Curriculum 2020

Philip Machanick*

Department of Computer Science
University of the Witwatersrand
2050 Wits
South Africa

philip@cs.wits.ac.za
http://www.cs.wits.ac.za/~philip/

Abstract

Over most of the second half of the twentieth century, much of the curriculum debate in Computer Science assumed programming was the fundamental tool of the discipline, and a key subject of debate was the first programming language. By 2020, the focus had changed to one of emphasizing the fundamentals first, and developing skills related to coding later in the curriculum. The intent was to ensure that real fundamentals were taught first, reducing the need for frequent curriculum upheavals in introductory courses, where stability was most important. Also, the new curriculum ordering was intended to break away from the hacker culture which is hard to avoid with students who learn programming before they have developed design and abstraction skills. This paper presents a proposal for Curriculum 2020, in which the order of topics is designed to produce graduates with a solid theoretical foundation, for whom programming is almost a clerical task. The basic educational philosophy is called abstraction-first. Students are first introduced to abstraction as a client of predesigned abstractions, and gradually led to the point of designing their own abstractions. Theoretical foundations are introduced first, followed by practical application—also following the abstraction-first philosophy.

1. Introduction

ACM/IEEE Curriculum ’91 [ACM 1991] is recognized today as the most important change in our conception of computer science as a discipline not for what it said, but for what it left unsaid.

Unlike earlier curriculum standards, although it suggested specific courses, the main content of the document was knowledge units (KUs), which could in principle be used in any order.

* At time of writing, on sabbatical at Department of Electrical Engineering and Computer Science, The University of Michigan, 1301 Beal Avenue, Ann Arbor, MI 48109-2122, USA.

A result of this new flexibility was a wave of experiments with variations on curriculum order over the next 20 years. This document summarizes the direction most widely accepted at the time of the 2020 Curriculum review, and contains a suggested curriculum summarizing that direction.

The biggest achievement resulting from curriculum work since 1991 has been the agreement on what the real fundamentals of the discipline are. Remarkably, in a discipline characterized by exponential performance advances across many areas, the Curriculum 2020 committee found little to change since Curriculum 2010. Most of those changes were in the relatively advanced courses, which is what one would expect of a mature engineering discipline. We have to compare this position with curriculum debates of the last century, when one of the most common issues was the first programming language, a debate which is of little interest today.

Since the changes have been relatively small, this document only contains a summary of the major issues raised in Curriculum 2010, and focuses on the changes. However, a retrospective of key milestones in the software industry is also provided, as background to the development of curricula since 2000.

In order to provide a framework for the Curriculum, the next section of this paper contains a brief review of the history of Computer Science as an Engineering discipline, with a discussion of the basis for the current abstraction-first approach to education. Section 3 recounts some events of the previous 20 years which influenced curricula over that time, and Section 4 contains the curriculum with brief annotations. Section 5 contains more detail of the new areas. Section 6 concludes the document with an overall discussion.

2. Computer Science as Engineering

It is now widely recognized that Computer Science followed the classic sequence of pre-engineering to engineering, just as happened in other engineering disciplines.

For example, before Newton's laws were in widespread use in areas like shipbuilding, major disasters could occur because there was no scientific basis for predicting whether a given design would work [Baber 1997*a*]. Today, any established engineering discipline is based on such fundamentals and any routine piece of work is mostly predictable in terms of technical issues (if not human issues).

An important step in the path towards establishing Computer Science as a proper engineering discipline was a proper understanding of what the fundamentals were—for example, that formal logic underlies much of the discipline. Indeed, some complained towards the end of the last century that the starting point of Computer Science should be based on the work of pioneering mathematicians like Alan Turing [Goldweber *et al.* 1997]. While there were certainly those who emphasized a mathematical approach to

introducing Computer Science as opposed to the other extreme of Computer Science as programming [Berztiss 1987, Gries 1991], this was very much a minority view.

