Essays on Computer Science Education

Vashti Galpin

vashti@cs.wits.ac.za

http://www.cs.wits.ac.za/~vashti

Technical Report: 1992-03

April 1992

Department of Computer Science
University of the Witwatersrand
Private Bag 3
Wits 2050
South Africa

The essays that appear in this report were written as course requirements for the Computer Science Education course that was presented as part of the MSc in Computer Science by Coursework and Research Report in 1991.

A Debate on the teaching of Introductory Computer Science

In this essay, I will discuss the issues relating to the teaching of computer science that were raised in the article "A Debate on Teaching Computer Science" [Dijkstra et al 1989]. The structure of the paper is as follows: first I will briefly summarise Dijkstra's contribution and his reply. I will do this to illustrate what I consider to be his main points. I will then discuss the relevance of these points to the teaching of computer science. In the section after that I will summarise the points raised by the other contributors. As many points were repeated, I will give an overview of all the issues raised and discuss them. In the final section I will address some issues that were not raised and bring together my conclusions.

Dijkstra originally presented the talk at the ACM Computer Science Education Conference in February 1989 and it was decided to print the text of the talk in CACM with other computer scientists entering into the debate. The editor of the Communications of the ACM, Peter Denning introduces the debate by describing Dijkstra as challenging "some of the basic assumptions on which our curricula are based" [Dijkstra et al 1989].

Dijkstra's basic position is that computer science consists of two radical novelties and that this has implications for the teaching of computer science especially introductory programming courses for first year students (I will use the terms "first years" and "first year students" to replace the term "freshmen"). A radical novelty is such a sharp discontinuity in thought that existing approaches cannot be used to reason about it and it is necessary to approach a radical novelty with a blank mind. The two radical novelties in computer science are the depth of conceptual hierarchies that occur in computer science and the fact that computers are the first large scale digital devices. Radical novelties require much work to come to grips with, and people are not prepared to do this, so they pretend the radical novelties do not exist. Examples of this in computer science are software engineering and artificial intelligence.

Dijkstra investigates the scientific and educational consequences, by examining what computer science is. He reduces this to the manipulation of symbols by computers. In order for meaningful manipulations to be made, a program must be written. He then defines a program as an abstract symbol manipulator, which can be turned into a concrete symbol manipulator by adding a computer to it. Programs are elaborate formulae that must be derived by mathematics. Dijkstra hopes that symbolic calculation will become an alternative to human reasoning. Computing will go beyond mathematics and applied logic because it deals with the "*effective* use of formal methods" [Dijkstra et al 1989]. He points out that this view of computing science is not welcomed by many people, for various reasons.

Dijkstra makes a number of recommendations for education. Bugs should be called errors since a program with an error is wrong, and lecturers should avoid anthropomorphic terminology as it causes students to compare the analog human with discrete computer and this leads to operational reasoning which is tremendous waste of time. When attempting to prove something about a set, it is better to work from the definition than with each individual item in the set. The approach can be applied to programming as well where programs can be reasoned about without dealing with specific behaviours of the program. The programmer's task is to prove that the program meets the functional specification. Dijkstra suggests that an introductory programming course for first years should consist of a boolean algebra component, and a program proving component. The language that will be used will be a simple imperative language with no implementation so that students cannot test their programs. Dijkstra argues that these proposals are too radical for many. The responses that he expects, are that he is out of touch with reality, that the material is too difficult for first years, and that it would differ from what first years expect. Dijkstra states that students would quickly learn that they can master a new tool (that of manipulating uninterpreted formulae) that, although simple, gives them a power they never expected. Dijkstra also states that teaching radical novelties is a way of guarding against dictatorships.

In his reply to the panel's responses, Dijkstra points out that this is only a recommendation for an introductory programming course for first years. He notes that functional specifications fulfil the "pleasantness problem" - whether the product specified is the product wanted -, and the proof fulfils the "correctness problem" - whether the product implemented is the product specified and that these two problems require different techniques. He admits that the choice of functional specification and notation is not clear. He addresses concerns about the possibility of a more formalised mathematics and gives a number of reasons for his belief that it will be developed. Well chosen formalisms can give short proofs, logic has not been given a chance to provide an alternative to human reasoning, heuristic guidance can be obtained by syntactic analysis of theorems and informal mathematics is hard to teach because of things like "intuition", whereas symbol manipulation will be easy to teach.

In this section, I will discuss some of the issues raised by Dijkstra. He has described the two radical novelties of computer science and he uses this as a justification for approaching the discipline with a blank mind, because previous knowledge cannot help in the understanding of computer science. He later advances an approach to computer science that involves taking a mathematical approach, by using existing techniques of predicate calculus to prove that programs meet their specifications. This would seem to contradict his argument that one cannot use the familiar to reason about a radical novelty.

Generally it is accepted that to restrict thinking to one particular framework is undesirable, and leads to the formation of dictatorships. Dijkstra's argument is that students taking the introductory course should only think in the specified way. In my opinion, different approaches to a topic can only help comprehension of that topic. A point raised in a number of letters that appeared in later issues of Communications of the ACM is that different students approaching learning new concepts in different ways and that teaching should cater for this [Bernstein 1990; Herbison-Evans 1991]. Dijkstra also seems to have a general suspicion of tools, even those that can help students (or professionals) better understand a topic. A more pragmatic issue is that some students doing an introductory course at university will already have been exposed to programming and therefore operational thinking. How are these students to keep their thinking "pure" when doing Dijkstra's course?

Dijkstra's approach to teaching seems to be one of training not education. He wants to teach students doing an introductory course, a rigid set of rules and that set of rules only. This does not leave room for intuition, judgement and discussion, which all relate to education. He also emphasises the need for a specific skill without grounding it in any larger context.

