

Principles Versus Artifacts in Computer Science Curriculum Design

Philip Machanick
School of IT and Electrical Engineering, University of Queensland
Brisbane, QLD 4072
Australia
philip@itee.uq.edu.au

Abstract

Computer Science is a subject which has difficulty in marketing itself. Further, pinning down a standard curriculum is difficult – there are many preferences which are hard to accommodate. This paper argues the case that part of the problem is the fact that, unlike more established disciplines, the subject does not clearly distinguish the study of principles from the study of artifacts. This point was raised in Curriculum 2001 discussions, and debate needs to start in good time for the next curriculum standard. This paper provides a starting point for debate, by outlining a process by which principles and artifacts may be separated, and presents a sample curriculum to illustrate the possibilities. This sample curriculum has some positive points, though these positive points are incidental to the need to start debating the issue.

1 Introduction

In discussions during the process leading up to the Curriculum 2001 standard, the issue of separating principles from artifacts was raised, but it was generally agreed that such a discussion would have taken too long to be practical in the time allowed for drawing up the new standard. This paper provides a starting point for such discussion, with the intention that the debate should be sufficiently advanced by the time the next curriculum standard is drawn up, that it will be practical to take this important issue into account.

Although a specific proposal is presented here, the intent is to start debate, rather than to argue for any specific curriculum design.

1.1 Why Separation of Principle and Artifact is Important

Why is it important that we separate out principle from artifact? Computer Science is a subject with a major image problem. We have difficulty in preventing outsiders from renaming the subject – in recent years, there has been tremendous pressure to rename it as IT. Just as an increasing number of Computer Science departments started caving in to the pressure, the fashionable name changed to information and communication technology (ICT). No one has attempted to rename theoretical physics as (for example) navel contemplation technology. These pressures appear to be a consequence of the fact that it is not clear to outsiders what the subject is. This paper argues that part of that confusion is the lack of clear separation of artifacts from principles: we study operating systems – but so does an *<insert your favourite brand>* Certified System Engineer. Mechanical engineering is the academic subject which qualifies someone to design a truck, with physics as the underlying theoretical discipline; no one expects mechanical engineering academics or physics professors to teach truck driving. The distinction between the academic discipline and practitioners is clear.

Further, by failing to distinguish clearly when we are addressing principle and when we are addressing issues specific to an artifact, we are exposing ourselves to commercial pressures and undue costs. I have observed vendors of commercial products who were extremely offended by universities which were not interested in a donation of their product, on condition that it form part of the curriculum – yet I do not imagine anyone would expect a university to have a Pontiac studies course within its mechanical engineering programme. On the cost side, since artifacts are seen as part of the curriculum, we are continually under pressure to have the latest hardware and software, with concomitant costs in upgrades, maintenance, capital outlay and obsolescence-driven curriculum revision. The concern about the lack of clarity about what the subject is about has led to movements such as pushing programming out of the start of the curriculum [GHMP01], but a more comprehensive reassessment of the curriculum is necessary to address the problem.

While there has been some debate in the literature along similar lines about the need to make Software Engineering education closer to that for a conventional branch of engineering [Bab99], that debate has also started from the point that Software Engineering is not Computer Science, but should be the application of the principles of Computer Science to an engineering approach to software development [Par99]. Clearly, this debate about Software Engineering Education requires that Computer Science be clear on where principles end and artifacts begin. The classic split between pure and applied science is at the point where the study of existing artifacts becomes more important than abstract principles; the split between applied science and engineering is where the study of artifacts progresses from understanding existing artifacts to designing new ones.

1.2 Proposed Approach

How are we to address this problem?

A good start would be to look at how a curriculum can be structured around principles. Making such a change in a single step may be too radical, as such a change would imply complete redesign of all curricula.

Accordingly, the approach adopted in this paper is one of identifying the underlying principles in a selection of existing courses drawn from the Core subjects of Curriculum 2001 [AI01]. Once these principles are identified, a model curriculum is proposed in which the principles become the starting point, and the artifacts examples to illustrate them. Finally, the resulting proposal is discussed, to see if any advantage accrues from changing to a principles-first curriculum. This evaluation is aimed at evaluating the broad idea, rather than a specific curriculum proposal but a specific example is useful to make the argument less abstract.

1.3 Remainder of Paper

The remainder of this paper is structured as follows. The Core defined in Curriculum 2001 is outlined in Section 2, with a division of the Core hours into “principle” and “artifact”. Section 3 splits off the “principle” hours, as a starting point for defining a curriculum in which “principle” would come first. In 4, a model curriculum is proposed, to illustrate how the separation of principle and artifact could be achieved in practice. In Section 5, the potential gains from the proposed change in curriculum strategy are weighed up against costs. Section 6 concludes the paper with an overall summary and some ideas to take the curriculum reform process forward.