The final big step came in realizing that the theoretical component of Computer Science, as with other engineering disciplines, can be presented in an introductory way. The key insight which was applied to putting theory into its proper place is that other sciences—especially as taught in engineering curricula—do not emphasize higher-level problem-solving skills in introductory courses. A classic distinction was between the Computer Science approach of teaching programming (which essentially is a design skill) before teaching theorem proving in formal logic (which is a relatively mechanical skill). Theory in Computer Science was widely considered to be "difficult" because it was taught as a problem-solving skill. Compare this with a classic engineering curriculum where theory was taught without reference to solving real-world problems (though such problems may be used as examples, they were not solved from scratch). A typical engineering curriculum in a four-year degree contained very little specific to a given branch of engineering in its initial stages, whereas Computer Science was almost the opposite: students were introduced to programming early, and the theory of programming late if at all [Baber 1997*b*].

A typical example of the difference between pre-engineering Computer Science and more established engineering disciplines is the way proof by induction used to be taught in Computer Science. If introduced at all, it was typically introduced in the form of finding a proof in an area such as algorithm analysis, where the formulation of the proof was not known in advance. Techniques such as finding closed forms of recurrence relations would be needed to decided what to prove. By comparison, students in other disciplines would have been taught proof by induction by being given a result to prove, a much easier problem to solve.

Many earlier attempts at placing theory in computer science either took the view that it was too hard and therefore should be done late if at all, with an opposing view that layered a heavy theoretical emphasis on the course but did not recognize the need to place design and abstraction skills later in the curriculum. A key realization among computer science education researchers early this century was that both schools were wrong, and that moving to a more classic "engineering" ordering of topics would make it more natural to introduce theory at an early stage of the curriculum.

Once this insight was in place, it became necessary to reconsider the order of topics in a curriculum. The idea of an *abstraction-first* order started to develop late in the century [Machanick 1998*a*]. The idea of the abstraction-first model was to start by introducing students to concepts without requiring them to develop design skills. Thus, algorithm analysis was introduced without a requirement to design programs. Class libraries were introduced as virtual machines for building software before

implementation detail was introduced. Theory was introduced in the form of proofs where what was to be proved was already known.

With this new orientation, it became possible to make the early part of the curriculum relatively fixed, and to move the technology-specific areas to later. Two useful consequences resulted from this change. A body of experience could be built up in teaching introductory material, the time when students were most often lost, and trend-dependent equipment purchases could be restricted to later years of study—as in other engineering disciplines.

A key insight in the transition to a modern-style engineering curriculum then was that the theory component of CS101 should be at the same level of difficulty as for example Calculus 101.

3. Key Events of the Early 21st Century

A few events have shaped our understanding of what the fundamentals of Computer Science really are, over the 20 years preceding the Curriculum 2020 process. Most significant of these is what has come to be known as the Millennium Muddle. Also influential however was the impact of widespread introduction of video on demand (VoD), subsequent to the roll-out of High Definition Television (HDTV). Finally, a significant issue is the underlying economics of system support, which changed our model of software development. This final issue is often referred to as the Bazaar Bungle.

This section first presents background on each of the Millennium Muddle VoD and the Bazaar Bungle, then outlines the lessons learned from both, in particular as they impacted upon curriculum issues.

3.1 The Millennium Muddle

As the year 2000 approached, there were increasing predictions of chaos resulting from what was then variously termed the Year 2K problem, the Millennium Bug, the Year 2000 Bug, and other variants on those names. The problem was seen as arising from old programs in which the year had been encoded as two digits, and which would do incorrect date arithmetic after the last two digits changed to zero. Note carefully, here we do not talk of the “new millennium” for reasons which are no doubt familiar to most readers but which will be explained shortly.

A large amount of effort was spent, with costs impossible to calculate, since many of the programs affected were very old and likely would have been replaced anyway. When the year 2000 arrived, a relatively small number of programs fell over, and mass chaos did not ensue as many had predicted—though the cost of fixing the problem was undoubtedly higher than would have been the case had it been fixed earlier.

