Higher Education Academy

TRANSFORMING TEACHING
INSPIRING LEARNING

Innovative pedagogies series: Innovative pedagogical practices in the craft of Computing

Dr Tom Crick, Director of undergraduate studies, Cardiff Metropolitan University

Professor James H. Davenport, Hebron and Medlock Professor of Information Technology, University of Bath

Alan Hayes, Director of teaching, University of Bath

Contents

Introduction and theoretical context

> **Knowledge without practice is useless. Practice without knowledge is dangerous** *(Confucius)*

Computer programming, the art of actually instructing a computer to do what one wants, is fundamentally a practical skill. How does one teach this practical skill in a university setting, to students who may not be initially motivated to acquire it, and who may have a variety of past experience, or none at all? How does one ensure that they progress to the rest of their studies with a firm background in programming, so that programming difficulties do not impede their other learning? Here National Teaching Fellows from two different institutions describe how they have addressed these challenges. Above all, they key is to *recognise* that Computer Programming is a practical subject, and needs to be taught and assessed as such. A model we use is that of 'apprenticeship' (Vihavainen *et al.* 2011), where the students learn how to do by seeing it done, and by being guided in the doing.

This model is related to, but different from, the "lab-first" approach discussed in Hazzan *et al.* (2011). Their own analysis (section 8.3) is worth considering:

> **The lab-first approach has both advantages and disadvantages. On the one hand, in the spirit of constructivism, its main advantage is expressed by the active experience learners get in the computer lab, which in turn, establishes foundations based on which learners construct their mental image of the said topic; on the other hand, the lab-first teaching approach involves some insecurity feelings expressed both by the Computer Science teacher and the learners.** *(Hazzan et al. 2011, p. 149)*

In our case, insecurity among the learners is a real concern, and manifests itself very clearly among students who miss the first two weeks of term and then start the course and its labs, even with the support of videos of past sessions.

Constructivists interpret student learning as the development of personalised knowledge frameworks that are continually refined. According to this theory, to learn, a student must actively construct knowledge, rather than simply absorbing it from textbooks and lectures (Mayer *et al.* 1990). Students develop their own self-constructed rules, or "alternative frameworks" (Ben-Ari 2001). For example, in programming, these alternative frameworks "naturally occur as part of the transfer and linking process" (Clancy 2004, p.87); they represent the prior knowledge essential to the construction of new knowledge. When learning, the student modifies or expands his or her framework in order to incorporate new knowledge. We have also had an interest in threshold concepts (Land *et al.* 2008; Meyer *et al.* 2010) - as a subset of the core concepts in a discipline - for Computing, and more specifically, programming (Khalife 2006). These are the building blocks that must be understood. In addition, they must be:

- *transformative:* they change the way a student looks at things in the discipline;
- *integrative:* they tie together concepts in ways that were previously unknown to the student;
- *irreversible:* they are difficult for the student to un-learn;
- *potentially troublesome for students:* they are conceptually difficult, alien, and/or counter-intuitive;
- *often boundary markers:* they indicate the limits of a conceptual area or the discipline itself.

Students who have mastered these threshold concepts have, at least in part, "crossed over from being outsiders to belonging to the field they are studying" (Eckerdal *et al.* 2006, p. 103), although there is some dispute on how they apply to Computer Science (Boustedt *et al.* 2007).

So what precisely is the 'field they are studying'? There are two key external reference points available to those who design Computing-based curricula, adding both national and international context. These are the Quality Assurance Agency for Higher Education in the UK (QAA) Subject Benchmark Statement for Computing

(2007) and the Association for Computing Machinery's (ACM)[1] Computer Science Curriculum 2013. At the time of writing this paper the benchmark was undergoing review, out for final consultation with a view to being published in late 2015. The benchmark revision process closely aligned itself to the ACM curricula recommendations - primarily the Computer Science 2013 Curriculum (ACM 2013), but also referencing the Software Engineering 2014 Curriculum Guidelines (ACM 2014) and the Information Systems 2010 Curriculum Update (ACM 2010). It recognised that computational and algorithmic thinking is central to the discipline; programming is the means by which this thinking is expressed. Consequently the art of programming is complex. It is the final step in an algorithmic approach to solving a given problem. The teaching of programming, therefore, requires much more than the semantic and syntactical aspects of a programming language.

Programming is a hard craft to master and its teaching is challenging. An apprentice model, where students learn their craft from a master, is an approach that can lead to improved student engagement (Astrachan and Reed 1994; Vihavainen *et al.* 2011). Although traditionally applied to physical and vocational skills, the apprenticeship model can also be applied to the acquisition of cognitive skills such as those required for programming. In this context, the master is required to focus on the programming process and demonstrates it through writing, debugging and running 'live' programs. This takes place while being observed by the student cohort. Scaffolding is provided through the provision of regular practical exercises with good quality formative feedback.

Why do universities teach a craft like programming?

We do not teach handwriting, or document composition: why, therefore, should we teach this means of expressing our thoughts? The simple answer is that, in the UK currently, computer departments have to teach it as students arrive with zero or minimal knowledge of it.

Eric Schmidt, Executive Chairman of Google, speaking at the MacTaggart Lecture at the 2011 Edinburgh International TV Festival, said that the UK, having invented the computer, was "throwing away your great computer heritage" by failing to teach programming in schools. "I was flabbergasted to learn that today Computer Science isn't even taught as standard in UK schools," he said. "Your IT curriculum focuses on teaching how to use software, but gives no insight into how it's made" (Schmidt 2011). While this is being addressed through the work of Computing At School[2] (CAS) alongside substantive curriculum and qualifications reform across the four nations of the UK (Royal Society 2012b; Crick and Sentance 2011; Brown *et al.* 2013; Brown *et al.* 2014) - each at various stages, with a new Computing curriculum in England (Department for Education 2013) that started in September 2014 - students presenting themselves for tertiary education in the field of Computer Science possess very little experience of Computer Programming. It consequently falls to the Computer Science education community within UK higher education providers (HEPs) to consider how best to introduce and develop the Computer Programming skill set required to implement the computational and algorithmic thinking required to address many of the classical problems in traditional Computer Science programmes.

In this paper we look at two complementary examples of how two different universities have approached these challenges. The University of Bath was looking to teach Computing to highly focused Mathematics

[1] ACM is the US-based, but in practice worldwide, learned society for Computing. These curricula are the result of a joint ACM and IEEE (Institute of Electrical and Electronic Engineers) task force.

[2] See, for example: **http://www.computingatschool.org.uk**

students, while Cardiff Metropolitan University offers degrees in Computing, Software Engineering and Business Information Systems to a broad range of students from diverse educational backgrounds.

One key principle unifies our distinct approaches: our belief that programming (as opposed to, say analysis of algorithms, a closely related theoretical skill) is fundamentally a craft that needs immersion and practice (Fincher, 1999). This is related to both general and discipline-specific pedagogy, as follows. Kirschner *et al.* write:

> **based on our current knowledge of human cognitive architecture, minimally guided instruction is likely to be ineffective. The past half-century of empirical research on this issue has provided overwhelming and unambiguous evidence that minimal guidance during instruction is significantly less effective and efficient than guidance specifically designed to support the cognitive processing necessary for learning.** *(Kirschner et al. 2006, p. 3)*

Connected to this general point is a curious paradox in the teaching of programming: we teach students to *write* programs, and not particularly to *read* them (essentially: code literacy), whereas in natural languages, be it the mother tongue or foreign language instruction, we teach students to read before we teach them to write. This is well brought-out by Stroustrup, who says (of C++, but the same is true of any programming language) "How does one write good programs in C++?"

