Computer Science education, teaching and learning - a discussion paper.

Professor Mike Holcombe, Dean of the Faculty of Engineering,
Department of Computer Science,
University of Sheffield.

Abstract.
Computer science, software engineering and artificial intelligence are dynamic, practical and popular subjects to study. They possess a short and turbulent academic history, a controversial and idiosyncratic curriculum and provide the basis for a wide range of lucrative careers. We consider why the current offering is inadequate for the challenge of the future and what might be done about it.

The ACM Computing Classification System K.3.2 Computer and Information Science Education

1. Introduction.

I'll start by considering the sort of statement that our universities are being obliged (by HEFCE) to create as part of their *Learning and Teaching Strategy*.

Leaving out the usual preambles (the University of XXXX is a research-led university that undertakes research and teaching of the highest quality in an international context... blah.. blah..), we find statements like this:

The university seeks to provide an education based around:
- *the development of knowledge and understanding;*
- *the development of subject specific skills;*
- *the development of an intellectual ability to analyse information rigorously, to use reasoned arguments to reach a conclusion or a problem solution;*
- *identifying authoritative propositions and question orthodoxy;*
- *critically reviewing their own work and those of others;*
- *developing an enthusiasm and appetite for lifelong learning;*

etc.

If that is what we are about, does computer science, in the widest sense, deliver? Does our curriculum deal with these issues? We consider some of these issues and also some raised in the ACM report [1].

What is computer science, is it a science and how does it relate to the engineering of applications and artefacts?

Although called a *science*, computer science has few of the attributes one might expect from looking at other, more established, sciences. We do not seem to carry out experiments although our laboratories are full. Rarely are our students trying to investigate fundamental truths or explore the behaviour or construction of objects in those laboratories. If our students are investigating anything it is more likely that they are trying to find out why their attempts at building things don't work and how to fix them. The lessons that they might learn from this are rarely reinforced in any systematic way in our lectures and commentaries. In other words, we emphasise facts about architectures and languages at the expense of understanding why these things are the way they are.

The main focus of the *scientific dimension* is theory - theory of computation, algorithmic complexity and models of concurrency. We use mathematics and formal logic as tools and languages for these topics and suffer the fact that our students know nothing of these subjects and do not, in the main, wish to.

When building software systems and artefacts we teach software engineering principles despite any scientific or theoretical justification for doing so. When faced with similar tasks in industry our graduates ignore all that we have taught - primarily because no-one in industry believes that they are practical or necessary.

The nearest we get to any scholarly analysis is in the treatment of complex application domains - particularly in artificial intelligence applications - scientific theories of perception (vision, speech) and in language. Here we can argue that we are being scientists but the products developed, e.g. speech recognisers, natural language processors etc. are built more by accident (and endless experiment and training) rather than by *design*.

The relative youth of the subject and the effect of rapid advancement in the technology have left us with a body of confused and largely irrelevant ideas and procedures that cannot be justified through hard scientific evidence. We pin our ideas onto partial and abstract theories that cannot stand up to the demands and the complexity that modern applications require and we rarely criticise them in class.

2. Content, fashion and prejudice.

In the past there were the *programming language wars*. Endless hours of departmental debate on which should be the first/main programming language have consumed our lives. The passions exposed are quite remarkable.

All that is now over. Java reigns supreme after a mere two years of existence! Nothing in the history of mankind has swept the world so swiftly and so completely, despite the many problems of teaching an extremely unstable and hype-loaded language. The attempts by some universities to introduce a functional language as a way of levelling the playing field are beginning to crumble. We rarely require A Level Computer Studies because we don’t approve of it but a large proportion of students take it and thus know *all* about programming. However we are happy to admit students who have never touched a computer before - hence the programming gap between our two cohorts of freshers. One half, having programmed since they were 8, know more about programming than we do, the other half never quite sort it out in the three short years they are with us!

A functional language is also supposed to be a *good thing* as it helps to develop “good” programming style, and to explore esoteric concepts such as higher order abstraction and lazy evaluation - despite the knowledge that apart, from a certain community of research workers, these languages are just curiosities. Maybe students who have done these subjects build *better* software, maybe not.