However, later in the year, a surprising number of systems did fall over, including major sections of the internet, the worldwide network in common use at the time. This occurred on 29 February 2000, because many programmers had not correctly implemented the test for leap years, in which a century divisible by 400 *is* a leap year. The problem was extremely widespread because so many programs had been rewritten to fix the Year 2000 problem without sufficient time to test them as thoroughly as might otherwise have been the case. Coincidentally, the resulting date errors interacted with routing software widely used at the time on the internet, resulting in significant outages. In April of the same year, another disastrous series of crashes occurred when the shift to daylight savings was made in the northern hemisphere, for similar reasons.

Finally, at the end of 2000, it appeared that all major problems had been cleared up, when another group of problems manifested themselves. For a reason not made clear at the time, Microsoft, then the biggest software company, had a feature in one of their operating systems which calculated the number of the current millennium and used this number in an internal patch to fix the Year 2000 problem. There was an error in this software whereby the new millennium was (correctly) picked up as starting in 2001 in some areas, but incorrectly as starting in 2000 in other areas. As a result of subtle interactions between these bugs, the problem was not picked up until the start of the year 2001, when all the affected software crashed.

A final problem resulted when a number of systems which had been rewritten to repair bugs in handling Year 2000 problems crashed in April when daylight savings time started in the northern hemisphere. The daylight savings bugs in some cases had always been present but had not manifested before other code was changed, and in other cases were introduced as a result of major rewrites of legacy applications, in which handling daylight savings was previously correct.

3.2 Video on Demand

The introduction of high definition television (HDTV) in the early part of the century coincided with a growth in demand for video on demand (VoD). VoD required very high network bandwidth, and in early designs, did not scale well with increasing numbers of users. The growth in demand for VoD spurred a major growth in the importance of networks as an area of research, since a mass-market technology has a much higher momentum behind it than a technology aimed at a smaller (for example, technical) market.

Up to the early part of this century, the two major drivers of technology innovation had been processor speed and memory density. While it had earlier been predicted that the limiting factor on processor speed and improvements in memory density would be the physics of the process technology of the time [Wilkes 1995]. However, by 2010, it

had become clear that the major limiting factor in further progress was moving sufficient data to the processor to justify improvements in speed. Similarly, increases in memory density were increasingly of less interest because very large data sets resulted in memory traffic dominating performance. Instead, the focus moved to making increasing use of high network bandwidth, a commodity easily scaled up firstly by the universal move to fiber optics, and secondly by the fact that bandwidth can easily be added by using a wider pipe (or multiple pipes).

A combination then of consumer market pressures and technology trends led to a decreasing emphasis on arcane details of computer instruction set-oriented architecture research, and an increasing emphasis in the architecture field on networking issues. This switch led to an increasing importance in algorithm analysis in the hardware field, as issues like network routing draw heavily on analysis of graph algorithms.

Other areas which grew importance included queuing theory (particularly as latency over long-haul networks is a major problem) and simulation.

3.4 The Bazaar Bungle

A highly influential paper which first appeared in 1997 with subsequent revisions in later years described two alternative software development models as the *cathedral* and the *bazaar* [Raymond 1997]. This paper led Netscape, a major supplier in those days of internet services, to offer a significant portion of the source code of their software free. The paper argued that the traditional development culture was like a cathedral (it was never made clear exactly what the analogy was here) in that it was controlled by some central architect, with strict limits on when the outside world could see it, and in how much detail. Raymond argued that another model, that used to develop the Linux operating system, which he called the bazaar model, was better. The fundamental idea was release early and release everything including source. In the bazaar model, the entire world becomes a talent pool, and having the source open makes it possible to uncover bugs as early as possible.

Several companies followed the Netscape lead in opening their source code, and in the ensuing chaos, some useful lessons were learned.

First, the cost of support is an important component of the lifetime cost of any computer system. In the open source model, the “bazaar” becomes people selling support. Since this is where the money is, there is little incentive to produce software that is easy to understand. Those major software vendors that stayed outside the open source world were forced to compete with every hacker in the world in matching the open software for features, without the benefit of worldwide review of their source. Consequently, conventional commercial software became more and more cumbersome and feature-laden, and more and more unreliable. At the same time, free software, while

increasingly popular, was also increasingly developed on the “bazaar” model, resulting in a significant growth in the complexity of free software.