> **There are two answers: "Know what you want to say" and "Practice: imitate good writing"** *(Stroustrup 2013, p. 18).*

There is one difficult point here: the lecturer's own code has to be presented as one (good) way of solving the problem, but not the (only) - or indeed optimal way - of solving the problem. This can be helped by pointing out that even Shakespeare seems to have changed his mind:

> **There are approximately 230 lines in Q2 [the second Quarto edition] that are not in F [the first Folio edition], and about 70 lines in F that do not appear in Q2. No one would argue that any of these 300 lines is not by Shakespeare.** *(Ward 1992, p. 289).*

Case study: University of Bath (James H. Davenport)

Introduction

The University of Bath's Department of Mathematical Sciences[3] has taught Computing to first year students since Mathematics was established at Bath at the start of the university's life (1966). Initially this was taught in the venerable programming language Fortran, to students on a range of Mathematics degrees, including Mathematics and Computing. At that time, the academic "Computing Group" was part of the School of Mathematics, and there was no separate Computer Science degree. Nearly two thirds of the students in the school did sandwich degrees, i.e. a year in industry (construed broadly, and until 1992 this could include teaching) between the second and final years of a BSc. In 1984, as a new lecturer who had used C in anger, I proposed, and led the process, of moving to C, not least because Fortran (at the time!) had no support for recursion or data structures. Gradually Computer Science degrees were established, and the Computer Group became a separate Department of Computer Science[4] in 2001. Computing skills were so embedded into the Mathematics curriculum, and needed by the sandwich students, that it was unthinkable at Bath not to have Computing in the Year one Mathematical Sciences syllabus, and the Computer Science Department

[3] See: **http://www.bath.ac.uk/math-sci/**

[4] See: **http://www.bath.ac.uk/comp-sci/**

provided a ‘service course’ (one semester, one module out of five) in Java Programming to the Mathematical Sciences Department, which, unlike C, did support object-oriented programming. This freestanding course was not without its problems: students repeatedly queried the relevance of this to the rest of the syllabus, not without reason. A consequence of this was a lack of student engagement with the course, and a relatively high failure/condone rate.

Starting in October 2009, Mathematical Sciences launched a significantly revised curriculum, based on more than a year’s worth of consideration in which I had been involved. One fundamental aim was to reverse the ‘module drift’ and re-think a holistic course rather than just a collection of modules. This re-design therefore cancelled the stand-alone Java Programming course. Its role in teaching programming was taken over by an all-year, one module out of five, unit called XX10190: *Programming and Discrete Mathematics* (described in more detail in Davenport *et al.* 2014). As ‘programming’ was 50% of a unit twice as long, there was notionally no change in the amount of programming in the syllabus. The other 50% of the syllabus was (relevant to Computing) Mathematics, notably the rigorous treatment of induction. The course was team taught, with, in general, one hour of programming and one hour of Mathematics being lectured each week, as well as a combined one-hour problems class. This is ‘teaching at scale’: the 2015/16 cohort is just over 330 students, with the A-level students typically having A*AA grades.

How this practice evolved

My research interests lie squarely across the Computing/Mathematics boundary. In the split of departments I ended up in Computer Science, but am still a member of Mathematical Sciences as well. Hence I was the obvious choice to lead the ‘Programming’ element of the new course. Immediately before this course was launched, I had been on sabbatical at the University of Waterloo. This was a research sabbatical, justified at the time on common research interests (and which indeed led to a large Engineering and Physical Sciences Research Council (EPSRC) grant, and numerous journals and conference papers, though mostly with the University of Western Ontario than with Waterloo). At Waterloo, I was working in a Faculty of Mathematics where there are 1500 students/year, and, though not teaching myself, I could appreciate the practicalities of teaching at scale in a way that, only physical presence, and hearing common room discussions etc., could have provided.

One strategic decision was taken almost by default. The course was resourced as a programming unit, i.e. with every student getting one hour of supervised laboratory time per week. This actually meant that there was twice as much practical time than in the previous regime. Having taught programming at both the ‘normal’ (at least for Bath) rate of two hours of lectures and one of laboratory and at this ‘parity’ rate of one hour of programming lecture and one hour of laboratory per week, I am convinced that the parity rate is more effective. By way of comparison, for its Computer Science students, where Programming is a double (24 CAT/12 ECTS credits) module, we provide three hours of lectures and two of laboratories/week: essentially a halfway house.

Another strategic decision was the assessment method for the course. Programming is a practical subject, and should be assessed as such. The high-level view of the assessment is 50% examination, 50% coursework. The detailed breakdown is slightly more subtle: the Mathematics component is 38% examination and 12% class test (with computers), while the programming component is 38% coursework and 12% examination. As of writing (July 2015) the course team is considering changes to the detailed breakdown, but the 50:50 split will remain.

One important tactical decision was focused upon which programming language to teach. Here Bath made the, rather unconventional for a Computing course, choice of MATLAB. MATLAB was already used in the Numerical Analysis courses later in the curriculum (where it is indeed the obvious choice), and was among the most common languages used by sandwich students. The department hoped that this would address some of the ‘relevance’ questions, but we were under no illusions that relevance would still be a key question (and indeed it still crops up at Staff-Student Liaison Committee, though far less vociferously than before). One

unexpected bonus of this decision was the knock-on effect in Statistics teaching. The initial Statistics module in Semester two of Year one overlaps with the second half of the *Programming and Discrete Mathematics* course, and is taught using the R language. As far as the students are concerned, R and MATLAB are sufficiently similar that it suffices to give them a one page 'differences between R and MATLAB' document, and the Statistics lecturer can concentrate on teaching statistics.

It is worthwhile considering the learning outcomes of the module (from the catalogue, our numbering):

> **After taking this unit, the student should be able to:**
>
> 1. **apply the basic principles of programming in studying problems in discrete Mathematics;**
> 2. **make proper use of data structures in the applications context;**
> 3. **demonstrate understanding of a range of mathematical topics which relate to computation, such as modular arithmetic, elementary graph theory and elementary computational number theory and their applications;**
> 4. **analyse the complexity of simple algorithms;**
> 5. **explain the use of some famous algorithms such as the Fast Fourier Transform;**
> 6. **use the MATLAB programming environment.**

Items one, two and six6 are directly practical skills. This is the rationale for the 50% coursework assessment mentioned above, and relates to my opening statement to the course: "My aim is to show you how to program, my and the laboratory tutors' aim is to help you with your programming, and your aim ought to be to learn programming by doing it". I have used this statement since 2009, but, since encountering Vihavainen *et al.* (2011), I have realised that this is really an apprenticeship model, with the students as 'apprentices', the tutors as 'journeymen', and the lecturer as the 'master craftsman'. There is a strong emphasis on 'learning by doing'. For example, in the first lecture I write in front of them (and definitely do not produce a pre-written one out of the hat) a recursive factorial program, and the first exercise is to adapt this into a Fibonacci number program. In fact, this "writing" process is iterative.

