HELPING STUDENTS UNDERSTAND PROGRAMMING THROUGH ACTIVE LEARNING

Chris Phillips, Marie Devlin

Newcastle University, UK
chris.phillips@ncl.ac.uk, marie.devlin@ncl.ac.uk

Abstract
This paper draws on the literature for teaching introductory programming and previous pedagogical research on learning and teaching in large classes. We discuss both the problems posed by large class sizes in lectures and the common problems many novice programmers face. We then outline some fun and practical ways to illustrate basic programming concepts in a large class setting by drawing comparisons with real-world scenarios. These methods can be adopted for other disciplines and help increase interactivity in large classes, giving students hands-on experience of problem-solving and promoting independent thinking.

Key words:
Programming, Large Classes, Active Learning, Problem Solving

I INTRODUCTION

Learning to program can be difficult for many first year students of Computer Science, particularly if they have no background to the topic. Programming concepts are abstract and therefore challenging to teach in an interesting way that novice programmers can relate to easily e.g. algorithms and data structures are difficult for many students to visualise or use to solve problems. Most approaches to the teaching of programming focus on the discussion of syntax and semantics, followed by application to ‘typical’ or ‘toy’ problems and the illustration of techniques to solve these during a one hour lecture class using patterns which can be adapted to other situations. This method of teaching means that students tend to experience the tutor’s way of solving the problem and they are often ‘lead’ to the solution. In this type of approach, students tend to stop thinking for themselves and often accept that the tutor has come up with the ‘best’ or only solution. Teaching in this way also often includes the use of PowerPoint presentations that contain diagrams and pictures to illustrate meaning and describe abstract concepts. However, students often struggle to relate to these ideas when it comes to their practical application in a problem-solving lab session. This difficulty is further exacerbated by the fact that most first year programming classes are quite large (often in excess of 100), students remain seated for the duration of the lesson, interactivity is often limited and some students can be intimidated by class size so much that they do not have the confidence to interrupt and ask questions for clarification. In essence, boredom
quickly sets in. This situation is again exacerbated by the fact that the class experience in programming is diverse, from the raw beginner to the student who has their own web development company. In this paper we outline some fun and practical ways to illustrate basic programming concepts in a large lecture class setting. These methods can be adopted for other disciplines and help to increase interactivity in large classes, give students hands-on experience of problem solving and promote independent thinking.

II PROGRAMMING IN A LARGE CLASS

II.1 Issues in the Lecture Theatre

Computer programming at university/college level is often taught in the traditional way, via lecture classes that give an overview of theory and concepts, followed by practical laboratory classes in which students can put the theory from lectures into practice and explore the mechanics of the programming language of choice. Lecturers first teach the basics of a programming language in the lecture and then guide students towards effective strategies for the whole programming process via examples on the board. (Lahtinen et al, 2005). However, programming is essentially a practical skill and most lecturers acknowledge, albeit ruefully, that a student may only truly learn programming as a personal skill outside of lecture time via the practical sessions, or private study in which they can explore the language through solving relevant problems. The typical programming lecture consists of students listening to the instructor and seeing how the instructor solves the particular problem being discussed. This means that students gain useful insight into an instructor’s personal approach to problem-solving and their programming style which are largely based on experience but also on personal preferences. Students are therefore not really challenged or encouraged to learn their own way of doing things or explore their own perceptions and ideas about solutions. In this method of teaching, students tend to bow to the instructor as the authority, the one with all the knowledge and therefore switch off their thinking, especially in a lecture class where they are a captive, seated and physically-restricted audience and communication is one way and impersonal. Programming in Computer Science is the one skill that you need to be good at, or so it is assumed by students (and most lecturers) and therefore this can be a pressure filled experience for those students who may find the subject daunting or the lecture environment with so many peers present, intimidating. Students may feel inhibited about asking or answering questions and therefore it is harder to assess if they are having difficulty. In general, students need to be able to "build a mental model of the program and track what is happening in order to predict its behaviour", (Milne & Rowe, 2002), so many programming lecturers now use animated visualisations as a way of helping students see how programs work. Visualisation can be used to engage students and help them learn programming concepts more easily. However, although useful in private study, these are not constructed by the student, and still do not involve the student interactively in program construction.
during lecture time. Instead students rather passively view the animation as they would when watching a movie and again, visualisations are created by the tutor so may have limited interactive capability for use in personal time and may pose only one or two methods for reaching the solution. These animations also take a long time to develop and many often lack illustrative quality.

