RESEARCH PARADIGMS IN COMPUTER SCIENCE*

Peter Wegner
Brown University

Keywords and Phrases: Paradigms, research, computer science, methodology, philosophy, empirical method, abstraction, software engineering, complexity, programming languages, analysis of algorithms.

"The empirical method is generally characterized by the collection of large amounts of data before much speculation as to their significance or without much idea of what to expect, and is to be contrasted with more theoretical methods in which the collection of empirical data is guided to a large extent by preliminary theoretical exploration. The empirical method is necessary in entering into hitherto unexplored fields and becomes less necessary the greater the acquired mastery of the field."

- P.W. Bridgman, McGraw-Hill Encyclopedia of Science and Technology

Abstract

This paper explores the ramifications of four influential definitions of computer science:

1. Computer science is the study of phenomena related to computers, Newell, Perlis and Simon, 1967
2. Computer science is the study of algorithms, Knuth, 1968
3. Computer science is the study of information structures, Wegner, 1968, Curriculum 68
4. Computer science is the study and management of complexity, Dijkstra, 1969.

The first definition reflects an empirical tradition since it asserts that computer science is concerned with the study of a class of phenomena. The second and third definitions reflect a mathematical tradition since algorithms and information structures are two abstractions from the phenomena of computer science. The fourth definition reflects the great complexity of engineering problems encountered in managing the construction of complex software-hardware systems. It is argued in section 1 that computer science was dominated by empirical research paradigms in the 1950s, by mathematical research paradigms in the 1960s and by engineering oriented paradigms in the 1970s. Section 2 illustrates how these three phases of development are reflected in the field of programming languages.

The remaining sections consider in greater detail how empirical, mathematical and engineering research paradigms have affected the development of computer science. Section 3 indicates that although the phenomena of computer science are created by man they can be studied using the empirical techniques of the natural sciences. Section 4 distinguishes between "micro computer science" concerned with the study of individual algorithms and "macro computer science" concerned with the study of mechanisms and notations for specifying all algorithms; and between intensional "how" specifications and extensional "what" specifications for programs and computing systems. Section 5 distinguishes between the uses of the term "complexity" in software engineering and the analysis of algorithms and suggests that different terms be used to denote these two kinds of complexity. In a final section it is argued that the diversity of research paradigms in computer science may be responsible both for our difficulties in deciding how computer scientists should be trained and for divergences of opinion concerning the nature of computer science research.

*This research was supported in part by AFOSR, ARO and ONR under contract N00014-76-C-0160.

1. Introduction

Computer science is in part a scientific discipline concerned with the empirical study of a class of phenomena[1], in part a mathematical discipline concerned with the formal properties of certain classes of abstract structures, and in part a technological discipline concerned with the cost-effective design and construction of commercially and socially valuable products [W1]. Research workers in computer science may think of themselves as empirical scientists, mathematicians or engineers. The research paradigms used by computer scientists accordingly include paradigms taken from science, mathematics and engineering. We shall briefly characterize the research paradigms of empirical science mathematics and engineering, and then consider the role of each of these classes of paradigms in the development of computer science.

The "classical" paradigm for the empirical sciences assumes that there is an initial descriptive data-collection phase during which a large amount of data concerning phenomena in the domain of discourse is collected. Observed uniformities and differences among phenomena lead to a classification scheme and eventually to the development of models and theories which account for the observed data in an economical (parsimonious) way.

Research in mathematics is concerned with the development of abstractions from a class of phenomena and with the study of properties of the abstractions independently of the phenomena from which they were derived. If the abstractions are realistic models of the

[1] The "phenomena" of computer science include digital computers, programming languages, algorithms, programs, and other man-made entities and concepts which owe their existence to the development of computers.

class of phenomena from which they were derived, then reasoning about the abstractions may provide useful information concerning the class of phenomena.

Research in engineering is directed towards the efficient accomplishment of specific tasks and towards the development of tools that will enable classes of tasks to be accomplished more efficiently. We are usually given a "what" requirements specification of what is to be accomplished and are asked to develop a "how" implementation. The "what" specification may generally be realized by a large variety of different "how" implementations. The problem-solving paradigm of the practicing engineer generally involves a sequence of systematic selection or design decisions which progressively narrow down alternative options for accomplishing the task until a unique realization of the task is determined. The research engineer may use the paradigms of mathematics and physics in the development of tools for the practicing engineer, but is much more concerned with the practical implications of his research than the empirical scientist or the mathematician.

Computer science may, as a first approximation, be characterized by three phases of development, respectively dominated by empirical, mathematical and engineering research paradigms.

1. A "data-gathering" phase from about 1950-1960 in which the prime activity was the discovery and description of computational phenomena, with hardly any work on the development of models, abstractions or theories. The paradigm appropriate to this activity is the paradigm of the empirical sciences.

2. An "elaboration and abstraction" phase from about 1961-1969 concerned with the extension and elaboration of computers and languages discovered in the 1950s, and with the development of abstractions to account for the observed properties of the phenomena of computer science. The paradigm appropriate to this activity is the paradigm of mathematics.