The first program I produce tends to look like (this is the 2014-15 version, but each year it is written from scratch in front of the students) the following.

```
function [fact] = Factorial(n)
%Factorial Computes the factorial function
if ( n == 0)
        fact = 1;
else
        fact = n * Factorial(n-1);
end
end
```

This works fine for non-negative integers, and I demonstrate this. Then I try a negative integer, for example, -1, and the program counts down to negative infinity: -1, -2, -3 ... -500 (at which point MATLAB's recursion limit is exceeded). I therefore add a check for negative n, test it, and then try `factorial(1.5)`. This tries to compute `factorial(0.5)`, then `factorial(-0.5)`, at which point it hits the previously-inserted check (and in fact behaves bizarrely for MATLAB internal reasons, which I explain). I believe that this makes points about the importance of testing, and of iterative development, that no amount of abstract lectures could do.

The laboratory sessions take place in five one-hour sessions at the end of the week (after the lectures and formal problem classes). They are held in 75-seater laboratories (arranged as five rows of 15 machines each), and the students are allocated to a specific row of 15 within that. Similarly, there are five tutors, each assigned to a specific row. Ideally, the same tutor stays with the same group of 15 all year. This firm allocation of students to tutors, and the briefing to tutors that they are responsible for the learning of their allocated

students, means that tutors do occasionally raise concerns about specific students with me in a way that tutors in a floating pool would not.

The role of practical work

It has been the authors' gut feeling for many years that weekly practice, and frequent assessment, are important in getting students into the habit of programming. This has recently been borne out by Willman *et al.* (2015), who state:

> **Students with high absolute submission counts during tutorials tend to significantly more often get a good grade from the course than those who have low absolute submission counts.**
> *(Willman et al. 2015)*

It is all very well having weekly laboratory sessions, but how, in a university context, does one make students use them? Our answer to this is that, in the weeks where there is no formal coursework being done, and especially in the first few weeks, we set formative weekly exercises. These are graded pass/fail, and are known as 'Tickables', since the laboratory tutor ticks them in the course of each laboratory session. The tutors are instructed that a certain amount of 'coaching' is permissible here (where it would not be in assessed coursework) and that one of the aims, again particularly in the first weeks, is to instil confidence in the students.

These exercises are sequential, building both on the lectures and on previous ones. The first 'Tickable' builds on the factorial program written in lectures (as shown above), and asks the students, based on the lecturer's program (which is supplied in the Learning Environment after the lecture), to write a Fibonacci number program, that is, F(n)=F(n-1)+F(n-2) with suitable base conditions. The students are also asked to reason about the running time of this program, which builds on the discrete Mathematics component of the course. The second 'Tickable' extends this to the matrix formulation, and so on. Each subsequent week, the students can build on their solution from the previous week, or on the lecturer's solution.

Simple pass/fail marking by the tutor seems easy from the lecturer's point of view, but the experienced academic will ask about:

- appeals from the tutor's decision not to tick;
- the incentives for the students to do the work;
- illnesses and other absences, and requests for extensions, which get in the way of providing the solution in a timely manner for next week's 'Tickable'.

The XX10190 solution to these, potentially very important, questions are as follows:

- I attend all the laboratories for the first 'Tickable', and resolve any queries on the spot (also helping the new tutors and giving them confidence), possibly announcing any adjudications if this seems appropriate. Thereafter, over a period of six years with 250 students per year doing ten more 'Tickables' each, I have not had a single appeal against the justice of ticking. (There have been appeals against the misrecording of ticks, and the tutors can be fallible here, and forget to copy a hand-written record into the virtual learning environment (VLE).)
- The incentive is that, if the student does not get 80% of the ticks (of which there are generally 15) for the course, then the assessed coursework mark is reduced pro rata. In practice, this is a rare event, as nearly all students do get over 80%, and those that undershoot do so drastically, fail the coursework anyway, and are generally those student who do not engage at all with the course. What does occasionally happen is that a student misses the first two laboratories (which the ticking process will record), is reported to the personal tutor (the department office gives me a list of personal tutors arranged by student computer username) and then starts attending seriously, at which point the missing ticks are condoned.
- Minor illnesses and absences are explicitly (this is explained as part of the course briefing, and repeated when requests are made) allowed for by the 80% rule, in that a student can miss three without penalty. More would indicate a more serious condition, which should be addressed programme-wide rather than

just in this unit. One case that this does not cover is that of sport players who may miss several laboratories, but this is known in advance, and the rule is 'submit in advance to your tutor by e-mail'. In particular, no extensions are given for 'Tickables' (unlike assessed coursework), and so the weekly rhythm of 'lecturer demonstrates; students work a similar example; lecturer shows his solution, uses it to lead into next demonstration' is not disrupted.

The journeyman

Just as in a traditional craft, the 'journeymen' have also to learn, and to be managed. I saw the importance of this at Waterloo, where the programming lecturer was supported by an entire cast of instructional support assistants (ISAs), instructional apprentices (IAs) and instructional support co-ordinators (ISCs) (essentially a full-time role recruiting and rostering the ISAs and IAs).

The tutors at Bath are generally PhD students from the two departments of Mathematical Sciences and Computer Science. One tutor in each department is formally designated as the senior tutor, with two sets of responsibilities:

- to the lecturer: to inform them when things are going wrong, and to ensure that the lecturer is aware of generic difficulties;
- to the other tutors, to act as a less formal source of advice and support than going to the lecturer.

These two senior tutors, and other experienced tutors, are consulted by the lecturer about changes, and often used to test new assignments or exercises. It should be noted that three of the seven authors of (Davenport *et al.* 2014) are or were senior tutors. I am often asked by tutors to write references for their teaching abilities as they move on to other jobs, and gladly provide them. One tutor returned to a lecturing post in her native Thailand and occasionally asks me questions on teaching practice, as well as using some of the ideas (such as the 80% rule) in her own university.

I find it important to remember a simple piece of arithmetic: in a given week I deliver about 1.5 hours of teaching (one lecture plus half a problem class), while the laboratory tutors deliver 25 between them. Hence, time spent writing briefing material for the tutors is as valuable as time spent preparing lectures. Every exercise is issued with support material for the tutors: sometimes 'what to explain' and sometimes 'what not to explain - they have to figure that one out'. This is necessary to ensure a uniform experience across the 25 tutorial groups.

On a more immediate level, I ensure that at least one, and generally two, of the five tutors in the lab for any hour are those who have done this tutoring in previous years. This is particularly important at the start of the year, when new tutors may not be that familiar with MATLAB, and have yet to learn to spot the baffling blunders that beginners can make.

To book or not to book?

An interesting question is whether there should be a course textbook. It is easy to make the argument that there is so much material online, both formal (and a commercial language like MATLAB clearly has a great deal) and informal, that a book is unnecessary and will not be consulted. On the other hand, I learned at Waterloo that lecturers attached substantial importance to appropriate accompanying textbooks, and over the decades have continued to use, adapt, and occasionally write textbooks (for example Cress *et al.* 1970, which I used while a tutor at Cambridge). One other thing I had observed at Waterloo is that they made significant use of custom books, that is, versions of a book, or even multiple books, tailored to the course at hand. I had heard of such things, but assumed that these were the domains of vast courses with multi-year enrolments with a thousand students per year. I was assured not, and made a few enquiries by e-mail to UK publishers, who, put bluntly, practically bit my hand off when I mentioned a course of first years (these are the ones who buy books) with an enrolment of over 200. In fact, I have since learned that the lower limit is

about 80, and is dropping over time as publishers' technology advances. The idea of a custom book appealed to Bath, since we were teaching what seemed to be a pretty novel course with no single textbook.

