# TEACHING PROGRAMMING – A JOURNEY FROM TEACHER TO MOTIVATOR

Tony Jenkins
School of Computing
University of Leeds
Leeds, UK.
tony@comp.leeds.ac.uk
http://www.comp.leeds.ac.uk/tony/

## ABSTRACT

*Few would disagree that learning to program is a fundamental part of degree-level education in computing. Fewer would disagree that teaching programming effectively in today's mass Higher Education is a problem. The graduating student who professes a complete inability to write the simplest program is commonplace.*

*This paper argues that the primary role of a teacher of programming is not as a communicator of information, as in many other subjects or areas of computing. Rather, the teacher's main role is that of a motivator. The students must be motivated to engage in tasks that will make them learn, and it is the teacher's job to ensure this. The role of communicating information such as the basics of syntax in a lecture theatre is very much secondary.*

**Keywords**

*Programming, Motivation, Expectancy, Diversity.*

## 1. INTRODUCTION

Few of those who have tried to teach it in Higher Education would argue that the teaching of introductory programming in post-18 education is a problem. Instructors will be familiar with the struggles of students as they try in vain to come to terms with this elementary part of our discipline. The phenomenon of the final year student, about to graduate, determined to avoid programming in any final year courses will also be familiar to many.

At first sight, teaching programming appears to be a straightforward task. Most staff who might be asked to teach a programming course will be fluent in at least one language, and will probably not be concerned at the prospect of learning another if necessary. There are plentiful textbooks covering all languages, paradigms and approaches. The material itself can be arranged neatly to fit into a lecture course over one or two semesters, and can be readily arranged in order of increasing complexity. Assessment is a simple matter of devising some suitable programming tasks.

This model, with slight local variations, is widespread. It is so widespread that it must be safe to assume that those teaching to it believe it works, or at least have had experience that it has worked in the past. How does this view fit with the phenomenon of the final year computing student who simply *cannot* program?

Can it be that the teaching techniques that were devised and first deployed in a Higher Education available only to an academic elite are no longer appropriate in these times of mass access?

2nd Annual LTSN-ICS Conference, London

## 2. A TEACHER OF PROGRAMMING

I have been teaching introductory programming in one of the largest Computing departments in the UK for some eight years. I came to the task by accident – a colleague was unavailable for one year, and I was asked to fill in. My teaching background was limited, and what I had was in the area of databases and fourth generation languages. Still, it was explained to me, teaching such basic material as programming is not a problem.

For my first delivery of the module, the language was Pascal in a Unix environment. I presented the lectures following the materials provided by my predecessor, and I dealt with the steady flow of students knocking on my door. Coursework was issued and marked, and I enthusiastically checked through it for evidence of plagiarism.

It was with genuine surprise that I discovered at the end of the module that many of my students simply could not write even the simplest programs. They had completed my assessments well enough and secured sufficient marks to pass (which was in itself worrying!), but they still could not *program* in any meaningful sense. What was I doing wrong?

Since this first outing I have spent many hours thinking about the ways in which I teach programming and have, I think, achieved some modest success. While taking a year's break from teaching programming (a shortfall in staff to teach computer graphics!), I found myself reading a section in John Biggs' *Teaching for Quality Learning at University* [3], which very closely mirrored what I had been doing over the previous few years.

## 2.1 Levels of Thinking about Teaching

Biggs presents three "levels" at which teachers may reflect on their teaching. He argues that the third level is the best (if not the only one that can be successful), and the first is the worst, being far too superficial. A teacher will tend to move through these levels, in order, as they gain more experience.

### 2.1.1 Level 1 – What the Student is

This is, for many teachers, the first and most attractive explanation for a group of students' failure to learn. There is simply something inherently wrong with the class – they lack motivation, they lack aptitude, or they are just not interested. There is nothing that the teacher could have done to make them learn. A teacher taking this view will habitually divide a class into "good students" and "bad students", the classification depending largely on perceived academic ability or summative results.