One of the participants, William Scherlis, raised the question of why use an imperative language if operational thinking is to be avoided. Luker raises the point made by Turner and Backus that variables and assignment statements in imperative languages make verification difficult [Luker 1989]. A rigorous formalism could be introduced using a functional programming language (or perhaps using a formal language that does not relate to a programming language) and an understanding of the use of formalisms could be related to various issues of computer science and mathematics. This would give a broader area of application for the course.

Dijkstra states that learning to manipulate uninterpreted formulae would be satisfying for a first year student as it would "give him a power that far surpasses his wildest dreams". I believe that a course of this kind would consist of boring and repetitive work that would become mechanical, as it is training and not education. It could give a student a sense of power, but only in a limited domain, and as I understand the course that Dijkstra has outlined, this knowledge could not be applied to other domains within computer science.

Dijkstra states that operational reasoning is a waste of time, however proving programs can be very time consuming (except with very powerful techniques that might be developed in the future). Currently there is no formalism that is powerful enough to allow for short proofs and generally proving that a program is correct is a very tedious job. [De Millo et al 1979].

I can agree with the basic theme of Dijkstra's recommendations. I do perceive that there is a lack of rigour in introductory courses in computer science. However I would not approach changing the curriculum as Dijkstra does and I do not think his suggestions have really challenged any basic assumptions about curricula. I feel that he is pandering to the view of "computer science is programming" by using program verification as a tool for developing rigour. I find it problematic that the more formalised mathematics required for proof have not yet been developed, as the current techniques seem insufficient. In my opinion, it would be more beneficial to introduce students to the concept of formal proof of programs and then discuss the current limitations of the approach. I would prefer an computer science discrete mathematics course to be presented during the first year curriculum, such as the suggestions made by Marion [Marion 1989]. In this course, rigour can be introduced but the material covered is related to computer science as a whole and not only to programming.

In this section, I will summarise the responses to Dijkstra. A number of panelists agreed with Dijkstra's general conclusions although most people disagreed with his argument to justify the conclusions. A number of panelists also responded to his attack on software engineering. The responses that follow are those that relate to the teaching of computer science. In engineering education both proof and testing are necessary techniques and are not taught as alternatives. Students require exposure to the whole process of software engineering. Dijkstra does not tackle the issue of formalisation (capturing a real world problem in a formal specification) and the fact that formalisation is difficult. Mathematical proofs are not found by symbol manipulation but by trial and error using intuition and any other tools that will help. Dijkstra is trying to restrict modes of thinking but intuition and operational thinking can help solve problems. There are a number of different approaches to programming such as imperative, functional and logic. Dijkstra's language is imperative and cannot be scaled up. Students need to be exposed to set theory and predicate calculus which can be used in both formal and informal proof, but there is more to computer science than symbol manipulation.

Most panelists lost track of the fact that Dijkstra was careful enough to discuss his thoughts in terms of a first year course only and their discussion related mainly to issues of definition of computer science which are beyond the scope of this essay. However this did bring out very clearly that there is a lot of debate about what computer science is and what should be taught . This debate has been continued recently by the ACM Task Force on the Core of Computer Science [Denning et al 1989] and ACM/IEEE-CS Joint Curriculum Task Force [Tucker and Barnes 1991].

My general feeling about this debate is that it did not address general issues of teaching computer science. In fact Dijkstra limits himself to discussing an introductory programming course and does not discuss the impact of this course on the rest of the curricula. The debate relates predominantly to the teaching of programming, a pervasive part of computer science (see for example the section in the Task Force on The Role Of Programming [Denning et al 1989]), although I believe that the main issue was about rigour and formalism in introductory courses. In the Core of Computer Science Report and the Curricula 1991 Report, there is a heavier emphasis on theory in the curriculum than previously, however the issue of where it should be placed in the curriculum was not discussed. I think that it has generally been agreed that rigour is necessary for a computer science undergraduate curriculum, but I think that the question of how the rigour is to be taught is still an open area and it is important to be aware that students have different learning styles. In my opinion, an essential point for any introductory course teaching rigour is that it must relate the material taught to the rest of computer science so that the students can appreciate its place in the curriculum and its relevance to the field.

References

BERNSTEIN, D.R., 1990, Letter to ACM Forum. *Comm ACM*, **33(3)**, Mar 1990, 264-265.

DENNING, P.J., COMER, D.E., GRIES, D., MULDER, M.C., TUCKER, A., TURNER, A.J., YOUNG, P.R., 1989, Computing as a Discipline. *Comm ACM*, **32(1)**, Jan 1990, 9-23.

DE MILLO, R.A., LIPTON, R.J. AND PERLIS, A.J., 1979, Social Processes and Proofs of Theorems and Programs. *Comm ACM*, **22(5)**, May 1979, 271-280.

DIJKSTRA, E.W., PARNAS, D.L., SCHERLIS, W.L., VAN EMDEN, M.H., COHEN, J., HAMMING, R.W., KARP, R.M., AND WINOGRAD, T., 1989, A Debate on Teaching Computer Science. *Comm. ACM* , **32(12)**, Dec 1989, 1397-1414.

HERBISON-EVANS, D., 1991, Letter to ACM Forum. *Comm ACM*, **34(4)**, Apr 1991, 24-25.

LUKER, P.A., 1989, Never Mind the Language, What About the Paradigm? *SIGCSE Bulletin*, **21(1)**, Feb 1989, 252-256.

MARION, W., 1989, Discrete Mathematics for Computer Science Majors - Where Are We? How Do We Proceed? *SIGCSE Bulletin*, **21(1)**, Feb 1989, 273-277.

TUCKER, A. AND BARNES, B., 1991, Computing Curricula 1991. *Comm ACM*, **34(6)**, Jun 1991, 69-84.