II.2 Issues in the Lab

When it comes to practicing new skills in the lab "novice programmers tend to approach program construction on a line-by-line basis, not thinking of the bigger picture i.e. bigger program structures", (Lahtinen et al, 2005). Studies of the literature on the behaviour of novice programmers including that by Robins and Lahtinen and colleagues, (Robins et al, 2003, Lahtinen et al, 2005), point out that whilst learning strategies and motivation will of course affect student behaviour on the whole, some students that are new to programming tend fail to apply the knowledge they have obtained in the lecture class to problem solving in the laboratory and many stop and give up when they find a particular task difficult. For many students the main source of their difficulty is not always the syntax and semantics of the programming language but rather on understanding how to combine the language constructs and basic concepts into a valid, working solution. (Lahtinen et al, 2005; Garner et al, 2005). Students often think they need to concentrate on the syntax of the programming language and this can lead them to concentrate on implementation issues rather than on activities such as planning or design or testing of their programs. Most novice students seek just to achieve a program that "simply compiles with no errors ", (McCracken, 2001). Larger class sizes make it difficult to design the learning experience so that it benefits everyone i.e. a learning experience that takes account of those with previous experience, those who have none and those who either find programming ‘easy’ or ‘difficult’ to learn. A large class in the laboratory can have a real impact on student learning because a laboratory session is typically staffed with only one tutor and perhaps a couple of demonstrators to assist. Students who are struggling may have to spend a lot more of the session working on their own to get their program working properly with minimal personal attention and this struggle may lead some to give up or think they are not ‘able’ to program, (Garner et al, 2005).

III Using ‘Active Learning’ in Lectures

Research indicates that "the most fundamental obstacles to learning programming are related to its problem solving character", (Deek & Turoff, 1999). If we are to assist novice programmers in their efforts to learn programming, it is therefore important to focus initially on the development of problem-solving skills. Also, much of the literature on active and deep learning suggests that most students do not internalise and cannot understand nor apply learning unless they are actively involved in it, (Higgs & McCarthy, 2005). With both these points in mind, over
several years at Newcastle we have adapted our introductory programming courses and curriculum to include a more active learning approach in lectures. There is no hard and fast definition of Active Learning, per se – the term depends really on the discipline but for a working definition, we take it to mean a learning environment and approach that allows students to participate in the search for meaning and understanding during a class, that focuses on greater student responsibility for learning and on the development of skills as well as knowledge. According to Cryer and Elton (Cryer & Elton, 1992) – “control of the learning and teaching should be shared” and active learning should focus on the ‘how’ of learning rather than the ‘what’ that most course learning outcomes emphasise. We try to help students to become more independent by changing lectures so they are not just places where students receive our wisdom or knowledge but where they can select and structure subject matter, solve problems, test out theories and also have some fun. According to Higgs and McCarthy, “An interplay of social and personal experience should be fostered in the process of learning.” We offer students opportunities for interaction in a large class and create activities that allow them to think deeply about the material and also to “re-order or re-structure any ideas” they may have in order to generate their own solution to the problem/problems that have been posed, (Higgs & McCarthy, 2005). Engaging students in an active learning activity that helps them visualise what is happening in the program and how it executes, is our way of doing this. We have tried computer animations and visualisations of algorithms in the past but investigations of such visualisations have reported mixed outcomes whereas students taught with active and cooperative learning showed far lower withdrawal and failure rates (Sheard, 2009; Gonzalez, 2006). We therefore focus on collaborative learning activities that emphasise the social aspects of learning.