Now we are faced with the UML bandwagon, trundling irresistibly in the wake of Java and comprising the most irrational, inconsistent, misleading and cock-eyed *rag bag* of concepts that have ever graced the academic stage, see [2, 3, 4, 5].

The sectarian nature of the subject is being replaced by the grey *uniformity* of a language and method, which would normally be very worrying, more so since it is accompanied by no demonstration whatsoever of its capability or coherence. Enthusiasts of the OO approach should ask themselves, before getting totally carried away, how they propose to test their creations, rigorously! How many realise that more time and money is spent on testing, review and debugging than on all the other stages of the lifecycle put together. We must raise the profile of principled testing and debugging in our courses. But if you were really interested in designing software that could be tested effectively and economically you would not start from here, i.e. UML and an object-oriented approach.

An interesting recent development is the emergence of the philosophy of Extreme

Programming [10] which seems to reject the heavy handed approach of conventional software engineering, almost rejecting design itself, concentrating, instead on small scale program development heavily constrained by rigorous and continuous testing managed in a more human-oriented and adaptable way. Part of the motivation is to try to handle changing and poorly understood requirements, partly it is a response to the fact that human nature does not seem to be very predisposed to the large methodologies that are recommended. If software products were not protected by wideranging disclaimers industry might be more interested in building it more carefully. We have used these techniques in some of the real projects run by our 4th year software company and they do seem rather effective, [7].

A further idea that is looking interesting and could possibly break the monopoly of the monolithic culture of database design is the emergence of XML. Building information systems that can *evolve* with the business process, and which do not need expensive maintainance by expensive systems analysts, is an important issue. The use of suitable dialects of XML and smart natural language processing techniques may well revolutionise things and provide much more power to end users.

3. Finding the essence.

The tension, that only exists in academic subjects with a strong relevance to the needs of society, consists of the widely differing perspectives of the academic research agenda and the practical needs of industry. Computing probably suffers the most of any subject from this.

If one looks at the sort of research that is popular, both with the practitioners and the research councils, we find a subject that is almost completely divorced from the principal pressures of industry, despite many attempts through collaborative projects, EU initiatives and so on. Compared with subjects such as materials and engineering the impact that the outcomes of projects in computing have had on industrial practice and projects is almost non-existent. Where there has been the biggest impact it is usually because the foundations of the problem domain have been based on proper scientific investigation rooted in the physical world.

Meanwhile the subject is driven on by industrial innovations from abroad, for example Microsoft, Sun. Although we will try to follow one (e.g. Java) we will ignore the other (who teaches Visual Basic as their main language?) despite the industrial demand.

If we are experimental scientists we should be doing some experimental science. We should be teaching our students how to investigate the properties and behaviour of artefacts. This could be done through *artefact classes*. In other science and engineering subjects there are laboratory classes devoted to taking things apart to see how they are made and function. Why could we not do the same? Our emphasis is always on *making* things, often trivial and useless devices, rather than looking at how things work. We should be taking some source code and seeing how different parts of it work, how by changing components or lines of code different behaviour results, recording the results rigorously. Looking at performance issues on different platforms and under different loadings. This provides a much clearer understanding than just reading theoretical texts.

Part of our problem is that we do not, ourselves, know how to conduct these experiments. Another problem is our lack of technical support. In good science departments each academic has at least one technician, if we had that level of support we could try to investigate our subject better, being funded as mathematics for so long has meant that our only tools for investigating our subject are theoretical and unvalidated.

However, there are things we can do to encourage a more experimental and reflective attitude amongst students from the start. Consider the typical programming assignment. We set the students a fairly simple problem and require them to develop some software to solve it. We may provide them with an informal statement of the problem, less commonly a formal requirements document or a formal specification. We might provide some outline program code or a library of classes or modules to base their solution around.

The things we expect back from them may include source code, designs, specifications, test reports and perhaps some informal personal evaluation. These are what we mark for correctness, style, consistency etc. Many students will proceed by programming what they can as soon as they can and any supporting documents, such as designs or specifications, are often constructed *after the event*! We might try to control this tendency by requiring a phased hand-in of things in the *correct* order. This is hard to manage since the intermediate products need to be assessed and returned before the next product is submitted, ideally. It is also very difficult to check that the designs are completely consistent with respect to the code, etc. We know nothing about *how the students have really created these artefacts*.