AN EXTENSION TO THE ACM DEFINITION OF COMPUTER SCIENCE

In this essay, I will investigate the possibility of adding a third dimension to the two-dimensional grid used to define computer science by the ACM Task Force on the Core of Computer Science [Denning et al 1989]. The third dimension would show the interaction of concepts such as social responsibility, ethics and parallelism with the subareas. The outline of this essay is as follows; I will first briefly describe the classification given by the Task Force and the decisions used to achieve this classification. In the second section I describe the suggestion that was presented at the discussion of the Computer Science Education class. I will next describe the approach I took to formalise the concept of the third dimension and to identify which subareas belong on this dimension. I then argue for the fact that social issues in computer science should be placed on the dimension of subareas and not on the third dimension. Finally I discuss some other omissions and then present the final classification.

One of the Task Force's three aims was to present "a description of computer science that emphasises fundamental questions and significant accomplishments" [Denning et al 1989]. The approach of the Task Force was to give a short definition and then to divide the field into subsections. The Task Force developed a two-dimensional grid to divide computer science into subareas. The first dimension has three areas or processes: Theory, Abstraction and Design. Each of these describe how practitioners of computer science achieve their goals, and these three areas also relate the discipline to the areas of mathematics, science and engineering respectively. For the second axis, nine subareas were identified. These are as follows:

algorithms and data structures
programming languages
architecture
numerical and symbolic computation
operating systems
software methodology and engineering
database and information retrieval systems
artificial intelligence and robotics
human-computer communication

The criteria that were used to distinguish subareas were "underlying unity of subject matter, substantial theoretical component, significant abstractions and important design and implementation issues" [Denning et al 1989]. The Task Force also thought that it was preferable that each subarea be linked with a research community or group of research communities that produces literature in that area. Figure 1 illustrates this two-dimensional

matrix. The Task Force Report notes that certain subareas for which there exist literature and research communities, do not appear because “they are basic concerns throughout the discipline” [Denning et al 1989]. These areas contain theory, abstraction and design and occur in all subareas. The examples given are parallelism, security, reliability and performance evaluation.

	Theor	**Abstractic**	**Desig**
1 algorithms and data structures			
2 programming languages			
3 architecture			
4 numerical and symbolic computatic			
5 operating systems			
6 software methodology and engineer			
7 database and infromation retrieval s			
8 artificial intelligence and robotics			
9 human-computer communication			

Figure 1. Definition Matrix of Computing as a Discipline [Denning et al 1989].

The suggestion that was raised in the Computer Science Education class was that a third dimension should be introduced to the matrix to allow for these ‘orthogonal’ subareas to be explicitly captured, so that the disparate items that are spread across the second dimension are actually listed in such a way that they are not forgotten or ignored. This would give greater cohesion to the discipline. This would be similar to the way that the first dimension of theory, abstraction and design unifies the field by recognising all its facets. The third dimension would enable those subareas that do thread through the nine given subareas to be recognised as distinct in their own right. Figure 2 illustrates this idea (In the figures that appear in the rest of the document the nine subareas will be represented by the digits 1 to 9).

The approach I took was to investigate previous classifications of the field of computer science. The ACM Task Force referenced the Computer Science and Engineering Research Study (COSERS) [Arden 1980] and therefore I did not use this report. Another classification system is the *Computing Reviews* (CR) Classification System [ACM Guide to Computing 1990] which was developed from a Taxonomy of Computer Science and Engineering [AFIPS Taxonomy Committee 1980]. I decided to compare the classification done by the Task Force with the CR Classification to see if there are any other subareas with should be placed on the third axis, or if any fields have been totally omitted from the classification.

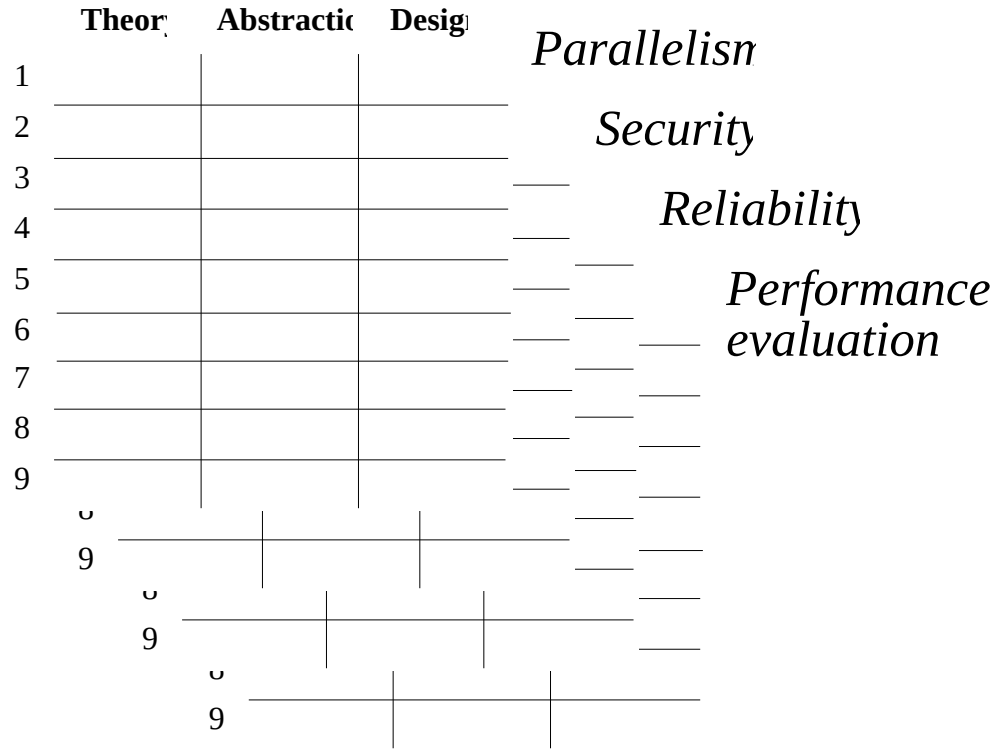

Figure 2. Suggested alterations to the definition matrix.

