Technical Report
CMU/SEI-87-TR-44
ESD-TR-87-207

Report on the SEI Workshop on Ada in Freshman Courses

Gary Ford, Editor

December 1987

Technical Report

CMU/SEI-87-TR-44
ESD-TR-87-207
December 1987

Report on the SEI Workshop on Ada in Freshman Courses

Gary Ford, Editor

Undergraduate Software Engineering Education Project

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, Pennsylvania 15213

This technical report was prepared for the

SEI Joint Program Office
ESD/XRS
Hanscom AFB, MA 01731

The ideas and findings in this report should not be construed as an official DoD position. It is published in the interest of scientific and technical information exchange.

Review and Approval

This report has been reviewed and is approved for publication.

FOR THE COMMANDER

Karl H. Shingler SIGNATURE ON FILE
SEI Joint Program Office

This work is sponsored by the U.S. Department of Defense.

Report on the SEI Workshop on Ada in Freshman Courses

Abstract. The Undergraduate Software Engineering Education Project of the SEI Education Program sponsored a workshop on *Ada in Freshman Courses* in June 1987. The workshop brought together several educators to discuss how the software engineering content of beginning programming and data structures might be improved. This report describes the workshop and summarizes the discussions and conclusions, and it also includes the position papers prepared by the participants.

1. Workshop Background

The SEI Education Program derives its mission from this sentence in the SEI Charter: "[The SEI] shall also influence software engineering curricula development throughout the education community." Because a large percentage of the next generation of software engineers in the mission-critical computer resource (MCCR) community will have only undergraduate degrees, it is important that software engineering education at that level be improved. The Undergraduate Software Engineering Education Project comprises SEI education efforts in this area.

A common complaint from industry is that new employees with a bachelor's degree in computer science are ill-prepared for the responsibilities of professional software engineering. In particular, these new employees have too much of a *programming-in-the-small* view of software. Because many attitudes about programming are formed in the first programming course in college, it is important to examine these courses and attempt to redirect them somewhat in the direction of *programming-in-the-large*, including team programming.

The Ada programming language has been developed to support software engineering, particularly for embedded real-time systems. It has been suggested by many educators and practitioners that using Ada in the undergraduate curriculum would contribute to improving the situation.

To examine the issues related to the use of Ada in beginning programming courses, the SEI Education Program sponsored a workshop, the *Ada in Freshman Courses* workshop. This report describes that workshop, its conclusions and recommendations, and SEI efforts that have grown out of those recommendations.

1.1. Goals

The following goals were established for the workshop:

- Determine possible changes in content or course structure for the freshman courses that would enhance the development of the software engineering profession.

- Identify specific tasks for the SEI to support educators in pursuit of better software engineering education.

1.2. Selection of Participants

The success of the workshop depended greatly on the quality of the participants. Educators with a variety of experiences and expertise were needed, including:

- *Textbook Authors*: A good textbook is critical for the first programming courses. Most Ada textbooks were written for relatively experienced programmers, and were not suitable for first courses. It was believed that textbook authors would bring an important perspective to the workshop with respect to the difficulty in writing an beginning textbook using Ada. At least seven of the participants had written textbooks.
- *Innovative Educators*: Educators with a reputation for innovation were chosen because they could provide ideas for new approaches to programming courses that might be valuable when introducing Ada in these courses. Four of the participants were known for their curriculum innovations.
- *Contributors to ACM CS1 and CS2 Curriculum Recommendations*: The ACM has a great influence on undergraduate computer science education through its curriculum recommendations. Two of the participants were members of the ACM Curriculum Task Forces for CS1 and CS2.
- *Educators Who Had Used Ada*: There is no substitute for experience, so educators who had used Ada in the undergraduate curriculum were selected. They would be able to identify some of the problems encountered, and perhaps describe some of the solutions. Four participants had such experience.

1.3. Position Papers

To stimulate their thinking before the workshop, the participants were asked to write position papers on one of several issues related to the beginning courses. The issues and the papers submitted by the participants appear in Chapter 3 of this report.

2. Workshop Summary

The workshop was held at the SEI on June 22 and 23, 1987. The agenda was established as:

June 22, morning:	identification of issues for the freshman courses
June 22, afternoon:	discussion of those issues
June 23, morning:	specification of tasks that the SEI might perform to improve the teaching of these courses
June 23, afternoon:	summary

2.1. Workshop Discussions

To open the discussion on identification of issues, four questions were placed on the table at the beginning of the first session:

1. How do computer science and software engineering differ?
2. What should be taught in the freshman year?
3. What is the impact of the programming language used?
4. What are some possible alternative course structures?

The discussion was wide-ranging and did not actually give clear answers to any of these questions.

The differences between computer science and software engineering were not easily identified, but there was a belief that identifying them would not contribute much to the design of the early courses. At this level, students of either discipline need essentially the same material.

Topics and concepts that are now taught in the freshman courses were identified and generally supported, including modern programming techniques, data abstraction, and the beginnings of analysis of algorithms. (See the position paper of Norm Gibbs for a list of topics that were generally accepted by the participants.) Three additional topics were suggested: communicating what you have done, increased emphasis on verification and validation, and the process of incremental software development.

The discussion of programming language issues focused on the deficiencies of standard Pascal for teaching data abstraction. The importance of data abstraction requires that educators change, as soon as possible, to a language that supports this concept.

Alternative course structures that were discussed included what were called *read before write* and *programming by components*. (See the position papers by Lionel Deimel and Gary Ford, respectively, for descriptions of these course structures.) Because Dr. Deimel was unable to attend the workshop, there was no champion of the former structure, so its possible impact did not receive a full discussion. The latter structure was seen to have considerable merit, but it requires a substantial amount of support in the form of pre-existing software components of all kinds. It was suggested that the SEI might provide some of this support.

Subsequent discussions could be grouped into three areas: the relationship between software engineering and the first courses, the kinds of support materials needed for improved courses, and the role of Ada in these courses.

It was generally agreed that most of the problems of software engineering cannot really be appreciated by students at the beginning programming level. Therefore, instead of trying to teach major pieces of software engineering, it was suggested that the role of the first courses should be to *plant seeds* of software engineering knowledge, and to start students in a direction that could more easily lead to advanced study of software engineering. Among the seeds that were mentioned were:

- the engineering process (the idea of tradeoffs: arriving at solutions to problems through analysis of alternatives, measuring or estimating costs and benefits, etc.)
- that there are definable and repeatable processes by which one can do requirements analysis, design, coding, and testing
- that large systems are built from small components
- that a software system is only very rarely built by an individual person
- that software persists after it compiles successfully and runs on a single test case
- that code is not the only software entity that can be expressed formally

Support materials were recognized as being a major part of the effort to develop better freshman courses with more software engineering content. As mentioned above, a programming by components approach requires a body of components. This includes not only libraries of reusable code, but other kinds of software work products. Interesting programs with associated specifications, designs, test plans and suites, and other documents, could be used by instructors as the basis for student projects. The instructor could give the students most of the program, along with good specifications for the missing piece, including its interface to the given piece; the students would then build the missing piece. Several different projects could be created from one such program. For example, early in the course the students might be asked to supply only a very simple piece, while later they might be asked to supply a more complicated piece.

Case studies were proposed as another useful kind of support material. These would include various kinds of software work products, up to and including entire systems, that could be studied. Included should be guidelines for instructors and students about what to look for or look at in the code or documents. Good exercises that do not involve coding were recognized as being valuable, and such exercises might be based on case studies.

Programming environments also were suggested as a kind of support that would improve the freshman courses. Professional software engineers are moving toward sophisticated software engineering environments. Students who have experienced working with a substantial number of support tools will be able to function better in that professional environment.

However, there is another kind of programming environment for students. That kind is designed to support learning as well as programming, and such environments will have a variety of capabilities to show the student what is really happening inside a program. Prototypes for such environ-

ments already exist in a number of places, and it would be desirable for educators to have more exposure to them and their capabilities.