2 Starting Point – Curriculum 2001 Core

2.1 Introduction

Curriculum 2001 defines the core curriculum as the minimal curriculum which any Computer Science degree program should include, but which is not itself sufficient to form a complete programme [AI01]. In other words, it is the common subset of all variations contemplated by the CC-2001 task force. The approach used here is to assume that the Core remains unchanged, but that a re-ordering of topics relative to current practice could achieve the goal of separating principles from artifacts.

In this section, the core is itemized as it stands, as a basis for isolating principles, but with hours split into “principle” and “artifact”. Some commentary on the likely split between artifacts and principles is made at this stage. However, it is recognized that such a split, in practice, cannot happen without considerable debate, so the split made here, while possible to defend, is not justified in detail but is instead presented as a starting point for debate.

The remainder of the section starts by itemizing the core with a breakdown into principle and artifact, with a brief justification of the breakdown. Next, an approach for using the split as basis for curriculum design is presented, followed by an overall summary.

2.2 The Core of Curriculum 2001

First, it is useful to write out the core on its own, since published versions of the curriculum standard generally show the core in relation to the complete body of knowledge, so here it is. Further, in addition to the usual annotation of core hours, a first cut at identifying a split of hours into principle versus artifact is made. In each case, the number of hours is split as $principle + artifact = total$, with the exception of the Discrete Structures knowledge area, which is all principle.

- DS. Discrete Structures ($43 + 0 = 43$ core hours)
 - DS1. Functions, relations, and sets (6)

- DS2. Basic logic (10)
- DS3. Proof techniques (12)
- DS4. Basics of counting (5)
- DS5. Graphs and trees (4)
- DS6. Discrete probability (6)
- PF. Programming Fundamentals (17 + 21 = 38 core hours)
 - PF1. Fundamental programming constructs (1 + 8 = 9)
 - PF2. Algorithms and problem-solving (6 + 0 = 6)
 - PF3. Fundamental data structures (6 + 8 = 14)
 - PF4. Recursion (3 + 2 = 5)
 - PF5. Event-driven programming (1 + 3 = 4)
- AL. Algorithms and Complexity (21 + 10 = 31 core hours)
 - AL1. Basic algorithmic analysis (4 + 0 = 4)
 - AL2. Algorithmic strategies (5 + 1 = 6)
 - AL3. Fundamental computing algorithms (5 + 7 = 12)
 - AL4. Distributed algorithms (1 + 2 = 3)
 - AL5. Basic computability (6 + 0 = 6)
- AR. Architecture and Organization (10 + 26 = 36 core hours)
 - AR1. Digital logic and digital systems (3 + 3 = 6)
 - AR2. Machine level representation of data (2 + 1 = 3)
 - AR3. Assembly level machine organization (1 + 8 = 9)
 - AR4. Memory system organization and architecture (1 + 4 = 5)
 - AR5. Interfacing and communication (1 + 2 = 3)
 - AR6. Functional organization (1 + 6 = 7)
 - AR7. Multiprocessing and alternative architectures (1 + 2 = 3)
- OS. Operating Systems (11 + 7 = 18 core hours)
 - OS1. Overview of operating systems (1 + 1 = 2)
 - OS2. Operating system principles (2 + 0 = 2)
 - OS3. Concurrency (5 + 1 = 6)
 - OS4. Scheduling and dispatch (1 + 2 = 3)
 - OS5. Memory management (2 + 3 = 5)
- NC. Net-Centric Computing (3 + 12 = 15 core hours)
 - NC1. Introduction to net-centric computing (1 + 1 = 2)
 - NC2. Communication and networking (1 + 6 = 7)
 - NC3. Network security (1 + 2 = 3)
 - NC4. The web as an example of client-server computing (0 + 3 = 3)
- PL. Programming Languages (7 + 14 = 21 core hours)
 - PL1. Overview of programming languages (1 + 1 = 2)
 - PL2. Virtual machines (1 + 0 = 1)
 - PL3. Introduction to language translation (1 + 1 = 2)
 - PL4. Declarations and types (1 + 2 = 3)
 - PL5. Abstraction mechanisms (1 + 2 = 3)