Discrete Mathematics book for the American market are not in short supply: the real problem was the MATLAB book. There are a large number of books extolling the virtues of MATLAB's numerous toolboxes etc., but books teaching MATLAB as a programming vehicle seemed to be in short supply. In the end I settled on Chapman (2009) without great difficulty, as it was the only book that came close to the requirement of teaching MATLAB programming as the core concept. Selecting this forced us to look at Discrete Mathematics books from the same publisher, and we chose Epp (2004). We took about 98% of Chapman (all but the answers) and 55% of Epp, which actually resulted in slightly more pages of Epp than of Chapman, and 856 pages in all.

In 2010 there was also 20 pages of our own material. This edition was used unchanged in 2011. In 2012, we went to a new book by Chapman, and a new edition by Epp. Both were substantially larger, but we had gained confidence in our custom book skills, and took slightly fewer pages of Chapman (65%), slightly more but a smaller proportion from Epp, and 26 pages of our own, totalling 862 pages.

In 2014 we took the braver decision to drop Epp in favour of our own material, which was 119 pages, for a total book length of 496 pages. In 2015 we will be moving to another new book by Chapman, tracking the evolution of MATLAB. In fact, this will allow us to drop about six pages of our own material, as we felt we had to track this evolution in 2014, before Chapman had moved.

Case study: Cardiff Metropolitan University (Tom Crick)

Introduction

Cardiff Metropolitan University (formerly, the University of Wales Institute, Cardiff or UWIC) was formed through the merger of a number of constituent colleges in Cardiff, joining the University of Wales as an autonomous body in 1992. It consists of five academic schools: Cardiff School of Art and Design, Cardiff School of Education, Cardiff School of Health Sciences, Cardiff School of Management and Cardiff School of Sport. The Department of Computing and Information Systems[5], formed as part of a restructuring in 2009, sits within the School of Management, with a strong undergraduate and taught postgraduate portfolio. It currently offers the following three or four year degree programmes: Computing, Software Engineering and Business Information Systems, as well as a franchised Foundation Degree top-up programme with Bridgend College in Management and Technology, all supported by a Foundation Year option leading onto these programmes. The overall undergraduate cohort size is *c.*250 students, generally evenly distributed across the three main programmes, with overall high student satisfaction in the National Student Survey (NSS) over the past five years. While the majority of students follow the three year degree programme, an increasing number (*c.*15%) of students are opting for the four year sandwich programme, obtaining high profile internships in competitive national schemes. The undergraduate degree portfolio is accredited by BCS, The Chartered Institute for IT[6]. In line with national trends for Computer Science (and perhaps more broadly, certain STEM disciplines), the undergraduate portfolio attracts a predominantly male demographic, with the primary age for entry onto the programmes under 21 years of age; students who have followed a traditional educational pathway. A majority of the undergraduate students are Welsh-domiciled, with increasing

[5] See: **http://www.cardiffmet.ac.uk/computing**

[6] See: **http://www.bcs.org**

numbers of European and international applicants. The typical qualifications profile of incoming students is mixed, consisting of A-Levels, BTECs and the Welsh Baccalaureate; the average entry tariff over recent admissions cycles has been approximately 300 UCAS points. While a majority of students have ICT or technology-related Level 3 qualifications, few students have A-Level (or equivalent) Mathematics, normally only GCSE grade C or higher. In light of recent curriculum and qualifications reform, we are also starting to see increasing numbers of applicants with GCSEs and A-Levels in Computer Science.

The current instances of the three main undergraduate programmes were validated in 2010 and 2011, with a periodic review scheduled for early 2016. A primary theme as the programmes have evolved since the last validations - and as we move forward into the 2016 revalidation - has been on developing mathematical foundations and transferable computational thinking skills, while maintaining strong practical programming and Software Engineering skills and industrial relevance and currency. A key social and professional focus has been on developing a culture of software carpentry, codemanship and software sustainability - developing useful and usable software artefacts.

How this practice evolved

I am currently Director of Undergraduate Studies in the Department of Computing and Information Systems at Cardiff Metropolitan University, having previously spent a number of years in the Department of Computer Science at the University of Bath (as an undergraduate, postgraduate research student and as a postdoctoral researcher). My research interests are naturally interdisciplinary: identifying large-scale data-driven and computationally-intensive problem domains, as well as targeting wider societal/policy impact: data science, analytics, reproducibility, optimisation, high performance computing, intelligent systems and Computer Science education. My research-informed teaching is thus focused on related underpinning themes: mathematical foundations of Computer Science, data literacy, computational thinking and principles of programming.

I have overall academic and pastoral leadership of *c.*250 students across five undergraduate programmes, managing a programme team of more than 15 members of academic staff. Over the past six years I have seen multiple cohorts progress through and graduate from our programmes, moving into employment, further study and research. Since our previous undergraduate portfolio validation events in 2010 and 2011, there have been a number of significant changes to Computer Science and related digital/technology disciplines in the UK, from both an educational and economic perspective. We have seen a range of curriculum and qualifications reforms across the four nations of the UK (Brown *et al.* 2014), including a new Computing curriculum that replaced ICT in England in September 2014 (Department for Education 2014), as well as imminent reform in Wales (Crick and Moller 2015): I co-chaired the official review of the ICT curriculum in Wales in 2013 (Arthur *et al.* 2013), whose recommendations have been endorsed via the wider independent Curriculum for Wales review published in February 2015. Furthermore, there has been a national focus on the high value skills required to support the UK's digital, creative and knowledge economies, with a focus on the employability of Computer Science graduates[7], relevancy and currency of UK degree programmes (with, perhaps, some commentators conflating education and training), as well as the value of accreditation by learned societies and professional bodies[8]. From a university quality assurance

[7] See: **http://www.software.ac.uk/blog/2013-10-31-whats-wrong-computer-scientists**

[8] Two high profile reviews were initiated by the Department of Business, Innovation and Skills and HEFCE in February 2015: the Shadbolt review of Computer Science degree accreditation and graduate employability: **https://www.gov.uk/government/publications/computer-science-degree-accreditation-and-graduate-employability-shadbolt-review-terms-of-reference** and the wider Wakeham review of STEM degree provision and graduate employability: **https://www.gov.uk/government/publications/stem-degree-provision-and-graduate-employability-wakeham-review-terms-of-reference**

perspective, there is the imminent publication of a new QAA Subject Benchmark Statement for Computing (QAA 2015), updating the previous statement for undergraduate degree programmes published in 2007. We thus appear to be at the start of further upheaval of Computer Science education in the UK, with many institutions potentially seeing increased diversity in qualifications and experiences from its incoming undergraduates rather than reduced, as anticipated from the reforms; for example, due to school resourcing and qualifications availability across the UK, it would be possible to accept two prospective undergraduates, one of whom has successfully achieved both a GCSE and an A-Level in Computer Science, with the other having no Computing or technology-related qualifications at Level 3.

