The impact of Ada on software engineering

by KENNETH L. BOWLES

TeleSoft and University of California at San Diego
San Diego, California

ABSTRACT

The term software engineering has traditionally been applied to extremely diverse activities, ranging from system programming to managing programmer teams. Ada appears destined to become the first widely used programming language designed to bring these diverse activities together in ways supported by both programmers and managers. Among many important aspects of the Ada language, the most important appear to be (1) its orientation to system construction using interchangeable building-block packages, and (2) strong standardization in the interest of program portability. These aspects should foster the emergence of a new kind of software component industry. A probable result will be an inversion of the traditional view of software as an added value for use on major hardware products. Instead, major machine-independent software systems will emerge, and hardware will be increasingly regarded as an added value.
BACKGROUND

Software large enough to require more than a month or two of design and programming effort and/or the collaboration of two or more programmers tends to be one of the most complex engineering activities undertaken by human beings. Often the complexity is illusory and ill understood by the sponsors or managers of a project—not to mention the project's participants themselves. The problems that result typically include the following:

1. Logical errors and unreliable program execution. Because computer systems are being used increasingly to control equipment or situations on which human life depends, the control and prevention of these errors is assuming great importance.
2. Much higher costs to maintain a software system than to create the system in the first place.
3. Great difficulties in project cost estimation and frequently spectacular cost overruns.
4. Frequent duplication of development efforts to accomplish tasks that are nearly identical on different machines or in differing application settings.
5. Divergence of the program behavior actually achieved from the behavior intended by a project's sponsors.

Characteristically, efforts to solve Problem 1 by increasing the size of one's programming staff results in retrogression because of the rapid escalation of required communication among the staff members.

Engineering is the art of applying relevant scientific knowledge to the solution of real-world problems. In the case of software engineering, the relevant knowledge includes both program design techniques and methods for managing and enhancing the work of groups of cooperating people. The two fields are related because the available program design techniques determine much of the style of interaction among the members of a design team. The Ada language design is the result of a major effort to distill the best current knowledge about programming language design into a single language powerful enough for use in embedded system design. In the second field a large-scale effort is under way to create a comprehensive set of software tools—called an Ada Programming Support Environment, or APSE—to serve both programmers and their managers. Both design efforts were initiated in the hope of alleviating the problems listed above.

Neither Ada nor an APSE will provide automatic solutions to the list of problems cited above. It will still be possible for ill-disciplined programmers to write erroneous and unreliable programs, and in many cases it will still be very difficult for managers to control the errors or enhance the reliability of products developed under their care. On the other hand, the combination of Ada and a disciplined programming style will make it easier for programmers to write correct programs. A good APSE will both enhance the productivity of the disciplined programmers and support effective management decisions affecting those programmers.

The Ada design has been criticized as too large and also somewhat dangerous for use in design of programs for which execution errors can be extremely expensive. The charge of excessive size has been debated extensively in the design process, and the result is an engineering tradeoff that will be tested as the language comes into widespread use. Ada draws its design philosophy from Pascal and from a large number of more recent research languages derived from Pascal. Standard Pascal has been found too small for practical software engineering, and most practical implementations embody a diversity of extensions to the language, thereby making most implementations nonstandard. The Ada design can be viewed as an effort to collect the most commonly demanded Pascal extensions into a single internally consistent language for practical system programming.

Whereas extended Pascal is unlikely to be covered by a widely implemented standard for many years, Ada is backed by a very strong standardization effort. All conforming implementations of Ada are expected to support the same language, with neither subsets nor subsets considered conforming. This approach is intended to permit the reuse of even large and complex Ada programs on many different machines and Ada implementations. Effective program portability was a major goal in the Ada design effort. To this observer, it seems unlikely that true portability can be achieved in a language intended for a very wide range of practical applications without making that language somewhat larger than ideal for any single application area. Portability will be achieved at the expense of a number of generally minor inconsistencies in the Ada design. Acceptance of the remaining inconsistencies is a tradeoff between the time needed to create a fully consistent language and the need to begin using a better software engineering language in the near future.

The charge that Ada may be dangerous seems curious in view of the lack of any other widely accepted language designed to enforce program correctness. Ada permits handlers for run-time exceptions which, in Pascal, might cause abnormal full termination of a program. Much of the charge against Ada is based on the concept that a programmer should be forced to test explicitly for any and all potential abnormal conditions (at least if he/she wishes to avoid full program termination). It is true that sloppy use of Ada exception handlers could result in continued program execution while masking the existence of serious execution errors. On the other hand, disciplined use of exception handlers permits the main
(unexceptional) program flow to be more readable by eliminating the intrusion of a large number of special-condition tests. Careful testing of an Ada module should insure that all possible exception conditions are properly handled. The Ada design encourages enumeration of the various possible exception conditions in each exception handler. Concentration of the programmer's attention on exception conditions in one part of a module should be an aid to comprehensive checking for all possible errors, but it is not an automatic solution to the problem. One objection to ADA exception handlers is that their behavior is still ill-understood in the context of automated proofs of program correctness. It is debatable whether such proofs will soon be well understood in complex real-time applications, with or without exception handlers.

