lecture room as possible. If necessary, a group of students may take the part of a single element. It could be argued, in fact, that this is a preferable situation as it might well serve to encourage discussion.

The array is set up (or “declared”) by selecting a “volunteer” to take the part of the first element. They are equipped with a large sheet of paper bearing their “array index” with space for them to record the contents of their element. They then select the next element in the array (a student who is not seated nearby), and the process repeats until the required number of elements is reached. The instructor then explains that each member of the array can remember only one value, but that the array as a whole can remember many values (this should be immediately apparent after the class on simple variables). It is then important to show how some operations can address the array as a whole, while others require each element to be used in turn.

Initialisation

The first step with the array set up is for the instructor to “initialise” the array by telling each student in turn to record some initial value. The other students will observe the necessary “loop” procedure. The process is first expressed in structured English:

Go to each element in the array in turn and set its value to zero.

This expresses the necessary loop informally. The instructor can then show that, in order for the initialisation to work correctly, it is necessary to know how many elements there are in the array, and where the first one is to be found. This leads to a refinement of the process to some pseudo-code:

Go to the first element of this array.
while there are elements left in the array
  Set the current element to zero
  Go to the next element

This code can then easily be translated into a Pascal loop as required.

Application

Once initialised, the array can be used to demonstrate various common operations. Examples might include computing the average of an array of numbers, or finding the highest and lowest values. A number of these, backed with pseudo-code and then Pascal serve to illustrate the concepts.

A feature peculiar to Pascal is the use of “packed arrays” to store strings of characters. This can easily be demonstrated by “packing” an array of students into the same row of the lecture room so that their “values” spell out some suitable string.
4.3. Procedures and Parameters

A particular problem area for novice programmers is the use of procedure (or function) parameters and in particular the difference between variable parameters (that are returned from a sub-program) and value parameters (that are solely supplied as input).

As before, the basic idea in this class session is to make “volunteers” from among the students play the parts of the variables and sub-programs. Using the same large sheets to store “values”, the instructor can graphically demonstrate how values and variables are passed between program modules.

Initialisation

To set up this exercise, a student (or small group of students) is chosen to play the part of a procedure. They are given a description of an algorithm (in English) which only they can see. This student is then asked how many values (parameters) are required to complete the task, and is asked to specify data types for these. The instructor then selects the appropriate number of other students to portray the required number of parameters. These variables are given their data types and are provided with some suitable initial values.

A “main program” is then devised in pseudo-code to allow the procedure to be tested. An example might be:

```
Variables
  Lydia : integer
  Elliot : integer

Begin
  Lydia is assigned 0
  Elliot is assigned 0

  Call Colette&Emma (Lydia, Elliot)

  Display the final values of Lydia and Elliot)
End.
```

The main program is controlled by the instructor, and a visual prop (such as a hat) is used to show how control is passing from the main program to the sub-program and back again.

Application

The two types of parameters are initially treated separately. At first the “procedure” has only value parameters; the “parameter” students can convey their values by shouting them to the “procedure” in the correct order (it is useful to get this wrong a few times to show the chaos that ensues). The final result is output when the instructor is told the answer and displays it. Again, the process is then expressed more formally in pseudo-code or Pascal and is repeated. Some further entertainment can be had by deducing what the procedure is doing!
The exercise is then extended to cover variable parameters. Again, a “volunteer” is provided with the outline of the procedure. Others are then given large sheets of paper to represent variables. To simulate the “passing” of a parameter, these sheets are passed across the class to the procedure, who writes the final values on the sheets, and “passes” them back across the class. An entertaining development of this idea proved to be the use of frisbees to replace the pieces of paper; this certainly serves to produce the required memorable experience!

**Extensions**

If time allows and the class goes well, more complicated parameter data types can be used. For example, an array may be declared and “passed” to a subprogram by having one student collect all the array elements and pass them over. This graphically shows the advantages of passing parameters as arrays rather than as individual values.

### 4.4. Pointers

The use of pointers to develop programs using linked list data structures is the most advanced part of the introductory programming course at Leeds. It is, not surprisingly, an area which causes the novice programmers a great deal of difficulty.

The key difficulty is to explain the difference between the pointer’s address and the value “pointed to” by the pointer. Our first class addressed this. From this point, we were able to build in a later class to a discussion of linked lists.

**Initialisation**

To explain basic pointer ideas, a few “volunteer” students are chosen to represent the pointers. They are asked to remember a value. At the same time, another student is appointed “memory manager” to record the name of the first student and their location in the theatre (we use the location to mimic the idea of a memory address). From this set-up the class can explore the various possibilities for accessing the values, and can consider which do and do not make sense.