3. A "technological" phase from 1970 onwards concerned with the management of the increasingly complex software-firmware-hardware systems required for the solution of system and applications programming problems. The paradigm appropriate to this activity is the paradigm of engineering.

It is easy to find exceptions to this characterization. For example, Turing's work on undecidable problems preceded 1950. However, there is considerable evidence that disciplines go through phases in which they are dominated by different paradigms [K4]. The characterization of the 50s as the age of empirical discovery, the 60s as the age of elaboration and abstraction, and the 70s as the age of technological consolidation seems to fit the facts remarkably well.

It is natural for disciplines to evolve from an initial empirical phase through a mathematical phase to a "practical" engineering-oriented phase. Computer science is distinguished by the rapidity of this evolution so that research workers representing all three paradigms are active in the same generation. The current divergences of opinion in the academic community concerning the nature of research in computer science and concerning the form of undergraduate and graduate education are in part due to the rapidity of this evolution.

2. The Development of Programming Languages

The role of the empirical, mathematical and engineering traditions in the development of computer science may be illustrated by considering the field of programming languages. The development of programming languages may be characterized by three phases which we shall call the age of empirical discovery, the age of elaboration and analysis, and the age of technology.

1950-60 The age of empirical discovery

A remarkably large number of the basic concepts of programming languages had been discovered and implemented by 1960. This period includes the development of symbolic assembly languages, macro assembly languages, FORTRAN, ALGOL 60, IPL V, LISP, COBOL and COMIT [S1]. It includes the discovery of many of the basic implementation techniques such as symbol table construction and look-up techniques for assemblers and macro assemblers, the stack algorithm for evaluating arithmetic expressions, the activation record stack with display technique for keeping track of accessible identifiers during execution of block structure languages, and marking algorithms for garbage collection in languages such as IPL V and LISP.

This period was one of discovery and description of programming languages and implementation techniques. Programming languages were regarded solely as tools for facilitating the specification of programs rather than as interesting objects of study in their own right. The development of models, abstractions and theories concerning programming languages was largely a phenomenon of the 1960s.

1961-1969 The age of elaboration and abstraction

The 1960s were a period of elaboration of programming languages developed in the 1950s and of abstraction for the purpose of constructing models and theories of programming languages.

The languages developed in the 1960s include JOVIAL, PL/I, SIMULA 67, ALGOL 68 and SNOBOL 4. These languages are, each in a different way, elaborations of languages developed in the 1950s. For example, PL/I is an attempt to combine the "good" features of FORTRAN, ALGOL, COBOL and LISP into a single language. ALGOL 68 is an attempt to generalize, as systematically and cleanly as possible, the language features of ALGOL 60. Both the attempt to achieve greater richness by synthesis of existing features and the attempt to achieve greater richness by generalization have led to excessively elaborate languages. We have learnt that in order to achieve flexibility and power of expression in programming languages we must pay the price of greater complexity. In the 1970s there is a tendency to retrench towards simpler languages like PASCAL, even at the price of restricting flexibility and power of expression.

Theoretical work in the 1960s includes many of the basic results of formal languages and automata theory with applications to parsing and compiling [A1]. It includes the development of theories of operational and mathematical semantics, of language definition techniques and of several frameworks for modelling the compilation and execution process [DSIPL]. It includes the development of the basic ideas of program correctness amd program verification [M1].

Although much of the theoretical work started in the 1960s continued into the 1970s, the emphasis on theoretical research as an end in itself is essentially a phenomenon of the 1960s. In the 1970s theoretical research in areas such as program verification is increasingly motivated by practical technological considerations rather than by the "pure research" objective of advancing our understanding independently of any practical payoff.

In the programming language field the pure research of the 1960s tended to emphasize the study of abstract structures such as the lambda calculus or complex structures such as ALGOL 68. In the 1970s this emphasis on

abstraction and elaboration is gradually being replaced by an emphasis on methodologies aimed at improving the technology of programming.

1970-? The age of technology

During the 1970s emphasis shifted away from "pure research" towards practical management of the environment, not only in computer science but also in other scientific areas. Decreasing hardware costs and increasingly complex software projects created a "complexity barrier" in software development which caused the management of software-hardware complexity to become the primary practical problem in computer science. Research was directed away from the development of powerful new programming languages and general theories of programming language structure towards the development of tools and methodologies for controlling the complexity, cost and reliability of large programs.

Research emphasized methodologies such as structured programming, module design and specification, and program verification [ICRS]. Attempts to design verifiable languages which support structured programming and modularity are currently being made. PASCAL, CLU, ALPHARD, MODULA and EUCLID are examples of such "methodology-oriented languages".

The technological, methodology-oriented approach to language design results in a very different view of what is important in programming language research. Whereas work in the 1960s was aimed at increasing expressive power, work in the 1970s is aimed at constraining expressive power so as to allow better management of the process of constructing large programs from their environments. It remains to be seen whether the management of software complexity can be substantially improved by imposing structure, modularity and verifiability constraints on program construction.