One other kind of support material was discussed: textbooks. The opinion was expressed that textbooks do not appear as fast as new ideas and methods of teaching beginning programming courses. Therefore it is desirable for there to be a more timely way of producing textbook-like materials to support, for example, programming by components or programming in Ada. Small technical reports on specific topics or methods, designed to be used by educators and students in conjunction with an existing textbook, were suggested as a possible solution to this problem.

Despite the title of the workshop, the discussion of Ada in the freshman courses did not occupy the majority of the time. This can probably be attributed to the fact that most of the participants had only a cursory knowledge of Ada's features and capabilities. One widely accepted suggestion was that the vast majority of instructors of freshman-level courses had even less knowledge of Ada, and that an effort by the SEI to raise that knowledge level would be beneficial.

The participants were unanimous that Pascal is insufficient for teaching data abstraction they way it should be taught. Modula-2 and Ada both provide substantially better capabilities, and there are now textbooks for data structures courses (CS2) using both of these languages. It was also generally accepted that it is undesirable to change the programming language from CS1 to CS2, so efforts should be made to influence educators to adopt one of these newer languages throughout the freshman course sequence.

Because of the lack of experience among the participants with either Modula-2 or Ada in freshman courses, there was no strong recommendation that either is better than the other. It was suggested that there is an advantage with using Ada because it has a standard definition, rather than Modula-2, because its inventor, Niklaus Wirth, continues to make changes in the language. Also, Ada provides some additional features that Modula-2 does not (such as exception handling and generics), although it was not clear how important these advanced features would be in freshman courses.

For a variety of reasons, it is unlikely that the educational community will adopt a single programming language for CS1 and CS2 anytime soon. Therefore it was recommended that the SEI concentrate on software engineering issues, and that it provide support materials in all three languages to allow the most widespread use of the materials. It was also suggested that the very limited number of low level Ada textbooks could be offset somewhat by providing Ada versions of the examples in existing textbooks that use other languages.

2.2. Conclusions and Recommendations

The workshop participants generally agreed on two major points:

- The freshman courses, assuming they are taught along the guidelines of ACM courses CS1 and CS2, do not need any *revolutionary* changes. They can be improved by a number of smaller changes, including changing the emphasis from building complete programs to building components, the inclusion of packaging concepts

(information hiding, modularity, separate compilation) earlier in the courses (which probably is a corollary of adopting the programming by components approach), and planting a number of seeds of software engineering that will have benefits in later courses.

- A language other than Pascal, probably Modula-2 or Ada, ought to be adopted for these courses. This will require a substantial support effort to raise the level of knowledge of instructors and to supply appropriate teaching materials.

The major recommendations to the SEI were the following:

1. The SEI should take the lead in promoting the kind of interaction among instructors of beginning courses that will lead to sharing of ideas and materials and to rapid improvement of the courses.
2. The SEI should become a clearinghouse for information about course structures, course content, textbooks, compilers, programming environments, and support materials and should disseminate this information widely.
3. The SEI should produce, both internally and with the help of faculty from affiliated colleges and universities, a variety of educational materials to support the improved teaching of the freshman courses. Foremost among these should be support for using the programming by components approach, support for using Ada, and for increasing the software engineering content of the courses.
4. The SEI should promote an increased knowledge of the capabilities and benefits of Ada as a beginning language, especially among instructors responsible for freshman courses.
5. The SEI should use its influence, where possible, to cause government agencies to provide more support for undergraduate education. A specific example is to influence the AJPO to address issues related to high quality affordable Ada compilers for the student environment.
6. The SEI should not promote an undergraduate curriculum in software engineering at this time, but instead should promote improvements in the undergraduate computer science curriculum. It should work with the ACM curriculum committees and other professional society curriculum efforts, rather than being seen as a competitor.

The project plan for the Undergraduate Software Engineering Education Project has been revised to reflect these recommendations. Activities have begun that implement some of these suggestions. For example, planning has begun for a workshop on data abstraction and object-oriented programming, including how Ada supports these concepts. The workshop will be held in February 1988 and will be jointly sponsored by the SEI and by the ACM Special Interest Group on Computer Science Education (SIGCSE).

3. Position Papers

The participants were asked to prepare position papers in advance of the workshop. They were asked to address one of the following issues:

1. To what extent should undergraduate education provide specific employment skills? Since most computer science majors are employed developing software, should software engineering have a more important role in undergraduate education. Should there be separate computer science and software engineering majors?
2. What are the concepts that must be taught in the freshman year? Are these different for computer science and software engineering? To what extent does the programming language used influence the concepts that are taught or how well they are taught? What are the relative merits of Pascal, Modula-2, and Ada in freshman courses?
3. Are there alternative course structures that would facilitate teaching better computer science or better software engineering in the freshman courses? Some examples of course structures are:
    - *Traditional*: programming-in-the-small; each program built from scratch, beginning with a "Hello, world" program
    - *Read before Write*: spend a substantial amount of time reading well-written, large programs; students modify these programs to change or enhance functionality rather than writing programs from scratch
    - *Programming by Components*: a substantial number of program components are available to students, and most programs are built by gluing those components together in appropriate ways
4. What materials, including textbooks, compilers, other software tools, and program components, are needed to teach effective freshman courses? What specific materials are needed to support some of the alternative course structures?

Position Paper

Lionel Deimel
Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213

Five years ago, David Moffat and I, in a paper called "A More Analytical Approach to Teaching the Introductory Programming Course"[1] suggested a course structure which deserves additional consideration in light of the increased interest in software engineering. The inspiration for this paper came to me one day as I was trying, as I often have, to decide why so many of my programming students did poorly. My conclusion at the time was that many students did not understand what the "game" was—they were being asked to write programs without understanding why and without understanding just what a program is. Instead of the traditional organization of the first course, we proposed a four-phase introduction in which the student would participate in the following activities in the order given:

1. Become a user of programs, preferably of many and varied programs.
2. Study programs and their algorithms. Reading and hand tracing are prominent here. The student learns about both algorithms and a particular programming language.
3. Test, debug, and modify existing programs.
4. Design and implement original programs.

Using programs was to provide motivation for the programming enterprise; the student could see why programs were useful and could begin to appreciate the role of user documentation and the significance of the user interface. Phase two was to demystify programs and to provide models for the student to emulate. Phase three was to provide a gentle introduction to program production and to emphasize the organic nature of software. The observation that some students are quickly overwhelmed by the programs they are asked to produce influenced our proposal here. In the final phase, of course, students do what they are "supposed" to do in a first course.

When we made our original proposal, I was most taken with the "read before write" aspect of it. Novelists, after all, read many novels before ever writing one. In retrospect, though, I think it is the way our organization leads the student to a more realistic view of the software life cycle which is the most important aspect of our proposal. I think the organization

- Highlights the fact that programs are not written for programmers.
- Forces the student to consider ideas easily overlooked in a traditional organization—the need for user documentation, the importance of the user interface, and the need for reasonable structure and comments in the code.

---

[1]Deimel, L. E. and Moffat, D. V. "A More Analytical Approach to Teaching the Introductory Programming Course." *Proc. NECC '82.* Columbia, Mo.: The Curators of the Univ. of Mo., 1982. The ideas in this paper were reformulated, expanded upon, and supplemented with anecdotal support in a later paper: Deimel, L. E., Hodges, L. F., and Moffat, D. V. "Restructuring the Introductory Programming Course. *AEDS Monitor 21*, 7-8 (Jan./Feb. 1983), 11-15.

- Encourages a more complete view of the software life cycle.
- Subordinates coding to a more realistic place in software development.
- Allows early on and in a very natural way for design, testing, documentation, debugging, and redesign (modification) exercises to be incorporated into teaching.

What would I advocate today, and how does Ada fit in? Certainly I would say certain things differently today, but I would not change the way the course begins and ends. There is no reason programming-in-the-small cannot be introduced in the context of programming-in-the-large. (The student can work on a "team," some of whose members are not present.) Using programs (an introduction we sometimes do quite well for "literacy" students) ought to come first and extensive code writing should come last. There is room for experimentation in the middle. Perhaps programming by components can be introduced early, for example.

