Historical perspectives on the computing curriculum

Michael Goldweber (co-chair)
Beloit College, USA
mikeyg@beloit.edu

John Impagliazzo (co-chair)
Hofstra University, USA
cscjzi@hofstra.edu

Iouri A. Bogoiavlenski
University of Petrozavodsk, Russia
ybgv@mainpgu.karelia.ru

A. G. (Tony) Clear
Auckland Institute of Technology, New Zealand
tony.clear@ait.ac.nz

Gordon Davies
The Open University, UK
g.davies@open.ac.uk

Hans Flack
Uppsala University, Sweden
hansf@docs.uu.se

J. Paul Myers
Trinity University, USA
pmyers@cs.trinity.edu

Richard Rasala
Northeastern University, USA
rasala@ccs.neu.edu

Abstract

Computing has become a diverse and multi-faceted discipline. It is imperative that computing curricula evolve so that they will effectively convey this breadth. An awareness of the societal implications of computing must also be at the core of all computing curricula. Furthermore, we observe that new computing curricula must be responsive to change, that pedagogy must be informed by reasoned judgement, and that educators function as reflective practitioners. This requires educators to respond appropriately to market pressures and technological innovations. This paper investigates some of the components of the discipline's evolving computing curricula from a variety of historical perspectives.

1 Introduction

Rapid changes in computing often motivate educators to introduce innovations in the curriculum and the classroom. The haste to do something new or adopt some current fad can cause teachers to overlook possible adverse effects of these innovations on students and the profession. The deployment of curricular or pedagogic innovations such as new languages and technologies may seem appropriate, but mistakes are costly. History is the best teacher to assess the worthiness of new ideas or products. The working group investigated these issues and developed guidelines for avoiding pitfalls when making innovations in curriculum or pedagogy.

Many institutions, departments, and programs of study introduce innovations into the computing curriculum on a trial-and-error basis. In doing so, they learn which such innovations are useful and which are not. However, in the haste to do something new or adopt some current fad, educators sometimes overlook adverse impacts of the innovations. Computing breakthroughs, when applied without the proper balance of idea versus application, can be costly. This paper reflects the exchange of ideas among educators on how innovations affect the computing curriculum and classroom. It focuses primarily on the broad philosophical issues of a computing curriculum.

The working group members identified and evaluated the status and trends of the computing curriculum. Historical facts and research findings were used to delineate the negative and positive aspects of an innovation. The members of the working group engaged in a pedagogic dialog on the topics researched. They focused their thinking toward the curricular issues and arrived at a consensus toward each issue. The working group achieved its goal to produce a document that would serve as a starting point for a discussion or project that includes innovations in the computing curriculum.

2 Historical background

Innovations continuously challenge computing educators. Hardly a week goes by without some new breakthrough or idea that fascinates the computing professional. Infatuated by these new ideas, educators are sometimes zealous to see that such novelties find their way into the classroom. Sometimes the leap is successful; sometimes it is not. Reflection upon the history of computing can be beneficial in evaluating which steps to take and how to take them. Not being too hasty to do something new or adopt some current fad is important.

2.1 Development of computing curricula

One of the first published reports dealing with computing curricula was the ACM publication known as Curriculum '68 [28]. This document provided recommendations for four-year programs in computer science. Curriculum '68 recommended a

set of courses that included computer programming, computer systems, computer organization and architecture, algorithms, data structures, operating systems, programming languages, and numerical analysis. It also recommended the study of discrete mathematics, calculus, linear algebra, and probability and statistics.

In response to the rapidly changing field of computing, the ACM published another report in 1978 entitled Curriculum ’78 [29]. Curriculum ’78 updated Curriculum ’68 and advocated a stronger emphasis on programming. It, even more so than its predecessor, became the model curriculum that many computing programs in the United States and throughout the world followed. In 1983 the IEEE Computer Society published its own recommendations for a computing curriculum [2]. In 1984 and 1985 the ACM published two recommendations [57, 58] that focused on the CS1 and CS2 courses.

By the mid-1980’s computing curriculum recommendations had become more reactive than pro-active; describing what institutions were already doing, instead of proposing future directions [21]. In the late 1980s a special task force addressed the issue of curriculum reform and published what became known as the Denning Report [31]. This report established nine topic areas that would become the definition of the computing discipline. Furthermore sensitivity to the social context of computing was defined as an accompanying thread to the nine primary subject areas. The ACM and the IEEE Computer Society embraced the elements of the Denning Report and in 1991 jointly published a curriculum recommendation known as Curricula ’91 [5]. Most recently, in 1996 the Liberal Arts Computing Consortium published a proposed computing curriculum for liberal arts colleges [101], and the ACM sponsored working groups to examine the strategic directions in computer science and engineering education [93] and the soon-to-be-published recommendations on information systems (IS’97) [47].

2.2 Lessons learned from the past

Today, computing educators are faced with many new ideas and technologies. The recent and rapid development of click-and-see Web pages can be dazzling and impressive. Unfortunately little is known whether the inclusion of such entities in a computing course is beneficial to learning. Will Web browsing increase the analytical skills of students? Will using this development make a person a better computing professional? Before this innovation is firmly placed in a computing curriculum, its curricular or pedagogic value should be well understood.

The literature shows many instances where caution should have taken precedence to implementation. For example, in a recent episode, an innovative programming design was used on a satellite rocket launcher. Because the software was never fully tested, the rocket exploded and crashed to the earth with its multi-satellite payload, costing its sponsors hundreds of millions of dollars [43]. Fortunately, it injured no one and the incident occurred with little news coverage. This example supports the idea that the testing of innovations requires the same scrutiny and diligence as testing sophisticated software found on missile systems. The pedagogical and fiscal damage caused by an unproved innovation in a universal curriculum may far outweigh the losses caused by malfunctioning software on a rocket.

As another example, in the late 1980s there were great advocations and movements to have closed laboratories for computing classes. On the surface, the idea echoed the laboratory settings of the natural sciences. Many computing educators leaped at what seemed to be a good idea. They championed the closed laboratory notion and convinced administrators to spend untold resources for equipment and staffing to support this innovative idea. Unfortunately, in the haste to develop closed laboratories, few had taken the time to really study the effectiveness of these laboratories before carrying out the idea. Sadly, a recent study suggests that closed laboratories are not more effective than the traditional open computer laboratories [22]. This assumption was very costly to many institutions. The result was a waste of already scarce revenues and increased skepticism by administrators. Had computing educators “thought before they leaped”, students and administrators would not have had to endure the pains of this idea.

New ideas and technologies may show great promise. Nevertheless, as good as they may initially appear, experience shows that their useful life-span is sometimes very limited. History is one good yardstick to assess the worthiness of products or innovations. Heeding the historical lessons of the computing field is important. Computing educators owe their institutions, society, and most of all their students the professional responsibility of including only proven elements rather than current fads in the curriculum. To err in this instance is professionally deleterious, fiscally unsound, and ethically unjust. Such actions harm the computing profession.

3 Foundations of the computing discipline

The computing discipline has many faces. Every computing curriculum is oriented, or prejudiced, by one or more of these perspectives. We hope that what follows is an exhaustive listing of such perspectives and their curricular implications. Furthermore Section 4 enumerates some of the pedagogical approaches suggested by these perspectives.

3.1 Computing as Mathematics

There are several ways in which mathematics can be viewed as a model or paradigm for computing. The theory and concept of an algorithm as a systematic computational process was invented by mathematicians. In traditional mathematics, only the steps of an algorithm are specified. It is assumed that the results needed along the way will be computed and saved by a human being. With the automation of computation, the question of how to store data also became important. The structures of linear algebra and combinatorial mathematics (vectors, matrices, trees, and graphs) provided the fundamental models for the computer storage. Thus, both algorithms and data structures owe their origin to mathematics.

Similarly the use of abstraction to manage complexity derives from mathematics. Abstraction is used, quite literally, in all aspects of computing; from the lowest level of hardware design to the highest levels of software systems. Currently, abstraction so permeates the notion of computing that design techniques have been developed to aid one from getting lost in a sea of abstractions. In many respects, it is the use of abstraction that permits one to create, manage, or perform ever increasingly complex computations.

On another level, the method of proof in mathematics is quite analogous to the design of a structured program. The decomposition of a mathematical theory into a systematic collection of lemmas, propositions, theorems, and corollaries is similar to the decomposition of a program into functions small and large. The invocation of a lemma in the proof of a theorem is similar to the use of a simple function to assist in the computation of a more complex function. With the introduction of object-oriented methods, computing has moved slightly beyond classical mathematical frameworks by distributing data and functions into a collection of distinct interacting objects.

Mathematical proof is of course the model for reasoning in computing, even if the format is often simply a "convincing argument that such-and-such is true". We use logical reasoning to assert that an algorithm or collection of algorithms work correctly. We use estimation techniques from combinatorics, probability, and calculus to determine the time and space requirements of computational processes.

Logic and discrete mathematics are critical for computing. Historically calculus was of immense importance to computing, which was primarily concerned with scientific and engineering problems. But presently such computing occupies a much smaller percentage. Many of the mainstream fields such as artificial intelligence, database, software engineering, programming languages, and hardware rely extensively on discrete mathematics: hard-core scientific computing, relying on calculus and analysis, has become a small sub-specialty within the computing community. A "thought experiment" should prove persuasive: randomly select computing publications, such as journals and tutorials, and count the occurrences of discrete mathematics topics as opposed to the occurrence of calculus topics. Unless a very specific source was selected, there would be virtually no calculus evident, but discrete mathematics would permeate the samples.

Indeed, the rising importance of discrete mathematics along with the decreasing importance of calculus are starting to be reflected in computing curricula [5, 72, 81, 101]. Given the pressures of maintaining a current curriculum in computing, adding additional discrete mathematics seems infeasible. Reducing a calculus requirement would enable additional coursework to be offered in discrete mathematics.

A major emphasis within discrete mathematics is logic. Mathematical logic and foundations are at the heart of computing. The astonishing results from the discipline known as the "foundations of mathematics", while curiously ignored by the mainstream mathematical community [30], now have tremendous currency in computing. In fact, the person sometimes named the "father of computing", Alan Turing, earned that distinction precisely through his forward-looking work in the foundations of mathematics — work that anticipated the computer, stored programs, universal machines, and artificial intelligence.