Thus, this case study presents a department and an undergraduate portfolio sitting at the cusp of these reforms, reflecting on how our undergraduate curriculum has evolved over the past five years to address specific national and institutional imperatives on graduate attributes and skills, employability, internationalisation, sustainability and entrepreneurship, as well as ensuring a rigorous Computing education and excellent student experience. Furthermore, it presents aims and intentions for the next five years in light of the upcoming periodic review of our entire undergraduate portfolio, contextualised by the upcoming new QAA Subject Benchmark Statement for Computing, reformed professional body accreditation, renewed focus on graduate employability, as well as addressing wider Welsh, UK and international educational and economic imperatives.

Interweaving theory, practice and professional issues

A primary aim for our undergraduate degree portfolio is to offer an appropriate combination of essential underpinning domain knowledge and theoretical foundations, coupled with industry-relevant practical skills and experience. This was a clear requirement from our programme validation events in 2010 and 2011, to ensure that all students have solid theoretical foundations, augmented by relevant professional practice and experience. There are, therefore, key anchor modules throughout the main three years of the degree programmes that provide a bridge from theory to practice and/or professional issues:

- BCO4008: Introduction to Information Systems (10 credits)
- BCO4012: Data Structure, Algorithms and Program Design (10 credits)
- BCO4013: Programming Fundamentals (10 credits)
- BCO4016: Mathematics for Computing (10 credits)
- BCO5001: Systems Development and Design (20 credits)
- BCO5023: Group Software Project (20 credits)
- BCO6003: Professional and Ethical Issues in Computing (20 credits)
- BCO6023: High Performance Computing (20 credits)

The anchor modules interact, intersect and provide the platform for cross-learning; for example, the Mathematics for Computing, Data Structures, Algorithms and Program Design and Programming Fundamentals modules are delivered concurrently and provide a structured plan for teaching principles of programming, mathematical foundations and algorithms and complexity, while developing introductory programming, program design/construction and software carpentry skills. High Performance Computing (HPC) is another good example: a Level 6 module that provides both substantive theoretical knowledge and understanding of parallel, distributed and HPC, as well as developing experience (and accessing training) in using cutting-edge infrastructure as part of the HPC Wales[9] project, along with engagement with a range of companies and industries using HPC.

[9] See: http://www.hpcwales.co.uk

Related to this, a further strength of our undergraduate programmes (as well as more broadly, all undergraduate programmes within the School of Management) is how they are supported and enhanced by compulsory work experience for all students (alongside sandwich year options). All students undertake a compulsory ten-credit work experience module at Level 5 to augment and ground their domain knowledge, skills and understanding in real world environments. In recent years, this has developed from one day a week placements, through to summer internships and sandwich years, into graduate positions.

Finally, to further strengthen (and make explicit) the link between formal academic study, thinking beyond the degree, professional practice and continuous professional development (CPD), we fully support and encourage engagement with professional societies, most notable BCS, The Chartered Institute for IT. The BCS is the professional body that champions the global IT profession, with over 70,000 members worldwide. Our undergraduate degree portfolio is accredited by the BCS and satisfies the academic requirements for professional membership and chartered qualification (after satisfying the necessary industrial experience). Over the past five years we have encouraged and supported our students to join the BCS as student members, to engage with local branch events and activities, as well as the student society becoming a BCS Student Chapter[10]. This provides access to activities and events for students (in particular, to gain knowledge and understanding of the wider IT industry), as well as to establish a professional network between the chapter, BCS member groups and relevant external bodies for the benefit of all.

Much of this work and associated impact on our students and undergraduate curriculum has been inspired by external engagement and policy work: in particular my work with schools, examination boards, governments and industry on Computer Science curriculum reform through Computing at School [11](CAS), as well as wider public engagement, science communication and widening participation activities in Wales and across the UK. For example, over the past three years I have received substantial funding through the Welsh Government's National Science Academy[12] grant scheme to work with schools, colleges and teachers in Wales to develop a network of excellence in teaching Computer Science, as well as identifying CPD and upskilling requirements alongside developing a pan-Wales community of practice. As part of these projects, we have used students as STEM Ambassadors to work with schools and colleges, developing programming, Computing and computational thinking skills, as well as changing perceptions of studying Computing and the potential career options.

Computational thinking and mathematical foundations

Computational thinking is taking an approach to analysing and solving problems, designing systems and understanding human behaviour that draws on concepts fundamental to Computing (Wing 2008). The interdisciplinary mindset and transferability of skills is key: it shares with mathematical thinking in the general ways in which we might approach solving a problem; it shares with Engineering thinking in the general ways in which we might approach designing and evaluating a large, complex system that operates within the constraints of the real world; it shares with scientific thinking in the general ways in which we might approach understanding computability, intelligence, the mind and human behaviour. All of these are fundamental aims of our undergraduate degree programmes.

More specifically, we explicitly try and develop higher-level computational thinking and problem solving skills before a significant focus on syntactic and semantic programming structures. While this abstraction can be conceptually difficult for many students transitioning from secondary education through to university, it

---

[10] See: **http://www.bcs.org/category/18176**

[11] See: **http://www.computingatschool.org.uk/**

[12] See: **http://gov.wales/topics/science-and-technology/science/nsa1/nsags-grant-scheme/?lang=en**

provides an opportunity to embed this at the start of their degree, intersecting across a number of the key anchor modules. Thus, with the emphasis on computational thinking as a problem solving process that includes a number of characteristics, such as logically ordering and analysing data and creating solutions using a series of ordered steps (algorithms), it starts to develop independent thinking and the ability to confidently deal with complexity and open-ended problems. While computational thinking is essential to the development of computer systems and applications, it can also be used to support problem solving across a number of other areas and disciplines, particularly Mathematics, Science, Engineering and the Humanities. Students who develop computational thinking skills begin to see a deeper relationship between disciplines as well as between their academic study and life outside (and beyond) the classroom.

This links to our focus on developing a deeper understanding of the mathematical foundations of the discipline, explicitly link to modelling and problem solving skills and ultimately, programming. While many students struggle with the concepts introduced in undergraduate Mathematics for Computing-type modules (for example, propositional calculus, set theory, number theory, etc.) - particularly if they have not studied A-Level Mathematics - grounding this knowledge and understanding with real world applications, as well as relevancy and currency for other topics they will study during their degree programme, provides a stronger platform for comprehension, retention and future success. In recognition of some of the challenges with mathematical and statistical competency for our programmes, in 2012, we were awarded £10,000 from *sigma*[13], under the National HE STEM Programme, to pump-prime the development of a Mathematics and Statistics Support Centre within the School of Management (primarily to support the Computing and Information Systems portfolio, alongside other STEM-related programmes across the university). The support centre was in operation from the 2011-12 academic year onwards, linking with the embedded tutor support centre within the School, providing a range of regular and *ad hoc* sessions and services for both undergraduate and postgraduate students.