Up to 2007, the trend in software appeared to be increasingly towards greater complexity of the product, and increasing fractions of budgets being spent on maintenance, training and support, to the extent that even companies which were still charging for software were frequently deriving 70% or more of their revenues from these sources, rather than from sale of their product.

While this change in software development strategy created a new industry in support gurus, it also created an opportunity for lighter-weight computers with very simple, easy-to-use software. An early prototype of this trend, the Apple Macintosh, was a limited success in countering the trend towards complexity in that it promoted ease of use as a design goal, but it too suffered from increasingly complex software being developed for it.

The trend in the early part of the century was towards a growing divergence between business and home computers, despite the pressure to run similar software at home. At first the trend manifested as set top boxes but with the growing sophistication of services like VoD, set top boxes started increasingly to resemble fully-fledged computers. While an early attempt at a minimal computer, termed the Network Computer (NC) was not a success, the proliferation of set top boxes created and opening for a new class of computer, the thin client (TC), capable of running limited software locally, but with emphasis on exploiting high network bandwidth. The adoption of the TC model at businesses was slow, until a large amount of simple, distributed software became available.

The original NC model had revolved around a language called Java, but support for Java collapsed as a result of a number of expensive lawsuits aimed at maintaining the standard as the property of a specific computer manufacturer. Instead, the dominant trend has become one of specifying interfaces clearly and precisely, so that a given unit of software can be seen as a black box. The language used has become unimportant, as long as the interface specifications are met. To guarantee that interface specifications are met, where reliability is important, formal proofs have increasingly been required over the last two decades. However, a more important realization has been that a programming language should make it easy to specify interfaces correctly. As a result, languages with features like automatic memory management (no pointers or aliases, but with garbage collection), mathematically precise semantics and support for well-defined interfaces have become increasingly popular.

Ironically, hiding detail and specifying interfaces precisely was one of the hard-won lessons in the early years of software engineering. In the twentieth anniversary edition of his influential book of essays on software engineering, Fred Brooks

conceded that information hiding was a better strategy than openness, the position he had taken in the original edition of his book [Brooks 1995]. Somehow, this hard-won lesson was lost when software moved from a centrally-controlled enterprise to a cottage industry on the internet.

It was only with the big software crash of 2007, when many major software vendors collapsed under their own weight, in their attempts at building up ever-growing feature lists to drown the competition, that the trend to large, bloated programs ended. The free software movement played a role in this trend in that the internet facilitated the construction of huge, complex programs by disorganized teams of volunteers, which large software vendors attempted to counter by producing huge, complex programs under controlled conditions.

In the last 10 years, with network bandwidth the major growth factor in the computer industry, the trend to simplicity in software has continued, as it has become a convenient model to distribute functionality. Large-scale data has come to be maintained on servers around the world, whereas latency of the network requires end-user interaction be controlled by local software. Moreover, businesses which have followed the simplicity trend have generally had much lower costs than those following the bazaar model, as a result of which, large-scale monolithic software has been on the retreat over the last decade. In retrospect, it becomes clear that problems in previous generations of software, such as the millennium muddle, were a result of over-large monolithic systems, with no clear interface specifications between their components.

3.3 Major Issues Impacting Curriculum Design

The Bazaar Bungle—as it has come to be known, the move to ever-more complex software with more and more emphasis on charging for support rather than for the software itself—led to some of the more significant changes proposed in Curriculum 2010. The most important of these changes was a move towards specification of interfaces at an early stage of the curriculum, with particular emphasis on viewing software components as black boxes. This change was in line with the philosophy of abstraction-first teaching, and led to an increasing perception that teaching programming first was not the correct introduction to computer science.

The Millennium Muddle was a further data point in the process which led to Curriculum 2010. Had software been composed of small modules with well-defined interfaces, it would have been much easier to isolate the effect of changes (for example, correcting the millennium bug should not have had an impact of parts of a program which implemented daylight savings).