IV METHODS TO PROMOTE ACTIVE LEARNING

We describe here a number of activities which students can relate to readily since they are likely to have met all of them at some stage in their lives. At first these activities are taken at face value, with no reference to computer programming. Once completed, students are asked to identify characteristics associated with each activity, and at that point analogies can be made to programming concepts. We consider these analogies in the next section, here we describe the activities themselves. As can be seen in Figure 1, the use of an everyday scenario such as making tea can be used to illustrate the concepts of stepwise refinement, decision making and looping. Stepwise refinement is a technique that students can use to decompose a problem into a sequence of programmable steps. There are also variations on using the concept of making tea that can be altered to illustrate a wide range of programming concepts e.g. There is the selection of the teabag in the first place (Green, Earl Grey, Assam, peppermint etc.). The beauty of using such a simple task as making tea is that it is a scenario most students will be familiar with, i.e. they know the steps so it is a good place to start when trying to create an algorithm.
Making Tea

1. Put teabag in cup
2. Fill kettle
3. Boil Kettle
4. Pour water in cup
5. Leave for a minute or two
6. Remove teabag
7. Add milk to taste
8. Add sugar to taste
9. Stir

Figure 1 : Algorithms derived from making tea

Decision Making

If there are no teabags in the cupboard
Then go out and buy packet
Else put teabag in cup

Looping

Repeat the kettle is not full
wait
Until Turn the tap off

IV.1 Take Your Partners

All students will be familiar with the concept of dance steps, and although their preferred mode of dancing may well be ‘free format’ dancing to modern music, it is very likely that they will be aware of more ‘structured’ dance steps, such as a waltz, tango, etc. which follow a clearly-defined pattern. In this exercise groups of students participate in one or more structured dance activities in front of the rest of the class. Although not essential, it is best if the dance involves a pairing. Conventionally a man and a woman, even if it is necessary that a man has to play a woman’s role or vice versa. This is ‘fun’ in that invariably students make wrong steps (and there are important lessons that can be learnt from this), and suitable music (something appropriate can readily be found on YouTube, Spotify, etc. adds to the atmosphere. If a group of students are able to demonstrate dance movements without prompting, the rest of the class can be asked to discern the pattern and write it down in some ‘language’ (e.g. reduced English). Alternatively, an ‘algorithm’ can be displayed which describes the dance steps and students have to follow them. After an initial failure it may be necessary to display a video of people dancing. The dance step chosen is largely immaterial, as long as it is structured – at Newcastle we have frequently used the Gay Gordons (a Scottish Country Dance), although the Argentinian Tango, Line Dancing, etc. are all legitimate. The only requirements are that the dance is able to be used to demonstrate the programming concepts of functional abstraction, repetition and selection.

IV.2 Phone a Friend

It is rare these days to find a student who doesn’t possess a mobile phone, so this is an activity that all students can participate in. Whilst there is some commonality in the way to operate a phone the precise operations required to make a phone call vary from one type of phone to another. Here, each student is asked to write down, in very precise terms how to make a phone call using their own mobile phone. They then exchange this list of instructions with a fellow student along with their phones, and then try to use the phone and the instructions to make a call to their own phone.
From this students should be asked to identify and write down with their colleague a core set of mobile phone operations that all phones ‘inherit’, and the ways that individual phone sets digress from this behaviour.

**IV.3 Little House on the Prairie**

Most students will have, at some time, made simple two-dimensional drawings either using pen-and-paper, or some low-level technology such as Etch-A-sketch. Many will have made a simple drawing of made up of basic shapes, such as squares, oblongs, triangles, etc. Here students are asked, individually, to write down a sequence of instructions that use these basic shapes to draw a house. The instructions will need to be parameterized, for example, draw a square of side length 2 units starting at some position measured horizontally and vertically from an assumed origin (e.g. the bottom left-hand corner of a piece of paper). They then pass these instructions to their neighbour who attempts faithfully to follow the instructions.

