Informatics for a New Century: Computing Education for the 1990s and Beyond

Mary Shaw

July 1990

CMU-CS-90-142

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213-3809

Prepared for IFIP Working Group 3.2 International Workshop
on Information Curricula for the 1990s

This report will also appear as Carnegie Mellon University, Software
Engineering Institute Technical Report CMU/SEI-90-TR-15, ESD-TR-90-142

Abstract

Information technology and computer science have not only reshaped computation, communication, and commerce; they have expanded the basic models and paradigms of many disciplines. Informatics education has obligations to all the communities that rely on information technology, not just the computing professionals. Serving this extended audience well requires changes in the content and presentation of computing curricula. This paper sketches the coming needs for information processing and analyzes the populations that will require informatics education. It considers curriculum requirements through two examples, one outside the traditional boundary of computer science and one inside.

This research is sponsored by the U.S. Department of Defense, in part by Contract F19628-90-C0003 with the U.S. Air Force as the executive contracting agent and in part by the Defense Advanced Research Projects Agency (DoD) and monitored by the Avionics Laboratory, Air Force Wright Aeronautical Laboratories, Aeronautical Systems Division (AFSC), Wright-Patterson AFB, Ohio 45433-6543 under Contract F33615-87-C-1499, ARPA Order No. 4976, Amendment 20. M. Shaw was also sponsored in part by the Software Engineering Institute under contract to the Department of Defense.

510.7808
C13r
90-14[illegible]
[illegible]

Keywords: Informatics education, computer science education, teaching programming, interdisciplinary computing education.

Table of Contents

Table of Figures

List of Tables

Informatics for a New Century: Computing Education for the 1990s and Beyond

Abstract: Information technology and computer science have not only reshaped computation, communication, and commerce; they have expanded the basic models and paradigms of many disciplines. Informatics education has obligations to all the communities that rely on information technology, not just the computing professionals. Serving this extended audience well requires changes in the content and presentation of computing curricula. This paper sketches the coming needs for information processing and analyzes the populations that will require informatics education. It considers curriculum requirements through two examples, one outside the traditional boundary of computer science and one inside.

1. Information Processing in the 1990s

Over the four decades of modern computation, information technology has assumed increasing significance as an economic force and a shaper of disciplines. Although the computing disciplines have been able to develop systematic models, theories, and tools at a quite respectable rate, the rapid growth of practical computing applications has generated demand for computational capability far faster. Educators have usually concentrated on the internal needs of the evolving discipline, often at the expense of the mutual benefits that could accrue from interaction with application areas.

The major tasks for computing education in the next decade will be updating the curriculum to incorporate improved models and theories as they emerge, matching content to changing needs, and expanding the communities served to include users of varying degrees of sophistication in various application areas.

1.1. Information processing as an economic force

Information processing is a large sector in both United States and world markets. For example, US industry sales forecasts were about $150 billion in 1989 and are projected to exceed $230 billion by 1992. This figure includes computers, peripherals, packaged software, and communication but does not include service, system integration software, and in-house software development. Table 1 shows US forecasts in more detail [DAG 89].

Category	1989	1992	Annual % growth
Multi-user systems	38.0	47.1	7.4
Supercomputers	1.9	3.5	22.9
Single-user computers	31.8	52.4	18.1
Printers	9.3	12.6	10.6
Storage	21.1	32.1	15.1
Communication equipment	8.6	11.4	9.7
Packaged software	23.7	37.5	16.6
Mfg auto/CAD/CAM/CAE	24.7	35.5	12.9

Table 1: US Sales Forecasts, $ billions

The cost of in-house software in the US alone may be in the range $150-$200 billion [CSTB 90]. It is not clear how much modification after release ("maintenance") is included in this figure. Thus, when services, system integration, in-house development, and maintenance are included, software now dominates the cost of information processing.

Assessment of the worldwide software market involves interpretation of varied and uncertain data. Despite the uncertainties, projections have been made; for 1990 they range from US$70 billion to US$180 billion, and projections for 1996 range up to US$340 billion. Whether in response to such projections or independently, an increasing number of countries are entering the international software market [World Bank 89].

