

# Case Study 1: Teaching and Assessing Programming

Geraint A. Wiggins

April 30, 2008

## 1 Introduction

Many people have an incorrect idea of what is taught in computer science departments. They believe that computer science is the same as the ICT programmes taught at school; indeed, the Government has added to the confusion by lumping computer science under the same heading. However, there is a useful and important distinction between the two. ICT, or Information and Communications Technology, is generally most usefully described as training for the use of general computer-based tools for purposes other than computing itself: operating systems like Mac OS X, software like Microsoft Office, information retrieval tools such as browsers and Google, network technology such as routers, switches and the like. Computer Science is perhaps best described by contrast with this: computer scientists are the people who design and create all these facilities, as opposed to those who merely use them. Of course, this implies a broad range of activity, from user interfaces right down to the electronic hardware. At Goldsmiths, we teach the software end of computing, the only exposure to hardware design and operation in our courses being theoretical.

There is one fundamental skill which is common to all software-based computer scientists (and most of the hardware specialists too, in fact) which is *programming*, the subject of this document. It is worth considering in detail what is involved in programming, before considering how to teach and assess it.

## 2 The Activity of Programming

Producing a computer program is a complex activity, involving most of the skills required of a level 3 student in most level-descriptor systems: creativity, analysis, synthesis and critical evaluation. The activities involved, at least in the didactic context, may be broken down into several stages, which are in fact rarely separate in practice:

**Problem/Task Analysis**, in which the programmer learns about the thing that the program is to do. (This is one of the things that makes computer science multi-disciplinary: one is constantly having to learn about one's application domain.)

**Specification Development**, in which someone (sometimes the programmer) writes down, more or less formally, what the program is to do and how it is to do it. This and the previous phase are sometimes referred to together as "Requirements Specification". There is often a loop between problem analysis and specification development, because the act of making precise the specification makes visible gaps in understanding. Specification development almost always begins with words, but there are mathematical languages, such as the Unified Modelling Language (UML) which aim to make the process more formal, and therefore more likely to be correct (in some sense).

**Program Development**, in which the programmer takes the specification and constructs a program which fulfils it. Again, there is a loop, back to specification development, because the level of precision needed to write a program is very great, and reaching it often involves making the specification more precise. Sometimes, it's necessary to revise the problem analysis too.

**Program Testing**, in which the programmer attempts to find the errors that will inevitably have arisen in the process of developing the program. These will be broadly of two kinds:

**Syntax Errors**, which are typographical mistakes in the spelling or “grammar” of the program;

**Logic Errors**, which are places where what was implemented did not match the specification.

Syntax errors can be detected, with varying degrees of accuracy, by the computer language used, and an “error message” results.

One important sub-skill of programming is *debugging*, where the programmer attempts, by reference first to error messages, and then to the program’s behaviour in comparison with its specification, to correct all the errors. It is not easy to imagine how difficult this is, until one tries it, because any non-trivial program will probably have interactions between errors, as well as the errors themselves; when this happens, the behaviour of the program can be utterly inscrutable, and, in general, it is not possible to ask the computer “why?”, because there are theoretical limits on what can be known about a program and its behaviour.

**Specification Testing**, which is the final stage, usually in conjunction with a client, where we decide whether the thing we specified was what we actually wanted. Often, it was not.

Evidently, this is a complex process, and attempts have been made to codify it, some of which we teach. The problem, however, is like the chicken and the egg. In order to understand why the cycle above is important, students need to be able to experience serious programming; however, to start programming at all, at least in many programming languages, one needs a large battery of conceptual tools, but the conceptual tools are only comprehensible in context of the languages one has not yet learned; to understand the error messages generated by many programming systems, one needs to be experienced, and therefore one cannot really learn from them. And because computers are complicated things, documentation, even when it is well-written, can be very daunting.

## 3 Teaching and Assessing Programming

### 3.1 History

To put this in context, it’s worth considering some history. Until the late 1950s, computer programming was an activity for scientists, all of whom had taught themselves. There were no undergraduate computing programmes. In the 1970s, “computer studies” was sometimes taught in schools. This was based around very simple programming languages, specially designed for teaching. The Computer Education in Schools programme did a reasonably good job of teaching the very simple programming instructions, and these were mostly applied to simple mathematical problems: it was taken as a given that if you were not good at maths, you would not succeed as a programmer, so it seemed not really to matter if that link was leaned upon.

With the arrival of the microprocessor in the late 1970s, relatively inexpensive computers, such as the Commodore PET, became common in schools, promoting a programming language called BASIC<sup>1</sup> which served very well the relatively small population of boys (sic) who were interested enough to use computers in their spare time. At this stage, there was almost no interaction between computing and the rest of the curriculum, and there were no formal qualifications. Word processors were only just being invented, so the only way to use a computer, if you wanted to, was still to become a programmer.

