velopments is Ada, a general-purpose high-level language of the 1980's. Ada not only reflects evolutionary improvements in language concepts, but indeed embodies features to encourage modern design and modularization. Perhaps the Ada philosophy is more of an advance than the Ada language, for it is the philosophy of modularization, of abstract data types, of hierarchical structuring. Ada is over-rich, a natural result of the process by which requirements were laid on its design. That is not fatal, for subsetted working vocabularies can solve the learning problem, and hardware advances will give us the cheap MIPS to pay for the compiling costs. Advancing the structuring of software systems is indeed a very good use for the increased MIPS our dollars will buy. Operating systems, loudly decried in the 1960's for their memory and cycle costs, have proved to be an excellent form in which to use some of the MIPS and cheap memory bytes of the past hardware surge.

Nevertheless, Ada will not prove to be the silver bullet that slays the software productivity monster. It is, after all, just another high-level language, and the biggest payoff from such languages came from the first transition—the transition up from the accidental complexities of the machine into the more abstract statement of step-by-step solutions. Once those accidents have been removed, the remaining ones will be smaller, and the payoff from their removal will surely be less.

I predict that a decade from now, when the effectiveness of Ada is assessed, it will be seen to have made a substantial difference, but not because of any particular language feature, nor indeed because of all of them combined. Neither will the new Ada environments prove to be the cause of the improvements. Ada's greatest contribution will be that switching to it occasioned training programmers in modern software-design techniques.

Object-oriented programming. Many students of the art hold out more hope for object-oriented programming than for any of the other technical fads of the day.[2] I am among them. Mark Sherman of Dartmouth notes on CSnet News that one must be careful to distinguish two separate ideas that go under that name: *abstract data types* and *hierarchical types*. The concept of the abstract data type is that an object's type should be defined by a name, a set of proper values, and a set of proper operations rather than by its storage structure, which should be hidden. Examples are Ada packages (with private types) and Modula's modules.

Hierarchical types, such as Simula-67's classes, allow one to define general interfaces that can be further refined by providing subordinate types. The two concepts are orthogonal—one may have hierarchies without hiding and hiding without hierarchies. Both concepts represent real advances in the art of building software.

Each removes yet another accidental difficulty from the process, allowing the designer to express the essence of the design without having to express large amounts of syntactic material that add no information content. For both abstract types and hierarchical types, the result is to remove a higher-order kind of accidental difficulty and allow a higher-order expression of design.

Many students of the art hold out more hope for object-oriented programming than for other technical fads of the day.

Nevertheless, such advances can do no more than to remove all the accidental difficulties from the expression of the design. The complexity of the design itself is essential, and such attacks make no change whatever in that. An order-of-magnitude gain can be made by object-oriented programming only if the unnecessary type-specification underbrush still in our programming language is itself nine-tenths of the work involved in designing a program product. I doubt it.

Artificial intelligence. Many people expect advances in artificial intelligence to provide the revolutionary breakthrough that will give order-of-magnitude gains in software productivity and quality.[3] I do not. To see why, we must dissect what is meant by "artificial intelligence."

D.L. Parnas has clarified the terminological chaos[4]:

> Two quite different definitions of AI are in common use today. AI-1: The use of computers to solve problems that previously could only be solved by applying human intelligence. AI-2: The use of a specific set of programming techniques known as heuristic or rule-based programming. In this approach human experts are studied to determine what heuristics or rules of thumb they use in solving problems. . . . The program is designed to solve a problem the way that humans seem to solve it.
>
> The first definition has a sliding meaning. . . . Something can fit the definition of AI-1 today but, once we see how the program works and understand the problem, we will not think of it as AI any more. . . . Unfortunately I cannot identify a body of technology that is unique to this field. . . . Most of the work is problem-specific, and some abstraction or creativity is required to see how to transfer it.

I agree completely with this critique. The techniques used for speech recognition seem to have little in common with those used for image recognition, and both are different from those used in expert systems. I have a hard time seeing how image recognition, for example, will make any appreciable difference in programming practice. The same problem is true of speech recognition. The hard thing about building software is deciding what one wants to say, not saying it. No facilitation of expression can give more than marginal gains.

Expert-systems technology, AI-2, deserves a section of its own.

Expert systems. The most advanced part of the artificial intelligence art, and the most widely applied, is the technology for building expert systems. Many software scientists are hard at work applying this technology to the software-building environment.[3,5] What is the concept, and what are the prospects?

An *expert system* is a program that contains a generalized inference engine and a rule base, takes input data and assumptions, explores the inferences derivable from the rule base, yields conclusions and advice, and offers to explain its results by retracing its reasoning for the user. The inference engines typically can deal with fuzzy or probabilistic data and rules, in addition to purely deterministic logic.

Such systems offer some clear advantages over programmed algorithms designed for arriving at the same solutions to the same problems:

- Inference-engine technology is developed in an application-independent way, and then applied to many uses. One can justify much effort on the inference engines. Indeed, that technology is well advanced.
- The changeable parts of the application-peculiar materials are en-

coded in the rule base in a uniform fashion, and tools are provided for developing, changing, testing, and documenting the rule base. This regularizes much of the complexity of the application itself.

The power of such systems does not come from ever-fancier inference mechanisms, but rather from ever-richer knowledge bases that reflect the real world more accurately. I believe that the most important advance offered by the technology is the separation of the application complexity from the program itself.

How can this technology be applied to the software-engineering task? In many ways: Such systems can suggest interface rules, advise on testing strategies, remember bug-type frequencies, and offer optimization hints.