As to the role of Ada, I do not think it has a proper role beyond that of any other programming language. Ada facilitates teaching certain concepts, of course, and makes some introductory approaches more attractive (programming by components, for example). The more important issue, however, is really how we introduce the student to the software development process generally, and this issue is not language specific. Further, the extensive use of program reading could make the use even of "messy" implementations of concepts acceptable so long as the concepts themselves can be communicated clearly. (There is even some advantage to using such implementations, though I would not want to press this point too far.) Information hiding can be disussed in the abstract and illustrated, even imperfectly, in, say, Pascal. It does not hurt to be using Ada, but to do so is not essential. The overall organization of the course and the pragmatics of the language vehicle used are really more important.

The question I would pose to the Workshop is this: What are we trying to accomplish in the first course and how should we go about doing it? It is, I submit, more important that the student come away with the right point of view than that he come away with the "right" bag of tricks in the "right" language. As for the notion that the needs of computer science and software engineering might be divergent in the first course, I think it can be dismissed quickly. Computer science has been and will continue often to be driven by practical considerations of developing real software. Even the computer scientist must know what computers are for.

Position on Alternative Course Structure

Michael Erlinger
Computer Science Department
Harvey Mudd College
Claremont, CA 91711

I would argue that the 4 issues expressed by Gary Ford are inter-related in such ways that it is difficult to present arguments that only address a particular issue. Thus, in the following position statement concerning alternative course structures I have included topics from the other issues. What follows immediately are expressions of my fundamental biases:

- Computer science and software engineering are really only different views of the same *discipline*. (I use the word discipline to try to avoid the confusion associated with the usual labels, i.e., computer science, software engineering, etc.) Both views must be presented in any major or course that purports to be concerned with the discipline. I think one of the fundamental problems faced in education and in industry is the failure to merge these concepts. I hear teachers saying "this is a small program, thus we can ignore any software engineering issues" or "this is such a theoretical approach that it has no applicability to practice." I hear design teams saying "we will just build the software system and then reverse engineer it."
- I believe software engineering must be taught in the introductory programming language course from the first day of class. Students must learn the interplay of software engineering, algorithm design, and programming.
- My final bias to be expressed here concerns the choice of programming language. In order to intergrate 4 years of varied courses and varied instructors, a *single programming language* should be used throughout the curriculum. (Such an idea seems to parallel the use of English in the humanities). I am not proposing that all other languages be omitted, but rather that a single programming language be the base language in all courses. I believe this provides at least the following benefits: the students know and understand a particular language in great detail; projects and examples can be easily integrated across courses; and with a fundamental knowledge in a single language, students can more easily understand other languages by comparing them against the base language. Also, in those courses where the base language is totally inappropriate, the students can learn why the base language is inappropriate. Thus a single programming language with capabilities in numerous areas is required in the curriculum.

I would approach (and plan to this fall) the freshman course with the following type of structure: a programming language with significant software engineering attributes and a single large example problem.

I believe Ada is the most appropriate language for the curriculum. It has the capabilities I require in the base language and I believe it is time (availability of compilers and tools) to install Ada as the base programming language. I will say no more about Ada as the base language as I am sure much will be said at the workshop, but I think it is necessary to expand on my approach to the course structure.

I foresee developing a problem that students are slightly familiar with, but not completely knowl-

edgeable of as the primary pedagogical example. My point here is that if students have a detailed understanding of the problem, they can be prejudiced in the solution. Also, a single large problem lets many aspects of the course develop simultaneously. The program design can be accomplished in whatever approach (top down, object oriented, etc.) appears proper. Also, the steps prior to design can be introduced, e.g., requirements analysis, user requirements, testing, and verification. Programming in the small can be used for the various modules of the solution. It is in this way that the various programming in the small features of the language can be introduced. The *read before write* approach can also be utilized because some modules can already be coded. Thus the students can read existing modules and discover how the modules they are building interface into the existing solution.

I think it is difficult in my approach to the introductory course to develop the large example problem. Besides the problem, parts of the solution must be established to show the interfacing aspects of software engineering and individual modules must be developed in an order that provides a method of teaching the programming in the small aspects of the language. I think such an approach is possible, but much work needs to be done in order for the approach to be practical.

Programming with Components

Gary Ford
Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213

A single change in the orientation of freshman courses might provide a large number of benefits. That change is to emphasize the construction of program *components* rather than main programs. Advantages of this approach include:

1. More interesting programs can be built earlier in the courses if the students supply some components and the instructor supplies the rest of the program. The instructor can assign to the students only those components that address the programming issues being taught at that time.
2. The emphasis on throw-away programs will be decreased. Students can learn to keep components and reuse them in later programs.
3. The students get a more realistic view of software development. It is rare in the professional world for a single person to write an entire program. Programming with components helps promote the ideas of a software team, with either student-student cooperation or student-instructor cooperation.
4. The importance of the specification of interfaces becomes more visible. When a single person is writing the code on both sides of an interface, it is possible to ignore the interface specification and tinker with the code on both sides to make it work. With components written by different persons, the issues related to interfaces can be seen and appreciated.
5. The concepts of abstract data types as program components can be taught more easily if the students are used to thinking in terms of components.
6. Students can be taught to work with higher level abstractions. For example, they can learn to *use* stacks and queues long before they learn to *implement* them.
7. The concepts of unit testing and building testbeds can be presented, rather than the monolithic testing approaches to entire programs.

All the topics in freshman courses can be taught with components as well as whole programs. Some, such as functional specification, interface specification, and verification, might be taught better at the component level than at the program level.

To adopt this approach to teaching requires a supply of components, main programs that use those components, and testbeds. The first two of these can also be used as class examples and programs for reading, assuming they are well written. Such libraries can be built in a rather short period of time if several educators share what they and their students develop. The SEI can contribute by soliciting, collecting, editing, documenting, and distributing these libraries.

The programming language used will have a significant impact on the kinds of components that can be built. Separate compilation is mandatory, so standard Pascal cannot be used. Pascal variants with separate compilation are available, but vary greatly in what they offer. Modula-2 permits the separate compilation of definition modules and implementation modules. These pro-

vide direct support for components such as abstract data types, and a somewhat clumsy support for components at the single procedure level. Ada compilation units include subprogram declarations, subprogram bodies, package declarations, package bodies, generic declarations, and generic instantiations. Nearly all of the kinds of components that might be taught can be structured in one of these ways.

Concepts for Computer Science and Software Engineering

J. D. Gannon
Computer Science Department
Institute for Advanced Computer Studies
University of Maryland
College Park, MD 20742

Introduction

The success and growth of any engineering discipline has never rested entirely on organization of trial-and-error knowledge. Application of deep theoretical results is also required to progress beyond the initial success that spreading common sense brings. The role of the engineer is sometimes to invent the required theory; more often it is only to apply an idea from a more abstract discipline to a problem the engineer understands. Furthermore, the application must meet a requirement peculiar to engineering: it must be in a form that can be used to solve practical problems.

We have developed a two-semester basic course for computer science much as calculus is a basic course for mathematics and the physical sciences, concerned primarily with methodology rather than subject matter. In fact we introduce a *program calculus* that deals with the functions computed by programs. Just as for ordinary calculus, there are two main problems in the program calculus. First, given a program, find its meaning (its derivative), and second, given a meaning, find a program with that meaning (its integral). This ability to derive functions from programs in the program calculus is of great value in computer science and engineering as well. First, it permits a mathematical treatment of program correctness, namely whether a program specifies correct behavior of the computer for every possible input. But even more importantly, it leads to a systematic design discipline for writing programs that are correct.

Programming Methods

Programming methods (i.e., stepwise refinement and data abstraction) are at the heart of any introductory course. Programming has two distinct phases:

1. *Design*—thinking out what the program should be in order to solve the problem.
2. *Development*—putting the program text in execution form.

In the stepwise refinement of a program, text designed to carry out a task in more detail is called a *design part*. A design part may itself contain more detailed task descriptions. The result of the design phase will be a hierarchy of design parts which collectively solve the problem at hand.