The major difference between the CR classification and the Task Force classification is the section on social and related issues. In the CR classification this is section K, Computing Milieux and it deals with the computer industry, history of computing, computers and education, legal aspects of computing, management of computing and information systems, the computing profession and personal computing. Dunlop and Kling are critical of the Task Force Report because it omits social analysis from its definition of computing [Dunlop and Kling 1991]. The ACM/IEEE-CS Joint Curriculum Task Force which investigated undergraduate curriculum and worked from the Report of the ACM Task Force on the Core of Computer Science, found it necessary add an undergraduate course to deal with social context of computing [Tucker and Barnes 1991, Denning et al 1988].

This area of social context could be added on any one of the three different axes. I will investigate the approach that was taken in Computing Curricula 1991 which is interesting because of the introduction of a third and possibly a fourth dimension. The underlying principles that are used for the curricula are as follows:

> "From the subject matter of the nine areas of the discipline, this article identifies a body of fundamental material called the *common requirements*, to be included in every program. The curriculum of each program must also contain substantial emphasis on each of the three processes, which are called *theory*, *abstraction* and *design*. In addition,

this article identifies a body of subject matter representing the social and professional context of the discipline, also considered to be essential to every program. Finally a set of concepts that recur throughout the discipline, and that represent important notions and principles that remain constant as the subject matter of the discipline changes, are an important component of every component of every program." [Tucker and Barnes 1991]

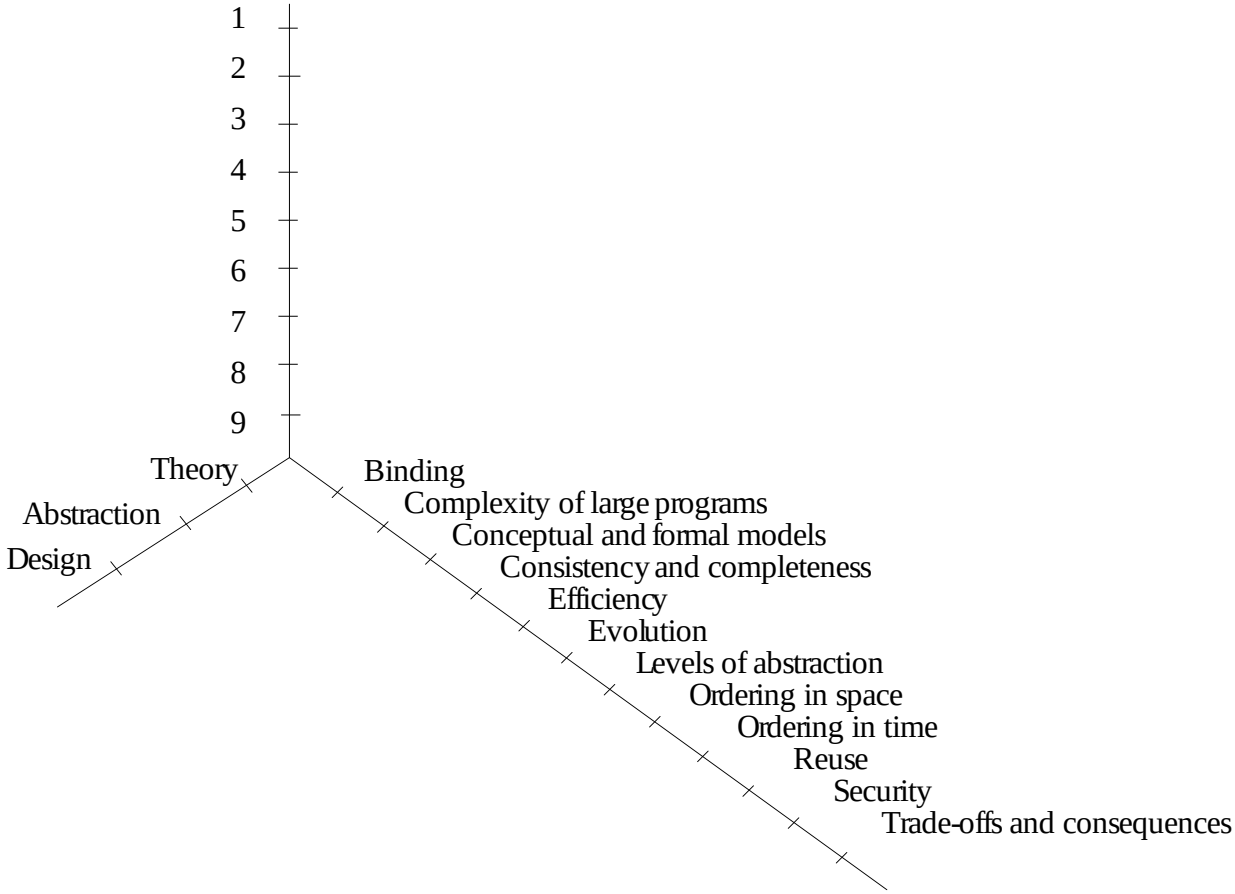

Figure 3. An interpretation of the underlying principles for curriculum design (excluding social context of computing).

The common requirements relate to the subareas, and the processes are as previously explained. The criteria for a recurring concept are that it occurs throughout the discipline, has a variety of instantiations and it is not technologically dependent. Most of the recurring concepts also occur in each subarea, and have instantiations at the theory, abstraction and design levels. I have interpreted these recurring concepts as a different third dimension as shown in Figure 3. Note that this third dimension consists of concepts whereas the third dimension suggested earlier in the paper consists of subareas. I will remain with the original idea of a third axis of subareas as opposed to one of concepts.

