DOCUMENT RESUME

ED 396 681 IR 017 857

AUTHOR Wolz, Ursula; Conjura, Edward
TITLE Abstraction to Implementation: A Two Stage Introduction to Computer Science.
PUB DATE 94
NOTE 7p.; In: Recreating the Revolution. Proceedings of the Annual National Educational Computing Conference (15th, Boston, Massachusetts, June 13-15, 1994); see IR 017 841.
PUB TYPE Guides - Classroom Use - Teaching Guides (For Teacher) (052) -- Speeches/Conference Papers (150)

EDRS PRICE MF01/PC01 Plus Postage.
DESCRIPTORS *Computer Science Education; *Course Content; *Curriculum Development; Higher Education; Instructional Effectiveness; *Programming; *Theory Practice Relationship; Undergraduate Students

ABSTRACT

A three-semester core curriculum for undergraduate computer science is proposed and described. Both functional and imperative programming styles are taught. The curriculum particularly addresses the problem of effectively presenting both abstraction and implementation. Two courses in the first semester emphasize abstraction. The next courses stress implementation as well as analysis. The third semester provides practical experience in programming. This curriculum has innovative aspects with respect to organization, methodology, and content. By integrating programming with mathematics in the first semester, abstractions and implementation become partners rather than competitors. In the later courses, implementation can be emphasized more naturally because of the early mastery of abstraction. The methodology emphasizes active learning via concrete laboratory experiences, and introduces multiple languages within the first year. (Contains 11 references.) (Author/BEW)

Paper (T4-201B)

Abstraction To Implementation: A Two Stage Introduction To Computer Science

Ursula Wolz
Department of Computer Science
Trenton State College, Hillwood Lakes, CN4700
Trenton, NJ 08650-4700
(609) 771-2241
wolz@trenton.edu

Edward Conjura
Department of Computer Science
Trenton State College, Hillwood Lakes, CN4700
Trenton, NJ 08650-4700
(609) 771-2766
conjura@trenton.edu

Key words: computer science curriculum, higher education, teaching of programming, integration of computer science and mathematics

Abstract

A novel three semester core curriculum for undergraduate Computer Science is described. It addresses the problem of effectively presenting both abstraction and implementation. Both functional and imperative programming styles are taught. Two courses in the first semester emphasize abstraction. The next course stresses implementation as well as analysis. The third semester provides practical experience in programming in the large.

Introduction

The last decade saw a major change in undergraduate foundations courses in Computer Science. Microcomputers and efficient Pascal compilers encouraged a standardization of introductory courses. Although other languages remained popular, Pascal became the standard as evidenced by the emphasis placed on its mastery in the Advanced Placement Exams and the plethora of Pascal textbooks. Yet many of the problems that the designers of Pascal attempted to address remain. Some students still fail to grasp underlying computer science concepts or develop strong programming skills (Koffman, Stemple, and Wardle, 1985).

In order to become proficient programmers, students must develop strong theoretical foundations as well as a firm grasp of how these concepts can be implemented on a modern computer. We claim that most current curricula muddle abstraction and implementation, diminishing students' ability to master either. Too often students learn an abstract concept as they learn to implement it in Pascal. Linked lists implemented with pointers is an example. Students fail to appreciate list structures because they never fully gain mastery of pointers. The approach described in this paper presents abstractions before implementations.

Two movements are currently encouraging the abandonment of Pascal as the language for introductory Computer Science (Decker & Hirshfeld, 1993; Roberts, 1993; Skubles & White, 1991). The "abstractionists" advocate functional (via Scheme) and object-oriented paradigms to stress concept over syntactic minutia. The "pragmatists" advocate C as a practical choice for "real world" programming. We take a middle ground in which both Scheme and C are introduced within the first year. We also expose students to object-oriented paradigms so that "programming in the large" can be addressed via C++ and CASE tools in the third semester.

