What are We Doing When We Teach Programming?

Sally Fincher, Computing Laboratory, University of Kent at Canterbury, UK

Abstract: *The academic discipline of Computer Science is confounded by the practice of its curriculum. Uniquely, it prepares students for future study by teaching the fundamental construct of its practice – programming – before anything else. The disciplinary argument seems to run that if a student is not versed in the practicalities, then they cannot appreciate the underlying concepts of the discipline. This may be true. However, an analogous situation would be if it were thought necessary for architecture students to be taught bricklaying before they could appreciate the fundamentals of building design. This argument is (in similar fashion) clearly flawed when compared to endeavours such as the study of English Literature, which makes no claim (or attempt) to teach the practice of producing work before the study of the products of others' work: similarly, academic philosophy is the study of philosophers, not the practice of thought.*

It is possible that this is an argument of disciplinary maturity – that all disciplines have passed through a similar phase. This paper will examine the emergent approaches currently being defined, all of which address the central concern of the teaching of programming and its relationship to the learning of Computer Science. It will examine: the "syntax-free" approach of Richard Bornat and Russel Shackelford, the "problem-solving" approach of David Barnes (et al), the "literacy" approach of Peter Juliff and Owen Astrachan and the "computation-as-interaction" approach of Lynn Andrea Stein. These approaches will be discussed both in their own terms, and also placed in a preliminary taxonomic framework for the teaching of programming.

Introduction

Once upon a time, thirty or forty years ago, people learned to program computers because they needed the computer to do something for them. Typically these people were scientists, engineers and mathematicians. They learned the languages and techniques of programming for a specific purpose. Over time, from these people there emerged a new discipline of Computer Science and they became Computer Scientists.

Traditionally programming has been taught as these scientists learned it, via syntax, through the vehicle of a single language. The limitations of this approach – that students get bogged down in the specifics of the chosen form, that (famously) they see programming as "fighting the compiler" – are frequently bemoaned and yet, as frequently, this is the approach that dominates undergraduate teaching. Very little attention has been paid to the rationale which informs the choice of *why* we teach the subject in a particular fashion.

As Computer Science as an academic discipline has matured, programming remains as a central and distinguishing feature of the curriculum. However, programming is now taught as a process separate from purpose. We no longer teach programming in order to get the computer to *do* something, but as a transferable skill in its own right. With this change of disciplinary construct other conceptual models and methodologies for teaching programming have been explored and developed. This paper outlines some of these directions and surveys some of the approaches being taken.

The "approaches" outlined here are my clusterings of examples of espoused (and therefore conscious) practice, from various sources in the literature and are chosen for their illustrative purposes. I do not include *every* example of the use of any give "approach", and inevitably there will be better examples for some which I have missed. Equally, it is certain that there are other examples in the literature which could be clustered as "approaches" in a similar fashion.

The "syntax-free" approach

If teaching programming via the vehicle of any given language constrains the learning process unacceptably, teaching programming without language would seem to have the attraction of avoiding all these pitfalls. And yet, such an approach is also paradoxical. Trying to teach the practice-based skills of programming without being able to demonstrate the practice sounds nonsensical. Nevertheless, teaching programming as a skill separate from coding is one that occurs often and has been instantiated in practice at several institutions.

However, whilst recognising the same core of the central problem, the approaches are often presented in terms of different rationales. For example, Richard Bornat in *Programming from First Principles* [1] describes the approach used by him (and his colleagues) at Queen Mary College[*] of the University of London as "the result of a six year experiment in undergraduate teaching". His rationale is "The 'damage' caused by early exposure to a particular code (BASIC is often singled out in this regard) is real enough but is not caused by the evil properties of any particular notation; it is the delusion that to lean a code is to learn to program which is truly harmful" [1, p.xvi].

The book which espouses and encapsulates this approach is divided into five parts: Basic Concepts,

[*] Now, due to a merger, Queen Mary and Westfield College

0-7803-5643-8/99/$10.00 © 1999 IEEE **November 10 - 13, 1999 San Juan, Puerto Rico**
29th ASEE/IEEE Frontiers in Education Conference

Structured Instructions, Some Extended Examples, Structures of Values and Transcribing into other Codes – reflecting issues of importance when using the imperative languages popular at the time (Pascal, COBOL, Algol-68, Ada etc.). The notation used for the examples is based on ISWIM (**I**f you **S**ee **W**hat **I** **M**ean) which is introduced gradually, throughout the text, as needed. All the exercises in the book can be done with pencil-and-paper; to use them in an electronic environment requires translation into a programming language.