The economic presence of information processing also makes itself known through the opportunity costs of system developments that fail or are not even attempted. Examples of costly system failures abound; less obvious are the costs of computing that is not even attempted: software development backlogs so large as to discourage new requests, gigabytes of unprocessed raw data from space, and so on. Despite very real (and quite substantial) successes the litany of mismatches of cost, schedule, and expectations is a familiar one.

The presence of information technology is qualitative as well as quantitative. For example, the (US) National Academy of Engineering recently selected the ten outstanding engineering achievements of the past 25 years [NAE 89]. Of the ten, three are informatics achievements:

- communications and information-gathering satellites
- the microprocessor
- fiber optic communication

Two more are direct applications of computers:

- computer-aided design and manufacturing
- computerized axial tomography scan

And most of the rest are computer-intensive:

- the moon landing
- advanced composite materials
- the jumbo jet
- lasers
- the application of genetic engineering to produce new pharmaceuticals and crops

[illegible] 370 [illegible]

→ computing science [illegible] [illegible] (el [illegible] scient)

1.2. Information processing as a shaper of disciplines

Information processing is both an amplifier of existing models and enabler of new ones. At a recent conference on computational science, several speakers observed that there are now three equal branches of science [Levin 89]:

- experimentation
- theory
- numerical simulation

All have the objective of providing insight and understanding; all complement each other. More generally, computational paradigms are emerging as alternatives to older scientific paradigms.

A National Research Council study on the impact of information technology on the conduct of research drew several conclusions, including:

> Information technology has already had a significant and widespread impact on the conduct of research. For the future, that impact amounts to a revolution. Computer and communication technologies are valuable to every scientific discipline and essential to a growing number of them. ...
>
> - As the power and speed of computers have increased, numerical computations of increasing complexity have become practical. The result has been more realistic simulation of systems either physically impossible or too costly to study directly.
> - Another consequence ... has been the capability to collect, store, retrieve, manipulate, and analyze enormous quantities of information. ... Huge information archives are not only feasible but also accessible ...
> - An additional consequence ... has been the capability to present the results of numerical computations as visual images, [which] allows speedier, more efficient interaction with models ...
> - Computers [can] take over monitoring and control of scientific instruments. This makes scientific observation more convenient, more reliable, and often lower in cost; in some cases, it has led to new computer-based instruments that extend the bounds of observation.
> - Information technology has greatly expanded the capabilities for communication among researchers [COSEPUP 89].

Not only does information technology supply intellectual resources to other disciplines, it also draws ideas, results, and problems from these areas. For example, numerical analysis, transactions and synchronization, and graphics were at various times strongly driven by application domains.

Because of the pervasive presence of software, the appropriate objective for its developers should be *the effective and economical delivery of computational capability to real users in forms that match their needs*. This objective requires computing to be embedded in applications; it requires computing to be packaged to meet the user's models and match his needs, and it requires computing that is comprehensible to its users in actual operation. The distinction between the computational component of a system and the application it serves is often very soft; the development of effective software now often requires substantial application expertise. Satisfying this requirement will require computing education for both the developers and the users.

2. Audiences for Computing Education

Personnel shortages in computing, and especially in software, are well documented. A study by the Office of Technology Assessment estimates a current shortfall of 50,000 to 100,000 software professionals in the US alone. A World Bank study identifies personnel shortfall as the number one software risk item [World Bank 89]. The same study shows substantial backlogs in data processing departments, as shown in Figure 1.