3. The Study of Phenomena Related to Computers

Computer science has been defined by Newell, Perlis and Simon as "the study of phenomena related to computers" [N1]. Although this definition may at first strike the reader as tautological, it in fact provides guidelines both as to the subject matter which should be studied and as to the methodology which should be used in studying the subject matter.

The definition emphasizes that physical computational devices as opposed to models or theories should be the central subject matter of computer science. This view is implicit in the choice of "computer science" rather than "computing science" as the name of the discipline. It is implicit also in the fact that the British and American professional societies call themselves the British Computer Society (BCS) and the Association for Computing Machinery (ACM).

The definition also suggests that computer science is like an empirical science in that it is concerned with the study of a class of phenomena. The phenomena of computer science include digital computers and "phenomena related to computers" like algorithms, programs and programming languages. The "empirical science" paradigm suggests the collection of observations (data) concering computational phenomena, and the determination of uniformities among computational phenomena. When a sufficient set of observations has been accumulated, theories which explain the phenomena may be developed.

Computer science differs from physics or botany in that its phenomena are not natural phenomena but man-made phenomena like digital computers, programming languages and algorithms. However, it is quite appropriate to think of a specific digital computer or programming language as a data point of a data space of digital computers or programming languages, and to develop simplifying theories which allow us to characterize spaces of computational phenomena so that we can more easily select or design digital computers or programming languages suited to a particular purpose.

The empirical science paradigm is appropriate to the early data-gathering stage of computer science. Moreover, there are certain areas of current research, such as the area of performance analysis and evaluation, where the empirical science paradigm is still the most appropriate. However, once a substantial amount of data concering a class of phenomena has been collected, it is inevitable that models and abstractions from observed phenomena will be developed by mathematically-oriented researchers who are more interested in the abstractions than in the phenomena from which the abstractions are derived.

Empirical research continues to be appropriate in computer science whenever the systems being analyzed are too complex to be understood by studying their internal structure and can be understood only by studying their behavior. However, it is important not to give up too easily in gaining a structural understanding of complex systems, since behavioral information is "qualitatively" different from structural information and can rarely, if ever, replace an understanding of system structure. Attempts to gain an understanding of system structure from an understanding of system behavior have been notoriously unsuccessful.

Theoretical computer science is mathematical not only because the process of model building and abstraction is inherently mathematical, but also because computational phenomena such as programs have natural abstractions as mathematical objects such as functions. Theoretical computer science is in this respect like theoretical physics. However, the kind of mathematics appropriate to the description of computational objects can be very different from the kind of mathematics appropriate to the description of physical objects.

4. Modelling and Abstraction

Modelling and abstraction in computer science makes use of mathematics in two different ways.

1. A model or abstraction of an object requires intuitive use of logical and mathematical notions which corespond at the formal level to the notion of a homomorphism.

2. In areas such as program verification, formal languages and automata theory, and analysis of algorithms, mathematics is required not only at the meta level in establishing the model but also at the object level in specifying objects being manipulated in the model.

As a discipline matures, there is a tendency to place increasing emphasis on abstractions used in modelling the underlying phenomena. Two important abstractions in computer science are the notions of "algorithm" and "information structure". These abstractions have led to the following two influential definitions of computer science:

A1: Computer science is the study of algorithms.
A2: Computer science is the study of representation, transformation and interpretation of information structures.

The definition A1 is due to Knuth and forms the conceptual cornerstone of his seven-volume treatise on "The Art of Computer Programming" [K1]. This view of computer science is also the starting point for work in computational complexity and the analysis of algorithms. These subfields of computer science have come into being only in the 1970s and have already replaced more traditional theoretical subfields such as formal languages a

automata theory as the dominant area of theoretical research. Knuth in [K2] asserts that "perhaps the most significant discovery generated by the advent of computers will turn out to be that algorithms, as objects of study, are extraordinarily rich in interesting properties".

The definition A2 was used by Wegner as the unifying abstraction in his book on Programming Languages, Information Structures and Machine Organization [W2]. This view of computer science has its historical roots in information theory [S2]. It strongly influenced the development of Curriculum 68 [C68] - a document which has been very influential in the development of undergraduate computer science curricula. It is implicit in the German and French use of the respective terms "Informatik" and "Informatique" to denote the discipline of computer science.

It is interesting to note that the British term "computer science" has an empirical orientation, while the corresponding German and French terms have an abstract orientation. This difference in terminology appears to support the view that the nineteenth-century traits of British empiricism and continental abstraction have persisted into the second half of the twentieth century.

The definition A2 abstracts from specific mechanical devices for transforming information structures such as digital computers and from specific conceptual models for function evaluation such as programming languages and chooses the notion "information structure" as a normal form for characterizing the phenomena of computing. The notions of representation, transformation and interpretation may be thought of as the counterparts of syntax, semantics and pragmatics. Syntax characterizes a set of representations independently of any transformational attributes they may possess. Semantics may, in the context of the information structure approach, be defined in terms of transformational properties of the representations. Pragmatics is concerned with the relation between computational objects and the human or mechanical interpreters which operate upon them.