A second classic implementation of this approach is described by Russel Shackelford from Georgia Institute of Technology [2]. The rationale for this introduction is slightly different from the earlier example. Although the course was designed for first-year undergraduate computer science students, in this case, they were not the only audience. The author says "We believe that all well-educated college students should be exposed to the basic tenets of algorithmic thinking. The algorithm-oriented agenda found in this book is not for CS-majors only. It is offered as 'foundational twenty-first century knowledge' for a broad population of students from across the family of academic citizens" and "To this end, our approach has three key goals: providing an introduction to the field, providing conceptual content and software skills and preparing students for programming". This course (Introduction to Computing) is now a core, required course for all undergraduates at Georgia Tech. For CS-majors is it followed in the second semester by "Introduction to Programming" which uses a given programming language.

The book is structured in three parts: the Computing Perspective, The Algorithm Toolkit and The Limits of Computing, the accompanying (lab-based part) of the course in five modules: Communications Tools and Facilities, Data Processing Tools and Facilities, Problem-Solving, a Taste of Programming and Lab Skills Evaluation. Shackelford uses a pseudocode as the teaching vehicle. This is rather closer to an actual programming language than ISWIM. It is called RUSCAL and "features the important ideas embodied in various languages such as Java, C++, Pascal and Fortran" [2, p.56]. This is supplemented (at least at Georgia Tech) with a system which allows fragments of this code to be compiled and run, sometimes against test suites, and electronically submitted as assignments.

The "Literacy" approach

Although somewhat differently implemented, the impulse to abstract the skill of programming away from the tightly associated skill of expressing a program in a code (and coding environment) is a common feature of both the syntax-free and literacy approaches.

It might be argued that this "literacy" approach, too, is an obvious one, given the nature of the problem. Almost every child in the world (and certainly every child who comes into tertiary-level education) has learned at least one totally abstract notation for the purposes of conveying meaning – the skills of the alphabet, of reading and writing. It is not such a far-fetched idea to assume that this process could be modelled for learning programming. However, a considerable problem in this area is that most literacy learning takes place in very young children. The learning patterns, styles and motivations (not to mention the physical structure of the brain) are very different in early life from later[†].

Given this, and accepting that we cannot teach material in the same ways (using the same methods) that we can with children, the question then is, what are the important features in the learning process by which we achieve literacy? These have concentrated on aspects of achievablity, motivation, relevance and what I term "the use of sharp tools".

Achievability and motivation are important because when children learn to read they do not (in general) know how to learn, what to learn or why what they are learning might be useful. Motivation in this sort of situation must, of necessity, be extrinsic, and supplied by the teacher.

The most illustrative example is that of Peter Juliff, in describing his approach to teaching programming to non-CS students [3]. He describes the implementation of the approach in a course to Business students at Deakin University which covers encompasses "A detailed study of algorithm design; multi-level decomposition; data typing and scope rules; logic constructs enabling structured programming and information hiding; abstract data structures and recursive processing techniques." Although this is the material to be covered, this description is not revealed to the students as they would see these aims as neither achievable nor relevant. These notions (of achievability and relevance) are reinforced through the course by ensuring that a student can achieve a (small) working application within a 2 hour laboratory session (which can then be built on and enhanced in later sessions) and setting problems and projects which result in software which resembles the look-and-feel of commercial applications they encounter in the "real world". Just as the majority of children are not motivated by learning rules of grammar, so the majority of students are not motivated by the construction and manipulation of complex internal data structures that involve little or no user interaction.

The third leg upon which this approach stands, in contrast to the syntax-free approach, is that of using a real, current, programming language throughout. Rather than teaching a quite separate notation (or a pseudocode) the

[†] There has been some interest in the parallel between acquiring a second language and the learning of programming, but this has manifested itself much less in the area of introductory programming. Some of the techniques for testing comprehension have, however, been transferred to the new domain.

students are given an actual language as their tool. This has obvious relevance for the students but also embodies the idea that learning to program involves the use of real tools and no service is done by making the students go through an intermediary step. In this, it could be seen to stand in opposition to the syntax-free approach described above.

A variation of this approach (rarely explicitly codified) is that students of programming (just like students who are learning to read) are capable of reading much more complex works than they are themselves able to produce. By presenting them with a structured series of good examples, the argument is that they will learn the desirable features of a good program, they will learn good style. This notion has been a feature of the "apprenticeship" model of learning described by Owen Astrachan which he utilises at Duke University. "This approach follows an *apprenticeship* model of learning, where students begin by reading, studying, and extending programs written by experienced and expert programmers"[4]. Astrachan's "applied apprenticeship" also has an emphasis on real-world applications and motivations " Under our applied approach, (1) students are able to learn from interesting real-world examples, (2) the synthesis of different programming constructs is supported using incremental examples, and (3) good design is stressed via code and concept reuse."