This is an attractive, simple, explanation with the added benefit that the "blame" is laid firmly on the students. The teaching has remained constant, so the change must be in the nature of the students. Typically, this view sees teaching as a process of *transmitting information* – programming is a skill, not a collection of information.

If this view is to hold, then we must presumably accept that University entry standards are too low and should be raised. We must accept that only a small percentage of the population can ever hope to learn to program, and that these are the only students we can hope to teach.

My own development in thinking about my teaching of programming started with these "level 1" thoughts. The students would learn from my lectures if they applied themselves, or if they were sufficiently motivated, or if they had an aptitude. On reflection, this view is far too simplistic. Any class of students starting an undergraduate degree is a body of intelligent people, who have excelled in their education to date. The only reason I happen to be teaching them is that I am somewhat (probably) older and possess a skill they do not. Can it really be the case that students today are less *intelligent* than students, say, fifteen years ago? Surely not!

### 2.1.2 Level 2 – What the Teacher Does

This view sees the learning of the class as a factor depending directly on the activities of the teacher (it follows that there are "good teachers" and "bad teachers", measured presumably in terms of the summative results of their students). This view remains firmly based on the concept of *transmission*, but now concentrates on the transmission of *understanding* rather than mere information. This seems to be much closer to that which we need to transmit when we teach programming.

Teachers thinking at this level will adopt innovative techniques. They will reject formal lectures and choose more participative methods, perhaps using physical props and entertaining demonstrations. These activities, the arguments goes, will engage the enthusiasm of the students, will motivate them, and they will learn.

This thinking leads to much of the published research in innovative methods for teaching programming (for example [1], [10], [15]). The problem with the majority of this work is that there is rarely any compelling evidence that these techniques bring with them any concrete *educational* benefit. It is true that students enjoy such classes, and so are more likely to remember them, but there is limited evidence indeed that they learn any more. These innovations have a place, but only as an "educational novelty" [13] – if they are overused they lose this novelty value and hence their effectiveness.

I have thought at this level. I have made students join in lectures, I have thrown frisbees at them to illustrate parameter passing, I have worn hats to indicate flow of control, and I have acted out algorithms. I have enjoyed these sessions, and I believe the students have enjoyed them too. But where is my evidence that they have learned?

No less a figure than Edsger Dijkstra held that any teacher resorting to metaphor in order to explain a programming concept was guilty of "contempt of the student body" [6]. Dijkstra's views on the teaching of programming are extreme, and are presented so as to deliberately provoke debate, but the suspicion must remain that some of the techniques proposed in the literature run the risk of trivialising the subject.

### 2.1.3 Level 3 – What the Student Does

This is the deepest and, according to Biggs, the best level. Here the focus is firmly on the activities in which the student engages as part of their learning. The teacher's role is to devise suitable tasks, to make sure that the students do indeed engage in them and to, as Biggs neatly puts it "get the students to agree that appropriate task engagement is a good idea" [3]. At the end of a course the degree to which a student has engaged will determine their final mark. There is no longer emphasis on transmission – teaching is much more a process of *motivation*.

In the context of teaching programming, it is trivial to identify the appropriate tasks. If a student is to learn to program, then they must write programs. A teacher's job is not to communicate the minutiae of syntax or the nuances of some particular language, but to persuade the students that learning to program (and so programming) would be a good thing. This is very different to "level 1" thinking.

If I see before me now an introductory programming class, I can no longer assume that those present have the slightest interest in learning to program (or indeed very much idea about what this entails). My first, perhaps only, task is to motivate them to want to learn, to engage their curiosity and interest, and to make them want to go away and write some programs. I can then move on to talk about some programming concepts, and even some details of syntax, but I can do this only when I have a suitably prepared, receptive and motivated audience.

It follows, then, that in my new role as a *motivator*, I must understand what motivates a programming student in the 21st century.

