

# Bandwagons Considered Harmful, or The Past as Prologue in Curriculum Change

David G. Kay  
Department of Information and Computer Science  
University of California  
Irvine, CA 92697-3425  
kay@uci.edu

**Key Words:** Curricular trends, first-year courses, curriculum design, academic decision-making

## Abstract

The field of computer science changes rapidly, and this change occurs as well in the introductory curriculum. Formerly advanced topics filter down to the first year, and even to secondary school; some topics disappear completely. These changes are good—they indicate a dynamic discipline and a still-emerging picture of the field's fundamental principles. But we must not let our revolutionary zeal blind us to the pedagogical need and conceptual value of time-tested material. Many topics and approaches that are well understood and now unfashionable should retain their place in the introductory curriculum, where they serve as intellectual ballast, foundation, and motivation for the more current and trendier content. We argue here for balance: that radical change be tempered by an appreciation for the place of long-standing approaches and underlying fundamentals. Those advocating curricular change must articulate their educational goals fully and consider explicitly what effect on those goals they expect the change to have; they must not throw the baby out with the bathwater.

## The Introductory Computer Science Curriculum Changes

The rapid development of computer science is by now a cliché. This rapidity occurs even in the introductory material; in what other discipline does the entire approach and content of the first course change so frequently? We decide that some topics are less important or unimportant, and new tools and technologies let us cover other topics in less time than in the past. This leaves room for more ad-

vanced material to filter down to introductory courses, though we might equally say that the advanced material often pushes its way earlier in the curriculum, sometimes faster than we can make room for it. The law of “Conservation of Curriculum” still applies: For any new material we add to the introductory curriculum, some old material—that once we thought important enough to cover—must give way.

Former fundamentals do fade: Few of us teach machine-level programming in introductory courses. Few spend weeks drilling on number-base conversions or teach circuit theory to novices. Yet once these topics were staples of introductory courses, regarded as fundamental prerequisites to the high-level language programming and algorithm analysis that most of us teach in the first course today.

We also decide to start our coverage at higher abstraction levels. We decide that some heretofore fundamental concepts have become underlying details. Historical perspective and maturing of the discipline contribute to these changes, refining our view of what is fundamental. We decide that binary search trees are fundamentally illustrative of certain concepts of algorithms and data organization, so they have made their way into the standard first-year curriculum, but B-trees and 2-3 trees typically remain in the realm of more advanced courses.

Better (or more accessible) textbooks allow us to cover more advanced material earlier: Many alternatives now exist to Knuth[1] for data structures, Jensen & Wirth[2] for Pascal, and even Abelson & Sussman[3] for programming, abstraction, and Scheme. These early texts are justifiably regarded as classics, but a proliferation of other books eases the migration of their topics ever earlier in the curriculum. Software tools as well promote this migration: “Student-oriented” compilers like WATFOR and PL/C, syntax-directed editors like parenthesis-matchers for Lisp, and program synthesizers like the CMU Pascal Genie all handle details and rough edges, allowing the instructor to devote more time to new substance and approaches.

As we add new material to our introductory courses, though, how carefully do we consider what concepts and skills we now must de-emphasize or omit? Changes in introductory curricula are made too often as a hasty attempt to jump on a current bandwagon, without thoughtful con-

sideration of pedagogical goals and the value of the topics being supplanted. The risk of short-changing meaningful deliberations may be greatest at research universities, where faculty often receive little reward or respect for time and attention paid to issues of introductory education.

## Introductory Curricula Today Are Diverse

Great diversity exists in today's introductory courses, on many different axes: emphasis on mathematical formality vs. production of actual programs, experience-oriented "closed" labs vs. design-oriented "open" labs, breadth of computer science coverage vs. depth of analysis (or of programming experience), analysis of existing "case studies" vs. synthesis of new programs, direct applicability in the real world vs. conceptual simplicity in the choice of programming language. This is all to the good; certainly we don't know any single best way to teach introductory computer science, and the exploration of new approaches is much to be encouraged.

But we must take great care that our enthusiasm for novel approaches not lead us to omit something vital. Going too far towards the end of any of these scales implies the omission of other topics and experiences, which may turn out to be equally fundamental.

## Curricular Debate Often Generates More Heat Than Light

An obstacle to reasoned curriculum development is the polarization that often arises between curricular innovators and traditionalists.