**IV.4 Piggy in the Middle**

The aim of this exercise is to take a group of students and sort them according to some criterion (alphabetically according to their first name, by age, or by height etc.). If nothing else, this exercise is a great ice-breaker in that you, and the students, getting to know the class better – they are no longer an anonymous collection of faces. Any sorting algorithm could be used, but quicksort, in which a value is placed in its correct position somewhere in the middle of the ‘data collection’ can be particularly enjoyable.

**V. HOW THESE RELATE TO PROGRAMMING**

At the heart of all (imperative) programming courses are the introduction of constructs such as repetition and selection, and whilst the syntax may vary from one language to another may vary, the principles do not. Functional abstraction is a key problem-solving tool that students have to learn, with the associated notion of parametrisation, something which students traditionally have considerable difficulty with. A further key problem-solving tool is type abstraction as implemented in object-oriented languages such as C++ and Java, and the manipulation of objects of those types. Many code implementations involve the use of arrays, or more generically collections of objects. And although not widely used, recursion can be an extremely powerful problem-solving tool. Each of the activities described in this paper are used to illustrate one or more of these programming concepts:

1. Dances such as the Gay Gordons can readily be used to illustrate the concepts of repetition and selection. A sequence of steps is repeated over and over until the music ends (illustrating a while loop), or a fixed number of times (a for loop). Each sequence can be broken down into a number of
phases (in the case of the Gay Gordons 4) and the behaviour in each phase is somewhat different, leading to the notion of selection (if this is Phase 1 then ..., but if this is Phase 2 ...). Further, the steps followed are often different for one partner than the other.

2. The operation of a mobile phone is normally sequential in nature, so there is not much here to illustrate in the way of repetition or selection (although there are some opportunities). Rather the activity is used to illustrate object-orientation, with the abstract concept of a mobile phone type, and an instance of that type being an individual student’s own phone. Object-orientation involves invoking operations on an object, and in the mobile phone example this involves pressing the keys on the keypad. Exactly what happens when a student presses a key is not readily apparent (how come when I press the “call” key there is an attempt to connect to my friend’s phone?) and this gives an example of functional abstraction. There is the opportunity for introducing the concept of a basic mobile phone type, with properties that all mobile phones exhibit, and phones that exhibit all of this behaviour an more (e.g. a Blackberry) and this gives students an early feel for what inheritance is all about.

3. Using simple shapes to draw a crude house is again likely to involve sequential operation but gives another example of functional abstraction – there is a ‘method’ to draw a square that I can be broken down into drawing four straight lines at right-angles to each other (and the concept of repetition can be illustrated here). But here there is the additional concept of parameterisation. It is not sufficient to simply write “draw a square” since the size and position of the square have to be specified. This could be extended to say that a fill colour of the square has to be specified also. There is a further opportunity to use this example to illustrate object-orientation here in terms of a Canvas type, with operations such as penDown, penUp, penMoveHorizontal, and some of these (e.g. how far to move the pen horizontally) will be parameterised.

4. An important aspect of any course in programming relates to data processing (the Data Structures and Algorithms CS2 course). Some of these processing algorithms, such as quicksort, can be rather complex and difficult to follow, and whilst computer simulations can be used to illustrate them, using ‘live’ data in the form of human beings gives students an experience they can relate to and remember. Using quicksort enables a tutor to demonstrate in a very practical way the important problem-solving tool of recursion, and if you felt so inclined, the potential for concurrency.