After the program has been entirely refined into a hierarchy of design parts, the translation into machine-readable form begins. A sequence of executable programs, each reflecting a larger part of the design, can facilitate orderly and systematic translation into a programming language. *Development programs* are accumulations of design parts, which grow in size until the entire design has been turned into a program. Each development program is defined so that it can be executed and tested to verify correct translation at each step of development. During program integration, a top-down, functional approach to testing integrates design parts into the development program.

Programming Languages

Programs are written in three increasingly complex subsets of the programming language Pascal. In the simplest subset, CF Pascal, there is but a single kind of data (characters) and a single data structure (files of characters that can only be accessed sequentially). Restricting our attention to so simple a language emphasizes program design rather than programming language features.

Small, but classical, problems lead to interesting program design problems in CF Pascal. Consider adding two hundred-digit numbers in different files and writing the result to a third file. The input files are read left to right, but digits must be added and carries computed from right to left. It is easy to see that the problem requires one pass over the two files for the add and carry logic but three file reverses. With an n^2 reverse, the solution will execute in $n + 3n^2$ time where it only takes n for what seemed the hard part. So reducing $n + 3n^2$ to $n + 3n \cdot ln(n)$ by finding an $n \cdot ln(n)$ reverse becomes an interesting problem. CF Pascal is an austere tool that requires a strong sense of abstraction.

The second language subset, D Pascal, permits the same functions to be created with smaller and simpler programs than is possible in CF Pascal. D Pascal also contains language features (type declarations and records) needed to implement data abstractions. Prior data abstractions become concrete language features in D Pascal. The final Pascal subset, O Pascal, introduces powerful control structures and data types to help optimize programs by providing random access to statements and data.

Mathematical Basis

The entire mathematical basis for the program calculus rests on just five discrete mathematical structures of character data: strings, lists, sets, relations, and functions. These five structures are not only sufficient to deal with program correctness and program design, but also admit treatment at various levels of formality with a mixture of English and mathematical notation.

Some sets are more easily and precisely described in English than in mathematics, but are sets no less because of the mode of their description. Many programming problems are better stated in English than mathematics, and we need to be able to treat questions of program correctness and design independent of the mode of description.

The mathematical property we study in programs is their effect on computer behavior. Understanding a program as a mathematical object is understanding the functional behavior it induces in a computer.

An execution state is a relation or function whose domain is the identifiers of a program and whose range is the values attached to those identifiers. The semantic meaning of a program will be a mathematical relation or function, a set of ordered pairs that defines a correspondence between one state (the inputs) and another state (the outputs). The meaning of a program will be taken to be the transformation implied by this correspondence; certain outputs are paired with certain inputs because given any such input, the program instructs the Pascal machine to compute that output.

Determining the Meaning of Program Parts

A *conditional assignment* summarizes the effects of several statements, mapping one state to another. For example, the meaning of the statement

```
BEGIN
   V1 := V2;
   V2 := V3;
   IF V1 < V2 THEN V3 := V1 ELSE V3 := V2
END
```

can be expressed as the conditional assignment:

(V2 < V3 → V1, V2, V3 := V2, V3, V2) | (V2 ≥ V3 → V1, V2 := V2, V3)

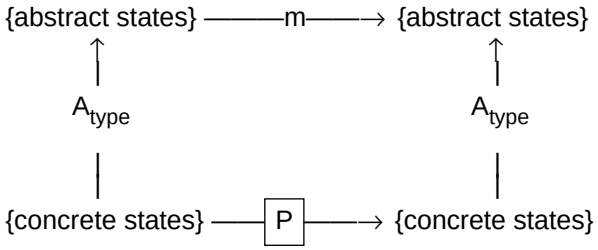
Symbolic execution is used to trace the values of variables through execution using only their names, not particular values.

Program Correctness

Given a program specification relation r and a program P, P is correct with respect to r if, for every member x of the domain of r (an instance of input data), P produces some member of the range of r which corresponds to x. That is, for each input x, P produces result y such that <x,y> ∈ r. What P does to input data not in the domain of r is not important since r should define all behavior important to the problem solver. Program P is correct with respect to specification function f if and only if f ∩ P = f.

Data Abstraction Correctness

The essence of data abstraction is captured by a diagram showing the relationship between the *concrete* world objects manipulated by Pascal procedures (e.g., P), and the *abstract* world objects the programmer manipulates with abstract operations (e.g., m) to achieve a solution. A *representation mapping*, denoted A_{type}, is defined between the values of concrete objects and the values of the corresponding abstract objects. By convention, for objects common to the concrete and abstract worlds, the representation mapping is identity. Then for any concrete state, the representation mapping can be extended to map the state to an abstract state.



Intuitively, an implementation is correct if its data objects are manipulated in such a way that the abstract objects to which they correspond, appear to be transformed according to the abstract operations. That is, correct implementation uses the concrete procedures and data, but in a way that mirrors the abstraction. To decide if this property holds, we show that the diagram commutes.

$$A_{type} \circ m \subseteq \boxed{P} \circ A_{type}$$

Of course abstract operations like m do not really exist except in users' minds. Pascal procedures implementing abstract operations are written with two sets of comments labelled "abs" and "con". The "abs" comments are added to modules so that users, those in the abstract world, need not examine the code (or even the "con" comments that document it). The "abs" comments replace the abstract operations in demonstrations that diagrams commute. If the implementation has been done properly, the abstract comment can be believed, and used in proofs at the abstract level.

Position Paper

Norman Gibbs
Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213

To what extent should undergraduate education provide specific employment skills?

Undergraduate students should be liberally educated with the goal of "training" them to learn on their own. I am not an advocate of professional undergraduate degrees and will encourage my children to study liberal arts first and postpone professional education until graduate school. This is why I have publicly opposed CSAB accreditation of undergraduate computer science degree programs. I believe Dartmouth College is correct in not awarding ABET accredited undergraduate engineering degrees, but treating engineering as a department with an "engineering major".

Being realistic, however, I know most universities are in the business of professional undergraduate training with success measured by the number of job offers their students receive and the magnitude of starting salaries. This being the case, it turns out that most computer science majors REALLY want to become professional programmers. That is why they "hold their breath and turn blue" over reasoning about programs, programming style, detailed design, documentation, complexity and formal methods while being willing to hack for hours in C or assembler language. Reality dictates that professional educators at the SEI need to attempt to influence most computer science departments to understand what it is they REALLY do and teach more software engineering. In particular, "throw-away" programs and independently graded exercises conflicts with what professional programmers or "software engineers" do in industry and government. I believe that many, if not most, undergraduate computer science programs really are oriented toward producing "pre-professional programmers" and as such are really not producing *computer scientists*. We will not see a distinction (in title) between undergraduate software engineering and computer science for at least a decade.

What are the concepts that must be taught in the first two courses in computing?

Absolutely essential are:

- algorithms
- stored program concept
- files
- programming in the small
    - correctness
    - preconditions
    - postconditions
    - invariants
    - functional and procedural abstraction

    - writing readable code
- complexity
- modularity
- information hiding
- data abstraction

The role of programming languages is that of representation. The danger with Pascal is that bad habits may have to be "unlearned" later. Pascal is based on Algol-60 ideas and codifies late 1950's and up to late 1960's thinking about programming languages. Modula-2 (3rd edition) includes ideas from the late 1960's and early 1970's while Ada captures some early to mid 1970's ideas. Universities should plan now to move away from Pascal. Unfortunately, I see a lot of the same reluctance to abandon Pascal that I saw in the 1970's about abandoning FORTRAN. Pascal is an inferior technology for representing programs that will be shared and developed by groups of people. It is not well suited for facilitating the management of the complexity of large programs for expressing abstract data types. It is not a matter of *whether* the academic community will abandon Pascal, but *when*!

In light of the above I believe that we ought to advocate more use of Modula-2 or Ada in the beginning courses. I think I slightly favor Ada over Modula-2 because I prefer packages over modules. I agree with Nico Habermann's statements about their being more room to grow within Ada than Modula-2.

Are there alternative course structures that would facilitate teaching better computer science or better software engineering in the first courses?