The view that information is the central idea of computer science is both scientifically and sociologically suggestive. Scientifically, it suggests a view of computer science as a generalization of information theory which is concerned not only with the transmission of information but also with its transformation and interpretation. Sociologically, it suggests an analogy between the industrial revolution, which is concerned with the harnessing of energy in the service of man, and the computer revolution, which is concerned with the harnessing of information in the service of man.

The algorithm and information structure approaches determine two different paradigms for the study of computer science. This is borne out by the fact that algorithms and information structure people have completely different attitudes to the study of computational problems, and tend to study different kinds of problems. Algorithms people are concerned with the design and analysis of efficient algorithms for particular problems, and with the attempt to find optimal algorithms for performing a particular task. Information structure people are concerned with the development of mechanisms and notations for computing all computable functions.

The distinction between the algorithm and information structure approaches is essentially a distinction between models for solving specific problems and models for characterizing general-purpose tools. The question of specificity versus generality is a recurrent theme in computer science which is so pervasive that it may be used as a basis for classifying work in computer science into two categories. We may distinguish between micro computer science concerned with the analysis and solution of specific problems and macro computer science concerned with the development of general-purpose tools, techniques and theories.

There are many disciplines in which there is a micro paradigm concerned with the analysis of small-scale phenomena and a macro paradigm concerned with the analysis of large-scale phenomena. Thus micro economics is concerned with the analysis of small economic units (individuals) while macro economics is concerned with the analysis of large economic units (such as the gross national product). In physics, quantum theory may be regarded as part of micro physics while relativity and the laws of thermodynamics may be regarded as part of macro physics.

In computer science the specific micro phenomena are individual programs and algorithms while the macro phenomena are mechanisms, notations and theories for the characterization of all algorithms.

The large amount of recent work on the analysis of algorithms is largely micro computer science. Knuth's volume 3 [K1], which devotes 600 pages to the analysis of sorting and searching, illustrates the rich and rewarding nature of work in micro computer science.

It is sometimes important to distinguish between the specificity of the problem being solved and the specificity of the tool being used to solve the problem. Many of the micro tools used in the analysis of algorithms may be applied to a very broad class of different micro problems.

Micro problems which at first seem too specialized to be interesting turn out to have unexpected ramifications because of "algorithm reductions" which allow an algorithm, say A, to be used as a basis for performing a second seemingly different algorithm, say B. For example, intensive study of 2×2 matrix multiplication led to the development of an algorithm (Strassen's algorithm [S3]) which allows 2×2 matrices to be multiplied in 7 rather than 8 multiplications. I personally found this result singularly unexciting when I first encountered it. However, this result allows us to develop multiplication algorithms for n×n matrices with an exponential saving in computation time, and to save computation time in certain graph computation algorithms, context-free language recognition algorithms and many other algorithms related to matrix computation. Thus, the Strassen algorithm for 2×2 matrix multiplication, which at first seems to be an incredibly specialized "micro" gimmick for a toy problem, turns out to result in nontrivial potential savings of computation time for quite a large class of non-toy problems.

Macro computer science includes the subfields of programming languages and operating systems. It includes the theoretical subfields of formal languages and automata theory, and of program verification, since these subfields are concerned with properties of large classes of algorithms rather than with properties of specific algorithms. The newly emerging subfield of software engineering is also macro computer science since it is concerned with the development of tools for managing the complexity of large classes of algorithms rather than with the analysis of specific algorithms.

The terms micro and macro computer science characterize the two extremes of specificity and generality. There is clearly a middle ground between the study of properties of specific algorithms and the study of properties of all algorithms, which is concerned with the study of properties of classes of algorithms.

One example of a class of algorithms is the class of all algorithms for realizing a given task (such as

sorting or matrix multiplication). Such classes of algorithms are extremely rich and may contain many essentially different algorithms for the same task whose equivalence is by no means obvious (consider the rich variety of different sorting algorithms). A task specification may be regarded as a "what" specification for the equivalence class of all algorithms ("how" specifications) which realize the task. The set of all algorithms in such an equivalence class generally has a non-denumerable number of elements, and is generally intractable in the sense that the problem of determining whether two algorithms are in a given equivalence class is unsolvable [W3]. The classical lower-bound problem in computational complexity is the problem of finding "optimal" algorithms for such equivalence classes. This problem is intractable for the same reasons that the equivalence problem is unsolvable.

A "what" specification of a task is referred to by philosophers as an extensional specification, while a "how" specification is referred to as an intensional specification [C1][2]. The problem of relating extensional "what" specifications of tasks with intensional "how" specifications by algorithms or programs is one of the fundamental problems of computer science. There are some tasks for which the extensional specification can be given by a simple mathematical input-output relation so that the extensional task definition is simpler than that of any intensional program realizing the task. However, there are programs (with an unsolvable halting problem) that cannot be defined by any input-output relation so that the class of extensional objects specifiable by programs is richer than the class of extensional objects specifiable by input-output relations. Moreover, there are many practically important computational tasks where the "what" specification may be more complex than the "how" specification, requiring many hundreds of pages. One of the prime sources of difficulty in large programming projects arises from the fact that we have no mechanism for "what" specifications that is as flexible as programming languages are for "how" specifications. A more flexible mechanism for "what" specifications would be a very important contribution to the development of computer science.