Innovators often meet resistance to change. Faced with this resistance, one tends at times to overstate one's case, exaggerating the harm of the status quo and demanding its complete reversal. The introduction of structured programming grew from a letter of Dijkstra's[4] into a raging controversy, with one camp regarding any use of a goto as a mortal sin and the other resentful of any attempt to rein in their creative freedom as programmers. Even Knuth's moderating voice six years later[5] did not mark the end of the controversy. We computer scientists are not immune to the human foible of regarding our own approach as the one best way and labeling as an idiot anyone who does not see things as we do.

A position stated in stark, extreme, revolutionary terms is more likely to generate attention and interest—necessary prerequisites to approval and funding—than one articulated with more balance or one that embodies an evolutionary approach. Sex, after all, sells, especially in a discipline where true order-of-magnitude changes do occur with frequent regularity. An example of this tendency towards exaggeration is Dijkstra's assertion that those who learn Basic as a first programming language are irreparably harmed as programmers. In fact, knowledge is dangerous only when it is incomplete or incorrectly applied. Many instructors assert that they would rather have a complete novice than one who has programmed in Basic, but if Basic

programmers were weaned onto a more powerful and conceptually stronger language after writing their first 25-line program, they would be ahead of complete novices in understanding interactivity, imperative programming and control flow, variables, the need for precise syntax, and so on. We do students a disservice by trying to protect them from "dangerous" knowledge, especially when they have already been exposed to it independently of us.

The pendulum of fashion swings as wildly in computer science education as elsewhere. Pattis[6], for example, describes how the "procedures early" approach gathered such momentum that many courses and texts eventually came to cover procedures much earlier than a pedagogical justification for them existed. He also describes the pendulum's reversal, which should provide some hope that equilibrium can eventually be attained, although the "procedures early" slogan still persists in textbooks' titles, prefaces, and advertising. As scientists and educators we owe it to our students to exert whatever moderation we can over our tendency to jump with both feet onto the newest, most attractive bandwagon.

As academic computer science matures, we must develop a respect for our pedagogical heritage and history, and not discard it willy-nilly as each attractive new concept comes along. Fewer and fewer academics' careers span the entire development of the field, so few of us have a complete personal perspective of how our discipline emerged. We have passed the point where we can build a new curriculum from the ground up every time a new approach warrants consideration.

## Innovations Have the Potential to Do More Harm Than Good

If we look at examples of four currently popular curricular trends, we see strong arguments in favor of each (which their proponents justifiably emphasize). But each also raises serious questions, which the trends' proponents seldom address. We argue here not against innovation (nor against *these* innovations), but in favor of a more careful, open, balanced debate, with less blatant advocacy and more reasoned discussion of the tradeoffs involved. Proponents of change must not fear honest dialogue. We are educators, and we must educate our colleagues rather than simply demanding that they share our vision on faith.

## Example—De-Emphasis of Programming

Introductory courses ten years ago focused largely on the construction of programs—sometimes just on coding, sometimes on larger design and abstraction issues, but always with a view towards writing complete programs to accomplish some task. Today the pendulum is swinging towards analysis and away from design. Current trends towards formality, case-study analysis, and structured, scheduled, analysis-oriented laboratory work reduce the amount of from-the-ground-up design an introductory student will carry out. Nobody should dispute the need for analytical skills and formal reasoning, and we should welcome a re-

treat from purely synthesis-oriented courses. Nevertheless, we must not deprive the students of some program design experience in the introductory courses, for many reasons.

First, design is an important part of the discipline; nobody would dispute that learning how to build software is a major part of an education in computer science. Although synthesis may have overshadowed analysis in the past, we must be careful not merely to reverse the imbalance.

Second, many of our introductory students come to us with many years' experience programming. We may find this experience to be haphazard and undisciplined, but that experience is probably what excited them about computer science and brought them to us in the first place. By ignoring the programming process, we risk alienating these highly motivated students, or giving them the impression that we have nothing to add to their existing programming knowledge (an attitude all too many of them come to us with already).

Third, the experience of trying to design something large without adequate complexity management tools is indispensable motivation for learning those tools. If computer science is largely about complexity management, people should learn by their own experience why those tools (be they good identifier names, source code indentation, modularity, data or procedural abstraction, or object-oriented programming) are essential. Programs of 40 lines (or even 240) fail to make the need for these techniques clear.