Obviously yes. There is less risk of failure when you adopt the "tried and true" methods and more potential risk with read before write and assembling programs from components. Of the more risky alternatives listed, I find programming by components more appealing. I suspect that hardware people would also strongly support such a notion. After all, electrical engineers do not build resistors or capacitors but use them to create more interesting devices. Teaching programming using conventional methods of explaining for statements and conditional statements is still at the "construct resistors out of carbon" stage and needs to evolve to the "integrated circuit" stage. Although we have experienced macro improvements in hardware technology we see only micro improvements in software technology. That is one reason why the SEI exists.

I have trouble with the read before write model in that humans begin to read at about age 5 or 6 and writing begins to "spiral" in a year or two later. Children do not write sonnets until very much later — drawing on all lots of previous experience. In programming we are in essence teaching reading and writing to adults with the constraint that the interesting part is the writing. Perhaps if all students did learn Logo first and computing was "spiraled" throughout a student's elementary education, we would have a chance of selling college students read before write. As it stands now I see little chance of this happening soon.

Position Paper

Elliot Koffman
Department of Computer and Information Sciences
Temple University
Philadelphia, PA 19122

Dr. Koffman's position on the freshman courses is reflected in the reports of the recent ACM Curriculum Committee Task Forces for CS1 and CS2, which he chaired. Those reports have been published in the Communications of the ACM.

Koffman, Elliot B., Philip L. Miller, and Caroline E. Wardle. "Recommended Curriculum For CS1, 1984". *Comm. ACM 27*, 10 (Oct. 1984), 998-1001.

Koffman, Elliot B., David Stemple, and Caroline E. Wardle. "Recommended Curriculum For CS2, 1984". *Comm. ACM 28*, 8 (Aug. 1985), 815-818.

Position Paper

Daniel McCracken
160 Cabrini Blvd.
New York, NY 10033

Let me address two issues: what should be done in the freshman year, and two interlocking problems in doing anything with Ada in education.

To respond to one of your questions, I don't think there is any difference at the freshman level between what a future computer scientist needs to study and what a future software engineer needs to study. There is a body of knowledge that, to my mind, is 99% common between the two. As a matter of fact, I will be interested to hear arguments to the contrary, as applied to the first three years.

Software engineering, as I understand the term, covers a very broad range of topics, only a small fraction of which can be dealt with until the student has assembled a significant part of his or her intellectual toolkit. Good coding style, sure, but that's hardly unique to a software engineering approach. Modularization and use of library modules/packages, sure, but people are doing that anyway, especially in the shift toward Modula-2.

Naturally, I assume that by freshman you mean essentially the first year of professional study. At some schools it would be impossible or difficult to take anything more than one introductory course in the freshman year; at a few, the entire freshman year is laid out, and consists of a common core with a liberal arts emphasis. Most of what I see around the horizon seems to be leaning in that direction, even in professional education; consider the recent reduction in CSAB computer science requirements, for example.

I'm not prepared to argue the relative merits of Ada vs. the others, and I'm reserving judgment on the best language for the first course, but I am really convinced that Modula-2 is a lot better for CS2 than Pascal. I approach this from the perspective of having just finished CS2 books based on both, as you know. I assume, from my limited command of Ada, that with suitable compilers and other tools, Ada would have the same (or similar) advantage.

And that is my final—and main—point. If Ada is going to get into the educational world, whether rapidly or slowly, and for whatever motivations, two things have to happen: there have to be textbooks, and there have to be student compilers. By the latter I mean simply a fast and cheap system with super diagnostics running on the most popular machines, backed up by easy-to-learn editors and helpful diagnostics. I'm thinking of things like WATFOR and Turbo Pascal, which revolutionized the way things were done in certain segments of the education sector, in a matter of a couple of years.

That's one absolutely crucial factor, and the other is texts. But, from my standpoint at least, there will not be texts until there are student compilers. For myself, I will not even think about doing anything in Ada until there is some convenient way to test the programs. I might not do it even then, mind you, but for now it is a total non-issue: I will not write a book with programs that have

not been run, to produce the output shown in the book. And if I had access to a mainframe system to do that, but the students in my market did not, it would be an economic disaster: the book could not sell.

I invoke the image (Isak Dineson, maybe?), of two locked caskets, each holding the key to the other. If SEI or anybody else wants to get Ada into education—freshman, graduate, continuing, or anything else—a creative solution to this impasse will have to be found. I have no idea what it might me. Free enterprise forces have done it with other languages in other years; as time passes and the same thing continues not to happen with Ada, I begin to wonder what's going on.

I look forward eagerly to a discussion of these and all the other issues at the workshop.

Position Paper

Terry Mellon
Computer Science Department
Arizona State University
Tempe, AZ 85287

I would like to address the question of *course structures*.

I feel that the programming assignments that are given in CS1 and CS2 are collectively the best vehicle we have for teaching the concepts that should be covered therein. Therefore, I feel we should attempt to produce some models and samples of sets of programming assignments.

Here is one such model for CS1:

1. An assignment to introduce the computing system, the editor the compiler, the printer, etc. No programming should be required — the student simply copies a sample program.
2. A "Hello, world" type program to introduce output formatting
3. A program to introduce expressions and assignments
4. A program to introduce subprograms
5. A program to introduce control structures
6. A program to introduce text files
7. A program to introduce arrays
8. A program to introduce records, pointers, and non-text files

I feel that assignment 2 should have the student modify an existing program instead of create a program from scratch. It is crucial that the student's first attempt at programming be based on an *outstanding* example thereof. This same technique should also be used at least for the assignment on subprograms. It likewise has application in CS2 where it can be used to facilitate meaningful, real-life assignments that would otherwise be too large.

I also believe that every assignment should specify exactly how the work will be evaluated. Criteria such as *readability, user interface, design, and correctness* should be defined, and the value of each should be stated.

We should also make some statement about the mechanics of evaluating and grading programs. Should one collect only hardcopies of the listing and output, or should one collect a machine-readable version of the source code, compile it, list it, and run it (i.e., actually test it)? I favor the latter, as I favor including the topic of "testing."

What about pseudocode? Should it be taught? Yes. Should guidelines or even a standard be established? Yes. Should it be required, collected and graded? No. It should be taught and used in lectures in such a manner that the students will use it as a "natural" part of the design of their programs. If we fail at this, requiring students to turn in pseudocode won't teach them how

to use it — they'll just write it *after* the code is finished. What about having them turn it in several days before the code is due (and before they could possible get the code finished)? I feel that this is also counterproductive because so much of beginning programming is trial-and-error coding. I much prefer to stress stepwise development, and to have them turn in at least one intermediate (working) phase for the larger assignments.

I feel that *separate compilation* (not just *include*s) should be taught as early as CS1. To allow students to use large source files (over 200 lines?), even if they are modular, is to teach a habit that will have to be broken later.

Position Paper

Philip Miller
Computer Science Department
Carnegie Mellon University
Pittsburgh, PA 15213

Employment Skills - I have never believed that undergraduate education should take as primary objective the preparation of programmers for today's industry. I believe that the university is the cutting edge, establishing what should exist in software engineering. It is the new knowledge, the new techniques, and the new way that should be taught in the university.

Concepts and Languages - I continue to believe that the concept of abstraction (control, procedural, and data) is the foundation of programming methods and therefore software engineering. The language has a huge impact on what is taught. This is true more in that the introduction of a language causes teachers to reconsider what they are teaching, rather than the intrinsic properties of the language.

Alternative Structures - I believe that Programming by Components is the best way to get students to understand procedural abstraction. I favor exploring courses that begin by having students glue together components. Read before Write (or more commonly, Case Studies) is a great way to teach the student about the decision points in program development. Mike Clancy articulates this position well. I believe that case studies can be smoothly integrated into programming by components courses. Finally I believe that programming in the small is an important component of a first course, but is the capstone, not the foundation. It can be achieved by having the student design and implement selected components, components which are not a composition of prepackaged tools.