The question still remains of where to put social context. Tucker and Barnes do not classify it as a recurring concept. However they both juxtapose it with theory abstraction and design, and give it an extra common requirement along with other subareas that were defined. So one option would be to add it as a new process on the first dimension and a second option would be to add it as a new subarea on the second axis. A third option is to add it to the third dimension of Figure 2. The fourth option is to add another dimension to the diagram but I believe that this would make the whole definition too complex. My personal feeling is that it should be added as a subarea as I feel that it fulfils the criteria for this and also because if it were added to the third dimension of subareas it becomes difficult to identify the issues that relate to a particular subarea on the second dimension. It is therefore necessary to define the theory, abstraction and design issues that relate to the social context of computing. Many of the ideas presented here are from Dunlop and Kling [Dunlop and Kling 1991]. The social context would be subarea 10 in the Task Force's classification scheme.

10. SOCIAL CONTEXT OF COMPUTING

This area deals with placing the discipline in a social context. Fundamental questions include: What is the social impact of computing? What code of behaviour should computing professionals adhere to? How can computers strengthen democracy? How should computer scientists be educated?

10.1 Theory

Major elements of theory in the area of the social context of computing are:

1. Philosophical questions relating to computing.
2. Ethics.
3. Health issues.
4. Democratisation, employment, class, gender and race issues.
5. Educational theory.

10.2 Abstraction

Major elements of abstraction in the area of the social context of computing are:

1. Educational issues.
2. Codes of conduct.
3. Critical computer systems.
4. Social impact of technology.

10.3 Design

Major elements of design in the area of the social context of computing are:

1. Curriculum design.
2. Reduction of health hazards when using computer equipment.
3. Design of critical computer systems.
4. Evaluation of social impact of technology.

There are two other fields that the CR classification contains - simulation and modelling, and text processing - and also a section called Computer Applications that are all omitted from the Task Force Report. The application of simulation to computers could be added to the third dimension which consists of subareas and in some sense modelling is captured in the Abstraction column. However, the actual field of using computers for simulation as opposed to using simulation to understand computers is not orthogonal to the other subareas on the second axis; in fact it is distinct. However it could be treated as an application of computers. Text processing could be placed in the Human-Computer Communication subarea which seems to be a catch-all for the topics that were omitted. Section J of the CR Classification is Computer Applications and it is too wide to try to bring into the existing framework. It could however become a list separate from the definition matrix that would allow for description of these applications.

To return to the three-dimensional model that was proposed, I have not found any more subareas that could be added to the four suggested by the Task Force Report, namely Parallelism, Security, Reliability and Performance Issues. The suggestion was made that social issues and ethics be added to this dimension, however I feel that I have made a strong case for adding it as a subarea on the second dimension. The next stage of this process is to elaborate on the details of each block in the three dimensional matrix, as was done for the two-dimensional matrix. This is beyond the scope of this essay, but I would like to note that it is possible at this level for certain blocks to be empty, i.e. for there to be no material that relates all three dimensions together given a specific process and two subareas. Another issue is that the three dimensional matrix does not replace the matrix given in the Task Force Report. It gives more detail about the discipline and allows for subareas that do not appear on the second dimension to be explicitly represented. The classification scheme given by the Task Force Report remains as is except for the addition of the Social Context of Computing.

In this essay, I have attempted to formalise the idea of a third dimension for the classification of the discipline of computer science. The ‘orthogonal subareas’ have been added as a third dimension and social context of computing has been added as a subarea. Although there are many ways to restructure the classification of the discipline, the approach taken in this essay

has given more detail and depth to the definition of computer science which will allow for a better understanding of the discipline.

References

ACM GUIDE TO COMPUTING LITERATURE 1990, ACM Press.

AFIPS TAXONOMY COMMITTEE, 1980, *Taxonomy of Computer Science and Engineering*. American Federation of Information Processing Societies, Arlington, Virginia.

ARDEN, B., (ED), 1980, *What Can be Automated?* Report of the NSF Computer Science and engineering Research Study (COSERS). MIT Press, Cambridge, Mass.

DENNING, P.J., COMER, D.E., GRIES, D., MULDER, M.C., TUCKER, A., TURNER, A.J., YOUNG, P.R., 1988, *Report of the ACM Task Force on the Core of Computer Science*. ACM Press.

DUNLOP, C. AND KLING, R., 1991, *Computerization and Controversy*. Academic Press.

TUCKER, A. AND BARNES, B., 1991, Computing Curricula 1991. *Comm ACM*, **34(6)**, Jun 1991, 69-84.

A Comparison of Teaching Methods

In this essay, I will look at both the lecture method and participatory methods and suggest as an alternative a third method that incorporates the advantages of the two styles when teaching large groups. This third method does have disadvantages - as will any teaching method - and these will be described. I will first outline the underlying argument and then give the details.

The basic argument rests on the fact that lecturing is as good as any other method for imparting information to students, however it does not promote thinking. Active or participatory methods help to engender thought skills, however they tend to reduce the content that can be covered in a fixed time period and they require small groups for effectiveness. An alternative method for large groups, which allows the use of participatory methods and reduces the amount of lecturing, is a method whereby students preread the material required for the course. This allows lecture times to be used for presenting difficult concepts (if necessary) and the remaining time can be used for participatory methods. This method thereby ameliorates the disadvantages of the two opposing methods given above and also gives abilities to students that they should have once they graduate. The main disadvantages associated with this method are that courses need to be prepared in advance, well-trained tutors are required for small group work and students must be mature enough to cope with this type of self-teaching approach.