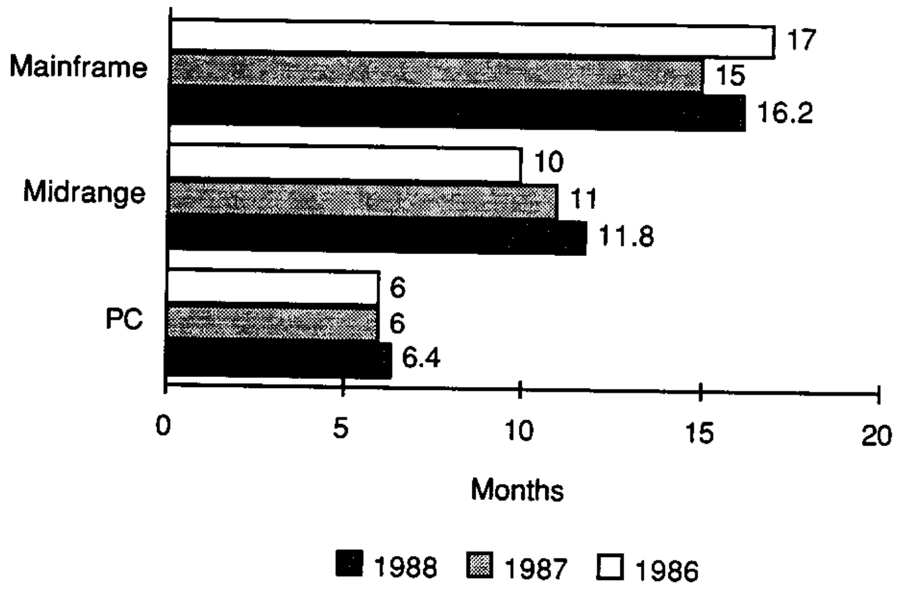

Figure 1: Application Development Backlogs (Months of Waiting Time)

As applications become more pervasive, computing becomes more accessible, and developers become relatively scarcer, we are seeing shifts in traditional roles, especially in the distinction between programmers and users. Three seem especially significant:

- *Softening of distinction between programmer and user:* Users are demanding more control over their own information processing, and modern personal computer systems such as spreadsheets, small databases, and text processing make this possible. Computations that once required a programmer can now be carried out by almost any user. The effect is simultaneously to make computation more effectively available and to transfer to the user some of the responsibility formerly held by the programmer.
- *Specialization within computing:* Enough technical knowledge is now needed for some tasks that it is no longer reasonable to expect every developer to master it all. This increase in the total of well-organized knowledge will surely lead to specialization, as it has in other fields. However, the lines of specialization are not yet clear. It could follow application lines, leading to specialties such as real-time processing, communications, artificial intelligence,

and database management. Or it could follow functional lines, leading to specialties such as security, reliability, and user interfaces.

- *Computational specialists in disciplines other than computing:* As disciplines become more sophisticated in their use of computing, it no longer remains possible to solve its computing problems with a disciplinary background and a smattering of Basic programming, nor with a sophisticated knowledge of computer science and superficial acquaintance with the application area. Disciplinary specialists with considerable depth in computation will be required.

These shifts in the relation between people and computers will affect the audiences for computing education. We can identify four groups of computer users with different kinds and intensities of involvement:

- *Computer scientists:* researchers, system developers, builders of the computational infrastructure that supports many applications
- *Computational specialists:* specialists outside computer science who use computational paradigms, application developers
- *Light-duty developers:* people who configure or adapt systems for personal use, occasional programmers, hobbyists
- *Casual users:* "informate" citizens (not just literate and numerate ones), indirect users (such as recipients of service or managers), users who aren't programmers in traditional sense

The responsibility for education in computing will be distributed across a number of institutions, including universities, grade schools, continuing education programs (both in-house and commercial), and the public media. I will focus here on the role of universities.

In the early 1980s, curricula were largely tuned to a population that had emerged over the previous decade or so. Many technical students and some nontechnical students took some sort of introductory programming course. Some of these -- perhaps half or fewer -- went on to take a few more computer courses, usually in programming topics. Although computer science majors were often popular, the absolute number of students who pursued a major was modest. Departments were often most concerned with the courses that supported the majors, offering few service courses beyond the introductory programming course. Figure 2 suggests this pattern. The vertical axis represents increasing technical depth in computing and the horizontal axis represents the relative number of students involved [Shaw 84].

But the shifts in needs and audiences described above will lead to a different demand profile in the course of the 1990s; Figure 3 suggests this new profile. Most students will be involved in computing at least casually; the best introduction for these students is almost certainly *not* the traditional first programming course. We can still expect students (perhaps in increasing numbers) who need some expertise in information technology but not enough to be professionals. The need for disciplinary computing specialists will create a demand that is not now widely recognized; this is discussed in some detail below. Finally, computer science majors will, of course, remain significant; it is possible that recognition of the needs for disciplinary specialists may decrease their numbers.