Consider an imaginary testing advisor, for example. In its most rudimentary form, the diagnostic expert system is very like a pilot's checklist, just enumerating suggestions as to possible causes of difficulty. As more and more system structure is embodied in the rule base, and as the rule base takes more sophisticated account of the trouble symptoms reported, the testing advisor becomes more and more particular in the hypotheses it generates and the tests it recommends. Such an expert system may depart most radically from the conventional ones in that its rule base should probably be hierarchically modularized in the same way the corresponding software product is, so that as the product is modularly modified, the diagnostic rule base can be modularly modified as well.

The work required to generate the diagnostic rules is work that would have to be done anyway in generating the set of test cases for the modules and for the system. If it is done in a suitably general manner, with both a uniform structure for rules and a good inference engine available, it may actually reduce the total labor of generating bring-up test cases, and help as well with lifelong maintenance and modification testing. In the same way, one can postulate other advisors, probably many and probably simple, for the other parts of the software-construction task.

Many difficulties stand in the way of the early realization of useful expert-system advisors to the program developer. A crucial part of our imaginary scenario is the development of easy ways to get from program-structure specification to the automatic or semiautomatic generation of diagnostic rules. Even more difficult and important is the twofold task of knowledge acquisition: finding articulate, self-analytical experts who know *why* they do things, and developing efficient techniques for extracting what they know and distilling it into rule bases. The essential prerequisite for building an expert system is to have an expert.

The most powerful contribution by expert systems will surely be to put at the service of the inexperienced programmer the experience and accumulated wisdom of the best programmers. This is no small contribution. The gap between the best software engineering practice and the average practice is very wide—perhaps wider than in any other engineering discipline. A tool that disseminates good practice would be important.

"Automatic" programming. For almost 40 years, people have been anticipating and writing about "automatic programming," or the generation of a program for solving a problem from a statement of the problem specifications. Some today write as if they expect this technology to provide the next breakthrough.[5]

Parnas[4] implies that the term is used for glamor, not for semantic content, asserting,

In short, automatic programming always has been a euphemism for programming with a higher-level language than was presently available to the programmer.

He argues, in essence, that in most cases it is the solution method, not the problem, whose specification has to be given.

One can find exceptions. The technique of building generators is very powerful, and it is routinely used to good advantage in programs for sorting. Some systems for integrating differential equations have also permitted direct specification of the problem, and the systems have assessed the parameters, chosen from a library of methods of solution, and generated the programs.

These applications have very favorable properties:

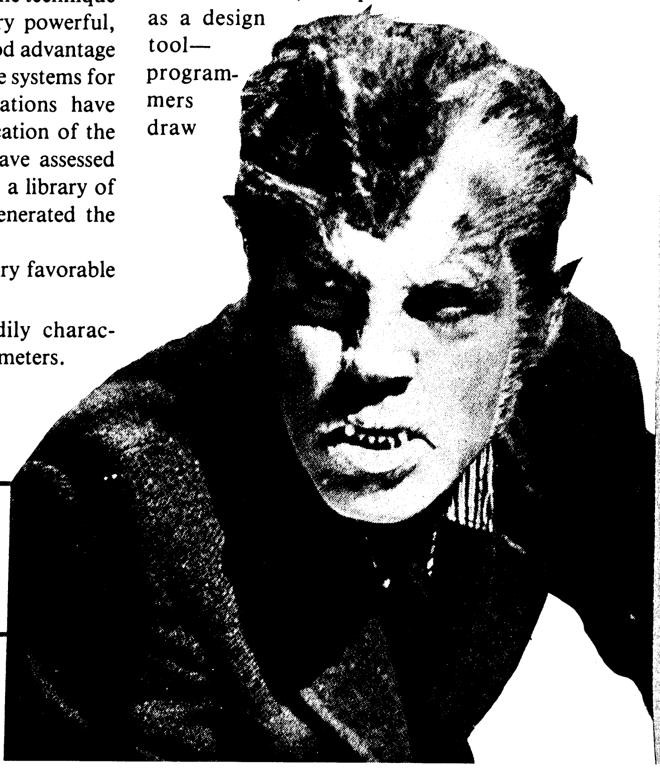
- The problems are readily characterized by relatively few parameters.
- There are many known methods of solution to provide a library of alternatives.
- Extensive analysis has led to explicit rules for selecting solution techniques, given problem parameters.

It is hard to see how such techniques generalize to the wider world of the ordinary software system, where cases with such neat properties are the exception. It is hard even to imagine how this breakthrough in generalization could occur.

Graphical programming. A favorite subject for PhD dissertations in software engineering is graphical, or visual, programming—the application of computer graphics to software design.[6,7] Sometimes the promise held out by such an approach is postulated by analogy with VLSI chip design, in which computer graphics plays so fruitful a role. Sometimes the theorist justifies the approach by considering flowcharts as the ideal program-design medium and by providing powerful facilities for constructing them.

Nothing even convincing, much less exciting, has yet emerged from such efforts. I am persuaded that nothing will.

In the first place, as I have argued elsewhere,[8] the flowchart is a very poor abstraction of software structure. Indeed, it is best viewed as Burks, von Neumann, and Goldstine's attempt to provide a desperately needed high-level control language for their proposed computer. In the pitiful, multipage, connection-boxed form to which the flowchart has today been elaborated, it has proved to be useless as a design tool—programmers draw


The Bettman Archive