One possible approach is to require a commentary - perhaps a lab book, which describes in a systematic way, the steps that were taken to build these programs. This would have to record every bug found, how it was found and what was done to fix it (whether in the specification or the design or the code). This would greatly increase the amount of reporting required and so time must be allowed for this. The other deliverable would be a report examining these bugs and reflecting on how they could be avoided. This might be more important to both student and teacher than the designs and the code. It should receive the majority of the assessment weighting. The personal software process approach encourages this, [11].

Once this type of mentality has been established, each student should be required to keep their lab book up to date and to use it every day when they are carrying out development activities. In this way we can try to encourage the sort of *personal software process* mentality of *recording*, *reflecting* and *improving* their basic engineering and science skills that are so vital.

4. The skills dimension.

The recent DfEE report “Skills in the Information Age”, [8], makes rather uncomfortable reading for academics. It is clear that the scientific and technical knowledge of their incoming graduates plays a very secondary role to the principal requirements of employers. What is wanted are bright students who have excellent soft skills, who can work well in teams, who can adapt quickly to new circumstances, who have a clear understanding of how business and enterprise works, who can communicate effectively and so on. The fact that they may have a deep understanding of denotational semantics or can prove impressive results about the complexity of obscure parallel algorithms or can create visually appealing (but probably meaningless) design documents in UML is of no consequence if they have never negotiated with a client or been able to test an application to make it acceptable to a user community.

We have got it all wrong. We are pedalling old ideas and theories (Z, etc.) of little real practical use, we are making our students perform difficult mathematical tricks for no reason, we are asking them to build software applications that nobody wants and we think that what we are doing is what the subject is all about.

The subject is changing fast and we are getting left behind. We need to realise that addressing industry’s needs requires more than putting on a dreary course on professional issues and lots of group work. Group work is almost always a disaster. It is very unpopular with students who regard it as a threat, it is often poorly managed and the assessment is usually a lottery. See [6, 7].

If the object of a group project is to develop a significant piece of software then we have to go about it in a sensible way. If the deliverable is a *bin job*, that is something that no one really needs and which will end up being thrown away once it has been marked, then it is seen as a waste of time and fails to motivate the students. If there is a real client, however, someone who really wants the software and who the students need to negotiate with, then this will achieve much, much more. The client's reliance on their work will also motivate them to produce a much higher quality product than if they know that it is to be binned after the marks have been awarded. This will give them a much greater understanding of quality issues, of testing and of trying to satisfy the client's needs than any phoney academic exercise. It provides a much deeper intellectual challenge for both students and *staff*.

The issue of critical review also needs to be addressed. It is bound up in quality assurance and must feature in real design projects. However, it is also important in an academic context. In a *hype laden* world it is vital that we encourage our students to take a highly critical perspective on the published literature. Give them some recent journal articles and ask them to present the ideas and findings to the class *and to mark the paper in a structured way*. This is a fascinating exercise rather like the fable of the Emperor's new clothes! Many papers that impress us can leave students cold, our prejudices and assumptions can prevent us from seeing things in a more dispassionate light.

Returning to the issue of experimental investigation, we need to directly address the issue of developing our students skills in *model building*. They need to know how to abstract real computational models, investigate their properties and to *validate* them against a real system that the model is supposed to represent. If we had done that seriously in the past many of the inadequate mathematical models and languages used in formal methods would have been rapidly consigned to history. How can we consider a software model that does not deal explicitly with time if we are claiming to be interested in safety-critical systems? The emphasis on modelling dwells too much on notations and the proving of simple facts and not enough on validation and evaluation. We are sending out graduates who are sceptical about what we have taught them because they know that it cannot cope with the demands of real systems and the *messy real world*.

Another weakness is the way we compartmentalise everything into modules etc. We ought to be looking at the breadth of ideas and finding ways of selecting the most appropriate for the job in hand. Why is it that considerations of topics such as intelligent agents is separated from discussions of algorithms, or formal methods or software engineering and quality assurance?