Perhaps, the central feature of this approach is that learning to program is a new (and difficult) skill. Students need their learning to be supported in such environments, and the supports this approach provides are those which mimic the acquisition of the skills of reading and writing prose. In this way, this approach is rather closer to the "traditional" model, of teaching something (int his case programming) which students can immediately use to achieve some other goal. "At the end of a day at kindergarten, youngsters like to have a painting to take home to show the family and to have it displayed on the door of the refrigerator. What makes us think that undergraduates are any different?" [3] In this sense it is a more practical, teacher-oriented approach than the others described here, as its methodology transfers directly to the classroom. It echoes the formal apprenticeships of other traditions used in the teaching of many skills-based activities, from craft professionals to engineers.

The "problem-solving" approach

The phrase "problem-solving" appears as a very common short-hand encapsulation of what the non-coding skills of programming are. Often it is used in conjunction with phrases such as "analysis" and "design". This is true to such an extent that there are books and even *series* of books with it in the title. For example, two published this year are *Ada95 Problem Solving and Program Design* [5] and *Problem Solving and Program Design in C* [6]. However, the approach they work on is not pedagogically very useful. Firstly they assume that problem-solving is what the student wants to do (when they may well be wanting to do something else – like learn C); secondly they do it via the medium of a single syntax (which brings back all the syntax-based problems previously observed) and thirdly they assume that the student does not know how to problem-solve.

A more pedagogically-based approach based on problem-solving has been described by Barnes et al [7]. Here they describe how programming tasks were re-conceptualized for the students away from a coding exercise towards an activity requiring a separate and distinct skill set. They derive a simple cycle of activity – Understand, Design, Write, Review – in part from previous work, influential in the field of mathematics [8]. This is then applied not only to programming tasks in a specific syntax, but across several courses and syntaxes (functional and imperative) and even to every-day material (that is, to things other than programming) with the intention of allowing the students to apprehend that problem solving is a distinct set of behaviours which can be applied to many areas. Their central concept is that problem-solving is a transferable skill, and one that can be presented (semi-) independently of domain.

Computation as Interaction

This approach is rather different from the others delineated so far, in that it has been described (and, presumably utilised) by only one author. Lynn Andrea Stein has defined a different approach to the teaching of programming over several articles [9,10]. Her approach has been influenced by a change in the paradigm underlying programming (and programming languages) and to the conditions and experiences of computing which students have before they come to be CS students.

In terms of the paradigm shift, her approach is in response to the massively increased popularity of object-orientation, where the world is modelled as a collection of objects which know certain things about each other and which communicate to get things done. This is fundamentally quite distinct from the stepwise decomposition of imperative and functional languages.

In terms of "the world" her argument is that all twentieth-century students, from a very early age, experience computers as multi-threaded, GUI–driven devices (which she terms experiencing "computation as interaction"). Her argument (and approach) is that to present these same students (when they begin to study Computer Science) with a model of single-threaded problem-solving "the sequence of calculations required to get from a particular instance of the question to the corresponding instance of the answer" is cognitively inappropriate. it "doesn't really correspond to the way that computation exists in the world at large".

The design of the course she uses (taken from [8]) reflects her intent:

Class Sessions		Laboratories
Introduction to Interactive Programming	Expressions and Statements	Spirograph (Expressions and Statements)
Objects and Classes	Interfaces and Exceptions	Nodes and Channels (Interactions)
Self-Animating Objects	Inheritance	Design Project
Student Holiday	Object-Oriented Programming	Balance (Classes)
Dispatch Mechanisms	Procedural Abstraction	Calculator (Procedures)
In-Class Examination	Events Driven Programming and java.awt	(Documentation Project)
Columbus Day	Event Delegation (and more java.awt)	Scribble (Events)
Safety, Livenenss, and Synchronization	Interfaces and Protocols: Composing Systems	*No Laboratory*
Push and Pull		Cat and Mouse (Systems of Systems)
In-Class Examination	Explicit Communication: java.io, java.net	
Veteran's Day	Servers	Final Project (Networked Interactions)
Arbitration or RMI	Design Architectures	
On Presentations	*Thanksgiving Holiday*	
Group Project Presentations		
In-Class Examination	Interactive Programming as Program Design	

It is interesting to note that she describes the advantages of teaching the "traditional" approach as a special-case, later in the curriculum.

Just as one might place the "syntax-free" and "literacy" approaches as standing in opposition, here one might place the "computation-as-interaction" next to "literacy" in regard to its emphasis on a "real-world" contextualisation of skills.

Provisional Taxonomy

These varying approaches clearly exist in some sense. However, they may only be available to those practitioners with the skills (and time) to reflect upon, examine and identify their own practice. Whether they can be transferred to other institutions and contexts is less clear, as these approaches are evangelical rather than evaluated. However, they may be made more accessible by placing them in a continuum of computer science pedagogy, enabling "chalk face" teachers to assess where their own practice stands in relation to these fixed points.