PROGRAM MODULARITY—PROGRAMMER INTERACTIONS
Perhaps the most important style issue in software engineering is the manner in which large programs are broken into modular pieces. Questions of modularity arise at several different levels of detail, such as the following, starting with the most detailed:

1. Isolation of small program control sequences in such a way that only one entry point and one exit point are used. This is one of the two main ideas on which the structured programming movement has been based.3
2. Isolation of related data objects in named data structures definable by the programmer. Again a part of structured programming.2
3. The recognition of distinct algorithms which can be isolated in their own subprograms. Reasons for using subprograms range from a desire to improve program clarity through the hiding of unnecessary details to an avoidance of duplicated program sequences.
4. The separation of major groupings of routines and data from other parts of a large program or system. Objectives range from the need to cope with limited main memory to a desire to subdivide the design work among several programmers.

The key idea running through all of these is the hiding of details except in localized areas where they can be handled in limited quantity. We human beings can concentrate on only a limited amount of detailed information at one time in our short-term memory. Other information, stored in our long-term memory, can be retrieved and actively used only at the expense of the displacement of other details from short-term memory. When concentrating on the overall structure of a complete system, or even on a subsystem, we must use abstract names and concepts to represent the details present at lower levels. When concentrating on a detailed level, we must put aside direct consideration of the overall structure.

The need for information hiding, and for isolation of modular groupings of details, arises not only because of the limits of human short-term memory, but also because of the high human communication overhead associated with the division of labor among several people working on a common project.
Whenever detailed information needs to be shared by two or more programmers, an effort is needed to insure that all participants have the same view of those details. Any change in the common information by one programmer must be reviewed with the others—often leading either to changes by several programmers to accommodate the revision or to arguments about the best way to confront the new situation.

The Pascal and C languages are probably the most widely used prototypical implementations of the first three program modularity concepts enumerated above. Both suffer in the fourth area, groupings of routines, because efficient communication among such groupings requires the use of shared data objects in common global areas of memory. At best, the shared data objects must be the subject of continuing communication among the several programmers on a team. At worst, the use of shared data objects is an invitation to programmer errors that result from an overload of detailed information, afflicting both individual programmers and groups of programmers. Insofar as it deals with the first three listed areas of program modularity, the Ada language is very similar to Pascal.

For some applications, the UNIX operating system provides an effective solution to the fourth modularity problem through the mechanism of pipes. The standard text stream output of one small UNIX program component can easily be connected via a pipe to the standard text stream input port of another program. From two to many such programs can be connected via a single pipe. This leads to a highly modular style of programming, described by Kernighan and Plauger.4 The UNIX pipe is a simple, easily described and understood abstraction for the interconnection of otherwise independent program components. All detail shared between these components is conveyed in the text stream passed via the pipe. To be useful, the logical structure of a text stream emitted by a program component must be well understood. The programmer writing a component designed to receive that text stream as input needs no other information about the emitting program. Indeed, many service program tools are designed to cope with text streams coming from a wide variety of sources, with no knowledge of details regarding those sources. The simplicity and generality of the pipe mechanism in UNIX permits the intermingling of program components written in C, Pascal, RATFOR (Rational Fortran, including various structured programming control constructs), and other languages. However, pipes are inefficient for many applications, and they often hide important information on data types that really should be communicated between modules.

Ada modules, i.e., both packages and tasks, are designed to meet needs associated with the fourth style issue listed at the beginning of this section. An Ada package is a collection of related subprograms, data objects, types, and constants, all of which can be separately compiled and stored in a library for later use. Most details regarding a package are hidden within the body part of the package and are not available for use or inspection by a using program. Only those carefully chosen details that the package writer wants used as an interface to the package are placed in the specification part of the package and thus made available to the using programs.

Many of the concepts that led to the design of Ada packages were drawn from research on abstract data types.5 For exam-
The idea of a software components industry, or marketplace, should lead to better programmer/manager interactions. Early experience in the use of Ada for design of large systems suggests that the necessary new style of programming demands substantially more advance planning before actual coding begins than does more traditional programming practice. The management school of software engineering has argued in favor of advance planning for some time. Early experience with Ada has made the benefits of careful design of module interfaces easily apparent to programmers. Though the effort to produce those interfaces in a consistent way is relatively large, the actual implementation of the underlying module bodies then turns out to be relatively simple. Moreover, the module interfaces are typically written at a level of detail easily understood by managers. As a result, the system granularity resulting from this style of Ada programming should lead to better programmer/manager interactions.