Looking forward to our undergraduate periodic review in early 2016, we are mindful of how we can better embed and support computational thinking and mathematical foundations from the start and throughout our programmes. In particular, we have identified a new 20-credit Computational Thinking module to sit at Level 4 for all of our programmes, to bridge from the mathematical foundations through to real world transferable and adaptable problem-solving skills. In particular, we wish to focus on connecting Computing to the real world, as well as key mental processes such as abstraction, algorithm design, decomposition, pattern recognition, etc., and tangible outcomes, such as automation, data representation, parallelisation, pattern generalisation, etc., to be able to solve problems in Computing. More generally, this entails students being able to create and analyse computational problems and artefacts. This will provide a platform and foundation for derived and more advanced modules as students progress through the three years of the programme. From a broader learning and teaching perspective, this refinement of how we develop and embed deeper mathematical competencies for Computing, as well as transferable computational thinking and problem-solving skills, has been actively developed in partnership with our student body and programme team; we acknowledge the value of the students as partners model and the work of the Higher Education Academy (HEA) in this area[14].

Programming, codemanship and software carpentry

There are a range of programming options across our undergraduate programmes, with a main strand of Java and object-oriented programming running through the three years. Furthermore, there is provision for low-level programming in C, as well as a range of modules with a focus on web programming and mobile

[13] See: **http://www.sigma-network.ac.uk**

[14] See: **https://www.heacademy.ac.uk/enhancement/themes/students-partners**

development. Nevertheless, this Java/OOP strand provides the main opportunity for teaching principles of programming (for example, the primary aim of the module is not necessarily to 'teach Java', but to teach the principles of object-oriented programming using Java), software engineering and developing practical programming skills, alongside key systems design and analysis modules:

- BCO4008: Systems Analysis and Design Techniques (10 credits)
- BCO4013: Programming Fundamentals (10 credits)
- BCO4014: Event-Driven Programming (10 credits)
- BCO5001: Systems Development and Design (20 credits)
- BCO5008: Object-Oriented Systems I (20 credits)
- BCO6008: Object-Oriented Systems II (20 credits)

Over the past five years, we have moved away from focusing primarily on syntax, to developing a deeper understanding of principles of programming, transferable language semantics, underlying constructs and structures, as well as developing useful and usable software artefacts: in summary, software carpentry and codemanship. This has a dual focus: firstly, it develops a high-level appreciation for why we are teaching programming - essentially to solve real-world problems, using the most appropriate languages, tools and environments; secondly, it enables us to embed the use of tools, methodologies and techniques so as to start to develop best practice for real-world software development. While we by no means claim to create industry-level programmers at the end of the degree programme, we foster and support the development of a particular culture around creating useful and usable software artefacts, underpinned by rigorous knowledge and theory, ensuring that students understand how software is designed, developed and maintained in industry, and have the internal framework for developing knowledge and understanding in new languages, tools and environments and methodologies when required to do so. Ultimately, we have seen this provide an excellent preparation for living and working (both technical and non-technical roles) in a world that is being "eaten by software", as Marc Andreessen (co-author of Mosaic, the first widely used Web browser) famously said in 2011 (Andreessen 2011).

Specific examples of how we foster this culture of appreciation in programming of software carpentry and codemanship - and ultimately, an apprenticeship model - can be shown by our approaches to engagement with industry, as well as linking domain theory and knowledge with real-world methodologies, practices, tools and environments. In particular, we focus on understanding and developing competencies in techniques such as (object-oriented) design principles, test-driven development, unit testing, refactoring and agile development. Furthermore, we encourage and support the use of industry-standard integrated development environments (IDEs) (such as Eclipse and NetBeans), source control (for example, using GitHub), as well as an awareness of working with large-scale collaborative software projects and how they are maintained (for example, continuous integrations tools such as Jenkins). In essence, we are developing code literacy: the ability to read, understand and work with existing code bases.

A key aspect of ensuring this industry relevance is via our engagement with employers, primarily through an active industry advisory board. This board consists of companies we have had a long-standing relationship with - for research, enterprise, policy, as well as learning and teaching - representing local employers, as well as national and international companies. They provide regular input into our programmes and activities, as well as providing a range of experiential opportunities for our students; for example, supporting labs and mentorship of projects and dissertations. However, we also have a wider national and international network of companies and organisations offer our students internships and sandwich placements, with *c.*15% of students opting for a full year sandwich placement alongside the compulsory work placements modules at Level 5; we envisage this number increasing year on year due to the increased interest from students (and employers) for practical experience on graduation, as well as having established strong links with local,

national and international schemes. As a department and a university, we recognise the broad imperative on graduate attributes and skills for employability[15]: developing and recognising achievements beyond the curriculum (for example, developing digital portfolios on GitHub and blogging), living in a sustainable society, alongside enterprise and entrepreneurship education (particularly through Cardiff Metropolitan University's Centre for Student Entrepreneurship[16]).

Looking forward to our undergraduate periodic review in early 2016, we plan to further develop ideas around software carpentry, codemanship and code literacy as students progress through the three years of the degree programme. We envisage rethinking modes of assessment for our key programming modules, both formative and summative, and how this enables progression and development. Furthermore, we intend to continue to embed industry engagement from the start of our programmes with new 20-credit group projects at both Level 4 and Level 5 (as well as the individual 40-credit dissertation project at Level 6), working with industry mentors to identify and solve real world software and systems problems.

Much of this work and associated impact on our undergraduate curriculum has been inspired by external research and policy work: in particular my work with schools, examination boards, governments and industry on Computer Science curriculum reform through CAS, as well as my engagement with the EPSRC-funded Software Sustainability Institute[17] - whose mission is to cultivate better, more sustainable, research software to enable world-class research ("better software, better research") - and with international initiatives such as Software Carpentry[18] and Data Carpentry[19], organisations who develop and teach workshops on the fundamental programming, computational and data skills needed to conduct research in and across disciplines. Furthermore, within the UK HE community, engaging with a range of key stakeholders, including BCS, The Chartered Institute for IT, the Council for Professors and Heads of Computing (CPHC), but in particular, the HEA Computing Subject Centre, including hosting workshops in Cardiff on rethinking the first year Computing curriculum as well as on teaching programming vs. software carpentry in 2014.

How others might adapt or adopt this practice

While we have presented case studies from two different institutions and departments, with different profiles and demographics, there is clearly shared ground, perspectives and common approaches to some of the problems presented. Invariably, these interventions will consist of a hybrid approach, addressing a combination of institution-specific challenges, but acknowledging the common approaches and best practices.

As we have seen, there are significant benefits to the apprenticeship model and approach. This has been highlighted in both the Bath and Cardiff Met case studies, across two different departments and discipline areas, but focusing on the teaching of introductory programming. The potential of early development of computational thinking and transferable problem-solving skills is clear, as well as fostering a culture of software carpentry and codemanship: developing useful and usable software artefacts. While this is an under-explored area from a pedagogic research perspective, we have seen the benefits from multiple cohorts at the two institutions.

---

[15] See: **http://www.qaa.ac.uk/assuring-standards-and-quality/skills-for-employability**

[16] See: **http://www.cardiffmet.ac.uk/business/cse**

[17] See: **http://www.software.ac.uk**

[18] See: **http://www.software-carpentry.org**

[19] See: **http://www.datacarpentry.org**

Many large-scale programming classes will rely heavily on tutors. They form an important part of the students' learning experience. The following points have been found useful at Bath, where we have large laboratories (efficient for timetabling, but harder to manage), but have also been applicable to Cardiff Met with a higher number of smaller labs and using academic staff as tutors:

- assign students to tutors, rather than having a 'floating pool'. We have certainly been helped here at Bath by having one large laboratory for 75=5x15, rather than smaller laboratories, as there can then be a mix of tutor experience in the room;
- brief the tutors on what they are expected to do, especially the role in helping (or not) students with weekly exercises versus the assessed coursework;
- active and engaged tutors are key: ideally they should not sit in labs waiting to be asked for help!
- debrief the tutors: they know far better than the lecturer what is actually going on in the laboratory classes and with the students, especially if the first point is followed;
- staff development of the tutors is important, and will pay off;
- encourage students to help each other, such as peer support in labs and online fora (for example, on Moodle or other VLE).