Course Materials - I believe in programming environments for teaching the freshman courses. Our MacGNOME environment is an example of what I want to use. It is a structure environment, it integrates tools of editing, compiling and execution, and it provides alternative program views. I believe that such environments dramatically improve what can be taught. And I have spent a great deal of my professional life on creating them. I believe that data visualization will prove to be very useful. I believe that texts, electronically integrated with programming environments will be prove to be useful. I believe that such systems will be tough to build.

Position Paper

Richard Pattis
Computer Science Department
University of Washington
Seattle, WA 98195

Background

I've taught CS-1/CS-2 courses continually for the past 5 years; first in Pascal and now (and for the last 3 years) in Modula-2. I plan to switch both courses to Ada by the Fall of 1988, based on the belief that (1) students can adequately learn and use Modula-2, and (2) Ada is a better language in which to teach the fundamentals. The pedagogical jump from Pascal to Modula-2 was quite large; the jump from Modula-2 to Ada (both being based on Pascal, and its shortcomings) will be less difficult: more syntactic in nature, although I plan to review completely the global structure of my courses, adapting them to Ada's unique features and the experience I've gained while teaching Modula-2.

Main Topic: Alternative Course Structures

1) The traditional programming-in-the-small style is unacceptable in languages like Modula-2 or Ada. First, students can and should learn how to read and use simple packages (e.g. I/O and utility operations: strings, random numbers, timers, modular numbers, etc). Once students learn how to write their own subroutines, they can easily be taught how to write their own packages (statements are to subroutines as subroutines are to packages; well as fields are to records — and I teach records before subroutines). By instructing students in the use of smaller, simpler I/O packages (for pedagogic reasons), they can be guided to design and implement small but interesting improvements: such as the function GetInRange (<message>, <low>, <high>), which has a natural use in writing programs that select menu items (see the DNA program described below). Assignments can alternate between writing or augmenting utility packages and writing or augmenting applications (using packages previously written by the instructor or the students themselves).

A note on input output — I favor assignments where students write embedded systems that call package operations to communicate with the real world (sense and alter the real world). Under a simulator, sensing is just reading data and acting is just printing data (of course the actual package implementation is hidden from the student, just as any I/O implementation would be — see the cardiac monitor described below). Also, some state may be present in the implementation, to tie future sensing operations to the results of previous acts. A more advanced testbed would run a simulation, calling student functions to make crucial internal decisions.

2) The focus of the first three weeks of the programming classes that I teach is learning the syntax and semantics of types, expressions, and statements (including various functions and procedures imported from easy to motivate and understand packages). To me, knowing the "meaning" of a construct means that students can efficiently hand simulate code containing that construct. Only after students have mastered these basics, do I teach them the syntax that encloses statements to make complete programs (although I assign them to compile/link/run

prewritten programs earlier). Thus, the first part of my class is mostly analysis of code fragments, which shows the students many important programming idioms, with only a small amount of synthesis.

I assign no "Hello World" programs; I wait for students to learn enough to write genuinely interesting programs (to do otherwise fosters a poor view of the uses of computers). My first assignment is for students to write a program that controls an implanted cardiac monitor and automatic defibrillator (with distress diagnosed via a simple zero-crossing algorithm). Later, I alternate programming assignments between writing complete programs and augmenting prewritten programs or packages. For writing complete programs, I either show my students a stepwise enhancement path to the solution (this is what I teach instead of top-down programming — both are a form of stepwise refinement), or ask them to write a program whose overall control decomposition is trivial (as a first 1-dimensional array assignment, my students write a small DNA analysis program, a lot like a one-dimensional editor, that consists of nine operations selected from a menu; so students can immediately decompose the program into nine subroutine-sized units. Thus, I still focus on coding at this level, and not so much on design: students need to master their tools first, but they can accomplish this goal while studying and writing very interesting programs.

3) Most current introductory programming books concentrate on programs as the unit of discussion. This unit is too large and unwieldy for illustrating programming concepts economically. Instead, the subroutine should be the unit of discourse, but generality is often a problem in "standard" Pascal, where a lack of arbitrary-sized array types — especially for strings — reduces the generality of code; also, in Pascal there is no convenient place to put (and refer to) such code. Ada goes a long way towards overcoming both of these problems, via unconstrained types and packages. Together these features allow subroutines to be effectively used as the unit of discourse. The students can augment/create various packages with subroutines. Thus, the focus can take on a distinct package/tool orientation (although a course that is focused too closely on writing utility packages can become inbred; every so often the students should be required to write a complete application program, using previously constructed packages). There is a danger that should be avoided: instructors may be tempted to show too many package specifications before students understand basic variables and control structures; this approach, similar to the "procedures first" approach for which I have many criticisms, will confuse students and be ineffectual. Instructors need more balance.

I center my CS-2 course on packages: the students study the specification of some obvious packages (lexical, clock, queue, priority queue, table, stack, list, tree, graph, etc), and experiment with these packages via some driver packages. As students learn new data structures and programming techniques, they can re-implement these specifications according to various time and space efficiencies. My first assignment in CS-2 is for the students to write a cross reference program (about 2 pages of code), using previously studied lexical, table, queue, clock and string packages. A later assignment requires them to re-implement tables as trees, and rerun the cross reference generator.

Alternative Topics: 2 Cents Worth on Each

1) Both engineering and science freshman take identical calculus, physics, and chemistry courses; if these disciplines don't have a reason to bifurcate their courses this early, computer science shouldn't need to split into science and engineering tracks yet either. I view the right first computer science course as a programming course not a survey of the field (just as a first math course calculus not a survey of mathematics, and a first physics course is mechanics not a survey of physics). In each of these courses, students learn some theory and standard models (and see how they are composed). In a programming course (or any other course where "design" is important) they will also learn some pragmatics of design, but the focus is still on analysis; design is difficult until students master their tools and are able to concentrate soley on design. Our undergraduate curriculum is too short for specialization at this level; our majors alternate between theory and programming classes; those who wish can take more "programming-based" senior electives (like graphics, or our one software engineering course). But to squeeze in more software engineering courses would squeeze out important "theoretical" courses that will ultimately benefit students more. I think that a detailed study of software engineering is more appropriate at the graduate level.

2) I break CS-1 concepts into three categories: algorithmics, abstraction, and analysis. The first covers BNF, primitive types, expressions, and control structures. The second covers composite/advanced data types (records, arrays, unconstrained types), subroutines (function and procedures), packages (normal and generic — just another form of parameters) and protection (private types). Analysis covers correctness proofs, big-O notation, basic numerical analysis (mostly in the forms of problems and warnings), and computer numeracy (speeds, sizes, costs, etc. of computers and communication hardware). Employers don't need syntactitians; they need programmers who are trained in broad concepts, not a particular language or its features. Of course, the language used for instruction should include the largest possible set of coherent programming features. Beginners pay close attention to syntax; the more these features are reflected in a language's syntax, the better (for example, the separation of specification and implementation should appear in the language's syntax, as it does in Modula-2 and Ada).

4) I am writing a book. I would like to have a cheap, fast (meaning compiling and linking — I'd be glad to use a compiler that correctly generates straightforward but naive code) compiler that prints good error messages. I expect to design and code all kinds of packages that I will provide to students (to use, but as importantly to read; so they must be cleanly written, well integrated, and commented) as an integral part to my book. I don't want a subset compiler for education (I've heard rumors of these). It will, no doubt, not include the language features that I consider important to teach beginners; what are those features? I'm not sure yet, which is why it is premature to specify such an educational subset. I would like to see good debuggers with this software, and an Ada profiler that yields statement counts (not timings).

Position Paper

Stuart Reges
Computer Science Department
Stanford University
Stanford, CA 94305

Each school must make its own decision about how much to stress employability. At Stanford we care more about preparation for grad school, so we emphasize concepts over skills and breadth over depth. The principles of software engineering are taught to all CS majors and a significant group project is required in the senior year, but *the* software engineering course is one of a number of project options. It is probably not possible to provide sufficient coverage of either SE or CS to allow someone with a BS in SE to keep current. Thus, a major in SE would probably create more problems than solutions, giving BS/SE graduates and their employers a false sense of competence.

