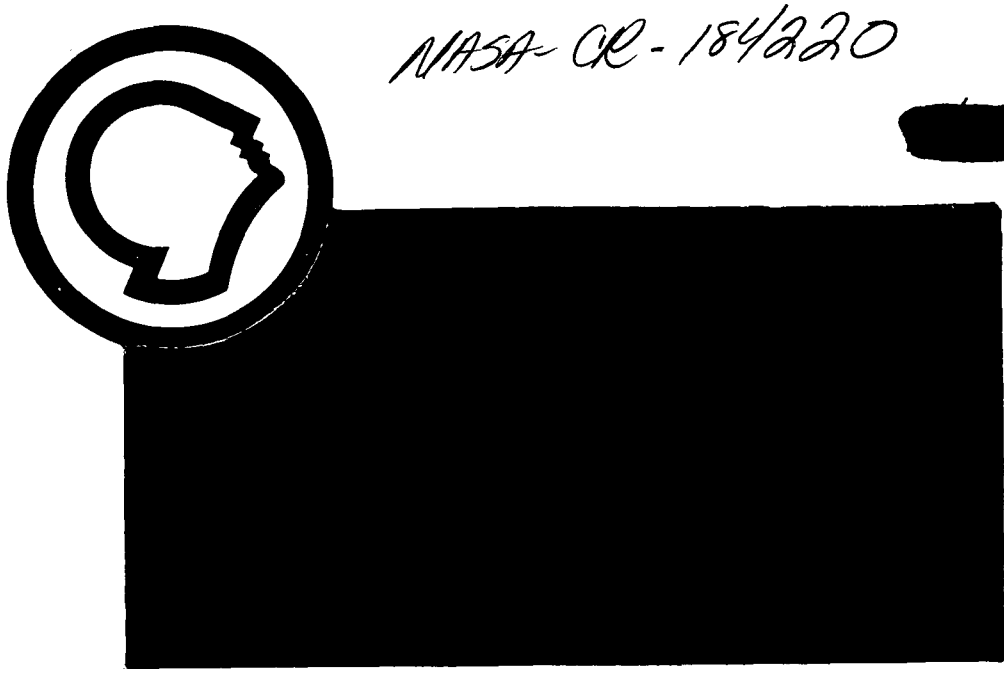
NAS8-36955
Marshall
[redacted]
P. - 105
NASA-CR-184220
[redacted]



The University of Alabama in Huntsville

Knowledge, Programming, & Programming Cultures: LISP, C, and Ada

JRC Research Report No. 90-04
February, 1990

Final Report D.O. 34
Prepared for:
Mr. Tim Crumbley
NASA Marshall Space Flight Center

By

Daniel Rochowiak
Research Scientist
Johnson Research Center
University of Alabama in Huntsville
Hutsville, AL 35899
(205) 895-6583
(205) 895-6672

PREFACE

This report contains the results of research conducted under NASA Contract NAS8-36955, D.O. 34, Task 2,"Ada as an implementation language for knowledge based system." The purpose of the research was to compare Ada to other programming languages. To this end the report focuses on the programming languages Ada, C, and LISP, the programming cultures that surround them, and the programming paradigms they support.

This report has been prepared for Mr. Tim Crumbley of Marshall Space Flight Center.

I wish to thank E. Howard Davis, Dion Boyett, and Ted Kerstetter who at one time or another worked on this project.

The papers based on this research contained in the Appendix provide an overview of the project.

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

If one wants to generate a debate at a party for persons connected with computer programming, just ask "What is the best programming language?" The result is often an outpouring of praise, curses, hyperbole, and technical detail that will either quicken the pulse or induce tranquil repose. Programming languages are at times treated as matters of religious fervor, and at other times treated as mere notational convention. All of this would be fine were it not for the demands for "good" software and the increasing size, complexity, and seriousness of software programming projects.