- PL6. Object-oriented programming ($2 + 8 = 10$)
- HC. Human-Computer Interaction ($2 + 6 = 8$ core hours)
 - HC1. Foundations of human-computer interaction ($2 + 4 = 6$)
 - HC2. Building a simple graphical user interface ($0 + 2 = 2$)
- GV. Graphics and Visual Computing ($1 + 2 = 3$ core hours)
 - GV1. Fundamental techniques in graphics ($1 + 1 = 2$)
 - GV2. Graphic systems ($0 + 1 = 1$)
- IS. Intelligent Systems ($7 + 3 = 10$ core hours)
 - IS1. Fundamental issues in intelligent systems ($1 + 0 = 1$)
 - IS2. Search and constraint satisfaction ($4 + 1 = 5$)
 - IS3. Knowledge representation and reasoning ($2 + 2 = 4$)
- IM. Information Management ($4 + 6 = 10$ core hours)
 - IM1. Information models and systems ($2 + 1 = 3$)
 - IM2. Database systems ($0 + 3 = 3$)
 - IM3. Data modeling ($2 + 2 = 4$)
- SP. Social and Professional Issues ($12 + 4 = 16$ core hours)
 - SP1. History of computing ($1 + 0 = 1$)
 - SP2. Social context of computing ($2 + 1 = 3$)
 - SP3. Methods and tools of analysis ($1 + 1 = 2$)
 - SP4. Professional and ethical responsibilities ($3 + 0 = 3$)
 - SP5. Risks and liabilities of computer-based systems ($1 + 1 = 2$)
 - SP6. Intellectual property ($2 + 1 = 3$)
 - SP7. Privacy and civil liberties ($2 + 0 = 2$)
- SE. Software Engineering ($6 + 25 = 31$ core hours)
 - SE1. Software design ($2 + 6 = 8$)
 - SE2. Using APIs ($1 + 4 = 5$)
 - SE3. Software tools and environments ($0 + 3 = 3$)
 - SE4. Software processes ($1 + 1 = 2$)
 - SE5. Software requirements and specifications ($1 + 3 = 4$)
 - SE6. Software validation ($1 + 2 = 3$)
 - SE7. Software evolution ($0 + 3 = 3$)
 - SE8. Software project management ($0 + 3 = 3$)

In some areas, it is obvious what separates out as principle as opposed to artifact. For example, the DS area is almost entirely composed of principle, so there should not be too much controversy about my placing the entire category in “principle”. At the other end of the scale, the SE area is mostly concerned about construction of artifacts, though some principles could be teased out; some may consider my allocation of nearly 20% of the time to “principle” as generous.

Some areas are harder to characterize. PL, for example, as the headings are labeled, could be almost entirely principle. Another spin could be put on the topics covered, making PL almost entirely concerned with artifacts. In the one case, a relatively theoretical view of languages could be presented (formal languages and automata, type systems, etc.). In the other, mechanisms and details of specific languages could be the entire focus of a course.

Clearly, a standards process would require detailed discussion of the breakdown, and such a discussion could also result in a redefinition of the Core. However, the breakdown given here is a reasonable starting point for discussion.

The overall effect of the breakdown is that 143 hours are “principle”, and 137 hours are “artifact”. In other words, on this (perhaps somewhat arbitrary) classification, the Core is slightly more than half principle, which is not a bad start – one would expect a substantial part of the subset of the discipline which is required of all students to consist of relatively immutable principle.

2.3 Likely Basis for Splitting

There can be very different interpretations of the Core as it is stated. These differences, as noted, could make for very different approaches to splitting between artifact and principle. Any basis proposed here, therefore (it is worth emphasizing), can only be a starting point for debate.

However, it is worth establishing some design principles as a starting point, to arrive at a consistent curriculum design. These principles are not important; others could be used, and a different design would result.

First, it is worthwhile establishing principles which are relatively immutable, and hence, can become anchors for a curriculum in which artifacts can and often do change frequently.

Second, it is worth establishing principles which can have as general an applicability as possible, so they can be taught in one place and reinforced by application across the curriculum. (As an example from disciplines where the split of artifact and principle is well-established, partial differential equations are widely used in Physics and various branches of Engineering.)

Finally, it is useful to distinguish principles specific to Computer Science, so parts of the curriculum which are properly part of the subject, as opposed to essential background, can be identified. Clearly, this last design principle is open to considerable contention, but this paper is about starting debate, so that is not a problem.

A general design principle which can be globally applied is to put more factual material earlier, while material requiring more problem-solving is put later; a finer distinction could be made using Bloom's Taxonomy [Blo56], which has been used as a basis for designing Computer Science curricula in the past [Mac00].