## 3. MOTIVATION

Measuring motivation, or discovering what motivates an individual is far from straightforward. It is possible to observe or interview an individual and from this deduce their motivations, but one can never be sure [2]. Moreover, in an increasingly diverse student population [11], we might expect to find increasingly diverse motivations.

Much has been written about the factors that might make a learner value a learning opportunity (for example [7] and [8]), and it is possible to extract some broad categories [12]:

- *extrinsic* – the motivation comes from some external source, probably an expected future reward (usually financial).
- *intrinsic* – the key is an interest in the subject itself.
- *social* – the main motivator is a desire to please some third party whose opinion is valid (family, sponsor, teacher).
- *achievement* – the motivation comes from the sense of "doing well" academically, and possibly the satisfaction of doing better than peers (this is sometimes called "competitive" motivation).
- *null* – there is no clear motivation (the student may have simply "drifted into" a programme of study, and may have no clear idea of why they did this).

It is clear that an accurate understanding of the main motivators of a class could inform the way in which a teacher approaches a subject. A student with extrinsic motivation, for example, will be keen to hear about lucrative careers in programming and will want to learn the very latest "in demand" skills. On the other hand, achievement motivation would lead a student to concentrate solely on those activities that were perceived to lead to high marks, and a desire that the teacher would dwell on these. Such a student would not necessarily want to "learn".

A level 1 teacher would assume that most students taking computing degrees are extrinsically motivated, with the main aim of securing financial rewards in a good job. Indeed, I have suggested this myself [11] as a primary cause behind the increasing diversity in the student population. However, more recent work [12] has indicated that, while eventual employment is indeed the most important factor for many, almost half our students remain committed to learning for its own sake, and profess an interest in the subject. The downside is that over half see their programming module simply as another subject to be studied – they do not share the perception of their teachers that this is a fundamental part of their education in computing.

While understanding how a student is motivated is useful, we would also like to have some measure, no matter how crude, of the *extent* to which a student is motivated. This is, of course, impossible – there is no scale of motivation. What we can have, though, is an understanding of the factors that control the extent of motivation.

The extent of motivation can be seen as a function of two factors, *expectancy* and *value* (for example [3], [13]):

$$\textit{Motivation} = \textit{Expectancy} \times \textit{Value}$$

It is important that these two factors are seen to *multiply* rather than add. If either factor falls to zero there will be no motivation, no matter how high the other factor becomes.

The second factor is closely related to those already identified; it is the reason why a student values success. In a programming context this might be future employment, high marks, and so on – it matters not why a student values success as long as they do. In this context it is important that a teacher also appreciates that different students will have different views of what constitutes success; some may be content to pass, some may demand first class results. A teacher's job should be to ensure that each student can achieve success, defined in the student's own terms.

It follows that a student must *expect* to succeed. If a student does not expect to pass a programming module then, no matter how much they value success, they will not be motivated, and will not engage in the tasks the teacher devises. There is a huge implication here for the way in which we teach, and in particular assess, programming.

It is, I suggest, safe to assume that at the start of a programming module all the students value the potential outcome. What is less certain is that they all expect to succeed. The first weeks of a degree course are a worrying, stressful time for many as they seek to find their feet in a new environment. In my institution, programming has a reputation for being "hard" and there is a powerful student grapevine that communicates this to the new recruits. I doubt very much that all the students who start a programming course expect to succeed.

A concept closely related to the expectancy element of motivation is *personal causality* [5]. If a student is to expect success they must believe that the factors that will determine their success or failure are within their own control. Again, there is an implication for the assessment of students here – the assessment must never be such that a student can feel there is no way they can complete the module successfully.

As a motivator rather than just a teacher, my role is now very different. I must understand why the students are taking this programming course, and I must exploit this to make them value the outcome. At the same time, I must ensure that they expect to succeed in whatever terms they choose to define. I must make sure that the assessment I devise gives them every chance of success but does not distract from the most important aspect – the learning.