Our goal is not only to make our students literate in two divergent paradigms. We explicitly focus on the dichotomy of abstraction and implementation. Scheme is used to introduce the "big ideas" of Computer Science in the first semester in "Computational Problem Solving" (CS1). This course is strongly articulated with a concurrent mathematics course, "Discrete Structures of Computer Science" (DS) that replaces the classic Discrete Mathematics course and also uses Scheme. The Scheme environment allows us to avoid the syntactic detail required in Pascal or C environments. The practical problem of implementation is addressed in the second semester in "Implementation and Analysis of Abstract Data Types" (CS2) ,when C and C++ are used to examine how abstractions are expressed efficiently. In the third semester "Programming in the Large" (CS3) provides experience in applying implementations of abstractions to large complex problems.

This curriculum has innovative aspects with respect to organization, methodology, and content. By integrating programming with mathematics in the first semester, abstractions and implementation become partners rather than competitors. In the later courses implementation and analysis can be emphasized more naturally because of the early mastery of abstractions. The methodology emphasizes active learning via concrete laboratory experiences, extending even to coverage of theoretical concepts in DS. Finally, the content is unusual because it includes the mathematical concepts underlying modern programming methodologies, introduces multiple languages within the first year, leads gracefully toward programming in the large, and presents the object-oriented paradigm throughout.

Each of the courses will be described in detail. A summary of our results to date is then provided.

Discrete Structures of Computer Science (DS)

The goal of DS is to present aspects of mathematics that relate to problem solving by computer so that students can apply them in diverse contexts. The content of DS is innovative because it places particular emphasis on those topics that relate closely to modern programming concepts, while the standard discrete mathematics course covers a more diffuse collection of topics. For example, we believe that lists, trees, and word algebras should be given the same early emphasis as sets. Laboratory work is included to foster exploration of concrete examples.

Several alternatives to the mainstream discrete mathematics course have recently been proposed. Henderson (1990) argues for a primary emphasis on problem solving, supported by appropriate laboratory experience. Maurer and Ralston (1991) minimize mathematical formalism and center on algorithms and problem solving via induction and recursion. Neff (1993) has supplemented the topics of the traditional course with laboratory experience using a complete logic programming system with Prolog syntax.

We attack the problem of non transferability of discrete mathematics through two mutually reinforcing, concurrent courses (DS and CS1) covering foundational topics in algorithm design, programming, and discrete structures, under the unifying principle of problem solving. We reinforce each important concept through appropriate laboratory experience. Freshmen are capable of understanding complex concepts and relationships, but often lack the motivation necessary for following formal textbook mathematics. The new DS course uses a strategy of involvement by example, via declarative programming.

Features of our new DS course include:

1. A small set of mathematical topics, closely related to the core of computer science covered in depth. Topics include: functions on lists, trees, and word algebras, the relational database model, polymorphic type inference, decompositions of digraphs, and formal languages. Each topic is presented as an application of general foundational principles, such as recursively defined sets and functions, structural and numerical induction, properties of relations, and informal predicate logic.

2. Concrete examples are explored both through paper and pencil exercises and laboratory work in Scheme and with specific software packages for abstract mathematics, such as ISETL. Articulation with CS1 is critical to insure that programming concepts are mastered before being applied to mathematical abstractions.

3. Work in logic programming utilizes a specially developed language known as Prologb (Neff, 1993). It implements a pure logic subset of the Prolog syntax, and, unlike Prolog, uses a breadth-first search strategy to guarantee solution for all valid queries, regardless of the ordering of clauses or goals.

Computational Problem Solving (CS1)

A first course in Computer Science should introduce the fundamental concepts of the field, and provide a strong foundation in the practical art of programming (ACM Curriculum Committee, 1991). There is conflict between teaching the abstract concepts and providing a practical experience upon which later courses rely. Our innovative approach is to introduce abstractions of all kinds (including OOP) in CS1 using a language such as Scheme that is suited to "hiding" implementation. The emphasis in our CS1 is not on functional programming per se. We exploit the high level nature of a functional language in order to avoid low level syntactic minutia.

The classic approach to introductory programming assumes that students must first learn the "primitive" constructs of a programming language before proceeding to the more abstract. For example, pointers are taught before lists. It is an historical artifact that data types such as lists and graphs are not considered primitive data types within a language like Pascal. Such limitations are creating an accelerating shift away from Pascal. For example, Roberts (1993) found Pascal inadequate as a base for teaching modern programming concepts, overly restrictive in design, and of limited usefulness in more advanced work, and opted instead for C. Skubles and White (1991) chose Smalltalk in order to stress modular design and reusability. Decker and Hirshfeld (1993) teach the object-oriented approach via Object Pascal.