All the approaches have in common the idea that coding is separate from programming. It is in the definition of what programming comprises that their differences are to be found. One possible axis is that of how closely the approaches model activity in "the real world". A possible representation might be:

Literacy — Computation as interaction — Problem-solving — Syntax-free

Closely modelled — Abstract

Another way of regarding the differences would be in their immediate applicability to the classroom situation. The Literacy and Problem-solving approaches both require minimal change to the curruiculum and are more dependent on the attitude, skills and learning objectives of the teachers. The Syntax-free and Computation-as-Interaction approaches require considerably more adaptation of existing materials, both in the course in which they are situated and in the remainder of the courses which comprise the program.

Summary & Conclusion

The debate about what we should be teaching undergraduate computer scientists is not particularly new. In the UK (at least) there has been fierce debate over the last decade as to whether Maths is a necessary pre-requisite, and this has extended to considerations as to whether mathematical ability (whilst necessary for certain areas of the curriculum, such as Theoretical Computer Science) is a predictor for success in programming. With increased numbers of UK universities no longer requiring Maths as a pre-requisite, this debate is moving to new grounds. Informal and anecdotal evidence suggests that Maths is not a predictor for success in programming so the debate is now focussed on what existing pre-dispositions (learning styles, abilities, skills) *might* be predictive.

What is new is the questioning of what we are aiming to do in the teaching of programming. Computer Science is a distinctive form of engineering and (especially compared to other engineering domains) professionally immature with little disciplinary consensus. It is debatable whether even today the majority of CS faculty have first degrees in the discipline. This means that they have taught CS as they have learned it and, in all probability, using their self-taught model and/or techniques from the disciplines they learned in formally – maths or science. This problem has been highlighted by Fintan Culwin [11] where he describes the state of mind of educators who believe that because they learned imperative programming before they learned object-oriented programming, that is the way in which everyone must learn it, because that is how the concepts are structured. Any other method would not deliver the required level of understanding.

By-and-large, the material covered by and delivered in these courses which are based on these "approaches" is the same. As Stein [8] says: “Like every introductory programming class, this class must begin with the mechanics of program-writing, introducing basic data types and programming constructs”. What distinguishes these approaches, then, is not *what* is taught, but *how* and *why* – and it is the *how* and *why* that creates the distinctive educational frameworks within which these educators teach.

The common *challenge* of these approaches is to the notion that there is a distinguished order to concept acquisition. Instead of accepting the view that students need to learn to code and that from this experience they will learn complex, transferable skills (analysis, design, problem-solving), leaving the students to abstract these for themselves, these approaches start from a position of identifying the acquisition of other skills as the ultimate objective and support student learning directly to this end.

Changing an approach to teaching requires first the knowledge that other approaches are possible; secondly it requires reflective practitioners. Perhaps this much we have. However, it also requires evaluation and evidences of the success of any given approach and there is little of this work in the literature, and much less which is comparable across institutions and diverse student populations.

This, then, is an open problem for the emergent specialism of Computer Science Education and for Computer Science Education researchers.

[1] Bornat, Richard *Programming from First Principles*. Prentice Hall International,1987

[2] Shackelford, Russel *Introduction to Computing and Algorithms* Addison-Wesley, 1998

[3] Juliff, Peter *Marketing Programming to Non-programmers* in Informatics in Higher Education , ed. Mulder and van Weert, Chapman and Hall, 1998

[4] Astrachan, Owen and Reed, David *AAA and CS1: The Applied Apprenticeship Approach to CS 1* In proceedings ACM SIGCSE Symposium, 1995

[5] Feldman, Michael *Ada95 Problem Solving and Program Design* Addison-Wesley, 1999

[6] Hanly, Jeri. R. and Koffman, Elliot B. *Problem Solving and Program Design in C* Addison-Wesley, 1999

[7] Barnes, David J., Fincher, Sally and Thompson, Simon *Introductory Problem Solving in Computer Science* In Daughton, Goretti and Magee, Patricia, editors, 5th Annual Conference on the Teaching of Computing, pages 36-39, Centre for Teaching Computing, Dublin City University, Dublin 9, Ireland, August 1997.

[8] Polya, George *How to Solve It*, Princetown University Press, 1957

[9] Stein, Lynn Andrea *What we Swept Under the Rug: Radically Rethinking CS1* Computer Science Education, Vol 8 (2), August 1998

[10] Stein, Lynn Andrea *Interactive Programming: Revolutionizing Introductory Computer Science* ACM Computing Surveys 28A (4), December 1996

[11] Culwin, Fintan *Object Imperatives!* in Proceedings ACM SIGCSE Symposium, 1999