In his influential text, *The Science of Programming*, David Gries writes:

> The research was fraught with lack of understanding and frustration. One reason for this was that computer scientists in the field, as a whole, did not know enough formal logic. … We spent a good deal of time thrashing, just treading water, instead of swimming, because of our ignorance. With hindsight, I can say that the best thing for me to have done 10 years ago would have been to take a course in logic. I persuaded many students to do so, but I never did so myself. [50]

But Gries' plea has been largely unheeded; logic is not a true part of the computing curriculum. Our students' only standard exposure to logic is in the already overworked course in discrete mathematics. Unfortunately, coverage is often limited to the trivial truth-table logic and rarely includes predicate logic to any depth whatsoever.

While not an emphasized part of the computing curriculum, the centrality of logic continues apace, as observed by Davis:

> When I was a student, even the topologists regarded mathematical logicians as living in outer space. Today the connections between logic and computers are a matter of engineering practice at every level of computer organization. [30]

Current strides are being made in artificial intelligence, programming languages, database systems, software engineering, formal methods, and hardware with techniques from deep areas of logic such as model theory, proof theory, combinatory logic, and non-classical logics, using such notions as completeness, consistency, axiomatics, natural deduction, Skolem functions, lambda calculus, and constructivity [72].

Regarding this last area, since computing professionals build algorithms and systems to transform actual inputs into actual outputs, the enterprise can be regarded as occurring with *constructive* mathematics: the true "classical" mathematics dating from the ancient times through Chronicler and prior to modern analysis. Indeed, there has been a renewal of constructivity since the advent of the field of computing. But apart from this acknowledgment, the implication of this turn of events to undergraduate curricula is unclear and should be explored [49].

Another mathematical view of programming (as expressed by Naur) is that programming is theory building.

> The building of the program is the same as building the theory of it by and in the team of programmers. During the program life a programmer team possessing its theory remains in active control of the program, and in particular retains control over all modifications. [74]

3.2 Computing as Engineering and Design

Computing professionals do not merely work with abstractions and passively study the artifacts in which they deal. They also create those artifacts — they build programs and systems. Thus, computing professionals do engineering as a daily part of their professional activities. The entire field of software engineering has been developed precisely because such an important facet of the discipline is engineering [5, 44, 96].

In traditional computing curricula, while students have considerable experience in the programming or coding phase

of the classical software life cycle, this work is primarily for very small "systems" [82]. Even for these small projects there is virtually no experience in requirements analysis, design, testing, and certainly not in maintenance. For example, the instructor's or textbook's statement of the programming assignment is the requirements analysis for that problem!

The vast differences between small and large systems are obvious and oft-discussed as the primary motivation for software engineering education. Brooks states:

> Software entities are more complex for their size than perhaps any other human construct … a scaling up of a software entity is not merely a repetition of the same elements in larger size. … In most cases, the elements interact with each other in some nonlinear fashion, and the complexity of the whole increases much more than linearly. [18]

It would be advantageous to provide students with experience in large (or at least larger) projects and in the other life cycle phases. However, as Miles states, "… teaching a course in software engineering is insufficient. Software engineering techniques must be promoted across the curriculum." [71] Furthermore, Booch has stated that the era of the solo, asocial programmer is over [17]. Good communications skills and teamwork are now regarded as crucial requirements for the computing professional and should be incorporated into computing curricula.

Curricula '91 [5] regards architecture as one of the foundations of computing. The analogies between computing and architecture include aspects of both the planning and building processes.

Architectural planning is an iterative process. The person or organization that desires a new structure will approach the architect with a combination of ideas, hopes, and constraints. The architect must develop plans, submit them for review by the client, make explicit changes that are requested, and ponder related changes that are implicit in the wishes of the client. This cycle iterates until a definite plan for the project has been agreed upon.

Once the overall features of a project are understood, the architect must make detailed plans. At this stage, goals are no longer the issue. The detailed plans address choices of technique and implementation: what materials, technologies, and construction methods will be used. The client is consulted only if issues arise concerning (un)foreseen variations in the financial, physical, or time constraints.

As an aid to understanding for both the client and the builder, the architect may construct a mock up or prototype of the structure as a whole or of a particular portion(s).

During the construction process, the contractor attempts to execute the architectural plans as faithfully as possible. Often changes will be made as the structure is being built. Some changes will be caused by errors in the design plans or by errors made by the contractor. Other changes will be requested by the client either to incorporate newly requested features or to restrict the project in light of financial or time constraints.

The process of software and hardware development parallels the above description of the architectural process in many ways. In the case of software, however, there is one critical difference. Whereas the product of architecture is a physical structure that more or less permanently occupies a specific location, the product of software development is a collection of source files and executables that can in principle be modified and enhanced in the future. To realize this potential, additional design considerations must be undertaken in the development process. In addition, the temptation to add unplanned features during the construction process (feature creep) is strong and this temptation must be resisted to preserve design integrity.

The process of software design has some characteristics that are not present to the same degree in engineering or architectural design. One may view the software design process as having three cyclic phases: inputs, planning, and outputs. While these phases are typically set in a project context, perhaps they should be more appropriately viewed as pertaining throughout the full lifetime of the product. The input phase consists of initial project scoping, identification of requirements, ongoing design drafts, and evaluations as the design proceeds. This phase can be difficult and amorphous since it may involve obtaining inputs from many stakeholders: customers, representative end users, managers, developers, societal representatives (such as the media), and members of the design team. A spirit of "participatory design" has much to recommend for this phase [25, 99]. The planning phase involves the aspects of actual design in which creativity, problem solving, and use of methodologies must all play a role. The outputs of design are plans that include overviews, models, decompositions, interactions, promises, and prototypes. As the three phases of the design process repeat, both the user community and the designers learn new things about the intended product, about what is possible (both to request and to achieve), and about how to perform design. More established engineering disciplines do not demand such intense on-the-fly learning. In the case of software design, frequently the product represents a wholly new way of supporting human activity for which past patterns do not exist.

Many of these points are increasingly represented in computing curricula, for example, the expanded treatment of requirements analysis, management, and systems engineering in new software engineering textbooks. But other implications of the above conceptualization may include a de-emphasis of such current "hot topics" as software reuse.

3.3 Computing as Art

To see how computing can be viewed as art, one might begin with a definition [20]:

- The application of skill according to aesthetic principles, esp. in the production of visible works of imagination, imitation, or design. (i.e. painting, sculpture, architecture etc.)
- Skill as a result of knowledge or practice, a technical or professional skill.

Consider how each of these perspectives may bear on computing.

3.3.1 Computing as Literature

The creation of software can be regarded as a literary process. In most countries intellectual property law respects the rights of ownership of computer programs, with copyright accruing to the writer of source code. “Source code for software is a ‘literary work’ and therefore protected by the copyright act …” [11]. Additionally there is Knuth’s notion of literate programming: design that incorporates elegance and other aesthetic characteristics [94].

There is an increasing use of research techniques drawn from the field of literature by information system researchers seeking a better understanding of information systems and their implementations. Typically these approaches apply interpretive methods, viewing the business organization as a text analogue requiring interpretation. Techniques such as critical hermeneutics and deconstruction offer new insights by viewing computing systems in a broader social and historical context. The folklore method is another analysis technique used in systems development whereby the myths and stories of an organization are investigated in order to understand the practices that are to be embedded in the computer systems [56].

3.3.2 Computing as an Artistic Endeavor

The prevalence of graphical user interfaces has brought the need for a new set of skills for software designers. Awareness of color, consistency of look-and-feel, and the subtleties of human computer interaction are all dimensions requiring a broader skill set than character mode screens demanded. Graphical design skills are becoming increasingly important, particularly in multimedia and hypermedia applications. The skills of the film producer and dramatic writer, the development of scripts, use of story-boards, designing phase-in/phase-out shots, use of photography, motion-video clips, sound, and editing, combined with navigation techniques and transitions, are all artistic undertakings.

Many initiatives in computing are attempts to minimize the “art” and replace it with more “method”. This is a deliberately stated goal of software engineering [82] and it is definitely one of the goals of formal methods and associated initiatives toward program and proof derivation.

Indeed, certain areas that have not yielded to method and procedure, areas in which the practice is still regarded as art, are considered failures in our understanding. The frustration is that since methods cannot be routinely applied nor taught, the activity must rely on human creativity, ingenuity, and insight.

Perhaps this is not so bad a state of affairs. Maybe there exists a bona fide essential aspect of computing that partakes more of art than of science. If so, this is an area in which very little discussion has occurred in recent years; hence, the curricular implications of such ideas are unclear. Whether this artistic component will ultimately be seen as permanent and essential or whether it will one day be replaced by method is unknown. In any case, for the immediate future, one pedagogical activity might be the use of professional software designers, programmers, and software engineers to speak to classes anecdotally about their experiences, insights, and creative solutions.

As another view of computing as art, one can simply mention that computing may be art in the same way that applying paint to surfaces is art; computing becomes a tool of the artist. This is rather obvious as computing has permeated the arts. Perhaps more controversial would be the notion that an actual program or system might itself be considered a work of art!

3.4 Computing as Science

To see how computing can be viewed as a science, one can begin with this definition [20]:

- An activity or discipline concerned with theory rather than method, or requiring the systematic application of principles rather than relying on traditional rules, intuition, and acquired skill.
- A branch of study that deals with either a connected body of demonstrated truths or with observed facts systematically classified and more or less comprehended by general laws and which includes reliable methods for the discovery of new truth in its own domain.
- … the intellectual and practical activity encompassing those branches of study that apply objective scientific method to the phenomena of the physical universe and the knowledge so gained.

Computing, while certainly not a natural science, has been termed a science of the artificial [95]. Design and engineering are very important to the field. But computing is most definitely an empirical science as well, where the subject being observed is not a natural object, phenomenon, or relationship, but is instead a human-made artifact and its behavior. As such, the computing curriculum lends itself both to observational laboratories and experiments. As is clear from these definitions, science includes both the systematic classification of knowledge using theory and a method for the discovery of knowledge using observation and experiment.