No wonder students revert to type when they start their individual projects and seem to forget much of what we have taught them, suppressed in the cause of hacking up some code to produce some "working" end product. There is no opportunity for them to take a principled look at the problem and choose the appropriate technology, intelligent or not, to solve it.

5. Does Computer Science have a future?

Computing degrees may continue to have a future but it is less clear as to whether they are really necessary. What is their purpose? Is it to provide employment of academic computer scientists and to feed into PhD programmes (thus enabling us to clone ourselves)? Or should these degrees try to provide a more industrial focus? Increasingly, application domains in the scineces, engineering and the humanities are generating their own software experts, people who understand the domain and can build reliable code as well as our own students can We, meanwhile, focus on obscure abstractions and notations, unwieldy design methodologies and research into irrelevant branches of the field while industry is marching on without us. Perhaps, in the not too distant future, there will be no Computer Science departments, just a few people in mathematics departments studying abstract notations and lots of people learning computing and practical software development in engineering, chemistry, biology, medicine and arts departments.

The successful computer science departments that do survive will try to address these issues with an increasing emphasis on specific application domains and on more practical project exercise involving real clients. They will get involved with building intelligent applications to support the analysis of digital objects - text, images, sound, etc. of interest to science, arts, engineering, business and others in collaboration with the domain experts and not in some isolated ivory tower that is today’s typical computer science department! The basic theory (Turing etc.) will still be part of our foundations but the current generation of formal methods and software engineering techniques will need to re-invent themselves into something that actually addresses the challenges of complex system design and construction and can be defended in a scientific way. What is really needed is a *software engineering science*. The stirrings are there but it won’t happen overnight, [9].

Some suggestions for developing our curriculum:

- some real experimental science;
- more focus on artefact studies;
- better integration of ideas;
- real projects with real clients;
- critical reviews and assessment of work;

And then there is assessment, but that is another story!

References.

[1] A B Tucker et al. “Strategic directions in Computer Science Education”, *ACM Computing Surveys*, 28, 1996.

[2] A J H Simons and I Graham, "30 Things that go wrong in object modelling with UML 1.3" , chapter 16 in: *Behavioral Specifications of Businesses and Systems* eds. H Kilov, B Rumpe, I Simmonds (Kluwer Academic Publishers, 1999), 221-242.

[3] A J H Simons, "Use cases considered harmful", *Proc. 29th Conf. Tech. Obj.-Oriented Prog. Lang. and Sys., (TOOLS-29 Europe*), eds. R Mitchell, A C Wills, J Bosch and B Meyer (Los Alamitos, CA : IEEE Computer Society, 1999), 194-203.

[4] A J H Simons and I Graham, "37 Things that don’t work in object modelling with UML", *British Computer Society Obj.-Oriented Prog. Sys. Newsletter*, 35, eds. S Kent and R Mitchell (BCS: Autumn, 1998)

[5] A J H Simons, M Holcombe & K Bogdanov, “Divide and conquer testing using hierarchical object statecharts”, Submitted.

[6] H Parker, M Holcombe and A Bell, “Keeping our customers happy: Myths and management issues in "client-led" student software projects”, *Computer Science Education*, 9, 230-241, 1999.

[7] M Holcombe, A Stratton, S Fincher, G Griffiths (Eds), *Projects in the computing curriculum*, Springer, 1998.

[8] See http://www.dfee.gov.uk/skillsforce/7.htm

[9] M Holcombe & F Ipate, *Correct systems: building business process solutions*, Springer Applied Computing Series, 1998.

[10] K. Beck, *Extreme programming explained: embrace change*, Addison Wesley, 1999.

[11] W.S. Humphrey, *Introduction to the Personal Software Process*, Addison Wesley, 1997.

Some of the work referred to in this article was supported by the UK Government’s Fund for the Development of Teaching and Learning initiative. The Industrial Software Project Support Network is based at the University of Sheffield and further information can be found at:
http://www.dcs.shef.ac.uk/teaching/fdtl/