SOFTWARE COMPONENTS INDUSTRY

Most software products sold today are complete programs. The idea of a software components industry, or marketplace, in which building-block software components are sold independently, was suggested by M. D. McIlroy. Since that time a flourishing marketplace has developed in building-block system components on printed circuit boards for use with several popular interconnecting bus standards. McIlroy’s idea was that a similar marketplace should be available for the interchange of software system components roughly equal in complexity to the board components. Implementation of that idea in connection with uses of the UNIX operating system within the Bell Telephone System (see, for example, Kernighan and Plauger) appears to have been very successful. However, the interchange of UNIX program components among users in general has remained informal and largely noncommercial. As described earlier, interchangeable UNIX components are generally connected by pipes—a useful mechanism for transmitting streams of text between components, but of limited utility in many other applications. In spite of that limitation, the rapid acceptance of UNIX for the current generation of desktop work stations may well encourage the emergence of a commercial market in UNIX components.

In principle, Ada modules can be interconnected in a wide variety of ways, including pipes. In practice, a flourishing market in Ada program components will only grow if most implementers adhere to a small number of conventionalized interconnection designs. The software components industry will require common bus designs similar to the standard bus designs that support the printed-circuit-components industry associated with single-board computers (SBCs). A hardware system design based on SBC components represents a trade-off minimizing development time at the expense of optimized performance. Similarly, a software system composed of building-block components connected by a general-purpose interconnection design will usually be completed in much less time, but perform somewhat less efficiently, than a system constructed specifically for the application at hand.

Thus the economics surrounding the software components industry can be expected to be similar in this respect to that characterizing the SBC components industry. Frequently, the decision whether to use building-block components or a specific design will depend on the number of copies of the system projected for delivery to customers and on the time available for completion of the design. A small manufacturer of hardware systems generally chooses a specific design if the number of system copies is projected to be in the thousands. The higher investment made in the design, compared with the use of building-block components, is then more than offset by the larger spread between manufacturing costs and prices paid by customers. If the expected number of identical system copies is only in the hundreds, the design cost per copy is a relatively large part of the potential sales price, and a design using commercially available building-block components becomes preferable. This suggests that system integrators will be willing to pay per-copy royalties for the right to distribute hundreds of copies of building-block software components. For projected duplication in the thousands (or more), integrators either will prefer to buy full rights to the software components they use, or will choose instead to develop the equivalent software in house. If the current shortage of qualified system programmers persists, as projected, integrators are likely to buy rights to fully developed software components rather than choosing in-house development, except for small parts specifically related to the purpose of the final design.
Among the most important domains for interconnection of building-block software components will be that associated with user-defined data records. One of the strong principles behind the design of Ada (and before it, Pascal) is compile-time checking to insure that incompatible data objects cannot be intermingled without explicit conversion instructions by the programmer. To be generally useful, a library of modules designed for use with data records will have to cope with data records of many different user-defined types. The following list gives examples of frequently needed modules:

Sort and merge packages
B-tree record storage and retrieval packages
Index handler packages
Data display and data capture packages
Report writer programs

Ada offers an alternative to the common method of using program generators to provide library modules like these. A program generator maps a user's input of specifications into a complete program written in one of the widely used programming languages. The generated source program is then compiled normally, as if handwritten. This is a relatively clumsy method. Either a substantial repertoire of specialized program preparation tools is needed, or the user needs to master a substantial volume of detail to make use of a program generator.

In Ada, these modules will commonly be provided as generic library packages. The user-defined record type of information will be partially supplied as parameters passed to the generic packages when they are instantiated in the user's program. However, an Ada generic library package will not be given direct access to the individual data fields within a user-defined record. That access can be provided through a Record_Access package supplied by the user (i.e., by the programmer working as system integrator). Subprograms such as Store, Retrieve, and Compare will be provided by the Record_Access package interface for indirect access to the fields of a record. These subprograms will also have to be passed as generic parameters to each library package. Although this implies processing overhead for accessing each field, it is a method that allows hiding details on how the data fields are stored except with the Record_Access package itself. Indeed, several different mappings of the same data items within different record formats could be used with the same set of generic library packages without change.

It may be seen that specifications for the interface part of the Record_Access package will provide the common meeting ground for both writers of the general-purpose library packages and writers of specific Record_Access package bodies. These specifications will take the role of a software bus for applications in the record domain.

Easily understood interface conventions will also be needed in several other domains where building-block Ada components are likely to be used extensively. For example:

1. Message passing among the several layers of a handler for a communication protocol such as X.25 or Ethernet. Here the messages are likely to be passed as complete units rather than as character streams, as in a pipe.
2. Packing and unpacking of data objects in the fixed-size information “containers” of a paged virtual-memory management scheme.
3. Interface controllers for peripheral devices ranging from disks, printers, and CRT display terminals to laboratory instruments.

REFERENCES