Two years ago the Stanford School of Engineering redesigned their undergraduate curricula and spent considerable time discussing the role of software engineering education. Engineering students establish engineering breadth by taking five courses in *engineering fundamentals.* This category includes basic electronics, thermodynamics, statics, engineering economy, and so on. Our introductory programming course was redesigned so that it would qualify. The most substantial change has been a shift away from programming skills and towards an understanding of larger issues. Concepts are emphasized in written assignments and are tested on exams and in oral grading sessions. Our first course no longer requires a gigantic project and instead takes time to more carefully introduce decomposition, program logic, testing, and reusability of code. Our second course now requires 200- rather than 500-line programs so that we have time to introduce verification, modules and opaque types. Because of our new emphasis, starting this year, students in mathematics and science can satisfy their *technology* requirement by taking our first course. Thus, software engineering also has a place in service courses for engineering, math and science students.

The concepts that should be stressed in the first year include the traditional topics of CS1/CS2 and those I have listed above. Beyond that, Stanford, like many other schools, has its own version of *computer science fundamentals.* Our two-quarter sequence first examines formal models and then implementation details. Thus, the first course is almost a course in paradigms of programming and the second is almost a traditional programming languages course. Students learn LISP and Prolog in the first half and UNIX/C and Smalltalk in the second. A parallel *fundamentals* course covers assembly language and introduces basic terminology for hardware/software systems.

The programming language used in the beginning courses can have a profound effect upon the education. For example, our Pascal system has an extension that allows us to create modules, but it is impossible to create truly opaque types. Many students never get the message and don't even know that they aren't getting the message because "it runs, doesn't it?" We will be switching our second course to another language next year and the only debate is whether to use

Modula or Ada. One side of the argument is that we should start our kids in Modula for the same reason that we put training wheels on their bikes, so that the set of possible mistakes they can make will be limited during their initial learning. The other side of the argument is that whatever pain might be experienced with Ada is outweighed by the payoff in the end, because they can do more with Ada and because the future of Modula is uncertain at best.

Our first-year courses can certainly be improved as we move towards Ada/Modula. I am somewhat skeptical of reading before writing, but I have not tried it myself. I have done some preliminary work in programming by components and I think there is a big payoff here for SE education.

We need some Ada and Modula textbooks with the same level of quality as the current Pascal books, but we don't need anything radically different. The new books need to adequately address issues like modules and opaque types, but will otherwise be almost the same.

Other than good textbooks, the only other prerequisite to moving towards a new course structure is some time off for instructors to prepare. Each school will want to approach software components in their own way, so it is unlikely that anyone outside the institution could help. For example, our second course is taken by industrial engineers who feed into a simulation course, so we will want to develop a module of useful simulation routines and other modules with commonly used data types. A well-orchestrated course can draw upon its students to perform much of the raw work. For example, we plan to give a "tools" assignment where students submit software tools for a class library. This library will be available for later programming assignments. The best support SEI could give to alternative course structures would be to bring together a number of educators for a significant period of time to rework their courses.

Position Paper

Frances Van Scoy
Department of Statistics and Computer Science
West Virginia University
Morgantown, WV 26506-6001

Question 1

To what extent should undergraduate education provide specific employment skills? Since most computer science majors are employed developing software, should software engineering have a more important role in undergraduate education? Should there be separate computer science and software engineering majors?

Background

West Virginia University is the comprehensive land grant university for the state of West Virginia. Additionally many of our undergraduate students are the first generation in their families to attend college. Parents of our students and tax payers see a major role for us being providing students with marketable skills. As a result, my department feels an obligation to provide specific employment skills.

On the other hand, computer science at WVU is in the College of Arts and Sciences. The requirements of the college include the usual liberal arts courses, including two years of a foreign language. The faculty in our department believe so strongly in the value of a liberal education for our students that last fall we argued against a proposed administrative reorganization that would have moved computer science to the College of Engineering.

The technical courses at WVU are in four main categories:

1. a 4 semester sequence of PL/I (two semesters), assembly language, and data structures
2. 1 year of calculus, 1 semester of statistics (with calculus prerequisite), 1 semester of discrete math, and 1 semester of numerical analysis
3. three required courses in compilers, operating systems, and data base systems (in each course a 4,000 SLOC term project is required)
4. three upper level computer science electives

We operate in the tension, then, of providing solid technical skills and also exposing students to the breadth of a liberal arts program.

The Issue of Employment Skills

A technology driven field has the need to give students current technical skills and also the skills (and mental attitudes) needed for learning new skills as the technology changes.

A similar problem is the extent to which different but relatively low level skills should be taught in a college program. For example, it is probably reasonable to expect sophomore computer sci-

ence majors to know two high level languages—say C and Ada or Pascal—reasonably well. However, if a department knows that likely employers of the students would like them to know Fortran, Cobol, or Modula 2, should the department respond by adding electives in those languages at the freshman or sophomore level? How should credit for those courses count towards the degree: as electives in the major (which might displace more advanced courses), as free electives towards the degree? If course credit for several introductory language courses is considered inappropriate, is it reasonable to expect students to learn a new language on their own and if they do so how can they document knowledge of a language to a prospective employer without a transcript entry?

At the very least, we need to teach students current best practice. Although most of us did not realize it at the time, our teaching Pascal to students in the late 1970's was good preparation for them to learn Ada on the job in the late 1980's. We need regular updating of undergraduate courses to reflect the best available technology.

The Role of Software Engineering

Could I single-handedly change the computer science curriculum at WVU I would leave the upper division program essentially unchanged but I would make major changes in the lower division program. I would convert the PL/I sequence of freshman year to an Ada sequence, move the data structures course from spring semester of sophomore year to fall semester of sophomore year (to be taken concurrently with the assembly language course or, possibly, in place of it), and add a new software engineering course.

I would change the emphasis of the data structures course to the writing of generic packages to implement abstract data types which would be used by the student in the advanced courses. As standardized generic packages become readily available, I would again change the emphasis of the data structures course to learning when to use which abstract data type.

The new software engineering course (at the second semester sophomore level) would be "Principles, Methods, and Tools of Software Engineering for Small Projects" and would be intended to bridge the gap between the 200 line programs of the current freshman/sophomore courses and the 4000 line programs of the three required project courses at the junior/senior level. I propose that we at the workshop compile a list of appropriate topics that should appear in such a course. The course I am proposing here would extract those aspects of software engineering best applying to 4000 line programs (1) to give the students useful tools for the required projects courses and (2) to give them an appreciation for the need for software engineering for constructing industrial sized software.

A Software Engineering Major?

Software engineering is a new field in the situation computer science was in 20 years ago—immature, with little theoretical basis, generally unrecognized as an independent discipline. An MSE program is very appropriate now as a professional degree, and there may well be cause for the establishment now of a few Ph.D. SE programs as well. Software engineering courses should begin to enter the computer science curriculum in forms such as the sophomore level course I'm proposing and the senior level projects course developed by Jim Tomayko, but the time is not yet right for a BSE major.

I see the migration path as being: a couple of SE survey courses in a BSCS program, some SE electives, an SE minor or elective in a BSCS program, and, ten years from now, a full BSSE program.

Question 2

What are the concepts that must be taught in the freshman year? Are these different for computer science and software engineering? To what extent does the programming language used influence the concepts that are taught or how well they are taught? What are the relative merits of Pascal, Modula-2, and Ada in freshman courses?

Concepts for Freshman

The freshman year is special. During the freshman year we need to lay the groundwork for future study in computer science and/or software engineering.

This spring at West Virginia University half of the students in my introductory computer science course (required of freshman majors but open to all) had no computer experience whatsoever. These students had no idea of what a computer can do or how to make a computer do anything, even run a word processing package or a game.

I believe strongly that students in a first semester computer science or software engineering course need hands-on experience writing computer programs. This gives them an appreciation for the ways a computer can be used.

Students at the novice level need to learn specific skills in three main areas: (1) algorithms, (2) language, and (3) mechanical.