We also have the following general suggestions:

- consider a custom textbook: it has certainly worked for Bath, and is more feasible than one might think;
- find a way (the Bath way is only one option) of ensuring weekly exercises are taken seriously by the students; engagement is key, especially for the apprenticeship model to work;
- since the aim is to teach the craft of programming, the lecturer should demonstrate programming, rather than just talk about pre-written programs; furthermore, they should emphasise that their solution is one of perhaps a number of 'correct' solutions, but may not be the optimal solution;
- the choice of programming language should suit the audience and the pedagogic goals, not some pre-defined idea of a 'good' language; be wary of jumping to new (and potentially faddish) languages, tools and environments;
- develop (and emphasise) computational thinking skills from the start, linking theoretical skills and understanding to real-world problem solving. A wide range of resources[20] already exists that can be easily adapted and adopted;
- the value of developing a culture around (and appreciation of) software carpentry and codemanship: essentially, creating useful and usable software artefacts;
- the importance of long-term industry engagement, from industry advisory boards, through to invited 'tech talks', internships, sandwich placements and graduate positions;
- the value of engaging with organisations such as the Software Sustainability Institute, Software Carpentry and Data Carpentry - the huge wealth of resource and experiences, as well as linking research and learning and teaching;
- have a strategic plan for developing these topics in future undergraduate and postgraduate programme development and periodic reviews, alongside the wider context of developing academic skills, graduate attributes, employability and entrepreneurial skills.

[20] For example, see **https://www.google.com/edu/resources/programs/exploring-computational-thinking**

Conclusion

This report has focused on the teaching of introductory programming as a craft skill, to be taught via an apprenticeship model, with further aims of fostering a culture of software carpentry and codemanship: developing useful and usable software artefacts. Teaching it this way has substantial advantages in terms of student engagement, participation and increased retention. There are other craft skills in Computing (for example, database design, advanced object-oriented programming, etc.) which could easily benefit from the same approach.

We have seen - and will to continue to see - significant changes to Computer Science education in the UK, from primary school through to FE and HE. From reform of the Computing curriculum in England, scaling and CPD challenges in Scotland, as well as expected future reform in Wales and Northern Ireland, the curriculum and qualifications landscape will most likely increase the diversity of qualifications and experiences of entrants to HE in the short and medium term. A renewed focus on pedagogy for teaching Computer Science and programming should be embraced; for example, the formation of the UK Forum for Computing Education[21], led by the Royal Academy of Engineering and supported by BCS, The Chartered Institute for IT, is a positive step to bring together key stakeholders in the UK.

Furthermore, as a discipline, we are currently in the midst of significant scrutiny, from both an educational, economic and policy perspective; this presents both opportunities and warnings: the opportunity to raise the profile and wider public perception of the discipline as an educational and economically valuable pursuit versus being driven to support the immediate and somewhat transient demands on the technology sectors. Recent and ongoing reviews that will have an impact on our discipline include: the aforementioned Shadbolt and Wakeham reviews of Computer Science and STEM graduate employability (as well as degree accreditation); the new QAA Subject Benchmark Statement for Computing to be published in late 2015; the 2014 UK Digital Skills Taskforce report[22] (*Digital Skills for Tomorrow's World*), the recent UK House of Lords Select Committee on Digital Skills' report[23] (*Make or Break: The UK's Digital Future*), as well as emergent effects from the recent or upcoming changes to the Computing curricula across the four nations of the UK.

Finally, we acknowledge the impact of the application of computational techniques across Science and Engineering and how this has fundamentally affected practices within those disciplines (Crick 2012). Computing is both a rigorous academic discipline in its own right and also facilitates and supports a wide range of other disciplines, from computational physics to computational social science; in essence it has become a bridge for interdisciplinarity: it now does not only support how Science is done, but what Science is done (Royal Society 2012a). Therefore, aspects from this report (and from across Computer Science) could also be applied to a number of STEM disciplines that now need to teach introductory programming (and how to leverage data and computation to solve domain problems) in their undergraduate curricula.

---

[21] See: **http://ukforce.org.uk**

[22] See: **http://www.ukdigitalskills.com**

[23] See: **http://www.parliament.uk/digital-skills-committee**

References

ACM (2010) *Information Systems 2010: Curriculum guidelines for undergraduate degree programs in Information Systems* [online]. New York, NY: Association for Computing Machinery. Available from: http://www.acm.org/education/curricula/IS%202010%20ACM%20final.pdf [Accessed 6 November 2015]

ACM (2013) *Computer Science 2013: Curriculum guidelines for undergraduate programs in Computer Science* [online]. New York, NY: Association for Computing Machinery. Available from: http://www.acm.org/education/CS2013-final-report.pdf [Accessed 6 November 2015]

ACM (2014) *Software Engineering 2014: Curriculum guidelines for undergraduate degree programs in Software Engineering* [online]. New York, NY: Association for Computing Machinery. Available from: http://www.acm.org/education/se2014.pdf [Accessed 6 November 2015]

Andreessen, M. (2011) Why software is eating the world [online]. *Wall Street Journal*, 20 August. Available from: http://www.wsj.com/articles/SB10001424053111903480904576512250915629460 [Accessed 6 November 2015]

Arthur, S., Crick, T. and Hayward, J. (2013) *The ICT Steering Group's report to the Welsh Government* [online]. Available from: http://learning.wales.gov.uk/docs/learningwales/publications/131003-ict-steering-group-report-en.pdf [Accessed 6 November 2015]

Astrachan, O. and Reed, D. (1994), *AAA and CS 1 The applied apprenticeship approach to CS 1*. Durham, NC: Duke University.

Ben-Ari, M. (2001) Constructivism in Computer Science education. *Journal of Computers in Mathematics and Science Teaching,* **20** (1) 45–73.

Boustedt, J., Eckerdal, A., McCartney, R., Moström, J.E., Ratcliffe, M., Sanders, K. and Zander, C. (2007) Threshold concepts in Computer Science: do they exist and are they useful? In: *Proceedings of the 38th SIGCSE Technical Symposium on Computer Science Education (SIGCSE'07), 7-11 March*. New York, NY: ACM Press. pp. 504-8.

Brown, N., Kölling, M., Crick, T., Peyton Jones, S., Humphreys, S. and Sentance, S. (2013) Bringing Computer Science back into schools: lessons from the UK. In: *Proceedings of the 44th ACM Technical Symposium on Computer Science Education (SIGCSE'13), 6-9 March*. New York, NY: ACM Press. pp. 269–74.

Brown, N., Sentance, S., Crick, T. and Humphreys, S. (2014) Restart: the resurgence of Computer Science in UK schools. *ACM Transactions on Computer Science Education,* **14** (2) 1–22.