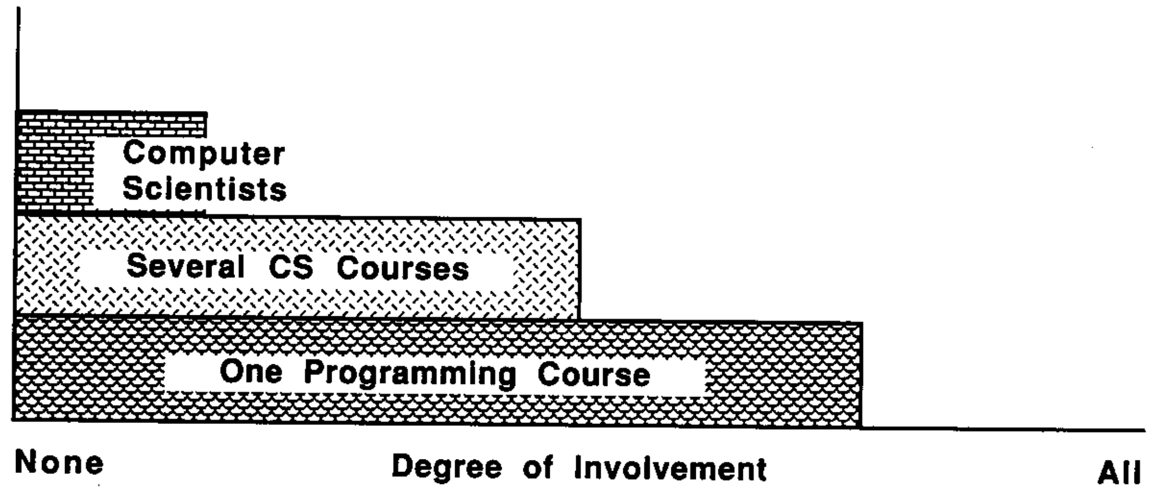

Figure 2: Undergraduate profile of the early 1980s

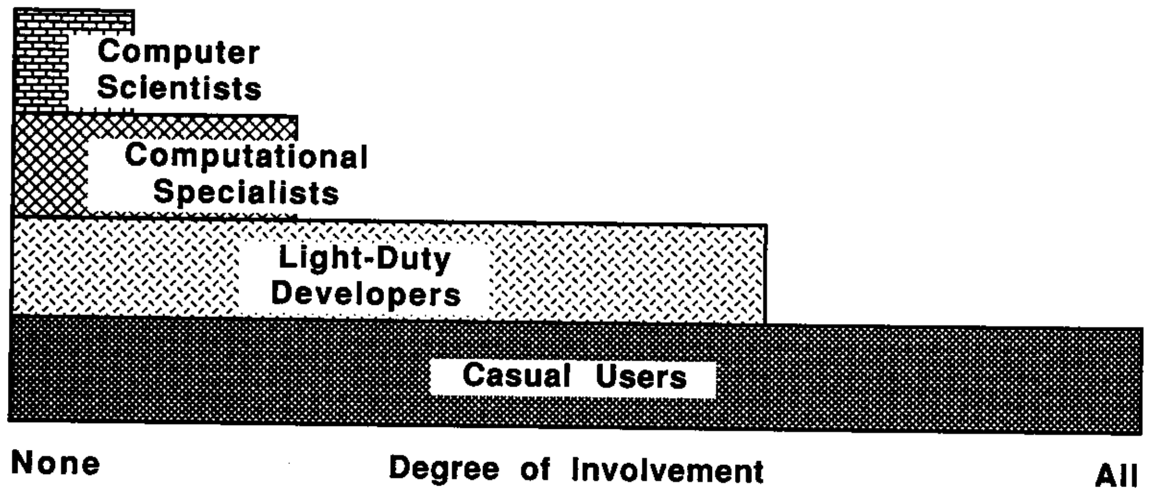

Figure 3: Undergraduate profile of the 1990s

Computer science departments have responsibilities to all these populations. They retain, of course, a primary responsibility for graduate and undergraduate (both terminal and non-terminal) education of professionals whose primary affiliation is with information technology. They share responsibility for the remaining groups with other departments in the university; this may take the form of consultation on curricula, of offering service courses, or collaborative development and presentation of courses. In addition, computer science departments must support the other kinds of institutions through teacher and curriculum development.