In computing research, both observation and experimentation are important. In studying the behavior of software, the collection and analysis of detailed time and space statistics is essential. Analysis will either confirm theoretical predictions or uncover discrepancies that must then be explained as errors in the theory or limitations of the model. When it is difficult decide upon a model, systematic experimentation must be undertaken to determine the important qualitative behavior and the detailed parameters of possible models. Experimentation may also be used to test whether a phenomenon is significant enough to warrant further study.

In computing education, much more emphasis has been placed on theory than on observation and experimentation. Students need to perform experiments and analyze the results. This will provide a hands-on acquaintance with computing phenomena that theory alone cannot provide.

In addition to performing experiments to test time-space performance, students should also experiment with user interface choices, the design of individual classes, and the structure of class hierarchies. This will help to foster an experimental attitude to complement the student’s theoretical foundations.

Holding a belief in the value of the scientific method is to take simultaneously an epistemological and ontological position that guides our views upon the nature of truth and the accepted means for determining it in the discipline. As an assumption

set, a belief in the value of the scientific method has been strongly held within the computer science and the American information systems communities. The European information systems community has held differing beliefs, which are progressively spreading within the community. These differing beliefs have acknowledged the value of more interpretive research methods (e.g. case studies), which address the qualitative aspects of computing systems and related phenomena, to derive new understandings that are less accessible via quantitative research methods. The particular value of such research approaches lies in their ability to reveal "the underlying connections among different parts of social reality" and thus explore aspects of the wider context in which computing systems are developed and implemented [78].

As a corollary, in a software engineering context the use of techniques such as soft systems methodology [23] and the interaction of social issues and software architecture [26] offer examples where the current pedagogy could be extended.

3.4.1 The nature of cognition and action

Is cognition a process of detached, abstract reflection and consideration, or rather a process of active engagement and action? Is the concept of decision a necessary precursor to action, or simply an after the event deduction?

Decision theoretical concepts developed from Simon's intelligence, design, and choice phases of decision making, have tended to emphasize deliberate, rational processes by developing normative models that separate cognition from action. This starkly contrasts with Langley et al's analysis of the reality of managerial work, in which they observe the intertwined, evolutionary, and event-driven nature of so-called decision-making. They go so far as to suggest that:

> … decision and decision process as decomposable elements tend to become mere figments of the researchers' conceptions, or artifacts of their methods. Or to use an even more graphic metaphor it is like a wave breaking over the shore —that is perhaps identifiable at some sort of a climax— then tracing a decision process back into an organization becomes much like tracing the origin of a wave back into the ocean. [62]

Likewise recent work by Maturana and Varela [68], initiating within the biological sciences, contradicts some earlier views about systems by observing the nature of autopoietic organisms structurally coupled with their environments. These kinds of organisms offered examples of systems which "were mechanistic but not programmed" [103].

These insights have been combined with a strand of philosophical thought through Heidegger, Merleau-Ponty, and Wittgenstein, to conclude

> … that perception could not be explained by the application of rules to basic features. Human understanding was a skill akin to knowing how to find one's way about in the world, rather than knowing a lot of facts and rules for relating them. Our basic understanding was thus a knowing how rather than a knowing that. [34]

The combination of these perspectives coalesce into a view that suggests that cognition represents a process of active engagement and is inherently embodied. Our internal systems are considered to be self-regulated and closed, but over time become environmentally adapted to our particular context. This in turn suggests that we have certain almost instinctual reactions based upon our biological, social, and cultural histories, with which we have become structurally coupled.

This model of cognition and action would suggest that:

> … the primacy of human being is social. The whole is conceived of as genuinely real and the part [i.e. the individual], is regarded as originally a differentiation. The model then is that human being is cultural, it is lived socially, and is therefore psychological and may be experienced as such. Culture is assumed to be a general and complex biological disposition that requires historically situated social constructional work to become a specifically lived-culture. [100]

Some implications of this perspective are that abstraction first and action next may not be wholly valid planning (or for that matter programming) techniques. Perhaps models that give primary emphasis to interaction may have more power — witness the success of the prototyping approach in software engineering. This notion also gives support to recent initiatives to distinguish differing learning styles of our students. A further implication is that we are inherently prisoners of our history and cultural context. The idea that the computer is an objective analytical engine, rather than an historical and social construction, and that systems have any detached, objective meaning may need to be reconsidered. Consideration of the overall context in which systems are to be developed may be more important than the abstraction processes by which their internals are constructed.

3.4.2 R&D and computing

Under the Organisation for Economic Cooperation and Development (OECD) definition:

> Research and Development comprises creative work undertaken on a systematic basis in order to increase the stock of knowledge, including knowledge of man, culture and society, and the use of this stock of knowledge to devise new applications. [76]

In relation to the connection with computing and science we can also note that:

> Any activity classified as R&D is characterized by originality; it should have investigation as a primary objective, the outcome of which is new knowledge, with or without a specific practical application, or new or improved materials, products, devices, processes or services. R&D ends when work is no longer primarily investigative. The definition of R&D in accordance with a change in OECD standards, now includes research into and development (or substantial modification) of, computer software, such as applications software, new programming languages and new operating systems. [91]

This heightened awareness of aspects of applied and practical computing as a research activity in its own right should be used by academics to advance the case that academic publishing is not the only way in which to have research activity recognized. Of course a corollary of this is that practitioners are actually researchers! Much the same point has been made recently by Glass noting that

> Not only is practice OK … but there are some important flaws in the theory and research of the field that may be of more concern than the software crisis ever was. … In any new discipline, it is often true people do things for which theory has no explanation and provides no foundation, and theory evolves only after practice has demonstrated that something works. … The notion of "best-of-practice" concepts emerges from the belief that practice can lead theory. [45]

A way in which such recognition might also impact pedagogy would be by enabling institutions to recruit practitioners as educators. This would encourage valuable cross-fertilization and closer ties with industry, while making it possible to adequately reward such staff by acknowledging certain kinds of professional experience as equivalent to research activity.

3.5 Computing as a Social Science

3.5.1 The Individual versus the Social

There are significant questions for computing that result from a perspective which is social rather than individual.

Psychology has been a foundation for a number of aspects of computing. The influence of Simon's information-processing model of cognition and the concept of bounded rationality related to the cognitive weaknesses of humans in decision-making contexts helped to develop a foundation for much work in the area of decision support systems [75]. Concepts such as the capacity, role, and mechanisms of short and long term memory have contributed to the work of system designers seeking to develop user interfaces that support rather than tire users by making unnecessary cognitive demands, where the processing power of the computer is able to effectively complement the capabilities of the user.

But as Laroche, a critic of the information processing model, comments,

> Simon's principle of bounded rationality implies looking at every individual as a decision-maker, because as the individual encounters major limits in his ability to process information, the outcome of his cognitive processes can neither be predicted nor relied upon. The idea of the individual as an imperfect information-processing machine was meant to destroy the illusion of a well-oiled organizational machine that functions in a predictable manner, as intended by its designers. Since the individual information-processing machine was a central assumption, it was very difficult for anything else to come out of this theory except an individual decision-maker. [63]

It is important for educators and researchers to be aware of the inherent bias in our assumptions when psychology is used as a reference discipline. The resulting tendency to regard computing from an individualistic basis must be acknowledged.

3.5.2 Technology as Tool versus Product

In a review of group support systems (GSS) research directions, De Sanctis has commented on the weaknesses of certain prevalent views [33]. In her review, the dichotomy between individualism and collectivism as normative views of organizations is used to illuminate the mental models of GSS researchers and illustrate implications for GSS development and research.

Under the individualistic view typically fostered by economic and psychological perspectives, the organization is an assembly of individuals whose aggregated activities constitute the whole. In this view [33], "technology is a tool to be applied to enhance individual power and overcome human limitations, such as limited strength or rationality".

Under the collectivist view, a view commonly advanced by sociologists and anthropologists, technologies such as GSS

> … are viewed as products of the social evolution of the organisation and the larger culture of which it is a part. Because cultures vary the meaning of technology varies. Thus any given GSS may mean different things within different organisations or even within different groups. Further, its roles and purposes may vary over time as the culture evolves. [33]

This implies that computing educators cannot assume technology can be developed in a deterministic fashion to meet given ends. Rather, the process of development and implementation must take into account social and human dimensions as critical determinants of success.

3.5.3 Anthropology and Computing

The field of information systems, by its very nature, is a pluralistic discipline. It has been argued that the science of anthropology should be considered one of its source disciplines [8]. Developments in anthropological thought on the central question of culture, a concept frequently used in the sense of organizational culture in the information systems literature, have moved from an early static view of the term as espoused by Benedict [13], to a more fluid one. The early definitions, borrowing from techniques used in the natural sciences, considered the

> … human world as composed of separate distinguishable entities. Each culture was thought to be a natural kind, just as entities of the physical world — kinds of animals, kinds of plants, kinds of minerals — are natural kinds. [8]

Benedict's imagery was

> … something like that of exhibits in a museum, where one finds an array of distinct, separate, integral objects, each unique, and yet each sharing some essential attribute with the others. [8]

Despite much disagreement over the concept of culture, anthropology as a discipline has long since moved on from this, perhaps unfairly criticized, "museum like view of culture".

Ironically Avison and Myers suggest:

> The predominant orthodox view of the culture concept in IS research, that culture is something which identifies and differentiates one social group from another (as per Schein's original formulation), in essence is not significantly different from Ruth Benedict's. … Although the concept of culture has been used rather narrowly in the IS literature, … IS research in this area would benefit if more attention was paid to the contemporary anthropological view of culture, which —as something which is contested, temporal and emergent— has the potential to offer information systems researchers rich insights into how new information technologies affect or mediate organizational and national cultures, and vice versa, i.e. how culture affects the adoption and use of IT. [8]

As a further elaboration of this view:

> Hirscheim and Newman (1991) argue that the symbolic concepts of myth, metaphor and magic facilitate a much richer understanding of information systems development than the conventional economic rationality model. [8]

The process of developing information systems, or for that matter any technology, could be defined as a process of designing tomorrow today. The designers (ideally in a partnership with the users), conceive a future world within which the users are to live. The system thus developed could be seen to be a reified social construct, which freezes into a software artifact a modified set of daily or periodic rituals, such as organizational routines and practices. The system thus becomes in effect a means for the transmission of culture in society. However, since the users both shape the system by their use and are in turn shaped by the system itself, the system becomes part of a broader culture which is "an ever changing emergent phenomenon through which people create and recreate the worlds in which they live" [8, 79]. A broader anthropological view, rather than a deterministic approach to systems development, may enable more successful implementation practices to be developed.