To be sure, software is more than the code for a program. Software, in the sense in which we will consider it, includes all of the information that is: (1) structured with logical and functional properties, (2) created and maintained in various representations during its life-cycle, and (3) tailored for machine processing. This information in large projects is often used, developed, and maintained by many different persons who may not overlap in their roles as users, developers, and maintainers. In order to develop software one must explicitly determine user needs and constraints, design the software in light of these, and, in light of the needs and constraints of the implementers and maintainers, implement and test the source code and provide supporting documentation.

The preceding dimensions and constraints on producing software can be looked at in two different ways. In one way, they can be thought of as, perhaps sequential, modules. In another way, they might be thought of as aspects of different moments in the software production process. We will adopt this latter, process-oriented, interpretation, and focus upon the programming moment. From this perspective we will examine the information that the programmers have and need, the languages that they use, and the paradigms they employ. Although somewhat indirectly we will examine their documentation needs from a management point of view. To this end we will not examine the technical details of the languages, their environments, and their tools, but will seek to illuminate the issues and debates that surround these.

Chapters 2, 3, and 4 examine the history and philosophy of three programming languages: LISP, C, and Ada. Each has been selected for reasons that are relatively external to their technical merits. LISP is the language of choice for projects in artificial intelligence or functional programming, while C and Ada are conventional languages that are less often used in AI research. C has a very large user base and is very portable; this is much less so for LISP and Ada. Ada is a highly structured language that has the support of the Department of Defense; this support is not so secure for C and LISP. Our discussions of these languages attempt to articulate their strengths and weaknesses.

Chapter 5 briefly examines the common programming paradigms. Although each of the languages is suitably universal so that each could use any of the paradigms, it is suggested that some of the paradigms are more readily supported in some of the languages that others. This is especially so for the object-oriented paradigm.

Chapter 6 briefly examines the idea of software engineering and suggests that special attention should be given to the the tools that are needed for "good" programming. Chapter 7 gives a brief account of why the idea of matrix management is relevant to the specific task of software management and indicates the effects that such practice might have on the more general economy.

The Chapter 8 brings together the disparate, yet related, elements of the previous sections in a set of tentative suggestions. The tentative nature of the selections reflects both the initial state of our research and the state of flux of the domain. We offer suggestions, rather than conclusions, because we believe that there is much work to be done and that the broad outline of that work is discernible.

The final chapter of this report argues for the central role of programming paradigms in understanding programming cultures and the desirability of the object-oriented paradigm. Further, it is suggested that the future evolution of the programming craft may make decisions about programming languages secondary, and questions about programming tools primary.

The Appendix contains papers that were presented at conferences based on the work in this report.

CHAPTER 2

LISP: HISTORY AND PHILOSOPHY

0. INTRODUCTION

The LISP programming language is used extensively in artificial intelligence research and development. Although LISP is one of the oldest high level computing languages, it has only recently come to be widely used. Critics of LISP raised many valuable objections, but the value of the LISP language in artificial intelligence work is undeniable.

1. HISTORICAL BACKGROUND

John McCarthy, who invented LISP, came to the idea of "an algebraic list-processing language" [McCarthy (1978), p. 174] as an outgrowth of the first formally organized conference on Artificial Intelligence (AI), the Dartmouth Summer Research of 1956.

McCarthy was trained in mathematics at Cal Tech and received his Ph D from Princeton in 1951. The conference resulted in an invitation for McCarthy to participate in IBM's New England Center for Computation at MIT and to work on an implementation of a language for AI on the IBM 704. The initial investigation was carried out in FORTRAN because of the ease of algebraic manipulation in that language and the readily available facility it offered for dealing with subexpressions.