3. Two Examples

Two examples provide concrete examples of the curriculum requirements implied by these computing trends and the consequent shifts in student populations. The first lies outside the traditional boundaries of computer science: the education of computational specialists in disciplines other than information technology. The second lies within the traditional boundaries: the education of software developers.

3.1. Computing specialists in disciplines outside information technology

Computational subspecialties are emerging in many disciplines, including architecture, astronomy, chemistry, economics, geography, geology, materials, physics, and psychology. The computational paradigms in these fields are sufficiently sophisticated that they require expertise in both the discipline itself and in computer science.

Mere programming skill will no longer suffice for these disciplinary specialists. Their expertise must include core knowledge of both areas, some specific computational models, and selected techniques from areas of computer science that depend on the application but include computer systems, algorithms, artificial intelligence, graphics, and theory.

There is a growing need for intermediate-to-advanced computer science education for computational specialists in application areas. This will require genuine competence in both fields, unlike simple application programming. Traditional mechanisms such a minors and double majors are not sufficient; they can cover cores of both subjects, but by their nature they fail to tie the two disciplines together. Further, they are likely to consume most of a student's elective freedom. Although this is better than nothing, it should be regarded as a basis for integration, not as a solution.

A joint degree program with close cooperation between computer science and the disciplinary department offers an alternative. Such a program would cover much of the cores of both fields but would likely require relaxation of some requirements in both departments. It would also include advanced courses in computational techniques of the discipline that relied substantively on prerequisites in both computer science and the cooperating department; this makes sense only if the cooperating department has a faculty member with professional interest in the computational specialty. The adaptation of requirements must be tailored to each discipline, and some repackaging of the usual courses may be appropriate.

Candidates for such joint programs might include network communications (with electrical engineering), scientific computation (with mathematics or physics), human-computer interaction (with psychology), music synthesis and composers' tools (with music), computer-aided design (with design or architecture), and information systems (with management). Programs of this kind are being developed at Carnegie Mellon University, George Washington University, the University of Illinois, the University of Colorado, the University of Toronto, and perhaps others [NSF 89, Shaw 84].

It is not always possible to include enough content in an undergraduate program to satisfy the needs of a profession. When the professional requirements are high, the undergraduate program should not be overloaded at the expense of liberal education or at risk of premature specialization that could stifle professional growth. Instead, joint programs leading to both bachelor's and master's degrees in five years could be considered. Continuing education is another alternative, though the amount of material involved is rather large for that medium.

3.2. Software developers

Even with the growth of applications, there will remain a market for system software. It includes production of simple software components, standalone systems, operating systems, the information processing infrastructure that supports applications, and some parts of system integration. As discussed above, we can expect specialization of some kind to emerge. This is likely to be coupled with a clarification of job categories that recognizes different skill levels: system integration is more demanding than component construction, and not every programmer should be called a software engineer.

These software developers who remain within the computing field need a sound basis in computer science. Although software accounts for most of the cost of using computers, it does not account for most of the substance of computer science. Students who expect to become software developers should nevertheless acquire a firm grounding in computer science, including its common core, the associated mathematics and engineering, and perhaps a non-software sequence either within computer science or in some application area. Potential software engineers are included among these students: an undergraduate software engineering degree is not yet justified by substantive, durable content.

The software content of the curriculum needs to be re-examined, both for topic selection and for presentation. Both our current understanding of engineering design and examination of the knowledge that programmers actually use suggest flaws in the current software offerings. Undergraduates are being prepared to think about algorithms and to write small programs, but they do not learn enough about existing software, about the incorporation of small program elements into large systems, or about the problems of long-lived software.

3.2.1. Routine and innovative design

Engineering design problems come in a number of forms; one of the most significant distinctions separates *routine* from *innovative* design. Routine design involves solving problems that resemble problems that have been solved before; it relies on reusing large portions of those prior solutions. Innovative design, on the other hand, involves finding new ways to solve unfamiliar problems. The need for innovative design is much rarer than the need for routine design, so routine design is the bread and butter of engineering practice. Most engineering disciplines capture, organize, and share design knowledge in order to make routine design simpler. Handbooks and manuals are often the carriers of this organized information [Marks 87, Perry 84].