Starting from these design principles, it becomes clear that the aspects of subjects which are clearly immutable, fundamental principles which clearly characterize the discipline ought to be taught relatively early in the curriculum as a basis for less fundamental concepts. Aspects which are fundamental but less specific to the discipline may also be taught relatively early, for a different reason: to allow students the flexibility to switch disciplines.

2.4 Summary

The existing core is just over half principle and just under half artifacts, according to one interpretation of the topics. On this basis, it is possible to proceed to identifying principles, and regrouping topics so that key principles appear first. Such an exercise can clearly be done with greater rigor than was applied here but a significant aspect of a realistic exercise would be soliciting opinion from a wide range of stakeholders, so an approximate exercise is the best which can be done without widespread consultation, and is sufficient to illustrate the idea of a curriculum based on separation of principle and artifact.

3 Isolating the Principles

First, let's consider the core pared down to the hours needed for principles, then look at whether to regroup some of the smaller areas.

Clearly, Discrete Structures stays unchanged (so there is no point in listing the detail again), but most of the other areas lose a significant number of hours.

Here is the modified core:

- DS. Discrete Structures (43 core hours)
- PF. Programming Fundamentals (17 core hours)
 - PF1. Fundamental programming constructs (1)
 - PF2. Algorithms and problem-solving (6)
 - PF3. Fundamental data structures (6)
 - PF4. Recursion (3)
 - PF5. Event-driven programming (1)
- AL. Algorithms and Complexity (21 core hours)
 - AL1. Basic algorithmic analysis (4)
 - AL2. Algorithmic strategies (5)
 - AL3. Fundamental computing algorithms (5)
 - AL4. Distributed algorithms (1)

- AL5. Basic computability (6)
- AR. Architecture and Organization (10 core hours)
 - AR1. Digital logic and digital systems (3)
 - AR2. Machine level representation of data (2)
 - AR3. Assembly level machine organization (1)
 - AR4. Memory system organization and architecture (1)
 - AR5. Interfacing and communication (1)
 - AR6. Functional organization (1)
 - AR7. Multiprocessing and alternative architectures (1)
- OS. Operating Systems (11 core hours)
 - OS1. Overview of operating systems (1)
 - OS2. Operating system principles (2)
 - OS3. Concurrency (5)
 - OS4. Scheduling and dispatch (1)
 - OS5. Memory management (2)
- NC. Net-Centric Computing (3 core hours)
 - NC1. Introduction to net-centric computing (1)
 - NC2. Communication and networking (1)
 - NC3. Network security (1)
- PL. Programming Languages (7 core hours)
 - PL1. Overview of programming languages (1)
 - PL2. Virtual machines (1)
 - PL3. Introduction to language translation (1)
 - PL4. Declarations and types (1)
 - PL5. Abstraction mechanisms (1)
 - PL6. Object-oriented programming (2)
- HC. Human-Computer Interaction (2 core hours)
 - HC1. Foundations of human-computer interaction (2)
- GV. Graphics and Visual Computing (1 core hours)
 - GV1. Fundamental techniques in graphics (1)
- IS. Intelligent Systems (7 hours)
 - IS1. Fundamental issues in intelligent systems (1)
 - IS2. Search and constraint satisfaction (4)
 - IS3. Knowledge representation and reasoning (2)
- IM. Information Management (4 core hours)
 - IM1. Information models and systems (2)
 - IM3. Data modeling (2)
- SP. Social and Professional Issues (12 core hours)
 - SP1. History of computing (1)
 - SP2. Social context of computing (2)

- SP3. Methods and tools of analysis (1)
- SP4. Professional and ethical responsibilities (3)
- SP5. Risks and liabilities of computer-based systems (1)
- SP6. Intellectual property (2)
- SP7. Privacy and civil liberties (2)
- SE. Software Engineering (6 core hours)
 - SE1. Software design (2)
 - SE2. Using APIs (1)
 - SE4. Software processes (1)
 - SE5. Software requirements and specifications (1)
 - SE6. Software validation (1)

If a course is roughly 40-45 hours, some of the “core” headings are a bit thin. Let’s group them into 4 major categories, which could provide a start towards designing an introductory sequence of 4 courses taken over 2 semesters:

- Discrete Structures (43 core hours)
- Fundamental Algorithmic Concepts (41 core hours)
 - PF1. Fundamental programming constructs (1)
 - PF2. Algorithms and problem-solving (6)
 - PF3. Fundamental data structures (6)
 - AR2. Machine level representation of data (2)
 - PF4. Recursion (3)
 - PF5. Event-driven programming (1)
 - AL1. Basic algorithmic analysis (4)
 - AL2. Algorithmic strategies (5)
 - AL3. Fundamental computing algorithms (5)
 - GV1. Fundamental techniques in graphics (1)
 - AL4. Distributed algorithms (1)
 - AL5. Basic computability (6)
- Principles of Systems (23 core hours)
 - AR1. Digital logic and digital systems (3)
 - AR3. Assembly level machine organization (1)
 - AR7. Multiprocessing and alternative architectures (1)
 - OS1. Overview of operating systems (1)
 - OS2. Operating system principles (2)
 - OS3. Concurrency (5)
 - OS4. Scheduling and dispatch (1)
 - OS5. Memory management (2)
 - PL2. Virtual machines (1)
 - PL3. Introduction to language translation (1)
 - NC1. Introduction to net-centric computing (1)
 - NC2. Communication and networking (1)
 - NC3. Network security (1)
 - IM1. Information models and systems (2)

- IM3. Data modeling (2)
- Human-Machine Interface (14 core hours)
 - HC1. Foundations of human-computer interaction (2)
 - SP1. History of computing (1)
 - SP2. Social context of computing (2)
 - SP3. Methods and tools of analysis (1)
 - SP4. Professional and ethical responsibilities (3)
 - SP5. Risks and liabilities of computer-based systems (1)
 - SP6. Intellectual property (2)
 - SP7. Privacy and civil liberties (2)

This revised grouping creates two relatively full courses: Discrete Structures and Fundamental Algorithmic Concepts, and two course fragments which together add up to 37 hours, Principles of Systems and Human-Machine Interface. The time that is left could be filled either by expanding the two smaller courses with more material, or by adding in other electives.

What is left out is 5 core hours of Programming Languages, 7 core hours of Intelligent Systems, and 6 core hours of Software Engineering. However, the principles in these areas become very abstract if not included with study of specific artifacts, or too theoretical for an introductory treatment, so leaving them until later is a reasonable choice to make.

The combination resulting from this grouping of “principle” areas selected from the core differs somewhat from convention, in that some areas are not conventionally included in one course, and a practical course covering so much ground would be difficult to construct with adequate laboratory time. However, if the emphasis is on principles, more ground can potentially be covered, as having students write programs and do other relatively time-consuming lab exercises could be avoided.

4 A Model Curriculum

Let us now consider a possibility for constructing a complete curriculum from the starting point given here. The idea here is to show feasibility of the idea, not to prescribe a specific model. However, the principles used are chosen to arrive at a potentially workable curriculum.

The curriculum proposed here assumes a 4-year program with 40-45 hours per course, and 4 such courses per year. It allows space for additional topics in other subjects, including mathematics and Computer Science electives. Space for electives is increased in higher years, though the curriculum is designed to allow for taking a smaller fraction of the total as part of a programme majoring in another subject.

It expands on the introductory core defined in Section 3, on the basis of expanding the hours allocated for the courses with a smaller number of core hours, and works from that start towards development of less study of principles and more study of artifacts in higher years.

Core hours left out of the introductory courses are added back, with the “principle” core hours earlier than the “artifact” core hours.

The progression from Year 1 starts from establishing principles, which leads first to understanding existing artifacts, then to building small-scale artifacts and, finally, to the full analyze-design-build cycle for relatively large-scale artifacts. By the end of Year 3, students not intending to do major software projects might wish to exit, so the Core is completed by the end of Year 3.

In the remainder of this section, each year is presented separately, concluding with an overall summary.

4.1 Year 1

Discrete Structures and Fundamental Algorithmic Concepts are not presented in detail, since they are not changed from the earlier versions. These two topics are the least difficult to explain in terms of how to present principle rather than focus on artifacts. Discrete Structures is an inherently mathematical topic, while Fundamental Algorithmic concepts can be presented largely mathematically, with emphasis on pseudocode to express algorithms, and emphasis on properties of algorithms (space and time complexity) rather than on implementations. Despite the relatively abstract presentation implied by a focus on principles, it is also possible to have a significant laboratory component to such a course, including exercises in predicting effects of scaling an algorithm up, and measuring the actual effect in the lab. Such empirical exercises are often neglected, but are a useful learning experience [San02].

Principles of Systems is likely to be the most controversial subject proposed here, because the obvious interpretation of the many of the headings is the study of specific artifacts, which implies much more time than proposed here. However, the ordering of knowledge units is intended to imply exploring the virtual machine concept, rather than digging deep into specific implementations. Principles which would be emphasized would include the theoretical aspects of digital logic, resource management, translation and data modeling. Naturally, these principles could not be explored in great depth in a single wide-ranging course, but they could be introduced with sufficient depth to give students a reading knowledge of the concepts.