Around this time, the first undergraduate computing programmes began to appear, sometimes initially as short programmes, taken in conjunction with another subject, because it was deemed that there was not enough to full a full three years. These programmes focused very much on programming, since, as we said above, there weren’t very many applications to use. One important feature was the use of multiple programming languages, for different applications, and with different styles.<sup>2</sup> This was termed “comparative programming languages”, presumably by analogy with literature: it is certainly true that the comparison between languages helps one understand the significance of what is common—and what differs—between them.

---

<sup>1</sup>Beginners’ All-purpose Symbolic Instructional Code, designed in the 1950, in fact.

<sup>2</sup>For example, this author, in a two-year computer science course, was taught Algol W, Algol 68, BCPL, Cobol, Simula 67, Prolog and Lisp, as well as two different machine codes, alongside a raft of applications-orientated material.

At this stage, teaching and assessment was done with the assumption that students would be able already to conceptualise and analyse problems: the teaching was merely “how to do” each of the things one “obviously” needed to make a program work, and the assessment was in terms of marking programs to do simple mathematical puzzles.

However, at this stage it became clear that it was no longer enough to work this way, and some computing departments (notably MIT and Edinburgh) developed teaching systems, intended to cut away some of the obscuring difficulty from programming, and to give the students direct access to the concepts, in the way that had been so, perforce, previously. These attempts did not generally catch on, possibly, we suggest, because the people teaching programming had had so long to learn it, step by step, that they had lost touch with just how difficult it is.

By the 1980s, computer science had developed into an altogether more professional pastime, with a UK professional body specifying the range of course content, and more advanced computer languages, with more complex facilities, to be taught. Because the emphasis was now on “general purpose” computer languages, it was believed that there was little value in proceeding with comparative language studies, and so a tendency developed to choose one programming language to teach with (for example, Modula 2 was designed for this purpose). In the 1990s, the arrival of enforced good practice<sup>3</sup> in Object-Oriented (sic) programming languages laid a further layer of complexity on top of what was already a very difficult thing to learn. By now, computer science was a main-stream subject in most universities, and even a cash cow, until the media dramatically exaggerated the effects of the dot-com crash.

And here is the important point: throughout this process, computing lecturers and professional trainers continually increased the content and level of their courses, never themselves having experienced the difficulty of having to learn a modern, advanced computer language *from nothing*. At the same time, government policy dictated that more students should be taught, necessarily inducing an average reduction in intake ability. So the subject has become harder and harder to teach, while the students have become less and less able. In particular, partly because of the lamentable state of the maths syllabus in schools, many of our students do not have the mathematical skills, nor the generic ability to abstract, that maths can give.

This is where we find ourselves. It is crucial, therefore, that we reassess how we teach and assess programming, this most central and most difficult of computer-science activities. And it is worth mentioning that the problem of how to teach programming effectively is a hot topic in every computing department in the country—even those that attract straight-A students. An interesting question to ask is: “Does Goldsmiths have anything to make it different from other institutions, that might help?”. We believe it does: creative practice.

### 3.2 Programming as creative practice

One important point that is often missed by computer scientists and others alike is that programming is an intensely creative activity, probably best compared with design. One aim, in teaching the subject, is surely to give students the confidence and ability to create (within the bounds of a specification), while another is to give them the tools to do so. The problem we have to solve is that it is very difficult to do one without the other.

Programming is probably best thought of as an advanced craft. There are tools and techniques (programming languages, development environments, data structures and algorithms), which need to be understood; there are materials (data) which need to be understood, in a different way; and there are methods and techniques for putting all this together into an approach to creating a finished product. The creative aspect of it all has been rather lost amidst the “professionalisation” of the field, led by industry and the British Computer Society, whose attitude is rather one of bean-counting, and, more problematically, of viewing computer scientists as “techies”, without an aesthetic viewpoint.

The particular, and difficult, way that programming is different from, say, sculpture, is that nearly all the activity involved is abstract, and requires the ability to imagine not just objects, but also processes, at extreme levels of detail. It is this that seems to be the primary problem for our students: because they find it hard to see these things through their mind’s eye, they find it hard to see the point of the tools and techniques, and quickly become dismayed with the whole thing. While it is necessary to acknowledge

---

<sup>3</sup>Or that was the intention, anyway.

that some students simply don't have the right ways of thinking ever to grasp programming to a highly competent level (and for whom, therefore, alternative useful learning provision must be made), we believe it is possible to encourage those who do have the fundamental aptitude to develop it, mostly through carefully controlled practice, as for a musical instrument—but most importantly, *creative* practice, and not merely rote learning.