We believe a first course must emphasize the fundamentals of programming, including iteration, recursion, decision making, data encapsulation, and process control and modularization, rather than the mastery of the details of a particular programming language. We emphasize the fundamentals through three means. First, CS1 itself provides an abstract conceptual framework through formal lecture and off-computer problem solving activities. Second, the course is articulated with DS so that the strong ties between the mathematical formalisms and the computational expression of those formalisms is

made clear to students. Finally, extensive experience in developing solutions to small programs is provided in both open and closed laboratory activities. Closed laboratory sessions provide invaluable guidance in important working strategies such as program organization, incremental testing, and use of the debugger.

Our current choice of language is Scheme. It allows us to dispense with much of the syntactic minutia associated with Pascal (or C) because of the absence of required data typing, the simple syntactic form, and the more natural expressibility of functions as individual entities via an interpreted environment.

A major theme is the practical illustration of top-down design via bottom-up implementation. The functional paradigm is used to stress the components of a computational process: input (via parameters), output (the returned value), and process (the body of the code.) Small programs (functions) can be designed, coded, tested and debugged as autonomous units. Once verified they can be combined into larger procedures, which themselves eventually form a fully working program. In essence, this is bottom-up implementation.

The stumbling block in many introductory computer science courses is convincing students that bottom-up implementation alone is not sufficient. Top-down design must precede it. The contribution of a discrete structures course is critical. By viewing a solution to a computational process as a static function, rather than an imperative process, students are encouraged to think of the composing function and its parts, regardless of how the details of the parts are constructed. We believe this will lead to a perspective in which top-down design is viewed as a natural evolution, rather than a form that is required by the instructor.

Features of CS1 include:

1. The course begins with the concept of a function. Both primitive and user defined functions are used to construct larger solutions to simple problems. Correctness is stressed by showing the correspondence between the informal denotational semantics of Scheme and the model of the problem.

2. Program control is introduced via recursion and conditional branching. The emphasis is on a strong association with a mathematical (and thus declarative) specification of a solution to a problem.

3. Data organization and typing are introduced via the use of nested list structures. Data types (integers, reals, Booleans, strings) as specializations of the concept of a symbol are presented. Data abstraction (including stacks, queues and trees) is shown to be an extension of nested list structures. Object-oriented ideas are shown by stressing operator definitions such as constructor and accessor functions and by showing that functions are first class objects.

4. The imperative framework is introduced. The iterative construct is presented together with an explanation of its correct use. Iteration is shown in relation to recursion; the recursion's precondition is precisely the loop invariant. Arrays (vectors) are presented within the context of efficiency of access. I/O is initially introduced via a discussion of the read/eval/print loop as a means of creating customized environments. More traditional perspectives on interfaces are then discussed and simple file processing is introduced.

Implementation and Analysis of Abstract Data Types (CS2)

The goal of CS2 is to provide insight into the implementation and efficiency of the structures and algorithms that were learned in CS1. A model of computation that is close to that of the actual hardware is required; hence an imperative framework is critical. The goal of CS2 is not to teach an imperative language, but to emphasize the analysis of the trade-offs between implementations. Within this context, the notion of efficiency, both of time and space, has special relevance. The major innovation of this course is that it is not simply an extension of CS1, but explicitly changes the focus from mastery of abstract concepts to analysis of implementation and efficiency. This approach is exactly the opposite of the traditional curriculum.

The traditional CS2 course is the place where the price is paid for any flaws in CS1. In the last decade, the curriculum has collapsed downward, and CS2 typically emphasizes intermediate programming, data structures and algorithm analysis, as well as software engineering. If the CS1 experience was in the "Pascal language features" form, students do not have requisite skills to master all of this new material. When the CS1 course excessively emphasizes low-level details, students lack experience in procedural organization of medium-sized programs. When discrete mathematics is not integrated with CS1, they do not have the requisite mathematical maturity to absorb algorithm analysis.