## 4. TRUST

Teachers are figures of authority. Motivators are rather different, and are more like trusted friends. If I am to successfully motivate students, they must, in some sense at least, trust me and I must trust them.

McGregor [14] describes two climates that may be found in an organisation, and these can be readily applied to the teaching relationship. The two climates, Theory X and Theory Y are based on different levels of trust.

In a theory X climate, the student is not trusted. The teacher makes the assumption that the students are taking the module simply to pass (level 1 thinking again), do not want to learn and will cheat if they believe they will not be caught. This leads to an environment where assessments are rigorously defined, deadlines ruthlessly enforced, and plagiarism zealously detected. It is hard to see how many students could like, let alone trust, their teacher in this climate.

Theory Y is the opposite. Here, the teacher believes the students want to learn, and trusts them. Assessments are devised to be loosely defined and interesting [9], a certain flexibility with deadlines may be allowed (after all, is it important that a student *does* a task or that they do it *before* some deadline?), and it will be assumed that students will not cheat.

Theory Y also satisfies personal causality – the students are much more in control of their own destinies than they would be under theory X. The teacher trusts them to take this control, and effectively delegates it.

Admittedly, it is unlikely that a strictly theory Y environment could ever be introduced. Academic standards demand attention to plagiarism and enforcement of deadlines. At the same time, students finding themselves in a world that is more theory Y than theory X are much more likely to trust their teacher. It follows that they are much more likely to be easily motivated.

## 5. IMPLICATIONS

Teaching programming is a problem. Over several years of attempts to impart basic Pascal or C++ I find that I have come to believe this more and more, but also that my views on the causes of the problem have changed. I have followed Biggs' three levels of teaching (without knowing of them at the time!), and now find myself, I think, at level 3.

Level 3 is a systemic view. The problem in teaching programming lies in a system of connected factors. I summarise these factors under three headings.

### 5.1 Expectations

Students and teachers both have expectations at the start of a programming course. Teachers need to be sure of what they expect from the class – they need to understand why the class are there, and what they are hoping for from the course.

We must acknowledge that many students are expecting programming to be difficult. While we would be wrong to tell them that it is easy, we must reassure them, we must provide support, and we must make them *expect* that they will succeed.

Teachers must ensure that students know why they are learning to program, and must make them value the outcome, ideally in terms above mere marks. It is no longer safe (if indeed it ever was) to assume that the students arrive expecting to learn to program and interested in doing so.

At the same time, it is to be hoped that students become more certain in their expectations [4]. Computing is still, for many, a course they chose for no clear reason, and programming is simply one part of that.

### 5.2 Motivation

I find that I had views about why students were motivated to be in my class. My views were anecdotal, based on what I heard in tutorials or deduced from what I saw. When I had the time to investigate this properly, I was amazed by what I found [12]. It appeared that I did not have a class determined to jump through "academic hoops" until they could command enormous salaries in industry, but a class many of whom were genuinely interested in learning what I was trying to teach them.

If I come to learn a new language I do not immediately seek out a lecture course. I buy a book and I spend a few days writing some sample programs. My book then becomes a reference, and I claim to know the language, safe in the knowledge that my skill will grow as I use it more. Why, then, can my students not learn in the same way? There is no reason; the only thing they lack is motivation.

My job as teacher has moved away from presenting lectures on syntax (which it has to be said are very boring lectures), to motivating students to engage in those activities from which I know they will learn.

### 5.3 Relationship

If this approach is to work, there must be a particular relationship between the teacher and the students. This relationship must be based on trust and, perhaps as part of this, mutual respect.

I trust my students not to cheat. At the same time, those who do not cheat trust me to seek out the ones that do. When these are found, as they inevitably are every year, my interest is very much in *why* they cheated rather than in applying penalties.