The proposed Human-Machine Interface course is a consequence of looking for new alignments. Participatory design [ABAK95, HHS91, KNF⁺01] and user-centred design [KAD⁺96] are ideas which have growing currency in the HCI field (or more generally, design for usability) and they imply a concern for the user. Usability, in general, implies some concern for the users' needs. It seems a reasonable generalization of such concerns to go further, and align HCI with social and ethical issues.

The proposed breakdown of course follows.

- Discrete Structures (43 hours)
- Fundamental Algorithmic Concepts (41 hours)
- Principles of Systems (44 hours)
 - PL2. Virtual machines (1)
 - AR1. Digital logic and digital systems (3)
 - AR3. Assembly level machine organization (1)
 - AR7. Multiprocessing and alternative architectures (1)
 - OS1. Overview of operating systems (1)
 - OS2. Operating system principles (3)
 - OS3. Concurrency (5)
 - OS4. Scheduling and dispatch (4)
 - OS5. Memory management (4)
 - NC1. Introduction to net-centric computing (1)
 - NC2. Communication and networking (10)
 - NC3. Network security (1)
 - PL3. Introduction to language translation (3)
 - IM1. Information models and systems (2)
 - IM3. Data modeling (4)
- Human-Machine Interface (42 hours)
 - HC1. Foundations of human-computer interaction (12)
 - SP1. History of computing (4)
 - SP2. Social context of computing (4)
 - SP3. Methods and tools of analysis (3)
 - SP4. Professional and ethical responsibilities (6)
 - SP5. Risks and liabilities of computer-based systems (4)
 - SP6. Intellectual property (4)
 - SP7. Privacy and civil liberties (5)

4.2 Year 2

In year 2, some of the missing “principle” core hours are added back, as well as “principle” knowledge units which are not part of the core. Artifacts are still mainly used as examples at this level, though some progression to tools required for construction of artifacts.

The emphasis at this stage is on establishing the necessary concepts to do efficient designs and implementations, which will be required in later years.

To establish a firm basis in tools, programming language core hours are completed at this level. Data structures and algorithms are covered in more depth to provide a basis for efficient design and implementation, and Systems topics are expanded to provide a more deeper understanding of how tools, hardware and software interact.

A possible course structure at this level, allowing space for electives is:

- Principles of Programming Languages (45 hours)
 - PL1. Overview of programming languages (1)
 - PL2. Virtual machines (1)
 - PL3. Introduction to language translation (2)
 - PL4. Declarations and types (4)
 - PL5. Abstraction mechanisms (4)
 - PL6. Object-oriented programming (8)
 - PL7. Functional programming (7)
 - PL8. Language translation systems (8)
 - PL9. Type systems (5)
 - PL10. Programming language semantics (3)
 - PL11. Programming language design (2)
- Principles of Efficient Design (45 hours)
 - AL1. Basic algorithmic analysis (1)
 - AL2. Algorithmic strategies (2)
 - AL3. Fundamental computing algorithms (7)
 - PF3. Fundamental data structures (6)
 - AL4. Distributed algorithms (2)
 - AL5. Basic computability (2)
 - AL6. The complexity classes P and NP (5)
 - AL7. Automata Theory (10)
 - AL8. Advanced Algorithmic Analysis (5)
 - AL9. Cryptographic Algorithms (5)
- Principles of System Design (45 hours)
 - AR1. Digital logic and digital systems (3)
 - AR3. Assembly level machine organization (2)
 - AR7. Multiprocessing and alternative architectures (2)
 - OS1. Overview of operating systems (1)
 - OS2. Operating system principles (1)
 - OS3. Concurrency (7)
 - OS4. Scheduling and dispatch (3)
 - OS5. Memory management (5)
 - NC1. Introduction to net-centric computing (1)
 - NC2. Communication and networking (6)
 - NC3. Network security (2)

- IM1. Information models and systems (2)
- IM3. Data modeling (2)
- IM4. Relational databases (6)
- IM5. Database query languages (2)
- Elective (45 hours)

4.3 Year 3

In year 3, the study of specific artifacts is emphasized, with courses on construction of artifacts introduced. Construction of artifacts at this level is relatively introductory in scale, since the principle-artifact relationship is still being explored, but the examples can be relatively sophisticated.

Year 3 could be a valid exit point for students more concerned with broad understanding of the subject than with the ability to construct major pieces of software.