3.5.4 Computing as Politics

How does an application move from conception to implementation without a certain amount of political skill on the part of the computer professional? After all, is software development not in the end an exercise of power? As Boguslaw points out:

> A designer of systems, who has the de facto prerogative to specify the range of phenomena that his system will distinguish, clearly is in possession of enormous degrees of power. … To the extent that decisions made by … participants in the design process serve to reduce, limit or totally eliminate action alternatives, they are applying force and wielding power in the precise sociological meaning of these terms. [16]

Is it not then inevitable that opposition will be encountered? How is this opposition to be surmounted without some degree of dissembling, artfulness, or cunning? Is it time that the view that politics is a managerial problem be abandoned? Is it time to acknowledge that the "designer as expert" model is inherently one that reinforces legitimacy and assigns computing professionals extra power —whether warranted or not— and is one of the "tricks of the trade" that helps one achieve their goals [53, 102]? One must take care to apply this art of cunning with conscious care, otherwise it becomes a black art, a form of sanctioned modern witchcraft carried out by the scientist. Perhaps in addition to ethics, organizational politics or political science should also be part of the computing curriculum.

3.6 Computing as Interdisciplinary

In a field so broad as computing, touching on so many aspects of human activity, the major challenge is not so much to define the discipline itself, but to maintain a degree of tolerance of differing views and permit cross fertilization of ideas from different strands.

This does not, however, offer much guidance to those who must produce relevant curricula based upon a consensus among academics and practitioners as to where the focus should lie. It has been suggested that anthropology, applied psychology, computer science, cultural studies, economics, ergonomics, ethics, history, linguistics, management, mathematics, philology, philosophy, semiology, sociology, and politics are some of the disciplines relevant to computing. It is clear that we cannot do justice to this diversity in our educational programs by applying a single disciplinary perspective.

The computer science and information systems communities have traditionally taken quite distinct approaches to the discipline, with some areas of overlap. As computer science curricula begin more actively to include social, ethical, and cultural dimensions some rapprochement may occur. Students may hasten these trends as they choose courses from both curricula in a more multi-disciplinary approach to their study. A side effect of this might be graduates better equipped for their future roles as practitioners. Likewise students studying computer science within a liberal arts program will no doubt identify parallels and draw links between their different studies.

One model useful for understanding the discipline of computing defines a broad continuum between the poles of engineering and commerce [54]. In this model, engineering, computer systems engineering, computer science, information systems, and business are the respective disciplines across the spectrum.

The nine subject areas, three processes, and twelve recurring concepts defined in Curricula '91 [5] give the discipline a unity based upon a consensus developed by the computer science and engineering communities. This unity emphasizes agreed ways of viewing a common body of knowledge and a commitment to the scientific method as frames underpinning the discipline. By contrast, the coverage of the social and professional context appears a little "tacked-on", seemingly as an afterthought based upon a view that the course was intended to develop practitioners as well as theoreticians.

Thus this model gives a tightly prescribed discipline, which can be taught in considerable depth and rigor in relative isolation from other disciplines. It does, however, acknowledge the value of mathematics and the physical and life sciences and suggests a half-year of study in each area to complement the computing aspects of the curriculum. So to this extent the curricular model could be seen to advocate at least a multi-disciplinary if not interdisciplinary approach.

However, a more interdisciplinary trend is becoming apparent in attempts to deal with the slightly ad hoc nature of the social and professional context coverage.

A recent proposal to introduce a social and ethical component into the computing curriculum has extended the definition of this area to cover five main topics [67]:

- Responsibilities of the computer professional.
- Basic elements of ethical analysis.
- Basic skills of ethical analysis.
- Basic elements of social analysis.
- Basic skills of social analysis.

This broadening of perspective is a significant change for the computer science community. It will require considerable effort and a degree of interdisciplinary teaching to do justice to such a broadened curriculum. There is an inherent danger with interdisciplinary study involving educators with limited knowledge in some of the constituent disciplines; flawed, outdated, or only superficial content may be conveyed.

If one considers the development of the information systems community during the 1980s, two distinct discourses were prevalent, with the North American and the European schools holding very different views. The European school criticized the American school for its restriction to the orthodox science model and its advocacy of "one universal scientific method" [12]. The Europeans "advocated greater pluralism, more diversity, greater use of methods that allow researchers scope for interpretation, and the adoption of theoretical perspectives that are not founded on a rational and mechanistic view of the world" [12]. This has generated considerable debate within the information systems community. However, over the intervening decade a more diverse and pluralistic approach has been accepted by both information systems communities.

The computer science community maintains some distance from this position. For instance, the definition of programming in the computing curriculum is based upon the concept of "activities that surround the description, development, and effective implementation of algorithmic solutions to well-specified problems" [5]. A perspective based upon social analysis would argue that there is no such thing as a "well-specified problem", let alone singular correct "solutions". How can these two value systems be reconciled? Would an interdisciplinary approach create a valuable opportunity to link the two disciplines, or would it simply mark the beginning of a loss of the focus and rigor that has helped bind the computer science community over the last four decades?

It may prove instructive to more closely examine the content of the information systems discipline. This discipline has also suffered from discipline-defining debates. The field is commonly held to encompass two broad areas [3]:

1. Acquisition, deployment, and management of information technology resources and services (the information systems function).

2. Development and evolution of infrastructure and systems for information use in organization processes (systems development).

The distinctions drawn among information systems, computer science, and software engineering are suggested to lie in:

> … the context of the work to be performed, the types of problems to be solved, the types of systems to be designed and managed, and the way the technology is employed. The context of information systems is an organization and its systems. The context of computer science is algorithms and system software. In computer science, the emphasis is on the "systematic study of the algorithmic process — the theory, analysis, design, efficiency, implementation and application — that describes and transforms information. The context of software engineering tends to be large-scale software systems of the kind found in command and control systems, military systems, communication systems and large inter-organizational systems. Although the methods can be applied in other smaller systems, the context for the methods is generally tied to the problems of large systems. [3]

Information systems as an interdisciplinary field has from its inception, been held to draw from several reference disciplines, including computer science, management science, cognitive science, organizational science, and economics. The recommended core curriculum for information systems include [3]:

- Information systems foundation
- Information systems theory
- Foundation for business knowledge
- Information technology
- Information systems development
- Information systems deployment and management process

One can observe a clear interdisciplinary flavor within this core curriculum. The curriculum also concentrates heavily on graduates developing effective skills as practitioners. There is a strong emphasis on students' written and spoken communication skills in addition to their qualitative and quantitative abilities. Furthermore,

> The IS student must also have the people skills and basic understanding of an organization and its people to effectively empower its knowledge workers. This requires an introduction to the basic principles of mathematics and the behavioral, social and natural sciences, as well as a foundational knowledge in the disciplines within business administration [where] the cross functional nature of systems requires knowledge in the areas of marketing, finance, accounting, production, distribution and human resources. [3]

With the revised information systems curriculum one now sees options offered for students to minor in the subject and an acknowledgment that there is a base of information systems knowledge that students from all disciplines might benefit from. To this extent information systems is offering itself as an interdisciplinary study option for students from other backgrounds. Likewise with proposals from the computer science community to offer computing as a major within a liberal arts curriculum [101], similar initiatives are being taken that acknowledge the value of interdisciplinary study.

Both these initiatives seem to be acknowledging the need to broaden the scope of teaching for computing and information systems, recognizing the ubiquity of information technology in everyday society and the need for a growing range of students to have exposure to what is becoming a set of infrastructural skills necessary to simply survive in a modern society. This brings further challenges to a definition of what constitutes the disciplines and what should be the curricula.

The pressures thus appear to be building towards computing becoming more interdisciplinary in nature. However, the nature of the term interdisciplinary must be questioned. In multimedia development projects, for instance, there are good commercial models of interdisciplinary collaborations. The project management, computing professional, marketing, graphic artists, and film producers collectively produce a CD-ROM product for sale. In the educational context, the term interdisciplinary should be more accurately named multi-disciplinary: namely the coverage of separate aspects of individual disciplines in an unconnected manner. Interdisciplinary programs involve the active collaboration of multiple disciplinary experts in a context where joint activity is necessary to produce a successful result. A truly interdisciplinary curriculum in computing may be the next challenge.

4 Pedagogy and methodology

Underlying the computing curriculum is the consideration that the students might some day become computing professionals. This has implications both for what is taught and how it is imparted. Computing curricula recognize aspects of learning to prepare for practice. They also recognize that techniques for imparting skills and developing know-how have to be embedded in the instructional methods that are used. Students increase in proficiency as they are progressively exposed to further concepts and practice their application. Graduating students are expected, for instance, to have achieved a level of mastery of programming concepts and practice. As practitioners they will be expected to further develop this expertise and demonstrate professional skills, including the skillful execution of workmanship as an object in itself.

4.1 Problem-solving strategies

In many ways, a substantial fraction of the literature on computing may be viewed as a discussion of problem solving. The material on problem solving emphasizes an openness to variation as a fundamental key to problem solving. Some of the important references to this material are [4, 27, 38, 42, 60, 61, 64, 65, 80, 98].

The implications for computing pedagogy is that one must emphasize how both problems and solutions may vary and not simply focus on “here is one problem and here is its solution”. It may be important to set up some laboratory exercises so that more than one solution to the given problem is required. One should consider multiple ways in which a single class can be implemented or an abstract class can be constructed using one of several possible concrete classes. One might choose to spend more time on the variety of solutions to a single problem than on attempting to “cover” the maximum number of problems within a course.