Finally the impact of new technologies like video on demand on the nature of computing has increased emphasis on distributed computing, in which small units of

computation are spread out over a network. The importance of networking has grown and along with it, the underlying theory of graph algorithms, as well as queuing theory. Given the move to abstraction-first learning, the Curriculum 2010 committee felt that increased emphasis on discrete mathematics as a precursor to programming was warranted.

4. Curriculum Proposal

Curricula in this century have emphasized topics in a given order—now that an order has been generally accepted. The following summarizes the accepted order, with few changes since the previous published curriculum. Note however that the order is specified in generic terms and there is a lot of flexibility as to what goes into each knowledge unit.

Note that a detailed curriculum is available in the full document to be published electronically; detail is not specified here as most is unchanged relative to the last published curriculum.

4.1 Introductory topics

focus: notation and techniques

- no problem analysis by students (but examples supplied), no design or implementation
- tools of the trade—the what and why (not how) of operating systems, programming languages, networks and formal methods
- problem-solving—divide and conquer, dynamic programming, systematic strategies, formal models of problems (all based on studies of given examples)
- basic theory—logic, sets, relations, graphs, trees, proof techniques, recurrences
- models of computation—formal models, including regular sets, formal languages, automata, Turing machines

4.1 Introduction to Analysis and Design

focus: problems, not on computer solution

- analysis of problems—introduction to solvability, efficiency (including complexity)
- problems and models of computation—given problem, derive FSM, Turing machine, etc.
- correctness of designs—given specification and design, prove design matches specification

4.2 Design and Implementation

given a design: implementation issues

- efficiency of designs—algorithm and data structure analysis
- empirical verification of analysis—given algorithm analysis and implementation, verify that run times etc. match analysis
- limits of computing—computability, complexity in more detail
- programming languages and paradigms

4.4 Engineering of Software

focus: correct design and implementation

- reusable components—emphasis on abstraction
- implementing a design—place coding in perspective as the ail-end of the software process
- advanced implementation—concurrency, OS kernels, networks, databases
- software life cycle—management, professional issues; life cycle models, retrospective on earlier parts of the course
- specific application areas
- major implementation and design issues
- designing a new software architecture

5. Major Changes

Since Curriculum 2010, the biggest changes have been in specific application areas and in software architectures, reflecting advances in high-speed networking and global inter-process communications models. Fundamentals in 4.1 through 4.3 have largely been left unchanged.

Most of the changes result from changes predicted as long ago as 1996 [Lewis 1996*a*, Lewis 1996*b*], in which global conductivity has increasingly become the largest resource at the disposal of the computing community, and global-scale compute servers have become a reality.

Also of interest is the way in which mass-market applications have begun to leverage off very high-speed interconnects, a trend predicted in the late 1990s [Machanick 1998*b*].

The full report contains more specific information; this section contains a summary of major changes. The areas summarized here are networking and distributed computing, and application architectures. Both areas are related to recent technology innovations.

5.1 Networking and Distributed Computing

Improvements in bandwidth resulting from wide-scale deployment of 2Tbit/s fiber imply a change in focus in network-related application areas, with wider coverage of issues in design of mass-market video on demand. Some issues which need to be introduced include:

- trading bandwidth for latency in information mass-transit systems
- design of distributed applications for global-scale interconnects
- fault tolerance in global-scale distributed applications

A few examples which may be added to illustrate the principles include:

- a home video on demand system: trade-offs between deployability and speed
- interactivity versus bandwidth conservation in large-scale weather modelling systems

5.2 Application Architectures

Changes in the computing model from a computation-based model to a network-based model have also resulted in changes in application models. The curriculum committee therefore proposes that the following new areas be included as examples of application architectures:

- mass transit-based interconnection (as opposed to classic client-server architectures)
- interaction-locality models for high bandwidth but high latency distributed systems (with emphasis on moving latency requirements close to the user)
- distributed object models (to some extent a revival of earlier 20th-century ideas like Common Object Request Broker Architecture, or CORBA—though modern languages allow a much simpler infrastructure to achieve the same ends; this is an example of a trend which has long been predicted [Pancake 1995] but has only recently been driven by the marketplace)

All of these areas can apply to video on demand and other interactive video areas, but represent more general computational models and are therefore worth studying as application architectures.