At the first semester freshman level, students are generally not able to construct elaborate algorithms. They must be taught explicitly many basic algorithms for operations such as computing the sum, finding the maximum or minimum, or sorting a collection of numbers.

Regardless of the language studied, students must learn basic syntax and semantics of the language and how to map details of an algorithm into language constructs. In the first course they need to learn assignments, control statements, arrays, and subprograms. By the end of the second course they need to be able to use record types, pointer (access) types, user-defined types, and files.

The mechanical details required in a freshman course are perhaps the hardest to teach. Novices need very detailed (where is the power switch?) unambiguous instructions. It is hard to give explanations of why certain things are done without discussing topics not covered until later courses.

Students using a mainframe need to learn procedures for logging on and off while students using micros need to know about system diskettes. All need skill in using a particular editor and in manipulating files (copying, deleting, printing).

They generally need on-site consulting, especially in the early weeks of the first course.

Freshman in Computer Science and Software Engineering

At the present time I believe software engineering is a discipline best taught at the graduate level, so I see no need for different freshman courses for the two groups of students.

Impact of Programming Language

The choice of first language does have an impact on the first course.

PL/I and Fortran 77 lack user defined types. The concept of a package is such a vital part of Ada that packages must be taught early in an Ada course (I introduced packages in the first week of a freshman Ada course last semester.) but is absent from Pascal and PL/I.

Relative Merits of Pascal, Modula-2, and Ada

Pascal processors are inexpensive and readily available. Most colleges already have Pascal capability (compilers and experienced faculty). Many students can afford to buy Pascal compilers or interpreters for their personal systems. Pascal is a small language with few enough features that the language can be taught in its entirety in one semester. It cannot serve as a student's primary language through college and onto a job in part because of the lack of separately compiled units in the standard and the requirement that the size of an array is part of its type (which has severe impact on the ability to build libraries of subprograms).

I'm less familiar with Modula-2 but I've had trouble porting code between systems because of very different I/O modules in two different implementations. I was surprised to learn last month that an area employer is now seeking people with Modula-2 skills; he considers Ada too expensive (cost of compilers, cost of computer, cost of training) but Modula-2 to have most of the advantages of Ada.

Validated Ada compilers tend to be expensive and require at least AT class systems (although that situation is changing). Ada jobs tend to require more system resources than do Pascal jobs or even PL/I jobs. A college may be reluctant to make the substantial financial investment generally required for Ada systems or reluctant to allow unrestricted use of Ada for fear of degrading system performance. Ada is a large language which requires at least two semesters to cover. (I am now teaching a second Ada course which emphasizes concurrent programming.) Sophisticated and simple features of the language are tangled together; for example, it is difficult to teach simple text I/O without mentioning generic packages. However, once learned, Ada can be used very nicely in nearly all other courses in the curriculum and on the job. Ada supports software engineering principles and practices, and therefore careful teaching of Ada should build good attitudes and habits in students. I recommend Ada as the language to be used in freshman computer science and software engineering courses.

Question 3

Are there alternative course structures that would facilitate teaching better computer science or better software engineering in the freshman courses? Some examples of course structures are:

1. Traditional: programming-in-the-small; each program built from scratch, beginning with a "Hello, world" program

2. Read before Write: spend a substantial amount of time reading well-written, large programs; students modify these programs to change or to enhance functionality rather than writing programs from scratch
3. Programming by Components: a substantial number of program components are available to students, and most programs are built by gluing those components together in appropriate ways

An Alternative Course Structure

I would like to teach the freshman course as a true laboratory course. I envision a large pool of small sample programs illustrating language details and algorithms.

Students would have lab manuals containing directed exercises at several levels.

1. What does program xxx do?
2. What would happen if you made the following change to this program? Why?
3. Modify the program to do such and such.
4. Construct a new program to do such and such.

This way the students would be encouraged to become familiar with all language features presented in class not just those used in the handful of programming assignments in a typical course.

I tried to teach PL/I in this fashion (in 1982 or 1983) but had to abandon the approach due to limited computing resources. Currently a master's student is working with me to develop a laboratory manual for such an Ada course.

My original expectation was that this kind of course would be used for a course emphasizing programming-in-the-small. I believe the approach could be easily adapted, however, to a course emphasizing programming by components (I envision an exciting graphical interface in this case) or, perhaps less readily, to a read before write course.

Concerning Programming in the Small

In general I believe that programming in the small is an appropriate approach for the first course. In learning reading skills, five and six year olds first need to learn reading readiness (those marks on the paper mean something) while nine and ten year olds can read simple novels. Teenagers can begin to analyze and critique serious works of literature. Similarly although our goal is that eventually our students will be able to work on a team in building and maintaining large software projects we deal in the freshman course with total novices who need some specific skills they can take pride in and can build on in later courses.

In many ways I'd prefer that a beginning sophomore be rather skillful in constructing 200 line programs rather than explicitly aware of many software engineering principles but unable to construct a simple program from scratch (without a specific library of components). I like for a student to leave the freshman course with skills that can be used on many different systems, not just within that particular course environment.

Concerning Read Before Write

This spring I taught Ada to a class of beginners, half of whom had no previous computing experience. During the first two weeks I introduced basic Ada concepts. The third week I presented an overview of OOD and worked through a design for a solution to the formatter problem as described in one of the SEI course modules. For two weeks I worked our way through an implementation of the text formatter and then assigned an enhancement of the system (adding break, center, and space commands and commands to allow the user to set the size of the top and bottom page margins). This was the least successful unit in my course although I'm eager to try it again now that I have a better idea of some of the pitfalls.

Concerning Programming by Components

I believe that as software engineering becomes a mature discipline and is taught as a distinct undergraduate discipline this will be the approach of choice for the introductory software engineering course.

For the time being (while software engineering is not an undergraduate discipline and standard libraries of reusable components are not available) I hope that, at least at semester's end, students in such a course would be given source code for all the components they used in the course so they could construct programs similar to those they built during the course on their own once the course was over and they no longer had access to the course environment.

Appendix: List of Participants

Lionel Deimel
Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213

Michael Erlinger
Computer Science Department
Harvey Mudd College
Claremont, CA 91711

Gary Ford
Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213

J. D. Gannon
Computer Science Department
Institute for Advanced Computer Studies
University of Maryland
College Park, MD 20742

Norman Gibbs
Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213

Harvey Hallman
Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213

Elliot Koffman
Department of Computer and
Information Sciences
Temple University
Philadelphia, PA 19122

Daniel McCracken
160 Cabrini Blvd.
New York, NY 10033

Terry Mellon
Computer Science Department
Arizona State University
Tempe, AZ 85287

Philip Miller
Computer Science Department
Carnegie Mellon University
Pittsburgh, PA 15213

Richard Pattis
Computer Science Department
University of Washington
Seattle, WA 98195

Stuart Reges
Computer Science Department
Stanford University
Stanford, CA 94305

David Rudd
Computer Science Department
University of New Orleans
New Orleans, LA 70148

Frances Van Scoy
Department of Statistics and
Computer Science
West Virginia University
Morgantown, WV 26506-6001

Note: Lionel Deimel was unable to participate in the workshop, although he had submitted a position paper. Harvey Hallman was then asked to participate, but he did not have time to submit a position paper.

Table of Contents

1. Workshop Background **1**
1.1. Goals 1
1.2. Selection of Participants 2
1.3. Position Papers 2

2. Workshop Summary **3**
2.1. Workshop Discussions 3
2.2. Conclusions and Recommendations 5

3. Position Papers **7**
Position Paper 9
Lionel Deimel
Position on Alternative Course Structure 11
Michael Erlinger
Programming with Components 13
Gary Ford
Concepts for Computer Science and Software Engineering 15
J. D. Gannon
Position Paper 19
Norman Gibbs
Position Paper 21
Elliot Koffman
Position Paper 23
Daniel McCracken
Position Paper 25
Terry Mellon
Position Paper 27
Philip Miller
Position Paper 29
Richard Pattis
Position Paper 33
Stuart Reges
Position Paper 35
Frances Van Scoy

Appendix: List of Participants **41**