The prominence of variation as a central theme in problem solving may also have implications for the holy grail of software engineering: “reusable software”. Perhaps, like the perpetual motion machine, reusable software is impossible. The essence of adapting software to new situations is to confront the variations in requirements, constraints, options, and outcomes from what currently exists. Doesn’t the experience of 50 years of software development suggest that no piece of software can be designed to be infinitely adaptable to all situations? A recent book on design patterns [42] attempts to classify patterns that are adaptable to some degree and inflexible to some degree, but no pattern in the book is infinitely adaptable. Perhaps an awareness of the problem-solving process will also permit us to wish for less “malleable software” and for ideas that are “reusable”.

While problem-solving strategies are useful, there is always the danger that a student may become confused with the multiplicity of ways in which problems are posed and how they are solved. However, using care and judicious selection of presentations, multiple strategies can stimulate students and generate effective results.

4.2 Apprentice approaches

The (so-called) apprentice-based approach to teaching computing has been developed at a number of institutions. A group of researchers working at Stanford have created a substantial quantity of C-based laboratories and support tools [35, 40, 87, 88, 89]. Another project, at Duke University, has led to a suite of laboratories that specifically attempt to use apprenticeship as the bridge from computer science to applications [7]. Finally, researchers at Northeastern University have developed laboratories and tool kits to support apprentice learning through both visualization techniques and the use of applications [19,36,85,86].

The theme common to all versions of the apprentice approach is that students will learn best if they develop significant examples with the help of code created by the faculty, as opposed to building smaller “toy” examples from scratch. Of course, from time to time students do build programs entirely from scratch but this is not the usual mode.

In the apprentice mode, the student is presented a problem, a framework, and a substantial set of tools. Although the tools differ from place to place, they commonly include operations to simplify input/output and calls to create graphics on the screen. They may also include file tools, array tools, and simple base classes.

The framework provided for a particular problem consists of routine segments of code that are not the main focus of the exercise plus special purpose code that may be beyond the student at that particular stage of the course. By giving students framework code, the problem can be larger and more interesting than would be possible if the program needed to be built from scratch. In solving the problem, students focus on the difficult algorithmic and design issues that are at the heart of the problem rather than reinventing for the *i*-th time simple input/output statements and other routine code.

The ability to focus on critical issues in the context of large problems is the key advantage of the apprentice approach. In addition, the use of tools and frameworks teaches students how to deal with abstractions that they have not themselves designed.

A key drawback to this approach is that the apprentice sometimes does not see the "big picture". The apprentice's view is always tightly focused and constrained. There is much to be gained from examining large systems and their design from the perspective of the whole.

4.3 Collaborative learning

Collaborative learning may be thought of as the instructional use of small groups, through which students work together to maximize their own and each other's learning [104]. This mode of learning is a generalization of the computer-mediated communication that uses computer technology to transmit, store, annotate, or present information. Examples of computer-mediated communication include email, chatrooms, and video conferencing. Collaborative learning is found in a variety of settings. Sponsoring institutions often derive different benefits by engaging in such experiences [24, 77, 92].

Consistent with the desire to better equip students for professional careers in computing, there is a need to link theory and practice in the educational process. This is accomplished through partnerships between educational, commercial, and community enterprises.

Models of collaboration such as live student projects, in which students work individually or in small teams to develop a commercial software application, have been part of the curriculum in some programs [24]. These projects, conducted on behalf of commercial or community clients, typically expose students to all aspects of the learning life cycle and develop a range of practitioner-relevant skills.

Caution should be heeded when using collaborative learning. Institutions should ensure that students are actually learning while they are collaborating and not assume that collaborative experiences automatically transform into educational ones. However, proper balance of collaborative experience and monitored learning can prove to be an effective way to elevate the student's insight into computing and to foster continued cooperation between educational institutions and industry.

4.4 Distance learning

Distance learning is not a recent phenomenon. However, with the proliferation of computers and computer networks, distance learning has become a significant component of education delivery, especially at the college and university levels. The International Centre for Distance Learning is an extensive resource in this area. It can be accessed through the URL http://www-icdl.open.ac.uk/icdl.

4.5 Multimedia and computing

Multimedia has come to mean a combination of still images, video, sound, music, and virtual reality. The purpose of multimedia is to provide a user with multiple pathways to information or just to provide enjoyment. In the domain of computing curricula, the full range of multimedia has yet to be realized. The bulk of the effort to date has focused on tools or programs to enable visualization or animation of algorithms, structures, and processes.

One of the earliest efforts in algorithm animation was the film "Sorting Out Sorting" [10]. As graphics workstations and personal computers became common, work shifted to creating visualizations and animations interactively rather than as canned films. The advantages of immediate display and interactive control were overwhelming. For a listing of some of the important visualization projects, see the reports by the working groups on visualization from ITiCSE'96 [14] and ITiCSE'97 [73] , including the "Visualization Repository" and a discussion of the potential benefits and drawbacks of using visualization tools.

There has been much less work done with sound and music. It has been suggested that one can use sound to enhance perception of detail. One can associate each level of gray with a tone and then, as the mouse passes over the image, sounds are produced. It is possible by this technique to detect variations in gray level that are not visible to the naked eye [36]. In a different direction, Rubenstein has guided interesting student projects on the generation and modification of MIDI sound [90].

Although substantial and important work has been done on the visualization and animation components of multimedia in computing curricula, it is fair to say that the full potential of multimedia has yet to be reached. One obstacle to further development in these directions is the cost of development systems and the associated software. Another obstacle is the time required to develop full multimedia educational software. Research needs to be done on what is possible in educational multimedia for computer science. If this research is fruitful then perhaps funds can be obtained to cover the development costs.

4.6 Networked computing as a resource

It is now relatively easy to access a computer network, either through a university system or an Internet service provider, thus enabling students to use the network to enhance their educational experience. There are a variety of ways in which this might happen.

1. Access to a computer network can provide access to a wealth of digital information, for example, in libraries or an organization's Web pages that hitherto would never have been possible.

2. Submission of assignments with on-screen marking can provide faster feedback to students.

3. Communication between students at a distance and between students and staff is enabled. For geographically dispersed students the advent of conferencing systems and cheap communications is possibly the most important technological development of recent years. Although the use of the network as a pedagogical device is still doubtful, as a medium for social interaction its importance is unquestionable.

A reference source discussing the Internet as a pedagogic resource is the report of the working group at ITiCSE'96 [51], but as this area changes rapidly, its value will diminish.

4.7 Introductory course considerations

The design of a computing curriculum is a highly complicated problem [31]. Educators need to prepare students with a common computing culture and a lifelong self-education ability. It may be useful to have two starting points to approach the problem.

First, there is a variety of topic areas that may be effectively used to achieve the goals of curriculum design, especially in the introductory sequence. These topic areas are fundamental and vary in depth and complexity.

Second, as students gain psychological insight, their self-estimation as a professional changes rapidly during their study. The changing process has specific features that should be taken into consideration during the introductory sequence and in the curricula design. This evolution may be considered a process of development into a computing professional.

The thinking of a new university student goes through a process of transformation. Starting as a novice, the student possesses few skills and does not have any insight into the profession. After a university experience, the student transforms into a beginning professional. The process is like the transformation from a grub to a graceful, fluttering butterfly. As such, the curriculum may be divided into two 2-year stages: core and advanced.

4.7.1 The core stage

A student's mind may be treated as gradually filling up a "tabula rasa" during the core stage. Students execute tasks very willingly and tend to trust teachers. They have no favorite approach or technique for problem solving. This stage is most favorable for forming a general understanding of computing and the development of self-education abilities. Knowledge and skills attained at this stage fundamentally influence the students' professional characteristics. The observation is implicitly confirmed in [5], where introductory courses are essentially different. This depends on the curricula orientation, such as computer engineering, computer science, or software engineering. At the same time, the core stage is the right place for equipping students with tools or "cultures and languages" (mathematical, algorithmical, architectural, programming skills). This core stage assists them to adapt and to increase rapidly their competence during the next stage.

Generally, at the core stage a student's motivation is high and directed to the foundations of computing. Students are intrigued by their new experiences. This means that the core should include in the introductory sequence some elements of the depth-first principle. This core should not be changed quickly. Students should be encouraged to evaluate the effects of the rapid changes in computing and to adjust to those changes.

4.7.2 The advanced stage

Increasing professionalism starts to influence students' attitudes toward study and the selection of courses during the advanced stage. Student group leaders start to emerge and their "professional" opinion is considered, in some cases, more significant than that of their teachers. The motivation at the advanced stage becomes more narrow and is directed to their work or to their attempts to get a steady job. This phenomenon is highly common [32]. Due to leadership and motivation factors, it becomes more difficult to fill omissions in the curriculum, and hence in students' competence, that occured during the core stage. For example, in the third and fourth years, students who acquired programming skills but lack a knowledge of architecture acquire a knack of resolving programming tasks without taking into account the architectural considerations. This leads to the formation of an incomplete notion about programming.

4.7.3 A possible modification of the introductory sequence

The introductory topic sequence [31] presents a well-founded core curriculum model. It formulates the principle that:

> Different kinds of institutions should exercise flexibility in determining the amount of coverage for each of the nine subject areas and three processes.

Following this principle, a reduction of some breadth topics in the core sequence can increase the depth of other topics in the core coverage.

Consider the following anecdote:

> Once a biologist, a physicist, and a mathematician saw a black sheep on a meadow in Australia. Oh, said the biologist, black sheep are to be found in Australia. No, said the physicist, we see one black sheep in Australia. No, said the mathematician, there is at least one sheep in Australia, and at least one side of it is black.

The anecdote illustrates the care and precision of the mathematician in observation and analysis. In the context of counting sheep, this seems amusing. In the context of designing and writing software to control complex hardware, such care and precision is absolutely essential. An important purpose of the core curriculum is to develop such attitudes.

The importance of algorithms and architecture subject areas force us to start teaching this combination of core cultures as soon as possible. One should observe that many elements of the introductory sequence [31] may be shown to students not only with a high-level programming language, but also with a low-level language using simple techniques [15], as well.