For example, the memory manager may record:

<table>
<thead>
<tr>
<th>Pointer Name</th>
<th>Location (Address)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Colin</td>
<td>Front Row, three seats from the left</td>
</tr>
<tr>
<td>Neil</td>
<td>Fifth Row, in the middle</td>
</tr>
<tr>
<td>Kate</td>
<td>Back Row, far right</td>
</tr>
</tbody>
</table>

This gives the class three pointers and access to them by means of their addresses. The only way to find the value “pointed to” by the pointer is to access the pointer by means of its address.

The Pascal syntax for a pointer gives the opportunity for the use of another prop. A pointer \( P \) is dereferenced with the syntax \( P^* \), which we would read “\( P \) hat”. The students playing the parts of the pointers are provided with baseball caps and are asked to record their values under their hats.
Application

The instructor now announces that the pointers must store some values. This is done by means of a dialogue with the “memory manager”, as follows:

**Instructor:** Right, I want to store a value at Kate’s location. Where’s that?

**Memory Manager:** On the back row, right hand side, by the door.

**Instructor:** Ah. I see her. Kate, store this value for me, will you?

**Kate:** Ok.

The value itself is passed written on a frisbee. This process clearly shows how the programmer (in the guise of the instructor) accesses a pointer’s memory by means of their address. A similar procedure can be used to access the value stored at a given address.

It is important that the students understand that two pointers can point to the same memory address. This can be shown by making two students occupy the same seat in the theatre. The instructor might declare some pointers:

<table>
<thead>
<tr>
<th>Pointer Name</th>
<th>Location (Address)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Colin</td>
<td>Front Row, three seats from the left</td>
</tr>
<tr>
<td>Jane</td>
<td>Back row, far right</td>
</tr>
<tr>
<td>Kate</td>
<td>Back Row, far right</td>
</tr>
</tbody>
</table>

A value is then stored using one of the students, and then the same value can be retrieved using the other.

**Extensions**

These ideas can then be extended to include linked lists (or similar data structures). In this case it is important to demonstrate the creation of the data structure and the procedures for the various operations performed on it.

A simple linked list consists of a pointer to the first element. Each element points the way to the next element in the list with a pointer, and the list is terminated with a null pointer. Our approach here is for one instructor to play the part of the first pointer in the list. The instructor selects a “volunteer” to be the first element in the list, writes down their name and location, and leaves the room. This graphically shows that the address of the first element in the list is the only information held. The second instructor then establishes the rest of the list through a series of “volunteers”, with each student asked to remember the name and location of the next person in the list. The final student is asked to remember only that they are the end of the list. It has also proved helpful for the links to be made physical with wool or string.

With the list established, the first instructor can return and operations on the list can be demonstrated. For example, the list can be traversed to seek for a certain value. The code for this can then be derived and examined. A particularly useful
demonstration relates to the procedure for deleting an element on the list where a third “pointer” is needed.

This is an area where this approach works particularly well. Pointers and linked lists are a very abstract concept, and it has proved invaluable to be able to give the students a more concrete set of images to form the basis of an understanding.

4.5. Observations

Since we were using a novel approach it was necessary for the instructors to learn as well as the students. The first thing that became apparent was that, as expected, it was extremely difficult to make the students join in. This is perhaps understandable; we had seen before [11] that a major problem we were facing was students’ unwillingness to expose their own ignorance. We were helped in addressing this by the relatively small size of the class, but some “ice breaking” activities were necessary.

On a similar issue, an effort by the instructors to learn the preferred names of all the students attending proved effective in promoting interaction. Indeed, in later classes we devised ways of stealthily recording unfamiliar students’ names during the classes themselves and of referring to them by name whenever possible. We might also hope that this “personalising” of the process would promote self-help activities among the students outside the classes.

The selection of “volunteers” proved to be crucial. The obvious approach is simply for the instructor to ask for volunteers but we supposed that this would lead to a small number of students monopolising the classes to the disadvantage of others. Since the idea of participation for all was key we decided that we needed a more devious tactic. Remarkably, the simple step of offering sweets to students who joined in seemed to be most effective. This was a small investment that greatly enhanced participation from all, and provides a promising light-hearted atmosphere for the class.

The classes were presented by two (and at times three) instructors. This is crucial. The amount of activity in the classes means that there is a great need to keep control and distribute materials. We found it useful to have one instructor whose primary rôle was to concentrate on the interaction and participation, while the other concentrated on the more formal content.

The changes also required modifications in the way in which classes were prepared. We found that there was little need for many materials prepared beforehand as would be the case with a formal lecture. The best approach seemed to be to devise an outline of the class and then “go with the flow”, preparing any materials as they were needed. It also proved useful to prepare a summary sheet for distribution at the following class to debrief what had gone on.