Fourth, we simply do not know how much coding experience is necessary as a background to understanding broader issues. Nearly every reader of this paper, we expect, learned coding before algorithm design or analysis, wrote imperative programs before functional ones and iterative programs before recursive ones, programmed in an Algol-like language (or Basic or assembler) before programming in functional or parallel or object-oriented languages. How much of that early experience provided us with motivation, fundamental grounding, and necessary ways of thinking that helped us learn and appreciate the later tools? Can we be certain that abandoning this history will allow us to teach as effectively? In particular, can students appreciate abstraction if they don't know what they're abstracting from? As C.A.R. Hoare put it, "You can't teach beginning programmers top-down design because they don't know which way is up." We know that a certain amount of active experience is necessary for learning complex technical details; none of us learns spreadsheets or word processors simply by reading the manual, nor do students write perfect first programs after reading the text and hearing our lectures. We don't know enough about the role of experience in learning computer science concepts to eliminate the programming portion of our courses entirely, no matter how sloppy, open-ended, demanding on the students, difficult to evaluate, and resource-consuming we may find them.

The issue of programming in introductory courses is but one illustration of the risks we run in our rush to innovate: We may excise too much, undercutting some fundamentals that we educators have internalized to the point that we re-

gard them as trivial. We disdain time spent on programming language syntax and we think that I/O details are uninteresting, but such "trivialities" still require non-zero time for the novice to assimilate.

### Example—Lisp as a First Language

Another trend with the potential of short-changing time-tested fundamentals (and a trend which the author has worked to advance—no reactionary Luddite he) is the teaching of Lisp dialects in introductory courses. A purely functional approach offers much in terms of elegance, provability, parallelizability, and mathematical analysis. But programmers do use straight-line imperative sequencing, for example in interactive data entry, and a failure to acknowledge this and exercise it misses a strong tie-in with practical reality. Students, after all, do not come to us pristine, unsullied, and devoid of real-world taint; confining them to the hothouse of pure functionality and interpreter-only interfaces fails to take advantage of their experience or allow them to make connections with the software they see (and have written) outside of school.

### Example—OOP as the Introductory Paradigm

Yet another trend that might displace important material is the introduction of object-oriented programming in introductory courses. The organizational benefits and practical applicability of the object-oriented approach are undisputed, but students must still understand input and output, variables and assignment, loops and conditionals, procedures and functions, arrays and records before they can write real object-oriented programs. What gives way to make time for classes and methods and inheritance? How do objects preclude the need to cover programming fundamentals?

### Example—C++ as a First Language

Even apart from its object orientation, C++ provides a standardized language available on all major platforms, with a well-developed mechanism for enforcing modularity and the ability to create "plug and play" exercises that allow students to program "real" things from the very beginning. C++ is also used widely in industry, which further motivates students to learn it. Yet its industrial-strength nature poses pedagogical problems. One of the lessons of the 1960s was that big isn't beautiful in programming languages. Languages like PL/I helped motivate Wirth to swing the pendulum towards simplicity with Pascal. This lesson is also one we learned in teaching computer science: We want to pare down the bells and whistles of the language we teach, so we can concentrate on the underlying concepts rather than distinguishing between two dozen ways of accomplishing the same task. Will the very richness of C++, including as it does much of the baroque syntax of C, land us back in the business of defining manageable pedagogical subsets, as PL/C and PL/zero were for PL/I?

Each of these trends has merit. But those promoting them must be prepared to consider and address the potential side effects and tradeoffs of their innovations. The reader is encouraged to think back on the curricular debates he or she has heard or read, whether on these four example issues or on others. Have the proponents tried to answer the hard questions, such as those raised above? Or has the discourse been primarily hand-waving advocacy (“This will work out great—you’ll see!”), better suited to the political stump or the revival tent than to scholarly deliberation?

## Pedagogical Goals

The key to implementing well-considered change is to articulate carefully one’s curricular goals and make a balanced analysis of how any proposed change might affect them.

Discussions such as the panel “Computer Science: The First Year Beyond Language Issues” at the 1996 SIGCSE Conference<sup>[7]</sup> contribute greatly to this deliberation.