The difference between micro analysis of a specific algorithm and "optimality" analysis for the class of all algorithms which realize a given task can be characterized in terms of a difference of the variable of quantification. When analyzing a specific algorithm we quantify over the set of all data values in the domain of the algorithm to obtain quantities such as average and maximum running times. When analyzing the class of algorithms for performing a given task, we quantify over a class of algorithms rather than over a class of data values. Since the class of data values for an algorithm is generally denumerable while the class of algorithms for a task is generally non-denumerable, optimality analysis for tasks is a less well-structured problem than running time analysis for specific algorithms.

Running-time analysis for algorithms and optimality analysis for tasks are subfields of the analysis of algorithms which require different analysis techniques and therefore different research paradigms.

Knuth in [K3] refers to the analysis of specific algorithms as type A analysis and to the analysis of families of algorithms for solving a particular problem as type B analysis. In terms of our terminology, type A analysis is micro analysis and is generally concerned with intensional "how" specifications of algorithms, while type B analysis falls in the middle ground between micro and macro analysis and is concerned with properties of equivalence classes of algorithms associated with an extensional "what" specification.

Another example of a naturally occurring family of algorithms is the class of "important" algorithms associated with a given area of application (such as inventory control or automated design of military systems). There is a great deal of duplication of programming among different system development efforts in a given application area because of the lack of effective technology transfer or module standardization in the application area. The development of standards for applications areas is an important practical problem that falls in the middle ground between micro analysis of properties of specific algorithms and macro analysis of properties of all algorithms.

Three kinds of standardization are currently being considered in the programming community.

1. Programming language standardization to facilitate portability of programs and cut down on programmer-training costs.

2. Software methodology standardization, including module interface conventions, programming style conventions, requirements specification conventions, documentation conventions, etc.

3. Applications area standardization to facilitate portability and technology transfer in a given application area.

Programming language and software methodology standardization are concerned with the development of standard notations and environments for expressing all algorithms. Applications area standardization is concerned with standardization for restricted environments and restricted families of algorithms. It is therefore a less ambitious and possibly a more immediately rewarding undertaking than software methodology standardization. The identification of major applications areas and the characterization of their structure for purposes of standardization may turn out to be a rewarding research and development activity with a high payoff.

5. The Management of Complexity

When computers were first developed in the 1940s, computer time was very precious and software costs were less than 5% of hardware costs. Computer pioneers like Von Neumann assumed that computers would be used primarily for solving well-defined numerical problems such as simultaneous or partial differential equations.

In the 1950s and 60s hardware costs decreased by a factor of 2 every two or three years and computers were applied to increasingly ambitious system programming and applications problems such as general-purpose operating systems, airline reservation systems, inventory control systems, command and control systems, aircraft and spacecraft guidance systems and natural language understanding systems. Programming systems to accomplish such tasks may require millions of instructions and millions of data items, and may require hundreds or even thousands of man-years to complete. The great increase in software costs combined with the decrease in hardware costs has led to a situation where software costs averaged 70% of total system cost in 1973, and are projected to average over 90% of total system cost by the year 2000. Thus, our problem-solving ability, which was limited by hardware costs in the 1950s and 60s, will in the future be limited by software costs.

The increased relative cost of software is due in

[2] Carnap defines two designators (of objects) to have the same extension if they are equivalent in a specific interpretation and to have the same intension if they are equivalent in all interpretations. This definition captures the intuitive notion that two objects have the same extension if they exhibit the same external behavior and the same intension if they have the same internal structure.

part to the fact that hardware costs have consistently declined more rapidly than software costs over the last 25 years, in spite of the fact that there have been considerable improvements in software technology. It may be due in part to the fact that the ratio of software effort to computation time for current large software projects is probably greater than that for the well-defined numerical problems being solved in the 1950s. However, an additional and ultimately more important reason for skyrocketing software costs arises from the fact that current large software systems are much more complex (by any measure of complexity) than the systems being developed 25 years ago or even ten years ago. It was pointed out by Dijkstra that the structural complexity of a large software system is greater than that of any other system constructed by man[3], and that man's ability to handle complexity is severely limited [D1,D2]. As a consequence our ability to manage large software systems simply breaks down once a certain threshold complexity is approached. The cost of systems which exceed this threshold becomes prohibitively large. Moreover, it is not only the high expected value of the cost, but also the very high variance in the expected value, and the difficulty of determining the correctness or degree of reliability of delivered systems, that makes such systems unmanageable.

Since our future problem-solving ability is limited primarily by our ability to manage complexity, the management of complexity is one of the most important current problems of computer science. It is no accident that the two most important new subfields of computer science in the 1970s are computational complexity (concerned with the theoretical study of complexity of problems) and software engineering (concerned with the practical management of complexity).