A possible course structure at this level, allowing space for electives is:

- Intelligent Systems Design (45 hours)
 - IS1. Fundamental issues in intelligent systems (2)
 - IS2. Search and constraint satisfaction (10)
 - IS3. Knowledge representation and reasoning (10)
 - IS4. Advanced search (5)
 - IS5. Advanced knowledge representation and reasoning (5)
 - IS6. Agents (5)
 - IS7. Natural language processing (8)
- Software Design and Implementation (45 hours)
 - SE1. Software design (8)
 - SE2. Using APIs (5)
 - SE3. Software tools and environments (3)
 - SE4. Software processes (2)
 - SE5. Software requirements and specifications (10)
 - SE6. Software validation (3)
 - SE7. Software evolution (3)
 - SE8. Software project management (3)
 - SE10. Formal methods (8)
- Operating Systems or Network Project (45 hours)
 - some PF lectures – fill in missing Core hours
 - emphasis on building on known APIs to build programming skills
 - use of OS or NW to make Systems concepts concrete
 - small examples but sophisticated to exercise PL, AL etc. skills
- Elective (45 hours)

4.4 Year 4

In the final year, the focus moves to design and implementation of specific artifacts, leading to being able to do relatively major projects.

Consequently, at this level, there are no required lecture-based courses in Computer Science, though there may be two Computer Science electives – with the option of any other electives which may help to establish a specialization.

There should be at least one major project, equivalent in credit to 2 normal courses. This project (depending on the emphasis of the programme) should either take the form of a major analysis, design and implementation exercise – along the lines of a traditional Software Engineering course project – or a research project.

4.5 Summary

5 Evaluation

5.1 Introduction

The model curriculum as outlined here has a few novel features, some intentional, others artifacts of starting from a Core which was not designed to be used in this way. Clearly, it would be possible to arrive at a very different outcome from the same starting point, so dwelling on the details of this curriculum at length is not a very useful exercise. However, the fact that it has novel features makes it worth considering whether the process by which it was derived was useful.

Consequently, this section briefly reviews benefits and costs of a change such as that proposed here, with some reference to the model curriculum – though also an attempt at generalizing beyond this specific example.

5.2 Benefits

A somewhat unexpected benefit of moving principles early is that the first-year curriculum could relatively easily be adapted to a variety of requirements, from an introduction for non-specialists through to an introduction for engineers and scientists. While some of the underlying basis for the material is strongly mathematical, the broad survey style of the topics (other than Discrete Structures) identified for the first year lends itself to an introduction for the non-specialist.

The first year courses could relatively easily be presented as one set of lectures to two different groups, with different laboratory and assessment materials. The scientists and engineers could be assumed to have done (or be doing) Discrete Structures, and be given lab work, tutorials, assignments and assessment incorporating a significant component of mathematical content. Students from other backgrounds, such as social sciences or arts, could be given very different work based on the same lectures.

Another clear benefit of moving “principle” earlier is that very little of the curriculum is technology-dependent, especially at the early stages. This is no big surprise, as this was an intended outcome.

Finally, the goal of making the distinction between the science (or engineering) aspects of computing, versus the use of the technology, is supported by reducing the use of artifact-associated keywords with courses, especially at early stages. A student studying a first-year programme with course titles like Discrete Structures, Fundamental Algorithmic Concepts, Principles of Systems and Human-Machine Interface is clearly not doing the same subject as someone learning to maintain a specific operating system or model of router.

5.3 Costs

A change like this cannot be without cost. Although much material across the curriculum would remain, the ordering is different. It is not simply a matter of using chunks out of existing texts in an interleaved order: a topic presented in an introductory way is not the same as a topic presented to a more advanced student, even if the content is the same. For an relatively inexperienced learner, the focus should be on lower cognitive skills; for a more advanced learner, on exercising more advanced cognitive skills [Mac00].

Aside from modifying a substantial portion of existing course materials, many academics would require a mindset change to adapt to the new approach. Learning artifacts first is a highly entrenched part of the Computer Science academic culture, to the extent that when I was chairing PFG1 (Introductory Courses) for Curriculum 2001, members of the focus group were surprised that I was able to avoid a “first programming language” war.

5.4 Conclusion

The potential benefits outweigh the costs. That is self-evident if the benefits are accepted, because the costs would be borne once (during the transition), whereas the benefits would be ongoing.

The big challenge therefore is to have the benefits accepted.

6 Conclusions

6.1 Introduction

This paper has introduced a starting point for debating separation of principles and artifacts in Computer Science curriculum design. This starting point is a data point for debate: though some interesting ideas have emerged, the model curriculum presented here is only for illustration of the process, and is not put forward as a proposal in itself.