McCarthy best describes his own language:

> As a programming language, LISP is characterized by the following ideas: computing symbolic expressions rather than numbers, representation of symbolic expressions and other information by list structure in the memory of a computer, representation of information in external media mostly by multilevel lists and sometimes by S-expressions, a small set of selector and constructor operations expressed as functions, composition of functions as a tool for forming more complex functions, the use of conditional expressions for getting branching into function definitions, the recursive use of conditional expressions as a sufficient tool for building computable functions, the use of [LAMBDA]-expressions for naming functions, the representation of LISP data, the conditional expression interpretation of Boolean connectives, the LISP function eval that serves both as a formal definition of the language and as an interpreter, and garbage

> collection as a means of handling the erasure problem. LISP statements are also used as a command language when LISP is used in a time-sharing environment.
> [McCarthy (1978), pp. 173-174]

LISP is a symbolic programming language, which is to say that it is a programming language that uses symbols (words, letters, numbers) for data (objects, actions and their relationships) to build up a domain of knowledge (or a "knowledge base") in a form that can be stored within a computer. Problems are solved in symbolic programming using inference:

> The inferencing program manipulates the symbolic information in the knowledge base through a process of search and pattern-matching. The inferencing program is provided with some initial inputs that provide sufficient information for the program to begin. Using these initial inputs, the inferencing program searches the knowledge base looking for matches. The search continues until a solution is found. The initial search may turn up a match that will, in turn, lead to another search and another match, and so on. The inferencing program performs a series of these searches that are chained together. This chain simulates a logical reasoning process.
> [Frenzel (1987), p. 10]

It is due to the structure of the IBM 704, and the problems encountered with creating list structures themselves, that the notions of LISP's first two primitives were born. The IBM 704 has a 36-bit word, divisible into two 15-bit parts and two 3-bit parts. The first 15-bit part was the address, hence, car ("Contents of the Address part of Register number"); and, the second was the decrement, hence, cdr ("Contents of the Decrement part of Register number"). Pointers, or more precisely, pointer data objects, are used to carry functions from one data location to the next. A solution statement is obtained when the nil pointer is achieved.

At MIT in 1958, McCarthy began acting as a consultant to Marvin Minsky on the development of a programming language for proving plane geometry theorems; and, he was carrying out his own work in AI on an Advice Taker system [a project that has never been completed]:

> ...[which] involved representing information about the world by sentences in a suitable formal language and reasoning program that would decide what to do by making logical inferences. Representing sentences by list structure seemed appropriate -- it still is -- and a list-processing language also seemed appropriate for programming the operations involved in deduction -- and still is.
> [McCarthy (1978), p.174]

At the invitation of the plane geometry team, McCarthy spent the summer of 1958 manipulating differential equations in FLPL. His work led to (1) an ability to create or

"define" recursive functions implementing conditional expressions, a recursion to which differentiation, by its very nature appeals, (2) the origination of mapcar which links the elements of a list to its functional arguments, thus mirroring an inherent list view of the world, (3) the key implementation of Church's [LAMBDA]-notations (1941) for functions, and (4) realization that a definite problem -- at that time without solution -- existed with regard to program-generated list structure.

The most important outcome for McCarthy's work through this period was the creation of a language with the unFORTRAN-like capabilities to perform both conditional expressions and recursive calls on subroutines. This arose in large measure from the unsuccessful attempts to run the differential equation program.

2. THE LISP LANGUAGE

We have noted that LISP is a symbolic programming language. Pratt carries us a step further by noting that pure LISP is also a universal programming language, where pure LISP is composed of only its basic primitives [CAR, CDR, CONS, EQ and ATOM], COND, recursion, functional linkages, atoms, and sublists -- but not number or property lists, and some other means for functions to be defined. By universal programming language, Pratt means:

> universal programming language is one in which any computable function can be expressed as a program. A function is computable if it can be expressed as a program in some programming language. This statement of the problem sounds a bit circular because we are up against a fundamental difficulty: how to define the concept of a computable function. To say a function is computable means, intuitively, that there is some procedure that can be followed to compute it, step-by-step, in such a way that the procedure always terminates. But to define the class of all computable functions, we must give a programming language or virtual computer that is universal -- i.e., in which any computable function can be expressed. But we do not know how to tell whether a language is universal; that is the problem.
>
> [Pratt (1984), pp. 350-351]

Several mathematicians in the 1930's invented simple abstract machines or automata that could serve to define the class of computable functions. The most widely known of these is the Turing Machine, named for its inventor, Alan Turing.