4.8 Curricular integration

There are a number of models for curricular integration that may be considered. Some of the interesting scenarios for the computing curriculum with respect to these models are the shared, threaded, and integrated models.

In the shared model, "Shared planning and teaching take place in two disciplines in which overlapping concepts or ideas emerge as organizing elements" [37]. For instance, a computer science and an information systems program could collaborate, possibly over software development and implementation, the social aspects of computing, and professional development as organizing elements.

The threaded model provides a "… metacurricular approach [that] threads thinking skills, social skills, multiple intelligences, technology, and study skills through the various disciplines" [37]. This model would require collaboration among different disciplines to investigate some broader transdisciplinary concept which would then weave its way through the instructional process, dipping from time to time into specific disciplines to explore the concept in more detail.

The integrated model is an "… interdisciplinary approach [that] matches subjects for overlaps in topics and concepts with some team teaching in an authentic integrated model" [37]. Integrated semester students are taught by a team of discipline experts who collaborate over the course of the program and

jointly negotiate issues of curriculum, methods of delivery, and assessment.

4.9 Design, algorithms and assessment

In addition to the recent attempts to enumerate the foundations of computing [5, 31, 52, 93] there has also been an attempt to enumerate the foundations of computer science education [83, 84]. In this model, there are three central components in computing education: design, algorithms, and assessment.

The design component deals with the overall approach to problem solving in various domains and with the management of complexity and interactions. A primary concern of design is the intellectual framework in which the problems of a domain can be viewed and the possible software frameworks in which these problems may be resolved. At the next level of detail, abstraction is concerned with the subdivision of the framework into coherent intellectual and physical components (data, functions, objects) that can capture the essential features of a system while encapsulating concrete details. The communication among these components is also an important design consideration.

The algorithms component is concerned with the efficient execution of the specific tasks that have been identified in the design of a system. Classical algorithms are used to build, traverse, reorganize, and destroy data structures. Domain-specific algorithms deal with numerical computation, operating systems, language processing, user interfaces, graphics, voice, video, speech, compression, cryptography, and artificial intelligence. These specialized algorithm families draw upon the classical algorithms, but are tuned to the needs of the particular domain.

The assessment component is concerned with the quality of the design and algorithms used to create a system as well as the overall quality of the system as a whole. The fundamental tools of assessment are theory, experiment, and observation. Theory provides analytic tools to assess new algorithms and a large catalog of existing algorithms with known performance characteristics. Theory also organizes knowledge about specific computational domains and appropriate design patterns for particular situations. Experiment is used to test theoretical predictions via simulation or to test actual implementation of full-scale systems. Observation adds the human element to the assessment process through design reviews, code reviews, and the use of debuggers. In addition, aspects of a system that are hard to quantify (such as the user interface) must be assessed by observation.

The three-component framework is simple enough for students to understand and motivates the major activities that occur throughout typical computing curricula. Additionally this framework helps to answer the perennial student question: Why are we doing this?

4.10 Classroom effectiveness

The classroom and laboratory is the traditional setting where the computing curriculum is instantiated. We observe that the physical environment can exert tremendous influence on how a particular curriculum is implemented. The physical environment can dictate the degree of collaborative learning and which classes, if any, can be supported with a closed laboratory.

The physical classroom and laboratory environments are changing. New technologies are supplementing the chalkboard, whiteboard, and overhead projector methods of content delivery. In addition to simply being "better boards/overhead projectors", these technologies have the potential to radically alter the nature of current pedagogy.

4.10.1 Physical environments

The introduction of the inexpensive chalkboard altered education pedagogy in a deeply significant manner. Some of the currently available (and soon to be available) technologies have the potential to effect a similar dramatic change.

It is no longer unusual for a classroom to be equipped with a computer and projection facility. The presence of the computer and projector allows one to develop a sophisticated delivery of the lecture material. The use of presentation software, authoring tools, and other multimedia creation packages is a vast improvement over the board/overhead projector method of conveying content. Some of the advantages include asynchronous access of "presented" material by the student and the inclusion of audio and visual enhancements, such as sound, color, and replay. While much has been written on this innovation, for the most part it is not significantly different than the introduction of colored chalk; pedagogy is not significantly affected.

The computer's presence in the classroom (and a network's presence on a campus) does have the potential to dramatically affect pedagogy. Due to the ubiquitous presence of computers in a computing curriculum, perhaps these changes will first be apparent in the computing disciplines. An instructor can conduct an experiment in real-time in front of the class, posing questions, formulating hypotheses, and actually running the computation to see what happens. Answers to questions can be more concretely visualized. Seeing results unfold on a screen can have far more impact on a student than hearing an instructor describe the result verbally or on the board/overhead projector. Without the computer, the class would have to "take the instructor's word" for what the outcome would be.

Computers have the potential to alter pedagogy in even more significant ways. Traditional classrooms reinforce the dualistic nature of the student-instructor relationship. The instructor, at the front of the room, controls access to the broadcast media: air (speech), the board, the overhead projector, and the presentation computer. Students have controlled access to the broadcast media (receive-all, arbitrated transmission) and uncontrolled access to their own notes. It is interesting to reflect how computing technology can alter this equation, in particular variations that improve on the traditional model. Some experiments with respect to these variations might include course-based newsgroups, etc.

Some of this redefinition of roles and the exploration of different learning styles is occurring in the collaborative learning movement. Proponents of such have learned that even simple things such as whether or not chairs are bolted to the floor can affect the degree to which roles can be redefined.

4.10.2 Assessment and evaluation

The debate between open and closed laboratories is old (by the metric of the age of the discipline). Articles abound declaring the advantages of one method over the other (see, for example,

any of the Proceedings of the SIGCSE Technical Symposium on Computer Science Education from the past ten years). Some recent research suggests that a closed laboratory does not increase student comprehension [22].

This is not to suggest that there is no valuable reason for conducting closed laboratories. On the contrary, this result shows that innovation in the computing curriculum needs to be scientifically evaluated. Anecdotal evidence is worthwhile to suggest avenues for experimentation, but strong curricular directions should only be set on the basis of such strong evaluative research.

Unfortunately the literature is sparse on formal assessments of curricular and pedagogical directions. There has been, on the other hand, an increase of activity in this area [46,66].

5 Professionalism and ethics

The growing emphasis upon ethical issues in computing delves into the realm of philosophy and the social human realm. For example, one recent proposal explicitly notes

> … from the perspective of computer science, every ethical concern is encountered at a particular level of social analysis. Only an analysis that takes account of at least three dimensions — the technical, the social and the ethical — can adequately represent the issues as they concern computer science. [67]

Professionalism is a relatively new component in the computing curriculum. The Denning Report [31] was the first ACM curriculum recommendation that discussed ethics. Furthermore, almost 10 years after its publication, a large number of "current" CS1 texts make no mention of professionalism/ethics nor even include the ACM Code of Ethics [1].

Professionalism in the context of a computing curriculum can mean a number of different things. These include responsibility for the quality of work, regard for the needs of a computer system's human users, societal impact, and a readiness to face the ethical dilemmas that computing technology can present on the job and in security.

5.1 Professional responsibility

In a broad sense, the entire literature of software engineering is concerned with the issue of quality of work. Recent books by McConnell [70] and McCarthy [69] do an excellent job of discussing software development as it is practiced in actual software companies. McConnell provides an in-depth analysis of various software methodologies that have been proposed and assesses their strengths and pitfalls. Both McConnell and McCarthy present case studies that expose "classic mistakes" and which highlight "best practices". One of the most basic mistakes is to accept unrealistic expectations concerning a project schedule and then to depend on individual heroics to attempt to bail out the project down the road. A basic aspect of professional responsibility is to resist unrealistic schedules at the start and to build in many checkpoints (zero defect milestones) that guarantee that some subset of the final product functionality actually works.

Both McConnell and McCarthy emphasize that the interaction of people in teams is critical to a software development project. Indeed, building the team and ensuring that each person can work effectively toward the team goals is as important as the schedule milestones or the details of software design and coding. When a team is working well, people are supportive and not competitive, communication is frequent and packed with information, and everyone takes responsibility for the quality of the product. In this atmosphere, they see discovering potential problems as a contribution to the team and not as a criticism of any individual developer. This permits the team to focus on strategies for solving the problems and moving on.

Of course, no project is without mistakes. If each individual has a commitment to learning, then we can use mistakes as the basis for what needs further study in the future. Companies can add value to this individual learning by conducting systematic postmortem studies on projects and making the results available to other teams.

5.2 Social and ethical issues

Beyond the individual project, computer professionals must consider why they write the software they do, the customers for whom they write, the larger impact of this software on society and, sometimes, the deeper legal and ethical issues. While the literature regarding these concerns is comparatively Spartan compared to that of software development, some recent texts [9, 39] make excellent contributions. At ITiCSE'97, the report of the working group on integrating societal and ethical issues into the computer science/information systems curriculum, more fully addressed these issues [48].

6 Fads: Choices, issues, and debates

6.1 Computing environments

An important influence on computing curricula, in addition to the recommendations of the ACM and the IEEE Computer Society, is the nature of the computing environment available. An environment may be categorized by the physical hardware (a factor of ever decreasing importance), the operating system(s), and the programming language(s)/development environment.

Arguably the most influential operating systems in academia in the last 20 years have been UNIX, DOS/Windows, and the Macintosh OS. Additionally, the integrated OS/language environments provided by Lisp, Scheme, and Smalltalk have had significant intellectual impact.

The significant role that UNIX has, and should continue to play, in computing curricula is due to many factors. Some of these include the technical merits of the system, its favorable price for academic institutions, and its widespread availability across many different popular hardware platforms. Unfortunately, the relative importance of UNIX has also blinded some to important innovations. In an interview, Hertzfeld recalls:

> I tried to make personal computers the topic of my graduate-school research, because to me they were the most exciting machines in the world. I was amazed that almost every professor in the department thought personal computers were the worst thing that ever happened to computer science … because personal computers were less powerful and had less memory than

> the big computers they were programming. … They just hadn't caught on to the thrills that ordinary people can have on these machines. … I guess a lot of academics couldn't have cared less. [59]

In hindsight, it is easy to recognize that the personal computer was one of the most important new technologies of the 1980s and that many innovations in the field of computing owe their existence to the personal computer. The breadth of application software and the advancement of user interface design could not have taken place without the impetus of a mass market of demanding end users.