The term "complexity" has different meanings in the fields of computational complexity and software engineering. Computational complexity may generally be measured by a quantitative index or objective function such as the number of instructions executed or the amount of memory space used. Such complexity may generally be referred to as quantitative complexity. In contrast, software complexity is not measurable by any quantitative index but is a function of the structure and complexity of interconnections of software components. Such complexity may be referred to as qualitative or structural complexity. Quantitative complexity may be characterized by a number, while qualitative (structural) complexity can be characterized only by fuzzy terms such as "large" or "very large".

Dictionary definitions of the word "complex" include phrases such as "not easily analyzed or disentangled" which suggest qualitative rather than quantitative complexity. The word "complex" is related to the verb "plex" which means "to interweave" and has been used in the computer literature to denote a very general and abstract notion of structure [R1]. Use of the term "complexity" in its quantitative sense in phrases such as "computational complexity" or "algorithmic complexity" does not accord with intuitive notions of what is meant by complexity or with the dictionary definition of this term. It is probably too late now to change this usage, particularly since it is difficult to think of some other appropriate term to denote quantitative complexity. However, in order to avoid confusion we should perhaps require the use of an adjective such as quantitative, computational or algorithmic when talking about quantitative complexity and assume that the term "complexity" occurring without an adjective always denotes qualitative complexity.

The notion of "structure" is a key term in the characterization of qualitative complexity. This term is etymologically related to the word "construct". Structures are central to computer hardware-software systems because such systems are "constructed" in a complex and subtle way from simple primitive components. The complexity of structure of computing systems is directly related to the complexity of the construction process. One of the unifying abstractions in the study of hardware-software complexity is the notion of a structure which can be broken down hierarchically into successively simpler component structure and ultimately described in terms of atomic, non-decomposable elements. Such a structure may be viewed in a bottom-up way in terms of how successively higher-level structures are constructed from primitive components, in a top-down way in terms of how a designer might develop realizations of a "what" specification in terms of successively lower-level "how" specifications, or in an undirected way as a structure through which it is possible to navigate in any way one pleases.

The activities of software engineering may all be viewed as attempts to grapple with the management of structure. It might be useful to undertake a taxonomy of notions relating to the study of structure, identifying certain notions such as general-purpose selection and construction operations as general-purpose notions applicable to the study of all structures [S4], and identifying other notions such as structured programming and parameter passing in subroutines as special-purpose notions applicable only to restricted structures.

Another important characteristic of qualitative complexity is that "the whole is more complex than the sum of its parts". Thus, if A and B are two subsystems with complexity C(A) and C(B), then the complexity of the combined system A+B satisfies the condition $C(A+B) \geq C(A)+C(B)$. This is the reverse of the triangle inequality.

One important concept which provides a starting point for the management of software complexity is the notion of the software life cycle, consisting of a requirements analysis and specification phase, followed by a program design, development and testing phase, followed by an operations and maintenance phase [ICRS]. Before the emergence of the life cycle concept, managers of software projects emphasized program efficiency and efficient program development, leading to local cost optimization of a part of the software life cycle, possibly at the expense of much greater costs in other parts of the life cycle.

The realization that the operation and maintenance phase was just as critical as the development phase or the requirements phase in software development caused many of the assumptions of programmers and programming language designers to be overthrown. For example, it turned out that readability of programs was more important in the maintenance phase than writability, and modifiability was in many instances as important as efficiency.

The life-cycle point of view allows us to systematically look at all the activities of the software life cycle and all the tools used in managing the software life cycle, to distinguish between activities which we can do well and those we do not know how to do at all, and to distinguish between critical activities whose improvement would substantially improve overall cost and non-critical activities. From this point of view programming languages may well have been critical to overall software management in the 1950s and 1960s, but may have become non-critical in the 1970s because further improvements due to further research may turn out to be marginal.

[3] This assertion of Dijkstra is disputed by Bell and Newell [B1] who assert that the structural complexity of an aircraft carrier or of a city is greater than that of even the largest programming projects.

Our ability to manage software complexity can be improved in the following two ways:

1. Provide a good management structure for managing software complexity over its life cycle.

2. Decompose the problem in such a way that it can be efficiently handled by the management structure.

The chief programmer team is an example of a management structure developed for handling a certain level of software complexity (programs of up to about 100,000 instructions). It has achieved certain spectacular successes but appears to be limited in its applicability both because the organizational structure cannot be extended to handle really large projects (with many millions of instructions) and because it handles only the program development part of the software life cycle and does not provide for adequate services in the maintenance part of the life cycle.

Structured programming is an example of a tool developed to handle program decomposition. It has proved quite influential in affecting programming style and programming language design but is limited in its applicability because it is concerned with program structure at the statement level (programming in the small) rather than with program interfaces at the module level (programming in the large). Systematic techniques for module decomposition and interconnection may ultimately yield greater benefits in the management of complexity than the statement-level modularity made possible by structured programming techniques.