The study of these simple universal languages and machines leads to the conclusion that any programming language that might reasonably be used in practice is undoubtedly a universal language, if bounds on execution time and storage are ignored. Thus, the

programmer who steadfastly refuses to use any language other than his favorite, e.g., FORTRAN, because he "can do anything in it," is in fact correct. It may be difficult, but it can be done. The differences among programming languages are not only quantitative differences in what can be done, but qualitative differences in how elegantly, easily, and effectively things can be done.

The origins of LISP's power, its simplicity and its universality lay in the developments that arose when McCarthy founded the MIT Artificial Intelligence Project in the fall of 1958. The implementation of LISP began here, by hand-compiling functions in assembly language to create a LISP "environment".

Recursive function definitions were originally achieved by overriding "unwritten" restrictions in FORTRAN precluding recursion. De facto standard external notation handled symbolic information within parentheses, a form now known as "Cambridge Polish" after Lukasiewicz. The idea for garbage collection was accepted because of its intrinsic quality; implementation came later. And, finally, cons became a function of two arguments, removing the second 15-bit part from use and leaving only the 15-bit address as a single type, "so that the language didn't require declarations."

> One mathematical consideration that influenced LISP was to express programs as applicative expressions built up from variables and constants using functions. I [McCarthy] considered it very important to make these expressions obey the usual mathematical laws allowing replacement of expressions by expressions giving the same value [or, in other words, symbolic mathematics]. The motive was to allow proofs of properties of programs using ordinary mathematical methods... This is an additional vindication of the striving for mathematical neatness, because it is now easier to prove that pure LISP programs meet their specifications than it is for any other programming language in extensive use. (Fans of other programming languages are challenged to write a program to concatenate lists and prove that the operation is associative.)
>
> [McCarthy (1978), pp.178-179]

Eval[e,a], whose a-argument is necessary for recursion, required the "creation" of some symbols which would represent LISP data as LISP functions; LAMBDA received this role. S.R. Russell is given credit for establishing eval as the interpreter for LISP.

3. ISSUES

LISP 1 was thus finalized and reported upon by conference paper to the computer programming world in 1960. In 1962, the MIT Press issued LISP 1.5 Programmer's Manual by John McCarthy, with P.W. Abrahams, D. Edwards, T. Hart, and M. Levin.

The 1962 publication lasted well into the mid-1980's as a standard text on the LISP language. Pratt considers it to be "one of the few programming language manuals that provides a fairly clear description of the run- time structures on which the language implementation is built". [Pratt (1984), p. 523]

McCarthy offers two reasons for the continuing success of LISP as the preferred AI/symbolic programming language in the United States :

> 1. Recursive use of conditional expressions, representation of symbolic information externally by lists and internally by list structure, and representation of programs in the same way will probably have a very long life.
> 2. The convenience that LISP offers for operations in higher level systems work, symbolic computation and AI, is unequaled by any other language.

McCarthy cites run-time interface between the host machine and the operating system and stresses the list structure of the internal language and the strength of its capabilities in compiling from higher level languages or systems producing assembly and binary code. The final operations feature mentioned is the availability of the interpreter:

> LISP owes its survival to the fact that its programs are lists, which everyone, including me, has regarded as a disadvantage. Proposed replacements ... abandoned this feature in favor of an ALGOL-like syntax, leaving no target for higher level languages... LISP will become obsolete when someone makes a more comprehensive language that dominates LISP practically and also gives a clear mathematical semantics to a more comprehensive set of features.
>
> [McCarthy (1978), pp. 182-183]