Chapman, S.J. (2009) *Essentials of MATLAB programming*. (2nd Ed.) Boston, MA: Cengage Learning.

Clancy, M. (2004) Misconceptions and attitudes that interfere with learning to program. In: Fincher, S. and Petre, M. *Computer Science education research*. London: Routledge. pp. 85-100.

Collins, A., Brown, J.S. and Holum, A. (1991) Cognitive apprenticeship: making thinking visible. *American Educator,* **15** (3) 6–11.

Cress, P., Dirksen, P. and Graham, J.W. (1970) FORTRAN IV with WATFOR and WATFIV. Upper Saddle River, NJ: Prentice Hall.

Crick, T. (2012) Computing: supporting excellence in STEM. In: *Proceedings of the 2012 Higher Education Academy STEM Annual Conference: aiming for excellence in STEM Learning and Teaching, 12 April*. York: Higher Education Academy.

Crick, T. and Moller, F. (2015) Technocamps: advancing Computer Science education in Wales. In: *Proceedings of 10th International workshop in Primary and Secondary Computing education, 9-11 November.* London: WiPSCE 2015.

Crick, T. and Sentance, S. (2012) Computing at School: stimulating Computing education in the UK. In: *Proceedings of the 11th Koli Calling International Conference on Computing education research, 17-20 November*. New York, NY: Association for Computing Machinery.

Davenport, J.H., Wilson, D., Graham, I., Sankaran, G., Spence, A., Blake, J. and Kynaston, S. (2014) Inter-disciplinary teaching of Computing to Mathematics students: programming and discrete Mathematics. *MSOR Connections*.

Davis, R. B., Maher, C. A. and Noddings, N. (1990) Introduction to Constructivist views on the teaching and learning of Mathematics. *Journal for Research in Mathematics Education, Monograph,* **4,** 1-3.

DfE (2013) *National curriculum in England: Computing programmes of study* [online]. Department for Education Available from: https://www.gov.uk/government/publications/national-curriculum-in-england-computing-programmes-of-study/national-curriculum-in-england-computing-programmes-of-study [Accessed 6 November 2015]

Eckerdal, A., McCartney, R., Moström, J.E., Ratcliffe, M., Sanders, K. and Zander, C. (2006) Putting threshold concepts into context in Computer Science education. In: *Proceedings of the 11th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education (ITICSE'06)*. New York, NY: Association for Computing Machinery, pp. 103-7.

Epp, S.S. (2004) *Discrete Mathematics with Applications*. (3rd Ed.) Boston, MA: Cengage Learning.

Fincher, S. (1999) What are we doing when we teach programming? In: *Proceedings of 29th Annual Frontiers in Education Conference (FIE'99).* pp. 1–5.

Hazzan, O., Lapidot, T. and Ragonis, N. (2011) *Guide to teaching Computer Science: an activity-based approach*. New York, NY: Springer.

Khalife, J. (2006) Threshold for the introduction of programming: providing learners with a simple computer model. In: *Proceedings of 28th International Conference on Information Technology Interfaces.* pp. 71-6.

Kirschner, P.A., Sweller, J. and Clark, R.E. (2006) Why minimal guidance during instruction does not work: an analysis of the failure of constructivist, discovery, problem-based, experiential, and inquiry-based teaching. *Educational Psychologist,* **41** (2) 75–86.

Kori, K., Pedaste, M., Äli Leijen, and Tönisson, E. (2016) The role of programming experience in ICT students learning motivation and academic achievement. *International Journal of Information and Education Technology,* **6** (5), 331-7.

Land, R., Meyer, J.H. and Smith, J., (eds.) (2008) *Threshold concepts and transformational learning*: *Educational Futures: Rethinking Theory and Practice*. Boston, MA: Sense Publishers.

Meyer, J.H., Land, R. and Baillie, C. (eds.) (2010) *Threshold concepts and transformational learning*: *Educational Futures: Rethinking Theory and Practice*. Boston, MA: Sense Publishers.

QAA (2007) *Subject Benchmark Statement: Computing* [online]. The Quality Assurance Agency for Higher Education. Available from: **http://www.qaa.ac.uk/en/Publications/Documents/Subject-benchmark-statement-Computing.aspx.pdf** [Accessed 6 November 2015].

QAA (2015) *Subject Benchmark Statement: Computing – Draft for Consultation* [online]. The Quality Assurance Agency for Higher Education. Available from: **http://www.qaa.ac.uk/en/Publications/Documents/SBS-Computing-consultation-15.pdf** [Accessed 6 November 2015].

Roediger III, H.L. and Karpicke, J.D. (2006) The power of testing memory: basic research and implications for educational practice. *Perspectives on Psychological Science,* **1** (3) 181-210.

Royal Society (2012a) *Science as an open enterprise* [online]. Available from: **https://royalsociety.org/topics-policy/projects/science-public-enterprise/report/** [Accessed 6 November 2015].

Royal Society (2012b) Shutdown or restart? The way forward for Computing in UK schools [online]. Available from: **https://royalsociety.org/topics-policy/projects/computing-in-schools/report/** [Accessed 6 November 2015].

Schmidt, E. (2011) Television and the Internet: shared opportunity [online]. *The Guardian*, 26 August. Available from: **http://www.theguardian.com/media/interactive/2011/aug/26/eric-schmidt-mactaggart-lecture-full-text** [Accessed 6 November 2015].

Stroustrup, B. (2013) *The C++ Programming Language*. (4th Ed.) Boston, MA: Addison Wesley.

Vihavainen, A., Paksula, M. and Luukkainen, M. (2011) Extreme apprenticeship method in teaching programming for beginners. In: *Proceedings of the 42nd ACM Technical Symposium on Computer Science Education (SIGCSE'11)*. New York, NY: Association for Computing Machinery, pp. 93-8.

Ward, D. (1992) The King and "Hamlet". *Shakespeare Quarterly,* **43** (3) 280–302.

Willman, S., Lindéna, R., Kaila, E., Rajala, T., Laakso, M.-J. and Salakoski, T. (2015) On study habits on an introductory course on programming. *Computer Science Education,* **25** (3) 276-91.

Wing, J. M. (2008) Computational thinking and thinking about Computing. *Philosophical Transactions of the Royal Society, A* 366 (1881), 3717-25.

Contact us

+44 (0)1904 717500 enquiries@heacademy.ac.uk
Innovation Way, York Science Park, Heslington, York, YO10 5BR
Twitter: @HEAcademy www.heacademy.ac.uk

Higher Education Academy (HEA) is the national body for learning and teaching in higher education. We work with universities and other higher education providers to bring about change in learning and teaching. We do this to improve the experience that students have while they are studying, and to support and develop those who teach them. Our activities focus on rewarding and recognising excellence in teaching, bringing together people and resources to research and share best practice, and by helping to influence, shape and implement policy - locally, nationally, and internationally.

HEA has knowledge, experience and expertise in higher education. Our service and product range is broader than any other competitor.

To request copies of this report in large print or in a different format, please contact the communications office at the Higher Education Academy: 01904 717500 or pressoffice@heacademy.ac.uk

Higher Education Academy is a company limited by guarantee registered in England and Wales no. 04931031. Registered as a charity in England and Wales no. 1101607. Registered as a charity in Scotland no. SC043946.