Many current CS2 books present data structures and algorithms in a cookbook fashion. When the "classic" algorithms and data structures are presented as complete Pascal code, there is little incentive for the instructor or student to go beyond the study of completed solutions. Such approaches de-emphasize trade-offs between solutions. Similarly, algorithm analysis is

often lost on students because they are shown techniques for polynomial time, but not logarithmic or exponential time, even though the trade-offs between the three is the real point of the discussion. The polynomial techniques are often learned by rote and consequently do not help develop intuitions about time complexity.

Our approach is to emphasize implementation and analysis over the simple mastery of data structures and classic algorithms. The difference is significant. In our CS1 and DS students will have been exposed to the "big ideas" of algorithm design and data structure choice. They should be comfortable users of abstract data structures. Since the concepts and mathematics have been covered, CS2 can efficiently focus on the imperative implementation of those concepts. Teaching the imperative framework is critical, not as a second approach to programming but as an introduction to the traditional processor. The emphasis requires exposure to an imperative language with object-oriented extensions. We have therefore chosen to introduce both C and C++.

Features of CS2 include:

1. Introduction to the features of an imperative language compared with those of a functional language (sequential nature of code, relationship between procedures, parameter passing, structure of a complete program).
2. Discussion of implementations of abstract data types in a strongly typed imperative language (e.g. stacks, queues, linked lists, trees) as both static and dynamic based implementations. The focus is on analysis of trade-offs: for example between static and dynamic implementations.
3. The object-oriented paradigm is introduced with an emphasis on implementation and efficiency both in reduction of coding time and transferability to other problems.

Programming in the Large (CS3)

The goal of CS3 is to provide students with real experience in solving large complex "real world" problems. The focus is the design, management, production, documentation and maintenance of large software systems. It strives to extend the notions of encapsulation, data and procedural abstraction, and inheritance by building on the students' prior experiences with the object-oriented paradigm. This course is innovative in its reliance on the previous mastery of abstract concepts, allowing management and implementation of a large problem solution to be emphasized without imposing huge time commitments from students.

The common requirements of ACM Curricula (1991) includes unit "SE2: The software development process", which covers life-cycle models, design objectives, documentation, configuration management and control, reliability issues, maintenance, specification tools, and implementation tools. Surprisingly, object-oriented technology is not listed among the topics within the ACM's topics for software engineering or advanced software engineering, although it is mentioned in some of the sample outlines for workshop courses.

Both the industrial and research communities recognize that OOP is an emerging technology (Osborne & Johnson, 1993; Skubles & White, 1991). The problems recognized in attempting to teach students the object-oriented paradigm via hybrid languages like C++ are great enough to require that their first exposure be through a high level language supporting full OOP. We believe that within a few years, experience with object-oriented technology as well as formal study of large-scale programming will be considered essential for computer science professionals.

CS3 builds on the early curriculum to produce an exceptional exposure to the areas of software engineering and the object-oriented paradigm. The course is taught on Sun workstations and employs the vast array of tools available in UNIX, such as SCCS, RSC for version control, XView for windowing, and SunWorks for software development. The object-oriented features of C++ are used extensively.

The course builds extensively on the experiences of the previous courses. In CS1 data and procedural abstraction, information hiding and inheritances will have been introduced. The object-oriented paradigm will have been continued throughout CS2 through Scheme, as well as C++. In CS3, Scheme will still be used to demonstrate and motivate and C++ will be the vehicle to achieve the results in a "real-world" language. The use of Scheme as a motivator/facilitator in CS3 is similar to that of the course described by Kay (1992). However, our approach is radically different in that we employ Scheme in the first course to implement concepts fundamental to computer science (where he uses Pascal). We move from the functional paradigm to the procedural one of C and C++ in CS2 (where he continues the procedural through C and introduces the functional through Scheme).

Another feature of this course is team projects. We address the pitfalls of team projects described by Pournaghshband (1990) by limiting team size, making the projects interesting and useful and requiring walkthroughs. CS3 also relies on a

required lab experience. The labs will be used for the demonstration and development of modules that will become component parts of a "large" software system. In some cases, needed components will be available in project libraries. Software engineering tools is demonstrated..