Similarly, engineers distinguish designs created from scratch (so-called *green-field* designs) from enhancements of existing systems; the latter are regarded as more constrained and hence more difficult (contrast this with the software developer's view of maintenance). Designs intended to be implemented once are also distinguished from those with replicated instantiations, and designs done by individuals are distinguished from large team designs.

Computer science teaching and research emphasizes innovative, green-field, once-used designs created by individuals. Software development and software engineering practice, however, involves work that frequently should be routine, that is most often enhancement of previous work, that is normally instantiated in many variants, and that involves large teams.

3.2.2. A cry from the wilderness

Practitioners recognize the need for mechanisms to share experience with good designs. This quotation appeared in a software engineering news group:

> In Chem E, when I needed to design a heat exchanger, I used a set of references that told me what the constants were ... and the standard design equations ...
>
> In general, unless I, or someone else in my engineering group, has read or remembers and makes known a solution to a past problem, I'm doomed to recreate the solution. ... I guess ... the critical difference is the ability to put together little pieces of the problem that are relatively well known, without having to generate a custom solution for every application...
>
> I want to make it clear that I am aware of algorithm and code libraries, but they are incomplete solutions to what I am describing. (There is no Perry's Handbook for Software Engineering.)

This chemical engineer--or former chemical engineer--is complaining that software lacks the institutionalized mechanisms of a mature engineering discipline for recording and disseminating demonstrably good designs and ways to choose among design alternatives. *Perry's Chemical Engineer's Handbook*, by the way, is the standard design handbook for chemical engineering; it is about 4 inches thick, 8-1/2 by 11 inches, and printed in tiny type on tissue paper [Perry 84].

As this developer seems to understand, software development in most application domains tends to be more often original than routine -- certainly more often original than would be necessary if we concentrated on capturing and organizing what is already known. One path to increased productivity is identifying applications that should be made routine and developing appropriate support. The current emphasis on reuse [Biggerstaff 89] emphasizes capturing and organizing existing knowledge. Indeed, subroutine libraries -- especially libraries of operating system calls and general-purpose mathematical routines -- have been a staple of programming for decades. But this knowledge cannot be useful if programmers do not know about it or are not encouraged to use it, and library components require more care in design, implementation, and documentation than similar components that are simply embedded in systems [CSTB89].

3.2.3. The nature of expertise

Proficiency in any field requires not only higher-order reasoning skills but also a large store of facts, together with a certain amount of context about their implications and appropriate use. This is true across a wide range of problem domains; studies demonstrate it for medical diagnosis, physics, chess, financial analysis, architecture, scientific research, policy decision making, and others [Reddy 88, pp 13-14; Simon 89 p.1].

An expert in a field must know around 50,000 chunks of information, where a chunk is any cluster of knowledge sufficiently familiar that it can be remembered rather than derived.

Furthermore, in domains where there are full-time professionals, it takes no less than ten years for a world-class expert to achieve that level of proficiency [Simon 89 pp.2-4].

Thus, fluency in a domain requires content and context as well as skills. In the case of natural language fluency, for example, Hirsch argues that abstract skills have driven out content; students are expected (unrealistically) to learn general skills from a few typical examples rather than by "piling up of information"; intellectual and social skills are supposed to develop naturally without regard to the specific content [Hirsch 88]. However, says Hirsch, specific information is important at all stages. Not only are the specific facts important in their own right, but they serve as carriers of shared culture and shared values.

A software engineer's expertise includes facts about computer science in general, software design elements, programming idioms, representations, and specific knowledge about the program of current interest. In addition, it requires skill with tools: the language, environment, and support software with which this program is implemented.

Hirsch provides a list of some 5,000 words and concepts that represent the information actually possessed by literate Americans. The list goes beyond simple vocabulary to enumerate objects, concepts, titles, and phrases that implicitly invoke cultural context beyond their dictionary definitions. Whether or not you agree in detail with its composition, the list and accompanying argument demonstrate the need for connotations as well as denotations of the vocabulary. Similarly, a programmer needs to know not only a programming language but also the system calls supported by the environment, the general-purpose libraries, the application-specific libraries, and how to combine innovations of these definitions effectively. Moreover, he or she must be familiar with the global definitions of the program of current interest and the rules about their use.