Bligh [Bligh 1972] looks at the experiments done pertaining to lectures and draws two conclusions, namely that lectures are as effective as other methods for transmitting (factual) information, but are not as effective as other methods in promoting thought. McLeish notes that the lecture system (using only lectures to present materials) gives “effective training for passing exams” and presents a number of experiments that question the efficacy of the lecture system [McLeish 1968]. Lectures can stifle self-expression and it becomes difficult for students to keep concentrating. Note-taking can be a good skill to develop, however it is a complex skill and it is susceptible to errors and can inhibit the absorption of information [Yorke 1981]. One of the main advantages of lecturing is that it is an efficient method of communicating informations to students, i.e. one teacher can teach a large number of students [McLeish 1968].

Jones [Jones 1987] notes that “participatory teaching methods may help by developing students with greater depth and breadth of thought”. The methods that Jones advocates are brainstorming, directed dialogues, small group discussions, role playing/drama, games, panel discussions, debates and Socratic dialogues. These methods are discussed in various places in the literature about the teaching process in higher education [Newble and Cannon 1989; Yorke 1981]. Jones raises a number of limitations of these methods, namely that the lecturer

is required to do more preparation, less content can be covered with the class, class size and arrangement becomes important and strategies must be developed to ensure all students participate. He suggests that in a large class, only the top students should take part and the mediocre students will benefit from listening. I find this unacceptable firstly as I believe all students should leave university with these skills of presentation and argument, or at least have the opportunity to acquire them. Secondly, it is also difficult to determine who the top students will be in advance.

Newble and Cannon [Newble and Cannon 1989] argue that small group activities can take place in a lecture theatre of any size, if the students are to work in relatively small groups. However, I believe that this would make it difficult for a lecturer to facilitate discussion within groups, as the lecturer would not be able to get physically close to a group in the middle of a tiered lecture theatre with fixed seating.

In the Computer Science Education discussion, the two experienced lecturers in the group pointed out that they liked participatory methods, but that to them it seemed difficult to use these methods with classes of 80-120 students. I would like to suggest a method of presenting Computer Science courses that allows the use of participatory methods for large groups. There are courses that are based on similar concepts presented in the literature [Hills 1976; Yorke 1981; Keller and Sherman 1974].

The crux of the approach is to leave the bulk of the content work to the students, i.e. it would be up to the students to have prepared the necessary work beforehand, therefore cutting down the number of lectures that are required. This in turn allows more time for participatory work, generally in small groups. It should be noted however, that there is extra time required by the students for preparation of work as compared with the lecture method. The lectures can be reserved for explaining difficult material that occurs in the curriculum when necessary. This material could be identified in advance by the lecturer, or by the tutors and students as the course proceeds.

There would be a number of weekly slots and at the beginning of each week, the types of presentations that would be occurring could be announced. Students who have covered the work that is being dealt with in lectures would not need to attend, however all students would be urged to attend tutorials and labs. Students that have not prepared the necessary work in advance would be unable to contribute, and a mechanism is required to ensure that students do not fall behind. For this, I suggest well-trained counsellors to help students allocate their time correctly and also a self testing procedure that allows students to evaluate their progress.

This approach has a number of implications, those that relate to the establishment and running of the course and those that relate to students. The course needs to be established in advance. It is not possible for a lecturer to run the course on a week-to-week basis, preparing the material as needed. As much material is required, it would be advantageous for the course to be set up to be used for a number of years. Yorke gives some guidelines for developing these materials [Yorke 1981]. This however requires some kind of curriculum stability, which seems to currently be lacking in the field of computer science, but the ACM Computing Curricula 1991 [Tucker and Barnes 1991] will help to give a basis from which to work. In computer science departments that are currently re-evaluating their curricula, it could be possible to design a number of courses in advance and to prepare the material for them. If an evaluation procedure for courses is instituted, it would become easier to re-evaluate and modify the courses when it becomes necessary.

As the aim of this method is to introduce more participatory work, which is generally small group work, it would be necessary for tutors to be employed. These tutors should be well trained, dedicated and committed to their jobs. It is necessary for the member of staff taking the course to let the tutors know what exactly is required of them and what teaching techniques could or should be used. A weekly meeting session between the lecturer presenting the course and the tutors would be used for the lecturer to describe the goals to be achieved in the week's teaching and also for the tutors to give feedback to the lecturer. This two-way communication is essential for the lecturer to stay in touch with what is happening with the students. The structure for this is suggested in Reges et al [Reges 1988]. The lecturer should be available for consultation and to observe or to take tutorials. The course should be run as a joint effort by the tutors and the lecturer, and it should not be the case that the tutors take on all of the teaching load.

The students need to be motivated to achieve the goals set for them. It would be likely that first year students would find this more difficult, especially if they have just left a school environment. The course outline should include a clear list of how far a student should should have progressed by a particular date, and how much time the department expects the student to spend on a particular section.

Students that are having problems reaching these deadlines could have problems for two main reasons. The first reason is that they are missing some crucial background and these could be remedied by extra tutorials or lectures. The second reason could be that they do not have the ability to organise their time or to use it effectively. This could be solved by having an accessible counsellor that can help the student develop these skills.

It could be argued that students are not mature enough to handle such an environment. This can be countered with a statement from the ACM Task Force on the Core of Computer Science Report that states "the curriculum should be designed to develop an appreciation for learning that graduates carry with them through out their careers" [Denning et al 1988] and I believe that if students are not encouraged to deal with self-directed study then they are not working towards achieving this goal.

When doing the reading for this essay, I became aware of a large body of research investigating teaching methods used in higher education. It is beyond the scope of this essay to evaluate my suggestion fully with respect to the literature and I have only touched upon some of the issues.