Below we present one framework for describing these goals, but this framework should serve only as one possible starting point for discussion; no single listing can be definitive. The categories and criteria, not to mention the priority assigned to each, will of course vary from one institution to another, depending on size, faculty, student body, available resources, and so on.

We start with a broader context—the goals of a university computer science department:

- Do good research and publish it
- Get extramural funding for research programs
- Increase the department’s national reputation
- Provide high-quality graduate and undergraduate programs
- Fulfill obligations to the rest of the campus (e.g., courses for non-majors)
- Maintain and improve the working environment for faculty

Next we list some goals of a complete (four-year) undergraduate program:

- Provide well-trained graduates to industry
  - Technical knowledge and skills
  - Ability to adapt to changes
  - Ability to communicate and work with others
- Provide well-schooled candidates to graduate schools
- Produce well-educated citizen decision makers
- Attract good students to the campus and the major
- Increase the number of students served
- Operate within constraints on finances and faculty time

Finally we enumerate the goals of a first-year course for computer science majors. Students should:

- Be prepared for further course work
- Learn fundamental computer science principles
  - Algorithms and data structures
  - Analysis, design, and tradeoffs among them
  - Abstraction and information hiding
  - Exposure to the breadth of computer science as a discipline
- Know computing concepts at an introductory level
  - Data representation (bits and bytes)
  - Basic computer architecture: processors, storage, I/O, and so on
  - Characteristics of secondary storage and peripheral devices
  - Basic software engineering: the process of developing software
  - Ethical, legal, and social issues surrounding computing
- Know how to use computing tools, such as
  - Word processors and spreadsheets
  - Network tools (Email, news, WWW)
- Develop programming skills
  - Mechanics of entering and running code in some environment
  - Programming “in the small”—single routines
    - Algorithms: insert into a list, find the maximum, linear search, sorting, ...
    - Data structures: Array/table, queue, tree, ...
  - Abstraction and information hiding
    - Data and procedural abstraction
    - Distinguishing interface from implementation
    - Abstract data types/objects/classes
    - Combining routines, using predefined “API”s
  - Programming paradigms (especially OOP and functional)
  - Reading, understanding, debugging, and modifying code
  - Analyzing code’s correctness and performance; mathematical tools
  - Adaptability to new concepts, tools, skills, environments, paradigms, and languages
- Decide whether computer science is the field they want to pursue
- Receive placement or credit for prior academic computer science work (e.g., Advanced Placement Computer Science)

## Conclusions

As noted above, specific goals and priorities vary from institution to institution. The effect of any proposed change on each goal is likewise subject to debate. But it is exactly this sort of focused, reasoned deliberation that we should conduct when we consider curricular change.

We need to explore new pedagogical avenues, but we cannot abandon existing topics and approaches without thorough consideration of the intellectual and experiential underpinning they provide. We cannot allow ourselves to become so enamored of our pet approaches that we fail to acknowledge existing alternatives, both old and new. We cannot impose our own version of “pedagogical correctness” on our students. We owe them a balanced presentation in the introductory course, a solid foundation on which their further education can build.

## References

- [1] Knuth, D. E., *The Art of Computer Programming*, vol. 1, *Fundamental Algorithms* (Addison-Wesley 1968, second edition 1973).
- [2] Jensen, K. and N. Wirth, *Pascal User Manual and Report*, second edition (Springer Verlag, 1974).
- [3] Abelson, H., G. J. Sussman, with Julie Sussman, *Structure and Interpretation of Computer Programs* (MIT Press/McGraw-Hill 1985).
- [4] Dijkstra, E. W., “Go to statement considered harmful,” *Communications of the ACM*, vol. 11 no. 3 pg. 147 (March 1968).
- [5] Knuth, D. E., “Structured Programming with go to Statements,” *Computing Surveys* vol. 6 no. 4 pg. 261 (December 1974).
- [6] Pattis, R. E., “The ‘Procedures Early’ Approach in CS 1: A Heresy,” *SIGCSE Bulletin* vol. 25 no. 1, pg. 122 (March 1993).
- [7] Astrachan, O. L. (panel moderator), “Computer Science: The First Year Beyond Language Issues,” *SIGCSE Bulletin* vol. 28 no. 1, pg. 389 (March 1996); additional materials available at <http://www.cs.duke.edu/~ola/slides/lang96.html>