6.2 Language fashions

There has been an even greater variety in programming language choices in academia over the past 20 years than in operating systems. Languages with recent substantial academic usage have been Pascal, C, C++, and Ada. Proponents of simplicity have favored Pascal or Scheme and those advocating structured design have favored Ada. The proponents of interactive design and the use of large tool kits have favored Lisp or Smalltalk. Bjarne Stroustrup remarks:

> One conclusion I drew … is that there is no agreement on what a programming language really is. Is a programming language a tool for instructing machines? A means of communicating between programmers? A vehicle for expressing high-level designs? A notation for algorithms? A way of expressing relationships between concepts? A tool for experimentation? A means of controlling computerized devices? My view is that a general-purpose programming language must be all of those to serve its diverse set of users. The only thing that a language cannot be —and survive— is a mere collection of 'neat' features. [97]

Stroustrup continues:

> The difference in opinions reflects the differing views of what computer science is and how languages ought to be designed. Ought computer science be a branch of mathematics? Of engineering? Of architecture? Of art? Of biology? Of sociology? Of philosophy? Alternately, does it borrow techniques and approaches from all of these disciplines? I think so. [97]

Despite this breadth of vision, Stroustrup's own creation, C++, is both widely admired and widely disdained. Pleasing everyone is not easy.

Currently, Java is the most recently developed language that has "great future potential". Java promises to enable true multi-platform interactive programming with just-in-time delivery over the World Wide Web. Will Java achieve its potential? Will its use one day be labeled a fad? Will Java be seen as a valuable addition to the mix of languages, but one destined to be no more dominant than many others?

These questions are hard to answer. One theory on the success of programming languages is that their acceptance is a social and evolutionary process, not a technological one. Furthermore, successful languages must require only minimal computer resources and not place any burden of mathematical sophistication on their users [41].

One then concludes that C is successful because the simplicity and directness of its programming model make it easy to learn and port to a variety of systems. This gives rise to the following opinion:

> Right now the history of programming languages is at an end, and the last programming language is C. [41]

Perhaps. Then again, perhaps not.

6.3 Computing for power users

A consistent theme in the development of programming languages has been the quest for greater productivity. While often more marketing hype than reality, programming languages that can easily be used by power users to develop their own systems have been touted for at least twenty years. Fourth-generation languages and rapid application development tools and techniques have made it possible to more quickly develop commercial applications of a moderate size. While most of these products, such as Visual Basic, Delphi, ORACLE, Progress, Paradox, MS-Access, Lotus Notes, and Power Builder, are really better suited for use by the professional application developer, intelligent and resourceful power users can develop their own applications. Microsoft is encouraging this trend with its integrated desktop software suite in which desktop applications inter-communicate via such mechanisms as OLE. The proliferation of Internet technology is supported by desktop products with facilities to publish reports to the Web or to relatively easily produce a Web page.

The developments on the commercial sector raise important curriculum issues:

1. Should computing educators be responsible for teaching about these tools and techniques?
2. What exactly should be taught and what should be left to the student?

In this context, is it not time for us to revisit the concept of programming and programming languages? Where does the boundary between the power user and professional programmer lie? Capers Jones refers to power-user-developed applications of 1000 function points [55]. Wasn't this once the mid-point on Albrecht's scale of commercial application software projects?

7 Conclusions

In this report a broad analysis of the status of modern computing has been conducted in the context of curriculum planning. Computing areas were observed not only from an implementation point of view (such as computer science and computer engineering), but from an applications point of view (information systems).

As demonstrated above, computing can be viewed from many perspectives. The paper has articulated several of these. This demonstrated breadth of the subject should be considered in new curricula accommodating these interdisciplinary dimensions. As the range of professions in computing proliferates and society broadens the nature of technology-related endeavors, graduates require a broader range of skills and insights in order to operate effectively as computing professionals in diverse

roles. Effective pedagogy demands serious consideration of professional and ethical conduct. An awareness of social and societal implications must be a core element of computing education.

When instructing students, educators should take advantage of students' natural process of maturing by introducing problem solving in stages ranging from smaller to larger exercises and projects. In that way teachers also encourage students to experience both the fruitfulness and the disappointments of working in groups of increasing size and to develop their capabilities in communication and interpersonal aspects of working in teams.

The working group strongly advocates more discrete mathematics and less calculus for any revised curricula, as the theory as well as the practice of computing is profoundly dependent on the former. In a crowded curriculum, calculus may be the aspect to be given less emphasis in less scientifically focused programs.

While educators must acknowledge market pressures and the impact of innovations in technology, they need to respond in appropriate ways. Pedagogy and programs must remain credible to students and to the wider community. The pressure of innovations in computing often leads to curricula overload and fragmentation. This demands that computing educators assess the impact of trends and changing paradigms and continue to evaluate the effectiveness of changes that are introduced. Computing is a field with a long history of rapidly emptying baths as a new model comes along, only to hurriedly gather up missing babies a year or two further on. Forthcoming curricula cannot exist in a vacuum and change can be considered constant. Pedagogy must be informed by reasoned judgment of educators, taking risks where necessary, but evaluating success continually. The best educators in any field operate as reflective practitioners [6]. As computing educators, that is no less our responsibility.

References

1 The ACM code of ethics and professional conduct. http://www.acm.org/constitution/bylaw17.txt.

2 *The 1983 model program in computer science and engineering*. Technical Report 932, Computer Society of the IEEE, 1983.

3 IS'95: Guideline for undergraduate IS curriculum. *MISQ*, (1995), 341–359.

4 Abelson, H., Sussman, G., and Sussman J. *Structure and Intrepretation of Computer Programs*. MIT Press, 1985.

5 ACM/IEEE-CS Joint Curriculum Task Force. *Computing Curricula 1991*. ACM Press, 1991.

6 Argyris, C, and Schon, D. *Theory and Practice*. Jossey Bass, 1974.

7 Astrachan O. and Reed, D. The applied apprenticeship approach to CS1. *SIGCSE Bulletin*, (March 1995), 1–5.

8 Avison, D. and Myers, D. Information systems and anthropology: An anthropological perspective on IT and organisational culture. *Information Technology And People 8*, 3 (1995), 43–65.

9 Baase, S. *A Gift of Fire: Social, Legal, and Ethical Issues in Computing*. Prentice-Hall, 1997.

10 Baecker, R. and Sherman, D. *Sorting out sorting*. 1981. 16mm color sound film.

11 Bell, S. Legal debate defines copyright. *Computerworld*, (April 1989).

12 Benbasat, I. and Weber, R. Rethinking "diversity" in information systems research. *Information Systems Research 7*, 4 (1991), 389–399.

13 Benedict, R. *Patterns of Culture*. Houghton Mifflin, 1934.

14 Bergin, J., Brodlie, K., Goldweber, M., Jiménez-Peris, R., Khuri, S., Patiño-Martínez, M., McNally, M., Naps, T., Rodger, S., and Wilson, J. An overview of visualization: its use and design. *SIGCSE Bulletin 28*, special issue (1996), 192–200.

15 Bogoiavlenski, I. A. and Pechnikov, A. A. Five years experience of architecture and assembly language introduction course for first year students. In *Proceedings of the Interdisciplinary Workshop on Complex Learning in Computer Environment (CLCE '94)*, 1994. URL: http://cs.joensuu.fi/~mtuki/www_clce.270296/louri.html.

16 Boguslaw, R. *The New Utopians*. Prentice Hall, 1965.

17 Booch, G. The future of software. Opening Address, *Fifth Annual CCSC Rocky Mountain Conference*, (October 17–18, 1996).

18 Brooks, F. *No Silver Bullet — Essence and Accident in Software Engineering*. Addison-Wesley Publishing Co., anniversary edition, 1995.

19 Brown, C., Fell, H., Proulx, V. and Rasala, R. Instructional frameworks: Toolkits and abstractions in introductory computter science. *Proceedings of the 1993 Computer Science Conference*. ACM Press, 1993.

20 Brown, L. (Ed.). *The New Shorter Oxford English Dictionary*. Clarendon Oxford, 1993.

21 Bruce, K. Thoughts on computer science education. URL: http://www.acm.org/pubs/citations/journals/surveys/1996-28-4es/a93-bruce/.

22 Burton, D. *The effect of closed laboratory activities on the comprehension of five concepts and the perception of effectiveness of the course in a second semester computer science course*. PhD thesis, University of Texas at Austin, 1992.

23 Checkland, P. *Systems Thinking, Systems Practice*. Wiley, 1981.

24 Clear, A. Quality control expert system: A project review. *New Zealand Journal of Applied Computing and Information Technology 1*, 1 (1997), 49–62.

25 Clement, A. and Van den Besselaar, P. A retrospective look at PD projects. *Communications of the ACM*, (June 1993).

26 Cockburn, A. The interaction of social issues and software architecture. *Communications of the ACM*, (October 1996), 40–46.

27 Corman, T., Leiserson, C., and Rivest, R. *Introduction to Algorithms*. MIT Press, 1990.

28 Curriculum Committee on Computer Science. Curriculum 68: Recommendations for the undergraduate program in computer science. *Communications of the ACM*, (March 1968), 151–197.

29 Curriculum Committee on Computer Science. Curriculum 78: Recommendations for the undergraduate program in computer science. *Communications of the ACM*, (March 1978), 147–166.

30 Davis, M. *Influences of Mathematical Logic on Computer Science*. Oxford University Press, anniversary edition, 1988.

31 Denning, P., Comer, D., Gries, D., Mulder, M., Tucker, A., Turner, A., and Young, P. Computing as a discipline. *Communications of the ACM*, (January 1989).