I have presented a suggestion to accommodate the use of participatory methods when dealing with large groups of students. It requires that each student learns the skills that enable the student to study independently and then to use the information from this study in an interactive environment. This would allow for courses that consist of large numbers of students and that encourage thought through participatory methods.

Note: Some of the above views relate to my own experiences as an undergraduate where lectures seemed to be an extreme waste of time. I would have preferred to have the whole course at the beginning and to have worked through it at my own pace.

References

BLIGH, D.A., 1972, *What's the use of lectures?*, Penguin.

DENNING, P.J., COMER, D.E., GRIES, D., MULDER, M.C., TUCKER, A., TURNER, A.J., YOUNG, P.R., 1988, *Report of the ACM Task Force on the Core of Computer Science*. ACM Press.

HILLS, P.J., 1976, *The Self-Teaching Process in Higher Education*, Croom Helm.

KELLER, F.S. AND SHERMAN, J.G., 1974, *The Keller Plan handbook: essays on a personalized system of instruction*, Benjamin.

MCLEISH, J., 1968, *The Lecture Method*, Cambridge Institute of Education.

NEWBLE, D. AND CANNON, R., 1989, *A Handbook for Teachers in Universities & Colleges*, Kogan Page.

REGES, S., MCGRORY, J. AND SMITH, J., 1988, The Effective Use of Undergraduates to Staff Large Introductory CS Courses, *SIGCSE Bulletin,* **20(1)**, Feb 1988, 22-26.

TUCKER, A. AND BARNES, B., 1991, Computing Curricula 1991. *Comm ACM*, **34(6)**, Jun 1991, 69-84.

YORKE, D.M., 1981, *Patterns of Teaching*, Council for Educational Technology.

HYPERTEXT AND HYPERMEDIA

Recently there has been much hype about hypermedia especially with respect to its application to education. I would like to take a more cautious look at the implications of hypermedia for education. The general focus of this essay is to investigate problems raised in the literature about using hypertext, hypermedia and multimedia for teaching, primarily with reference to tertiary education. Since these are general issues that relate to teaching tools, I will not specifically discuss the teaching of Computer Science. When I was reading the literature, I discovered that it was often difficult to determine what type of interaction the student was expected to have with the hypermedia, and I will present a classification scheme that describes 8 different levels of interaction. I will give some examples of hypermedia systems that are used for teaching and classify them according to the eight levels. Finally I will outline some problems that occur when using hypermedia in teaching.

First I will give some definitions of hypertext and hypermedia/multimedia. Jakob Nielsen uses the following definition:

> "*Hypertext* is non-sequentially linked pieces of text and other information. If the focus of such a system or document is on non-textual types (for example graphics, sound, moving images from videodisks, executable programs) of information, the term *hypermedia* is often used instead. In traditional printed documents, practically the only such link supported is the footnote, so hypertext is often referred to as "the generalised footnote"." [Nielsen 1988]

Marchionini differentiates between hypermedia systems and the products of such systems:

> "The term "hypertext" describes the electronic representation of text that takes advantage of the random access capabilities of computers to overcome the strictly sequential medium of print on paper. "Hypermedia" extends the nonlinear representation and access to graphics, sound, animation, and other forms of information transfer. To distinguish between the hypermedia system used to create and display information, and documents created and used with this system, the term "hyperdocument" is used to refer to the actual text, lesson, or product." [Marchionini 1988]

Hyperdocuments consist of nodes and links. Nodes contain information, and links are relations between pieces of information in the nodes. Links can be unidirectional or bidirectional. There are a number of different hypertext and hypermedia systems and they take different approaches to implementing these principles. I will not discuss these differences in detail.

Often it is not clear for what level of interaction, educational hypermedia systems are designed, and I have developed a classification system to describe these different types of interaction. (a) is the most restricted type of interaction and (h) the least restricted. The classification is as follows:

(a) The user follows a predetermined path. This is a linear approach and is also called a guided tour [Trigg 1988]. It can be useful when introducing users to a hypermedia environment.

(b) The user chooses a path. This is called navigation and relates to browsing [Marchionini and Shneiderman 1988].

(c) The user "trail-blazes". This means navigating a path and storing it for other users to follow.

(d) The user employs non-hypermedia search techniques such as Boolean connectives, string search and proximity limits. This relates to fact finding [Marchionini and Shneiderman 1988].

(e) The user adds new links between pieces of information. This allows other users to follow new paths.

(f) The user adds new information to nodes, without adding new links. This is called annotation although this term is sometimes used to refer to (g).

(g) The user adds new nodes and new links to an existing hyperdocument

(h) The user creates a new hyperdocument. This is called hypermedia authoring or composing.

I will now investigate some hypermedia systems that are used for teaching and classify them according to the above list.

- Multimedia Works allows students to create their own hyperdocuments using a small database of text, pictures and video clips related to a given topic [Pea 1991, Soloway 1991]. They do this in small groups and then present the final product to their classmates and teacher. A major problem that I see with this approach is an emphasis on presentation as opposed to content. It also seems a very expensive and very time consuming method of learning. The fact that it is time-consuming is not necessarily a criticism, if it can be shown that it is an effective method of learning. However this has not yet been shown. Multimedia Works is aimed more at primary and early secondary education. This would be classified as type (h).
- The Perseus Project is an interactive hypermedia system that is managed as a journal [Crane and Mylonas 1988]. Its emphasis is material concerned with Greek civilisation. Its aims are for first year students to learn to use material in an analytical way, and to build an environment in which different kinds of knowledge are accessible. I think that some filtering mechanism is needed, especially when there is very technical and detailed

knowledge required for experts present in the hyperdocuments, as first year students are likely to want more simplified knowledge. This is classified as (b).