Our use of the active methods outlined, over the years, has been largely experimental but our findings are consistent with those of Gonzalez, (Gonzalez, 2006), i.e. greater participation in class, more interaction between students and lecturers, students thinking more actively and increasing in confidence and problem-solving ability, all of which has led to a reduction in the number of students leaving the course in their first year. However, given the nature of different cohorts and the
number of measures for retention of students that we have in place, it has been
difficult to distinguish is if any of our methods have been the sole deciding factor for
these changes. What has changed is our approach to teaching introductory
programming. We have come to realise the weaknesses in the didactic method used
in traditional lectures and their detrimental effect on student acquisition of the skill
of programming, which is essentially a creative endeavour as well as a practical task.
It is for these reasons that we have changed from ‘transmitting knowledge’ to
students in the lectures to allowing students to create their own knowledge by
exploring and visualising algorithms physically, interacting with others rather than
being passive listeners, and by challenging themselves to solve the problems posed
effectively, without influence of the tutor’s view on the ‘correct’ answer.

VI RELATING THESE METHODS TO OTHER DISCIPLINES

Currently Higher Education, (in the UK, at least), produces “a curriculum driven
student, often used to solving problems in a theoretically coherent framework, who
is used to a classroom in which instructors instruct and learners learn ” (Denicolo et
al., 1992). This is in direct contradiction to the learners that we all say we want to
develop at the end of their program. Ideally, we would all like our graduates to leave
being able to communicate well, take direction yet be self motivated, problem solve
and find things out, make intelligent judgements, test ideas etc. Large group teaching
makes it even more difficult to help students develop these qualities, especially those
who are struggling with the material during a lecture class. To help overcome the
limitations of large classes in lectures and reduced individual contact-time, students
can be drawn into reflecting on the subject matter by posing questions, highlighting
puzzles or enigmas, being drawn step-by-step into an analysis of a case study or the
discovery of a new process or phenomenon in any discipline (Elton & Cryer, 1992a).
The ability to problem-solve is part of the skill set that every discipline aims for their
students to learn and because expert thinking skills cannot be asorbed, “the best way
to learn to solve problems is to be given a problem to solve” (Elton and Cryer,
1992a). The main focus of all problem-solving activities used should be on relating
everyday scenarios that students can identify with, to the concepts of the material
being taught. Whereas this might not always be possible to do directly, depending on
the discipline, there are techniques that can be adapted to every discipline if the
notion of problem-solving as a basis for the teaching is used. If the tutor relates the
problem scenario to something a student may have encountered previously e.g. a
banking scenario, a travelling scenario, a budgeting scenario, a sporting scenario or a
personal interaction that takes place in their everyday world of work and living, this
will make it easier for the student to relate to, to understand and comment on and to
think about in new ways.
VII CONCLUSION

In this paper we have highlighted some methods used by at Newcastle to help students visualise programming concepts and construct solutions for themselves using everyday experiences and objects. Actively involving students in the class keeps them interested, allows them to explore their ideas and understanding of the concepts under discussion in the lecture session and also allows them to interact with each other and is therefore more interesting and intellectually stimulating than traditional lectures. We have also outlined how a similar problem-solving approach such as this could be used in other disciplines to engage students actively in the lecture, in spite of large class size. Such an approach allows students to think more independently about the material being presented, to discuss the issues brought forth by the differing perspectives of their classmates, to explore their own ideas, create their own knowledge and form their own independent solutions, rather than passively absorbing the material and not being able to transfer it to other scenarios in the laboratory or accepting that the tutor’s view is the only correct solution.

REFERENCES

Cryer P and Elton L, (1992a, 2nd ed), Active Learning in Large Classes and with Increasing Student Numbers, Module 4, Effective Learning and Teaching in Higher Education, CVCP Universities’ Staff Development and Training Unit, Sheffield.

Cryer, P., and Elton, L., (1992b, 2nd ed), Learning Actively on One’s own, Block B: Preparing Self-Instructional Materials, Module 8, Effective Teaching and Learning in Higher Education, CVCP Universities’ Staff Development and Training Unit, Sheffield.