While we might have foreseen some of these issues, it is clear that the presentation of classes such as these is a continual learning process. We will learn more in future sessions as we refine the approach and gain more feedback.
5. Evaluation

A short evaluation was carried out after the first presentation of the class on arrays using an approach similar to the “Instant Questionnaire” described in [8]. The main purpose of this was to give the instructors an indication of whether the classes were likely to be useful, and whether the idea was worth pursuing. Unfortunately, it has not yet been possible to undertake an evaluation that directly compares the classes with traditional lectures on the same material.

The evaluation was organised as follows. The students were asked before the class to grade their understanding of the topic on a scale of zero (no understanding) to ten (complete understanding). After the class, they were asked to repeat this, and the results were collected anonymously. A simple analysis of these numeric responses would give a very general indication as to the effectiveness of the session.

The results of the analysis were encouraging. The students initially rated their understanding at an average of 3.4. After the class their estimate had increased to 6.4. It might be argued that this improvement could have been brought about by a traditional lecture. However, all the students had previously attended a semester long lecture-based course on Pascal. Significantly, 91% of the class felt that the participative style of presentation was far preferable to the lectures they had had before.

There were, of course, other factors influencing the students’ learning. Most importantly, the class was significantly smaller (60) than the equivalent class had been for lectures (about 130). Some students commented that this also improved their learning; certainly it might be expected to offset some of the “embarrassment” factors noted in [11]. Overall, the evaluation seemed to indicate that a change to a participative approach was an effective step.

6. Conclusions

So far, the results from these innovations are encouraging. Certainly the students say that they are enjoying and remembering the classes more; this was also reported in [1] and [16]. As Race [20] suggests, we found that the students did indeed say that classes are worth attending if only “to see what will happen next”. A happy side effect of the classes is that the interaction appears to foster a better, less formal, relationship between the instructors and the students and among the students themselves. This also persists outside the classes, and the resulting atmosphere can only aid the learning process.

The introduction of these classes is also a learning process for the instructor. There is a constant need to devise more exercises and to present them in ways that avoid the pitfalls of “artificial, inauthentic, and awkward” [24] presentation. Experience has also shown, for example, that it is essential to make sure that all the students actively participate in as many classes as possible, and that the classes do not become dominated by a clique.

Presentation of the classes is certainly a different undertaking to the presentation of a normal lecture. There is, for example, a strong element of loss of control and unpredictability that some might find worrying. On the other hand, these aspects
make the classes more enjoyable and memorable for the students, so the risk is worthwhile.

While the classes are undoubtedly enjoyable for the instructors, and appear to be so for the students, there is a need to keep a focus on the academic content. The summary sheet was an essential tool in doing this; in future presentations we will look at the possibility of having the students themselves produce this sheet, perhaps on a rota basis.

A side effect of the introduction of these classes was that it made the instructors consider issues of motivation more closely than before. It has been shown, we think, that we were guilty of over assessing the students. Race [20] describes assessment as “the engine which drives motivation”; it was clear that our assessment was driving the students towards the learning strategies (surface) that we wanted to avoid. With the change to C++, it is now planned to reduce the amount of assessment in the course considerably.

The ideas presented here are not proposed as a total replacement for traditional lectures in programming courses. These classes are presented as a complement to more traditional forms of teaching, and are especially targeted towards those students who find the topic the most difficult. Indeed, it might be suggested that students with previous experience would find the whole approach somewhat patronising. We suggest that these students are still better served with traditional lectures; they have mastered the process of programming, and need concentrate only on the syntax and semantics of a language.

After the first introduction, there seems to be sufficient evidence of success to merit further effort in the development of these ideas. They will become an integral part of the School’s new module in C++ from session 1998/99, and will continue to be developed and enhanced.

7. Acknowledgments

I am eternally indebted to William Towle who assisted in the development and deployment of the approaches described here. He is a better teacher of programming than I will ever be.

Sarah Fores took care of all the coursework marks to let me concentrate. She also lent me the wool. Peter Jimack made sure that the experienced programmers kept out of my way.

Clarissa Price, Jane Muir and Emma Tyson all spent many many hours making sure I understood precisely how difficult programming can be. Thanks.

Maggie Boyle and Martyn Clark (the original lecture theatre terrorist) assisted with several very useful discussions. Maggie and David Gardner of the Teaching and Learning Support Unit provided the frisbees.

I have quoted Sally Laycock and John-Paul Duckworth without their knowledge. I just hope they don’t find out, and that Martyn doesn’t spot it.
Simon Myers assisted with the intricacies of the word processing. It’s useful to have a computer scientist around, and at least we didn’t have to use Microsoft!

None of this work would have been possible without the students who came along and acted as guinea pigs. Thanks, people.

8. References


http://www.lgu.ac.uk/deliberations/eff.learning/happen.html