32 Denning, P. Educating a new engineer. *Communications of the ACM 35*, 12 (1992), 83–97.

33 DeSanctis, G. *Group Support Systems: New Perspectives*, chapter Shifting Foundations in Group Support System Research. MacMillan, 1993.

34 Dreyfus, H. and Dreyfus, S. *Mind Over Machine*. The Free Press, 1986.

35 Feldman, T. and Zelenski, J. The quest for excellence in designing CS1/CS2 assignments. *SIGCSE Bulletin*, (February 1996), 319–323.

36 Fell, H. and Proulx, V. Exploring Martian planetary images: C++ exercises for CS1. *SIGCSE Bulletin*, (February 1997), 30–34.

37 Fogarty, R. Ten ways to integrate curriculum. *Educational Leadership 49*, 2 (1991), 61–65.

38 Foley, J., Van Dam, A., Feiner, S., and Hughes, J. *Computer Graphics: Principles and Practice* (2nd). Addison–Wesley, 1990.

39 Forester, T. and Morrison, P. *Computer Ethics: Cautionary Tales and Ethical Dilemmas in Computing*. MIT Press, 1990.

40 Freund, S. and Roberts, E. THETIS: An ANSI C programming environment for introductory use. *SIGCSE Bulletin*, (February 1996), 300–304.

41 Gabriel, R. P. *Patterns of Software*. Oxford University Press, 1996.

42 Gamma, E., Helm, R., Johnson, R., and Vlissides, J. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

43 Gates, B. *The Road Ahead*. Penguin Books, 1996.

44 Gibbs, N. The SEI education program: The challenge of teaching future software engineers. *Communications of the ACM*, (May 1989).

45 Glass, R. The relationship between theory and practice in software engineering. *Communications of the ACM*, (November 1996), 11–13.

46 Goldberg, M. WebCT and first year: Student reaction to and use of a Web-based resource in first year computer science. *SIGCSE/SIGCUE Conference on Integrating Technology into Computer Science Education*, (Uppsala, Sweden, June 1997).

47 Gorgonne, J., IS '97, ACM, 1997.

48 Granger, M. J., Little, J. C., Adams, E. S., Björkman, C., Gotterbarn, D., Juettner, D. D., Martin, C. D., and Young, F. H. Using information technology to integrate social and ethical issues into the computer science and information science curriculum. *SIGCSE/SIGCUE Conference on Integrating Technology into Computer Science Education*, (Uppsala, Sweden, June 1997).

49 Greenleaf, N. Algorithms and Proofs: Mathematics in the Computing Curriculum. *Proceedings of a Summer Symposium*, (San Antonio, Texas, June 1991). Springer-Verlag, 1992.

50 Gries, D. *The Science of Programming*. Springer-Verlag, 1981.

51 Hartley, S., Gerhardt-Powals, J., Jones, D., McCormack, C., Medley, M. D., Price, B., Reek, M., Summers, M. K. Enhancing teaching using the Internet. *SIGCSE Bulletin 28*, special issue (1996), 218–228.

52 Hartmanis, J. and Lin, H. *Computing The Future: A Broader Agenda for Computer Science and Engineering*. National Academy Press, 1992.

53 Hirscheim, R. and Klein, H. Four paradigms of informations systems development. *Communications of the ACM*, October 1989.

54 Hudson, H. *Report of the Discipline Review of Computing Studies and Information Sciences Education*. Australian Government Publishing Service, 1992.

55 Jones, C. Software productivity research: What are function points. URL: http://www.spr.com/library /funcmet.htm.

56 Kendall, R. and Losee, R. Information system folklore: A new technique for system documentation. *Information Management*, (February 1986), 103–111.

57 Koffman, E., Miller, P., and Wardle, C. Recommended curriculum for CS1: 1984. *Communications of the ACM*, (October 1984), 998–1001.

58 Koffman, E., Stemple, D. and Wardle, C. Recommended curriculum for CS2: 1984. *Communications of the ACM*, (August 1984), 815–818.

59 Lammers, S. *Programmers at Work*. Microsoft Press, 1986.

60 Langer, E. *Mindfulness*. Addison-Wesley, 1989.

61 Langer, E. *The Power of Mindful Learning*. Addison-Wesley, 1997.

62 Langley, A., Mintzberg, E., Pitcher, P., Posada, E., and Saint-Macary, J. Opening up decision making: The view from the black stool. *Organization Science*, (May 1995), 260–279.

63 Laroche, H. From decision to action in organizations: Decision-making as social representation. *Organization Science 6*, 1 (1995), 63–75.

64 Levine, M. *Effective Problem Solving*. Prentice-Hall, 1994.

65 Liskov, B. and Guttag, J. *Abstraction and Specification in Program Development*. MIT Press, 1986.

66 Makkonen, P. Does collaborative hypertext support better engagement in learning of the basics in informatics. *SIGCSE/SIGCUE Conference on Integrating Technology into Computer Science Education*, (Uppsala, Sweden, June 1997).

67 Martin, C. D., Huff, C., Gotterbarn, D., and Miller, K. Implementing a tenth strand in the CS curriculum. *Communications of the AC*, (December 1996), 75–84.

68 Maturana, H. and Varela, F. *Autopioesis and Cognition*. Reidel Pub., 1980.

69 McCarthy, J. *Dynamics of Software Development*. Microsoft Press, 1995.

70 McConnell, S. *Rapid Development: Taming Wild Software Schedules*. Microsoft Press, 1996.

71 Miles, G. One approach for teaching software engineering across the undergraduate computer science curriculum. *Computer Science Education*, (January 1988).

72 Myers, J. P. Jr. The central role of mathematical logic in computer science. *SIGCSE Bulletin*, (February 1990).

73 Naps, T., Bergin, J., Jiménez-Peris, R., McNally, M. F., Patiño-Martínez, M., Proulx, V. K., and Tarhio, J. Using the WWW as the delivery mechanism for interactive,

visualization-based instructional modules. *Conference on Integrating Technology into Computer Science Education*, (Uppsala, Sweden, June 1997).

74 Naur, P. Programming as theory building. *Microprocessing and Microprogramming 15*, (1985), 253–261.

75 Newell, A. and Simon, H. *Human Problem Solving*. Prentice-Hall, 1972.

76 OECD. *The Measurement of Scientific and Technical Activities*. OECD, 1993. Frascati Manual.

77 Olesen, K. The use of synchronous and asynchronous methods of computer mediated communication in collaborative learning.

78 Orlikowsi, W. and Baroudi, J. Studying information technology in organizations: Research approaches and assumptions. *Information Systems Research*, (March 1991), 1–28.

79 Orlikowski, W. The duality of technology: Rethinking the concept of technology in organizations. *Organization Science 3*, 3 (1992), 398–427.

80 Polya, G. *How To Solve It* (2nd). Princeton University Press, 1957.

81 Prather, R. E. *A Freshman-Sophomore Curriculum Integrating Discrete and Continuous Mathematics*. MAA, 1989.

82 Pressman, R. *Software Engineering: A Practitioner's Approach* (3rd). McGraw-Hill, 1992.

83 Proulx, V. and Rasala, R. Outline for the SDCR position statement on the future of computer science education. URL: http://www.ccs.neu.edu/home/rasala/ProulxRasalaOutline.html.

84 Proulx, V. and Rasala, R. SDCR position statement on the future of computer science education. URL: http://www.ccs.neu.edu/home/rasala/ProulxRasala.html.

85 Proulx, V., Rasala, R., and Fell, H. Foundations of computer science: What are they and how do we teach them. *SIGCSE Bulletin*, (June 1996), 42–48.

86 Rasala, R., Proulx, V., and Fell, H. From animation to analysis in introductory computer science. *SIGCSE Bulletin*, (March 1994), 61–65.

87 Roberts, E. Using C in CS1: Evaluating the Stanford experience. *SIGCSE Bulletin*, (March 1993), 117–121.

88 Roberts, E. *The Art and Science of C: An Introduction to Computer Science*. Addison-Wesley, 1995.

89 Roberts, E. A C-based graphics library for CS1. *SIGCSE Bulletin*, (March 1995), 163–671.

90 Rubenstein, R. Computer science projects with music. *SIGCSE Bulletin*, (March 1995), 278–282.

91 Science Ministry of Research and Technology. *New Zealand Research and Experimental Development Statistics; All Sectors*. Publication no. 15, 1993.

92 Schlimmer, J. C. Practicum/powerpen: Four year student software teams. *Forum for Advancing Software Engineering Education 6*, 11 (1996).

93 SDCR Working Group on Computer Science Education. Strategic directions in computer science education. *ACM Computing Surveys*, (December 1996), 836–845.

94 Sewell, W. A *Weaving a Program: Literate Programming in WEB* . Van Nostrand Reinhold, 1989.

95 Simon, H. A. *The Sciences of the Artificial* (3rd). MIT Press, 1996.

96 Sommerville, I. *Software Engineering* (5th). Addison-Wesley Publishing Co., 1995.

97 Stroustrup, B. *The Design and Evolution of C++*. Addison-Wesley, 1994.

98 Tarjan, R. *Data Structures and Network Algorithms*. SIAM, 1983.

99 Trigg, R. and Anderson, S. Introduction to this special issue on current perspectives on participatory design. *Human-Computer Interaction 11*, (1996), 181–185.

100 Varela, C. R. Harre and Merleau-Ponty: beyond the absent moving body in embodied social theory. *Journal for the Theory of Social Behavior 24*, 2 (1994), 167–185.

101 Walker, H. and Schneider, G. M. A revised model curriculum for a liberal arts degree in computer science. *Communications of the ACM*, (December 1996), 85–95.

102 White, L. and Taket, A. The death of the expert. *Journal of the Operational Research Society 45*, 7 (1994), 733–748.

103 Winograd, T. and Flores, F. *Understanding Computers and Cognition*. Ablex, 1986.

104 Wolz, U., Palme, J., Anderson, P., Chen, Z., Dunne, J., Karlsson, G., Laribi, A., Männikkö, S., Spielvogel, R., and Walker, H. Computer-mediated communication in collaborative educational settings. *SIGCSE/SIGCUE Conference on Integrating Technology into Computer Science Education*, (Uppsala, Sweden, June 1997).