The engineering of software would be better supported if we knew better what specific detailed content a software engineer should know. We could then organize the teaching of this material so that useful subsets are learned first, followed by progressively more sophisticated subsets. We could also develop standard reference materials as carriers of the content.

3.2.4. Ways to get information

Given that a large body of knowledge is important to a working professional, we turn now to the question of how software engineers should acquire facts, either as students or as working professionals. Generally speaking, there are three ways to obtain a piece of information you need: you can remember it, you can look it up, or you can derive it. These have different distributions of costs:

	Infrastructure Cost	*Initial Learning Cost*	*Cost of Use In Practice*
Memory	low	high	low
Reference	high	low	medium
Derivation	medium-high	medium	high

Memorization requires a relatively large initial investment in learning the material, which is then available for instant use. *Reference materials* require a large investment by the profession for developing both the organization and the content; each individual student must then learn how to use the reference materials and take the time to do so as a working professional. *Deriving information* may involve ad hoc creation from scratch, it may involve instantiation of a formal model, or it may involve inferring meaning from other available information; to the extent that formal models are available their formulation requires a substantial initial investment. Students first learn the models, then apply them in practice; since each new application requires the model to be applied anew, the cost in use may be quite high [SGR 89].

Each professional's allocation of effort among these alternatives is driven by what he or she has already learned, by habits developed during that education, and by the reference materials available. At present, general-purpose reference material for software is scarce, though documentation for specific computer systems, programming languages, and applications may be quite extensive. Even when extensive documentation is available, however, it may be under-used because it is poorly indexed or because software developers have learned to prefer fresh derivation to use of existing solutions. The same is true of subroutine libraries, though incorporation of a library in the programming language (as in Common Lisp) provided better documentation and visibility to the routines that are added to the language.

3.2.5. Flaws in current software curriculum and ways to improve the situation

This examination of knowledge appropriate to a software developer suggests short-comings in the current software curriculum:

- *Programming from scratch:* Most courses teach program construction from a blank sheet of paper, rather than by modifying existing programs or by working from models of good solutions. Moreover, students rarely read good programs; it is as if we asked students to write good English without reading good prose.
- *Equating program text with software:* A complete software product includes also the analysis that led to the design, the user documentation, the test suites, and records of design decisions that will be important to the maintainer. Students too often focus on the code, do ad hoc testing, neglect the user documentation, and fail to deal with anything else.
- *Abstract skills at the expense of specific content:* Our curricula are very strong in techniques for formulating solutions from first principles. They present too few well-known examples of good solutions for study and emulation.
- *Programming before reasoning:* Although the situation is improving, coding and debugging still seems to win out over specification, analysis, and careful construction or derivation.
- *Throwaway exercises:* When assignments are discarded as soon as they are graded, there is no clear incentive for creating comprehensible, well-documented, maintainable software.

Some ways to remedy these flaws and improve the software curriculum include:

- *Study good examples of software systems:* To do this properly requires the development of case studies intended for presentation. However, careful guided reading of good code would be an improvement, as would assignments that start from code distributed by the instructor.

- *Learn more facts:* Software developers will not use resources they do not know about. We should teach more specific substance such as the available subroutine libraries and interface standards; we should reinforce this with assignments that require their use.
- *Modify and combine programs as well as creating them:* Students should learn to work with program structures devised by others, to reuse components, to adhere to standards, and to value good documentation.
- *Incorporate reference material as it becomes available:* There is currently a dearth of good reference material to help software developers avoid re-invention. As such material becomes available, it should be incorporated. Meanwhile, students should be taught to use reference manuals, library documentation, and the like effectively. Plan for continuing improvement in this area.
- *Present theory and models in the context of practice:* The curriculum should emphasize durable ideas that will transcend a major shift of technology. These ideas are often learned best when coupled with concrete examples of their application; if the examples are selected well they may themselves be worth remembering for reuse.

4. Conclusion: Adapt or Die

Examination of the roles of computing in the economy and society showed needs that computing education must satisfy. Two examples elaborated those need in particular areas. This led to specific recommendations for changes in the curriculum. We close with a set of summary observations.