Features of CS3 include:

1. The course begins with a review and extension of prior OOP experiences through solution design using a true object-oriented variant of Scheme followed by re-implementation in C++. Comparisons between solutions are made.

2. Students are introduced to software engineering concepts through formal lectures and closed lab. A problem situation will be presented and student teams will design, code, test and document a large software system to completely or partially solve the problem. Topics include: software development life cycle, walkthroughs, software tools and environment, requirements specification, design—structure charts, testing, and maintenance.

3. The required laboratory component of the course will involve producing small modules that serve as building blocks for a large software system. This lab experience will also provide demonstration and training on software engineering tools.

4. Activities centered on code reuse and the advantages of OOP in this process will be stressed through lab activities and required in the design and implementation of each team's project.

Summary and Status

Versions of CS1, CS2 and DS that led to the curricular detail described here were offered between Summer 92 and Spring 93. In Fall 93 fully articulated CS1/DS courses were offered for computer science majors. In Spring 94 the new CS2 was offered for these students. CS3 will be offered in Fall 94.

To date, we have seen a marked improvement in performance among students in the new DS, CS1/CS2 sequence. They demonstrated the ability to articulate a problem solution both in English and in Scheme. Quick mastery of a second programming language occurred in CS2, but more significantly, students employed a programming style based on the procedural emphasis from their experience with Scheme. We believe this resulted from a firmer grasp of the abstractions, and because we could separate the abstractions from the implementation. Our results to date have been informal and anecdotal. Course notes and laboratory exercises have been prepared, and are being formally tested and evaluated during the 93-94 academic year.

We expect that the new courses will contribute to a redefinition of the objectives of the initial core courses in computing and supporting mathematics. In mathematics, we moved from a "grab bag" of isolated topics to a package of concepts particularly useful and concurrently applied to computer science. In computing, we moved from courses with syntax-centered emphasis to ones focused on the application of theory and high level computing concepts to problem solving.

Because of the superior theoretical background provided by DS, CS1, and CS2 we are able to more successfully meet objectives. The potential impact of the new DS, CS1, CS2 and CS3 sequence is to demonstrate a teaching methodology that provides students with strong conceptual foundations, while simultaneously providing practical experience in developing efficient implementations of those concepts.

Acknowledgments

This project is supported by NSF Grant DUE-9254108. Norman Neff, a project senior investigator was responsible for the DS course. Peter Henderson, Jim Dunne and Eugene Spafford provided valuable advice on course content and evaluation.

References

ACM Curriculum Committee on Computer Science (1991) Computing Curricula 1991, Report of the ACM/IEEE-CS Joint Curriculum Task Force, ACM Press.

Decker, R. & Hirshfeld, S. (1993) Top-Down Teaching: Object-Oriented Programming in CS 1. SIGCSE Bulletin, 25, (1) 1993.

Henderson, P. B.(1990) Discrete Mathematics as a Precursor to Programming, SIGCSE Bulletin 21(1).

Kay, D. (1992) A Balanced Approach to First-Year Computer Science. SIGCSE Bulletin, 24 (1).

Koffman, E. B., Stemple D., & . Wardle,C.E. (1985). Recommended Curriculum for CS2, 1984: A Report of the ACM Curriculum Committee Task Force for CS2, Communications of the ACM , 28 (8).

Maurer , S. B. & Ralston A. (1991) Discrete Algorithmic Mathematics, Reading MA: Addison-Wesley.

Neff, N. (1993) A Logic Programming Environment for Teaching Mathematical Concepts of Computer Science. SIGCSE Bulletin 25, (1).

Osborne, M.& Johnson, J. (1993) An Only Undergraduate Course in Object-Oriented Technology. SIGCSE Bulletin, 25, (1).

Pournaghshband, H. (1990) The Student's Problems in Courses with Team Projects, SIGCSE Bulletin, 22, (1),.

Roberts, E. (1993). Using C in CS 1: Evaluating the Stanford Experience. SIGCSE Bulletin, 25 (1).

Skubles, S. & White, P. (1991) Teaching Smalltalk as a First Programming Language. SIGCSE Bulletin, 23 (1).