- The use of hypermedia in computer science has generally been related to programming using HyperTalk, the scripting language of HyperCard [Katz and Porter 1991; Fritz 1991]. This can be classified as (h), however the emphasis is more on the process of scripting the hyperdocuments than on the final hyperdocument.
- A plant cell biology course and English literature course have been implemented in Intermedia [Yankelovich et al 1988]. There were 8 authors, and 80 students using the hyperdocuments as browsers. This is a type (b) interaction, although some students did create their own documents. Intermedia provides webs which contain a subset of node and link information and users can add node and link information to their own webs. Intermedia therefore allows all interaction types.
- Project Jefferson is a hypertext system to assist first year students in writing assignments about the American Constitution [Parsaye et al 1989]. Students browse the hyper document and can "take photographs" of information found and then use that information in the word processor that is provided. This is also a type (b) interaction.

Although the above list is not complete, it seems that most hypermedia educational systems are geared towards browsing (b) or full authoring (h). I feel that the other types of interaction especially (c) to (g) offer scope for further development.

A number of problems that relate to using hypermedia for teaching have been identified in the literature and I would like to discuss these briefly. There are three areas that will be discussed - navigational support, authoring of hyperdocuments for teaching purposes and learners' usage of hypermedia.

It is easy for users to become "lost in hyperspace" or disoriented. Wright [Nielsen 1989b] identifies five navigational issues:- going to a known place, going to an ill-defined place, going back, going somewhere new and knowing how much information there is. These issues can be resolved by providing navigational support. Navigational support can be given by overview mechanisms which allow the user to see "where" they are and filtering mechanisms which restrict which nodes and links a user can access. Filtering becomes even more important when users can add their own links, as a hyperdocument with too many links is as bad as one with too few.

Blattner and Dannenberg [Blattner and Dannenberg 1990] note that currently hypermedia composition is intuitive and that not enough is known about the design of hyperdocuments. Wright points out that the cognitive costs on authors is high [Nielsen 1989a], therefore it would be undesirable for teachers to have to create their own hypermedia, in fact experts are

required to design hyperdocuments [Morariu 1988]. If the aim of hypermedia is to cater for different styles of learning, then designers must ensure that they do supply these capabilities. Teachers also need the tools to assess learner progress and to check that the student has not gained incorrect concepts in their self-directed learning [Morariu 1988]. Marchionini [Marchionini 1988] notes that hypermedia gives learners freedom of choice, but that educators must now develop methods for learners to manage this choice.

One of hypermedia's claims is that it models human associative memory [Marchionini 1988], however as it is not known exactly how humans' learn, this cannot be verified. Waller [Nielsen 1989a] points out that there will be cultural problems in coming to grips with new types of media and that it will take time for the full implications and usage to emerge. Students will need to develop new skills to be able to deal with hypermedia and this will be problematic as there are no role models to learn these skills from.

Hypertext, hypermedia and multimedia appear to be interesting tools to use in education, however their implications are not fully understood and they should not be seen as a panacea for all educational ills. Only time will tell what the full application of these tools will be, and their advocates need to remember that "the goal isn't multimedia, but effective communications" [Grimes and Potel, 1991].

References

BLATTNER, M.M. AND DANNENBERG, R.B., 1990, CHI '90 Workshop on Multimedia and Multimodal Interface Design, *SIGCHI* **22(2)**, October 1990, 54-58.

CRANE, G. AND MYLONAS, E., 1988, The Perseus Project: An Interactive Curriculum on Classical Greek Civilisation, *Educational Technology* **28(11)**, November 1988, 25-32.

FRITZ, J.M., 1991, HyperCard Applications for Teaching Information Systems, *SIGCSE Bulletin*, **23(1)**, March 1991, 55-61.

GRIMES, J. AND POTEL, M., 1991, What is Multimedia?, *IEEE Computer Graphics and Applications*, **11(1)**, January 1991, 49-52.

KATZ, E.E. AND PORTER, H.S., 1991, HyperTalk as an Overture to CS1, *SIGCSE Bulletin*, **23(1)**, March 1991, 48-54.

MARCHIONINI, G., 1988, Hypermedia and Learning: Freedom and Chaos, *Educational Technology* **28(11)**, November 1988, 8-12.

MARCHIONINI, G. AND SHNEIDERMAN, B., 1988, Finding Facts vs. Browsing Knowledge in Hypertext Systems, *Computer*, January 1988, 70-80.

MORARIU, J., 1988, Hypermedia in Instruction and Training: The Power and the Promise, *Educational Technology* **28(11)**, November 1988, 17-20.

NIELSEN, J., 1988, Trip Report: Hypertext '87, *SIGCHI*, **19(4)**, April 1988, 27-35.

NIELSEN, J., 1989a, HyperHyper: Developments Across the Field of Hypermedia - A Mini Trip Report, *SIGCHI*, **21(1)**, July 1989, 65-67.

NIELSEN, J., 1989b, Hypertext II, *SIGCHI*, **21(2)**, October 1989, 41-47.

PARSAYE, K., CHIGNELL, M., KHOSHAFIAN, S. AND WONG, H., 1989, *Intelligent Databases: Object-Oriented, Deductive Hypermedia Technologies*, John Wiley.

PEA, R.D., 1991, Learning through Multimedia, *IEEE Computer Graphics and Applications*, **11(6)**, July 1991, 58-66.

SOLOWAY, E., 1991, How the Nintendo Generation Learns, *Communications of ACM*, **34(9)**, September 1991, 23-26.

TRIGG, R.H., 1988, Guided Tours and Tabletops: Tools for Communicating in a Hypertext Environment, *ACM Transactions on Office Information Systems*, **6(4)**, October 1988, 398-414.

YANKELOVICH, N., HAAN, B.J., MEYROWITZ, N.K. AND DRUCKER, S.M., 1988, Intermedia: The Concept and Construction of A Seamless Information Environment, *Computer*, January 1988, 81-96.