This section summarizes the ideas established in this paper, and proposes a way forward.

6.2 Summary

Division of the existing Curriculum 2001 Core into principle and artifact can be done, as illustrated here, though a more rigorous process would have to be based on broad consultation, since opinion plays a big role in curriculum design. However, this paper has presented, by way of example, a possible division.

Given such a division, it is possible to group a significant fraction of the “principle” knowledge into earlier courses; such courses then have a natural tendency to have names which do not reflect artifacts.

From this starting point, it becomes possible to work towards final-year courses which are strongly based in creation of specific artifacts, with convenient exit points for students with less interest in large-scale software development. The first year sequence, possibly minus mathematical content, could be a nice package for the non-Computer Science major who needs some exposure to basic ideas of the discipline; the first 3 years a useful exposure for students who do not want to develop major software systems.

6.3 How to Move Forward

To take these ideas forward, an advance working party for Curriculum 2011 could be established, to build consensus on how principle and artifact should be divided, within the framework of the existing Core and Body of Knowledge. This split could be used as a basis for a more rigorous development of a model curriculum than that presented here.

In the longer term, it would be useful to redefine the body of knowledge using keywords which are less specific to artifacts. For example: resource management, queuing theory, formal languages, complexity theory.

In the long term, separating principle and artifact could become a basis for establishing clearly boundaries between subjects such as Computer Science, Information Systems, Software Engineering, Electrical Engineering and Information Technology.

6.4 Overall Conclusion

This paper presents some ideas for curriculum reform, building on discussions which took place during Curriculum 2001, but which were not possible to include in that proposal for lack of time.

Since the kind of reform envisaged here would take considerable effort to work through properly, it would be useful if the debate could start as soon as possible. Hence, this paper places the issues in the public domain to encourage discussion.

Whether such change is worthwhile is a matter for debate; having considered the idea for some time, I believe that the issue is important and worth taking forward.

References

- [ABAK95] Divyakant Agrawal, John L. Bruno, Amr El Abbadi, and Vashudha Krishnaswamy. Managing concurrent activities in collaborative environments. In *Conference on Cooperative Information Systems*, pages 112–124, Vienna, Austria, 1995.
- [AI01] ACM and IEEE Computer Society. *Computing Curricula 2001 Computer Science*. ACM/IEEE-CS, December 2001. <http://www.computer.org/education/cc2001/final/index.htm>.
- [Bab99] Robert L. Baber. Software engineering education: issues and alternatives. *Annals of Software Engineering*, 6:39–59, 1999.
- [Blo56] Benjamin S Bloom, editor. *Taxonomy of Educational Objectives: Book 1 Cognitive Domain*. Longman, London, 1956.
- [GHMP01] Judith Gersting, Peter B. Henderson, Philip Machanick, and Yale N. Patt. Programming early considered harmful. In *Proc. SIGCSE-2001*, pages 402–403, Charlotte, NC, 21-25 February 2001.
- [HHS91] R. R. Harper, J. A. Hughes, and D. Z. Shapiro. *Studies in computer supported cooperative work: theory, practice and design*, chapter Harmonious working and CSCW: computer technology and air traffic control, pages 225–234. North-Holland Publishing Co., Amsterdam, 1991. JM Bowers and SD Benford (editors).

- [KAD⁺96] John Karat, Michael E. Atwood, Susan M. Dray, Martin Rantzer, and Dennis R. Wixon. User centered design: quality or quackery? In *Proceedings of the CHI '96 conference companion on Human factors in computing systems : common ground*, pages 161–162, Vancouver, British Columbia, Canada, 1996. ACM Press.
- [KNF⁺01] Scott R. Klemmer, Mark W. Newman, Ryan Farrell, Mark Bilezikjian, and James A. Landay. The designers' outpost: a tangible interface for collaborative web site. In *Proceedings of the 14th annual ACM symposium on User interface software and technology*, pages 1–10, Orlando, Florida, 2001.
- [Mac00] Philip Machanick. Experience of applying Bloom's Taxonomy in three courses. In *Proc. Southern African Computer Lecturers' Association Conference*, pages 135–144, Strand, South Africa, June 2000.
- [Par99] David Lorge Parnas. Software engineering programmes are not computer science programmes. *Annals of Software Engineering*, 6:19–37, 1999.
- [San02] I. D. Sanders. Teaching empirical analysis of algorithms. In *Proc. 33rd SIGCSE Technical Symposium*, pages 321–325, Cincinnati, Kentucky, 27 February–3 March 2002.