6. Conclusions

Compared with Curriculum 2010, there were relatively few changes in Curriculum 2020. Reflecting the development of Computer Science into a proper engineering discipline, the latest curriculum only needs changes in the later components, where application-specific knowledge units are dealt with.

It seems likely that future curriculum documents of this century will mainly focus on refinements of the accepted basic theory, with most changes occurring in the later knowledge units.

As a result, education research is likely to focus on issues such as classroom and lab practice, rather than on curriculum.

Acknowledgments

This work was largely done while on sabbatical at the Department of Electrical Engineering and Computer Science, University of Michigan. I would like to thank Trevor Mudge for hosting me, and my own University as well as the Foundation for Research Development for supporting my sabbatical.

References

[ACM 1991] A Summary of the ACM/IEEE-CS Joint Curriculum Task Force Report: Computing Curricula 1991, *Comm. ACM*, vol. 34 no. 6 June 1991, pp 68–84.

[Baber 1997*a*] Robert L Baber. Comparison of Electrical "Engineering" of Heaviside's Times and Software "Engineering" of Our Times, *IEEE Annals of the History of Computing*, vol. 19 no. 4, October/December 1997 pp. 5–17.

[Baber 1997*b*] Robert L. Baber. CS Education and an Engineering Approach to Software Development, *Proceedings of the 27th Southern African Computer Lecturer's Association Conference*, Wilderness, South Africa, June 1997 pp. 22-24.

[Brooks 1995] Frederick Brooks. *The Mythical Man-Month: Essays on Software Engineering* (anniversary ed.), Addison-Wesley, Reading, MA, 1995

[Berztiss 1987] Alf Berztiss. A Mathematically Focused Curriculum for Computer Science, Communications *of the ACM*, vol. 30, no. 5, May 1987, pp. 356-365.

[Goldweber *et al.* 1997] Michael Goldweber, John Impagliazzo, Iouri A. Bogoiavlenski, A. G. Clear , Gordon Davies, Hans Flack, J. Paul Myers, Richard Rasala Historical. Perspectives on the computing curriculum, *ITiCSE-WGRSP '97. Selected Papers from the Proceedings on Integrating Technology into Computer Science Education: Working Group Reports and Supplemental Proceedings*, Uppsala, Sweden, 2-4 June 1997, pp. 94-111.

[Gries 1991] David Gries. Teaching calculation and discrimination: a more effective curriculum, Communications *of the ACM*, vol. 34, no. 3 March 1991, pp. 44-55.

[Lewis 1996*a*] Ted Lewis. The next $10{,}000_2$ Years: Part I, *Computer*, vol. 29 no. 4, April 1996, pp 64–70.

[Lewis 1996*b*] Ted Lewis. The next $10{,}000_2$ Years: Part II, *Computer*, vol. 29 no. 5, May 1996, pp 78–86.

[Machanick 1998*a*] Philip Machanick. The Abstraction-First Approach to Data Abstraction and Algorithms, *Computers & Education*, 1998 (in press).

[Machanick 1998*b*] Philip Machanick. *A Scalable Architecture for Video on Demand: SAVoD*, Technical Report, <http://www.cs.wits.ac.za/~philip/papers/SAVoD.html>, 1998.

[Pancake 1995]. Cherri M Pancake. The Promise and the Cost of Object-Technology: A Five-Year Forecast, *Comm. ACM*, vol. 38 no. 10 October 1995, pp 33–49.

[Raymond 1997] Eric S. Raymond. *The Cathedral and the Bazaar*, <http://earthspace.net/~esr/writings/cathedral-bazaar/>, 1997 (revisions 1998).

[Wilkes 1995] M.V Wilkes. The Memory Wall and the CMOS End-Point, Computer Architecture News, vol. 23 no. 4, September 1995, pp 4–6.