Information technology will continue to evolve. New theories and techniques will mature; they will often be more concise than the material they replace. This kind of change is inevitable. Not only will it affect our curriculum, it will affect our student's professional growth. Thus we must not only be prepared to adopt new results ourselves; we must also prepare our students to do so.

A curriculum design should be a document of aspiration, not merely documentation of the status quo. It should say where we want to be, not just where we are. It may be ambitious, but it must be at least moderately realistic. Design involves balancing conflicting requirements for scarce resources -- in curriculum design the scarcest resource is course time, or curriculum space. Hard decisions must be made; including too many topics will have the effect of diluting attention, reducing the depth of each to superficial survey. The core should be based on sound, durable fundamentals. The current fashion will change, but students who have learned the fundamentals well will be equipped to learn new fashions.

The packaging of material should match students' needs. Practitioners will apply their knowledge to immediate problems, whereas researchers will build on their knowledge to obtain new results. The same introductory courses may well serve both, but this may not be true for all advanced courses.

Computation is joining the scientific paradigms of experimentation and theory. We should present computation in its own terms; we have no need to disguise it in some other form.

Computer science departments have an obligation to help with the education of all the audiences for computing education. This is not to say that they must assume sole responsibility, except for their own majors. But if they fail to respond to the emerging needs, they risk becoming irrelevant.

5. References

[Biggerstaff 89] Ted J. Biggerstaff and Alan J. Perlis. *Software Reusability*. Two volumes, ACM Press, 1989.

[COSEPUP 89] Committee on Science, Engineering, and Public Policy, National Research Council. *Information Technology and the Conduct of Research*. National Academy Press, 1989.

[CSTB 89] Computer Science and Technology Board, National Research Council. *Scaling Up: A Research Agenda for Software Engineering*. National Academy Press, 1989.

[CSTB 90] Computer Science and Technology Board, National Research Council. *Keeping the US Computer Industry Competitive*. National Academy Press, 1990.

[DAG 89] Data Analysis Group. *Computer Industry Forecasts*, Fourth Quarter 1989.

[Hirsch 88] E. D. Hirsch, Jr. *Cultural Literacy: What Every American Needs to Know*. Vintage Books 1988.

[Levin 89] Eugene Levin. Grand Challenges to Computational Science. *Communications of the ACM*, 32, 12 (December 1989), pp. 1456-1457.

[Marks 87] L. S. Marks et al. *Marks' Standard Handbook for Mechanical Engineers*. McGraw-Hill, 1987.

[Martin 71] William A. Martin. Sorting. *ACM Computing Surveys*, 3, 4 (December 1971), pp.147-174.

[NAE 89] National Academy of Engineering. *Engineering and the Advancement of Human Welfare: 10 Outstanding Achievements 1964-1989*. National Academy Press, 1989.

[NSF 89] National Science Foundation. *Report on the National Science Foundation Disciplinary Workshops on Undergraduate Education*. National Science Foundation, 1989.

[Perry 84} R. H. Perry et al. *Perry's Chemical Engineer's Handbook*. Sixth Edition, McGraw-Hill, 1951.

[Reddy 88] Raj Reddy. Foundations and Grand Challenges of Artificial Intelligence. *AI Magazine*, 9, 4 (Winter 1988), pp. 9-21 (1988 presidential address, American Association for Artificial Intelligence).

[SGR 89] Mary Shaw, Dario Giuse, Raj Reddy. *What A Software Engineer Needs to Know I: Vocabulary*. CMU-CS-89-180 and CMU/SEI-89-TR-30 ESD-TR-89-40 Tech reports, August 1989.

[Shaw 84] Mary Shaw (ed). *The Carnegie Mellon Curriculum for Undergraduate Computer Science.* Springer-Verlag 1984.

[Simon 89] Herbert A. Simon. Human Experts and Knowledge-Based Systems. Talk given at IFIP WG 10.1 Workshop on Concepts and Characteristics of Knowledge-Based Systems, Mt Fuji Japan, November 9-12, 1987.

[World Bank 89] Robert Schware. *The World Software Industry and Software Engineering: Opportunities and Constraints for Newly Industrialized Economies.* World Bank Technical Paper 104, August 1989.