We are not the first institution to propose this kind of approach (Teague and Roe, 2007; Price, 2007; Routledge et al., 2007; Griffiths et al., 2007); indeed, some proposals go much further than merely changing one's way of thinking!

### 3.3 The Current State of Affairs

For the past two years, the Department of Computing has been running a new way of teaching programming in first year, based on the Oxbridge tutorial technique. Part of the motivation for this is that programming is a creative activity (see above) and creative activities are often best taught in a sort of apprenticeship model, rather than by conventional lecturing, assessing and then marking. At the same time, we introduced a series of “challenges” (designed by Sebastian Danicic), for coursework, through which students should proceed, one at a time, throughout the year. These challenges are assessed as simply acceptable or not: there is no gradation of marking, but they are given detailed feedback in tutorial. Thus, the students have a clear path to success by means of their practical ability, for 20% of the course mark. The remainder is by written exam.

There are two problems. The first is that this practical learning seems not to be accompanied by theoretical understanding: pass rates and averages for the first year programming course are extremely low, even for those students who do well practically. There is a considerable amount of evidence that this mismatch is partly because students lean too hard on each other when they work together: in many cases, this is manifest as plagiarism, with work copied word for word. However, this author believes that the larger part of the problem (which gives rise to despair and hence cheating in students) is due to the nature of the material being taught<sup>4</sup>, which is abstract in the extreme, and really quite advanced for first year students.

### 3.4 Assessing programming

All of this raises important questions not just about how to teach programming, but about how to assess it. It is certainly true that there are absolute senses in which a program can be wrong (syntax or logic errors) but there are equally many different ways in which any program of more than trivial complexity can be right, and there are notions such as elegance which combine to give a genuine aesthetic, just as in design. Ideally, we would like to foster these aesthetic values in our students. This suggests that at some stage we should be assessing their work and giving them feedback on its aesthetic quality and not just on whether or not it is error-free, which is the norm in most computer science departments; while some of our tutors do this, it is not clear that all do. And it is far from clear that students view programming as a creative activity; we should encourage them to think this way. If we can do this, we may even get them excited about it.

## 4 A New Approach

### 4.1 Curriculum

We propose, therefore, to update not only assessment styles, but also the delivery of the programming curriculum. The principles are:

- Steady on-going development of students' programming skills throughout their programme, with the target that computer science students should have the skills necessary for individual projects by the beginning of 3rd year. This is in contrast to current attempts (here and elsewhere) to teach the whole thing in 1st year.

---

<sup>4</sup>And it is appropriate at this point to emphasise that absolutely no criticism is intended of those teaching it.

- Use of multiple teaching and assessment styles, to reach as many students (and their individual learning styles) as possible. This includes using multiple programming platforms, allowing students to see for themselves what is common and what is different.
- Focus on activity-based learning, with a graduated scheme of abstraction (e.g., in first term, very little abstraction, lots of low-level practice; in second term, abstraction based on real-world objects; in third term, theoretical models of abstract processes and their links with actual programs).
- Integration of supporting theory (in particular, maths) with programming practice, helping students understand why they are learning things. Focus this integration around real-world problems, not artificial mathematical ones.
- Ensure that assessment is not about “can” and “can’t”, but about how well things are done, with feedback about what is good and what isn’t. This leads into plans for a common first year with end-of-year progression hurdles based on programming development, channelling students into programmes appropriate to them.

The upshot of these is that full-on Java programming should be introduced in second year, and not at all in first. In first year, instead, we should use mixed approaches to the various different aspects of the programming process outlined above. One current proposal is to use the Alice animation system to encourage skills of analysis and specification-understanding, and to introduce basic programming constructs in a way which connects directly to students’ experience in the real world. Another is to use one of several small hardware units currently on the market which allow students to plug electronic components into their PCs and program them to behave in ways which are directly visible. Thus, an element of “instant gratification” is evident: these systems themselves are both simple and sophisticated enough to give students constructive feedback on what is happening.

## 4.2 Delivery Style

Given that this proposal entails deferring abstract technical thinking to 2nd year, the question is immediately raised of whether a conventional delivery style, with abstract lectures, is appropriate, or whether learning by guided discovery, in a practical context, would be more appropriate. Feedback suggests that, where there is tutor engagement, small-group teaching is effective, but students also want larger scale practical classes. Interestingly, they are also requesting more compulsory sessions in first year, perhaps indicating that they understand their own inexperience at managing their own learning. Experience in our current second-year programming module, where large-group, mixed-mode teaching is employed, is that students engage much more here than with more traditional methods. This in itself presents new opportunities for assessment (particularly formative), with students assessing each other’s work.