Operating a system based on trust would not come naturally to all. Many staff (and students) would prefer to operate a "them and us", theory X, climate. This might well be possible, or even appropriate, in some subjects or disciplines, but surely it is not in programming. Programming is a skill, not a subject. It is best learnt when the learner has ready access to a skilled programmer for advice. This will not be the case in a theory X climate.

## 6. CONCLUSIONS

I talk to many students during their first year. If I mention their academic progress, the talk will always turn to programming. More often than not, the programming course is seen as a nightmare that has (hopefully) passed, and the student is keen to see how they can avoid programming in the future. I have on several occasions seen students leave the university simply to avoid programming. Every year I see students spending untold hours on their assignments. I see suffering.

This cannot be right. One topic should not dominate the curriculum, and the student experience, to this extent. It has to be the case that there is something fundamentally wrong with the way we teach programming. One step (and I do not claim a panacea) is to appreciate the crucial role that motivation has to play in teaching programming.

A student will not learn unless they are motivated. It must be a teacher's main task, therefore, to ensure that all their students are properly motivated.

## 7. ACKNOWLEDGMENTS

I am indebted to all the students who have tolerated my attempts at teaching them to program.

This paper is based on a presentation I gave at the First LTSN-ICS Conference on Teaching Programming. My thanks to LTSN-ICS for provoking me into writing it.

Thanks are also due to Ken Brodlie for letting me take a year away from teaching it to allow for some serious thinking.

## 8. REFERENCES

[1] Astrachan, Owen. *Hooks and Props in Teaching Programming.* In Proceedings of ITiCSE '98, pp 21-24, ACM, 1998

[2] Ball, Samuel. *Motivation in Education.* Academic Press, 1977.

[3] Biggs, John. *Teaching for Quality Learning at University.* OUP / SRHE, 1999.

[4] Clark, Martyn and Jenkins, Tony. *What are they going to do now?* In L Brooks and C Kimble (eds), UKAIS '99: Information Systems The Next Generation, pp 755-764, McGraw-Hill, 1999.

[5] DeCharms, Richard. *Personal Causation – The Internal Effective Determinants of Behavior.* Academic Press, 1968.

[6] Dijkstra, Edsger W. *On the Cruelty of really Teaching Computer Science.* Communications of the ACM, 32 (12), pp 1398-1404, 1989.

[7] Entwisle, Noel. *Motivation and Approaches to Learning: Motivation and Conceptions of Teaching.* In Sally Brown et al (eds), "Motivating Students", pp 15-23, Kogan Page 1998.

[8] Fallows, Stephen and Ahmet, Kemal (eds). *Inspiring Students: Case Studies in Motivating the Learner.* Kogan Page, 1998.

[9] Greening, Phil. *Students seen Flocking in Programming Assignments.* Proceedings of ITiCSE 2000, pp 93-96, ACM, 2000.

[10] Jenkins, Tony. *A Participative Approach to Teaching Programming.* In Proceedings of ITiCSE '98, pp 125-129, ACM, 1998.

[11] Jenkins, Tony and Davy, John. *Dealing with Diversity in Introductory Programming.* In Proceedings of 1st Annual LTSN-ICS Conference, pp 81-87, 2000.

[12] Jenkins, Tony. *The Motivation of Students of Programming.* In Proceedings of ITiCSE 2001, pp 53-56, ACM, 2001.

[13] Keller, John M. *Motivational Design of Instruction.* In Charles M Reigeluth (ed), "Instructional Design Theories and Models – An Overview of the Current Status", pp 383-434, Lawrence Erlbaum Associates, 1983.

[14] McGregor, D. *The Human Side of Enterprise.* McGraw-Hill, 1960.

[15] Siegel, Eric V. *Why do fools fall into infinite loops: Singing to Your Computer Science Class.* In Proceedings of ITiCSE '99, pp 167-170, ACM, 1999.