A good management structure allows the threshold of complexity of problems handled by the organization to exceed the threshold which can be handled by an individual. The complexity threshold for an organization is very sensitive to both the formal and informal relations among individuals of the organization, and is probably very time-dependent[B2]. Human factors research to determine good robust organizational structure for handling a high complexity threshold would probably be very rewarding. Research of this kind has almost certainly been done by industrial engineering researchers and/or behavioral psychologists. Managers of computer projects would probably benefit by finding out about such work.

A complex problem may be decomposed into subproblems in an enormous variety of different ways. A "bad" way of decomposing a problem may be exponentially worse in its complexity than a "good" way of decomposing the problem. Good problem analysis is probably more critical than good management structure in pushing back the complexity barrier, and one of the objectives of management structures like chief programmer teams is to ensure good problem analysis. Ideally, it would be nice to find an "optimum" way of solving the problem by minimizing its complexity. However, in practice there is no precise way of measuring qualitative complexity and we must be satisfied with fuzzy notions like "good" and "better".

Software engineering, just like other branches of engineering, is concerned with the construction of a product. Thus, civil engineering is concerned with the construction of buildings and bridges, computer engineering is concerned with the construction of computers, and software engineering is concerned with the construction of software.

Software is like other engineering products in that it is constructed out of primitive components (raw materials) by a sequence of operations controlled by humans. Tools may allow part of the construction process to be automated, just as in the case of other products. The total cost of the product may be measured in terms of the direct cost in human effort and the indirect cost of tools. The construction process may be capital-intensive (making heavy use of tools) or labor-intensive (building the system from scratch).

However, software differs from other products in that it is a mental rather than a physical product. It is not possible to feel or smell software. The degree of completion of a building can be measured directly by looking at the partly completed structure, while the degree of completion of a piece of software cannot be determined by looking at the partly written code. (Note that the difference here is primarily due to the fact of our confidence in the structural soundness of a partly completed building and our lack of confidence in the soundness of a partly written program. Thus the difference between software and other products resides more in our inability to guarantee the correctness of components than in the difficulty of "seeeing" the partly completed structure.)

One of the prime objectives of software engineering is to develop standards for software construction that will allow the same standards of quality and production control to be applied to software as to other engineering products. These objectives require the development of techniques for testing and certifying the correctness of components.

One of the chief differences between software and other engineering products lies in the great complexity of software components and the subtlety of possible interconnections among components. Control over software products will require restriction of possible interconnections among modules to a subset whose effect may be easily understood.

When a civil engineer designs a bridge or a building to certain requirements, he selects the final design from an enormous variety of designs satisfying the requirements, using both explicit and implicit criteria for narrowing down the choices. A software engineer similarly has an enormous number of potential designs satisfying the requirements and must similarly use both explicit and implicit criteria for narrowing down the set of choices. The explicit criteria are part of the emerging "science" of software engineering while the implicit criteria are part of the "art" of computer programming. One of the objectives of software engineering is to increase the science component and decrease the art component in the development of large software systems.

The fact that software engineering aims to increase the science component and decrease the art component in the programming process has both its good and bad aspects. It is good in that the programming process becomes more predictable and hopefully cheaper and more reliable. But this is achieved at the cost of imposing severe constraints on the freedom of action of the programmer which may in certain instances prevent the programmer from finding the best solution to the problem.

The discipline imposed by software engineering may be seen by certain free spirits as a form of fascism while the lack of programming restrictions of the 50s and 60s may be seen by these programmers as a form of free enterprise. In the programming community, just as in society at large, it is sometimes necessary to trade some personal freedom in the interests of efficiency of society as a whole. The degree to which this tradeoff is worth while in computer programming is an issue that is still to be decided.

6. Conclusions

A number of different research paradigms have been considered in the body of this paper. It is evident that each of the paradigms plays an important role in the development of the discipline as a whole, and that the "educated" computer scientist should feel at home

with all of the approaches to research that have been discussed. The computer scientist should be a "universalist", having the enquiring mind of the empirical scientist, the modelling and abstraction ability of the mathematician, and the tool buildling and implementation ability of the engineer.

The training of such universalists presents a problem in computer science education, especially in an age where the tendency is towards increasing specialization. There have been a few examples of universalists in the 20th century, such as the World War II physicists who overcame formidable problems in physics, mathematics and engineering in the development of the atom bomb. However, these physicists were primarily educated in prewar Europe, and the educational systems of postwar America or even postwar Europe simply have not generated the intellectual excitement and the personal dedication necessary for the creation of a generation of universalists.

There have been a number of attempts to develop ambitious computer science curricula which combine a thorough training in mathematics, statistics, operations research and numerical analysis with a solid system programming base and an introduction to management techniques for complex organizations. However, such programs have consistently attracted only small numbers of students. There is some concern among academic computer scientists concerning the fact that we may not be providing the right training for computer science undergraduate and graduate students, but it is very difficult to come up with a constructive viable program either at the undergraduate or at the graduate level. The curriculum committee of the Association for Computing Machinery developed an influential curriculum report in 1968 [C68], but has found the task of preparing a revised report appropriate to the mid-1970s much harder than anticipated.