We are surrounded by good practice in assessing creative work; we need to look at and learn from practice around the College, and combine it with methods we need for the technical aspects of what we teach.

## 4.3 Assessment

Assessment will be in several styles, depending on the nature of the material. Currently, we anticipate there will be three categories of activity:

**Mathematics** It is almost impossible to program without maths, and an abstract understanding of maths does need to be examined. However, maths will be taught in context of application: each time concepts are needed, they will be supplied (or, in fact, the curriculum will be planned in an integrated way, with activities designed to introduce these concepts).

This will be reinforced by regular on-line quizzes, which will be automatically assessed, giving immediate feedback. These will not count to a final mark, but students must complete all of them to pass. There will, necessarily, be a summative theoretical exam at the end of each module. By this time, explicit coursework and formative quizzes will have helped students abstract mathematical methods from the applications with which they are taught.

**Specification understanding and problem analysis** The Alice animation system, and another slightly less friendly/more advanced one called Squeak are ideal for learning by doing. We can begin with scripts, inviting students to become film directors. Those scripts can become progressively less prescriptive and more open-ended; there can be a choice of subject too, since Alice has worlds suitable for all interests: from skaters and furry animals to lunar modules. Advanced users can even design their own worlds.

Students will be asked to self-assess, and to assess each other's work, on the basis of a standard mark-sheet. They will keep a log of their progress, as they get better at the tasks. We may introduce a system of awards, like the coloured belts in Judo, to promote (healthy, constructive) competition, and we will certainly introduce at least one competition in which the winner(s) are elected by the student cohort. Ultimately, there will also be staff assessment of students' work, of course, which will be both in terms of correctness according to specification, and of creative quality (originality, though, is not normally an issue here).

**Programming primitives and their application** Alice and Squeak incorporate all the control structures of advanced, object-oriented languages, but absolve students of the need to learn complex and fiddly syntax to use them. Alice is now in use in 10% of US universities that teach CS. What is more, we can teach multimedia applications, an important area for Goldsmiths computing, at the same time. Thus, students learn difficult and abstract concepts by experience; it is subsequently much easier to transfer them truly into the abstract. Later in the course, we will introduce the Arduino, a small electronics board which plugs into a computer. With the addition of a few small, cheap components, an Arduino can be made to do some quite interesting things—particularly multimedia: light shows, sound synthesis, and so on—and it is programmed in a simple subset of the C programming language, a standard which is likely to remain so for the foreseeable future. Thus, the students graduate from visual programming to “real” programming while maintaining a grounding of what they do in the real world.

Assessment here needs to be more traditional, but is likely to be similar to our current system: students must complete certain tasks, and simply pass a hurdle for doing so. A final mark is calculated from how many hurdles they pass. Again, a system of awards as in Judo, may help here.

As students move through their programme, we will aim to increase the open-ended-ness of their course-work, and, where appropriate, to apply creative practice assessment styles, encouraging them to think around problems and to be confident in their own creativity.

## 5 Threats

All of this presupposes that College facilities will be available: for example, a very large room will be needed, for numerous hours per week, for the programming activity discussed above. It is imperative that we receive the active support of Estates in this plan.

## 6 Action Plan

There is much work to be done before this plan is installed in October 2009. The next steps are as follows:

**08/08** Install and test software and hardware on student laptop; ensure familiarity of development team with facilities and approaches. Begin to draft detail of course structure.

**09/08** Develop simple, short introductory course; invite new undergraduates to take part in a small study of its efficacy, paying them if appropriate.

**10/08** Finalise first draft of course structure. Identify detailed materials requirements. Begin development.

**to 03/09** Develop materials and delivery methods for new courses.

**to 09/09** Further development, enhancement, slippage. Begin planning 2nd year curriculum.

**10/09** Delivery. Continual introspection and modification.

**06/10** Review and further modification if need be. Begin detailed work on 2nd year curriculum.

## **Acknowledgements**

Thanks to Tim Blackwell, Sebastian Danicic, Sarah Raugas, Christophe Rhodes, Daniel Stamate and Marian Ursu, all of whom have contributed enthusiastically to this debate.

## **References**

Griffiths, R., Holland, S., and Edwards, M. (2007). Sense before syntax: a path to a deeper understanding of objects. *ITALICS*, 6(4).

Price, C. B. (2007). From kandinsky to java (the use of 20th century abstract art in learning programming). *ITALICS*, 6(4).

Routledge, G., Aminaei, A., and Benachour, P. (2007). Developing understanding of programming principles using flash actionscript. *ITALICS*, 6(4).

Teague, D. and Roe, P. (2007). Learning to program: Going pair-shaped. *ITALICS*, 6(4).