One clear conclusion is that the management of complexity is a critical problem which must be addressed if we are to make further progress in the computer management of complex systems. There are indications that complexity is becoming a key problem in areas other than computer science. The availability of computers has greatly increased our ability to create complexity and the world in the second half of the twentieth century has, partly as a result of computers, become much more complex. This complexity has caused social, economic and political problems to become unmanageable to the extent that we may be strangled by the complexity and unmanageability of the institutions that we have created. The problems of complexity appear to be particularly acute in "democratic" societies, and there is a tendency both in society as a whole and in the community of programmers to restrict the freedom of individuals in order to simplify problems of management.

The two critical characteristics of computer science in the 1970s are complexity and universality. Although complexity has in recent years received greater attention than universality, it may well be that the universality problem is just as fundamental and difficult to handle as the problem of complexity. The relation between complexity and universality may be characterized as one of depth versus breadth. A complex problem generally has a structure with several levels of hierarchy and may be understood only by intensive study of details of the particular problem. Universality requires an overall understanding of broad concepts and methodologies and in some ways represents the antithesis or perhaps the dual of the notion of complexity. The "compleat" computer scientist must master both complexity and universality. That is, he must combine the traits of the code jock with the ability to handle broad concepts and perspectives. People who possess all the above traits are few and far between, although Knuth is perhaps an example of a compleat computer scientist in our generation.

Current work in software engineering and the analysis of algorithms has made us aware of some of the problems and bottlenecks which stand in the way of the further development of computer science, but we have not made much headway in the solution of these problems. It is not at present clear whether new methods of managing complexity will allow us to push back the complexity barrier by several orders of magnitude or whether such techniques will result in only a trivial increase in our ability to solve complex problems cheaply and reliably. The optimists feel that the complexity of our current problems is due entirely to the fact that we have not yet found the right algorithms, languages and methodologies for solving them, while the pessimists feel that the complexity is inherent in the nature of the problem and that there is no way of substantially pushing back the complexity barrier.

Acknowledgements: Ken Magel, Bob Sedgewick and Andy van Dam provided constructive criticism in the development of this paper.

References:

[A1] Aho, A.V. and Ullman, J.R., The Theory of Parsing, Translation and Compiling, Prentice-Hall, Vol. 1, 1972, Vol. II, 1973.

[B1] Bell, C.G. and Newell, A., Computer Structures: Readings and Examples, McGraw-Hill, 1971.

[B2] Brooks, F.B., The Mythical Man Month, Addison-Wesley, 1975.

[C1] Carnap, R., Meaning and Necessity, University of Chicago Press, 1956.

[C68] Curriculum 68 - A Report of the ACM Curriculum Committee on Computer Science, CACM, March 1968.

[DSIPL] Data Structures in Programming Languages, Proceedings of a Symposium, SIGPLAN Notices, February 1971.

[D1] Dijkstra, E.W., Notes on Structured Programming, in Structured Programming, Dahl, Dijkstra and Hoare, Academic Press, 1972.

[D2] Dijkstra, E.W., A Discipline of Programming, Prentice-Hall, 1976.

[ICRS] Proc. International Conference on Reliable Software, April 1975.

[K1] Knuth, D.E., The Art of Computer Programming, Vol. I: Fundamental Algorithms (1968), Vol. II: Seminumerical Algorithms (1969), Vol. III: Sorting and Searching (1973), Vols. IV-VII to be published, Addison-Wesley.

[K2] Knuth, D.E., Computer Science and its Relation to Mathematics, American Mathematical Monthly, 1973.

[K3] Knuth, D.E., Mathematical Analysis of Algorithms, Proc. IFIP 1971, North-Holland, 1972.

[K4] Kuhn, T.S., The Structure of Scientific Revolutions, University of Chicago Press, 1970.

[M1] Manna, Z., Mathematical Theory of Computation, McGraw-Hill, 1974.

[N1] Newell, A., Perlis, A.J. and Simon, H.A., Computer Science, Science 157: 1373-74, 1967.

[R1] Ross, D.T., Plex 1: Semantics and the Need for Rigor, Softech Report, Dec. 1975.

[S1] Sammet, J., Programming Languages: History and Fundamentals, Prentice-Hall, 1968.

[S2] Shannon, C.E. and Weaver, W., The Mathematical Theory of Communication, University of Illinois Press, 1962.

[S3] Strassen, V. Gaussian Elimination is not Optimal, Numerische Mathematik, Vol. 13, 354-356, 1969.

[S4] Standish, T.A., Data Structures - an Axiomatic Approach, Computer Science Dept., Univ. of Calif., Irvine, May 1976.

[W1] Wegner, Peter, Three Computer Cultures, Computer Technology, Computer Mathematics and Computer Science, in Advances in Computers Vol. 10, Academic Press, 1971.

[W2] Wegner, Peter, Programming Languages, Information Structures and Machine Organization, McGraw-Hill, 1968.

[W3] Wegner, Peter, Abstraction - a Tool for the Management of Complexity? Proc. 4th Texas Conference on Computing, Nov. 1975.