LISP's role in AI is rather like FORTRAN's for scientific and mathematical computations and COBOL's for business applications. But, it is even more entrenched in its particular culture than these other two languages which have enjoyed similar longevity. This is to say that McCarthy is probably correct in his assessment of LISP's ability to endure. Ada and C are certainly no intellectual threat to LISP's role in AI, and even the most ardent proponents of producing AI-type machines with Ada and C interfaces are quick to admit that the kernel of these systems is "owned" by LISP. [Manuel (1986), p.59-65]

Notwithstanding this, LISP has had some major defects over its relatively long lifetime that have been resolved only in the last two years. The major breakthrough in LISP programming is the advent of a LISP compiler within the market range of many smaller and newer businesses. This mechanism, born thirty years after the language itself and after ten years of intensive research at MIT and Stanford, has finally removed legitimate complaints about the amounts of time, money and memory that LISP once absorbed. [Manuel (1987),

p. 110] There remains the criticisms that LISP outside of a LISP environment is not very fast and absorbs vast quantities of memory.

Merrill Cornish, who is cited by Winston and Horn, tells a story that is worth repeating:

> Although LISP... is the second oldest computer language (FORTRAN is first; COBOL is third), it only recently entered the public spotlight, replete with rumors. When I first began working on Texas Instruments, Inc.'s Explorer LISP machine project, I was openly incredulous about the wild claims that my friends were making about LISP. I had programmed in assembler, FORTRAN, Pascal and C, plus I had a passing acquaintance with Ada, Cobol and PL/I. This background told me that no matter how well-suited a language might be for one purpose, no single language could deserve the praise heaped on LISP. As I worked with LISP (and with LISP-ish people), I began to notice that the claims were true. But few actually concerned the LISP language. Instead, people were talking about LISP in its LISP machine environment. Since these particular people had little contact with LISP outside its unique environment, there was no compelling reason for them to separate the two ideas in their everyday conversation. I have since devised a... test to apply to wild-eyed nouveau "LISPers" to see if they are talking about LISP the language or LISP the environment. Ask them if they would be willing to program LISP in batch mode. If they stare, stammer and sputter at the very idea, you know you're talking to a LISP environmentalist.
>
> [Cornish (1987), p. 75]

This story represents more than any set of arguments the sort of personal experience with LISP that a multilingual computer programmer might have. It encapsulates the one or two oft-repeated reasons that will keep LISP in the forefront of AI research in the United States. The LISP system of language and machine embodies the meaning of the word "integrated", according to Cornish.

None of this is to say that LISP functions paradigmatically outside of its environment: the cold hard fact is that is doesn't. LISP is often inefficient on conventional machines. LISP, however, offers several advantages when implemented within the framework of its own "world" that make it a force to be reckoned with by serious student of computer science. While the LISP machine was literally designed as a personal computer, it can transparently access a local-area network. The tremendous amount of storage that is now afforded by the LISP machine allows users to encapsulate both system source and their own source codes on-line throughout use. Historical reference files are automatically created for fail-safe code changes, so that original programs can be retrieved. Thus a synergy between disk space and a massive amount of virtual memory is achieved. Cornish notes:

> The LISP world's approach to programmer productivity is totally at odds with the classic notion that says minicomputer users should conserve memory and disk space and then try to regain productivity through things like structured programming disciplines.
>
> [Cornish (1987), p.76]

This is a key element in asserting the power and simplicity of LISP.

Further, language issues and comparisons are almost obviated by virtue of the integrative arguments in favor of the LISP environment. Manipulation of the LISP machine allows flexible interaction patterns during "binding" (i.e., freeze) time which can speed or slow processing of data through the interpreter and compiler. This is achieved because the machine does not differentiate between interpreted and compiler code. The implementation uses an artificial lookup table for all function names; the lookup step -- which would be an astronomical addition in cost in a conventional environment -- is inherent in the LISP machine and achieves optimum speeds in debugging, reconstructing or altering programs.

What essentially is happening is that for the first time symbolic tasks are being carried out on equipment that is designed to handle symbolic processing. There is a vast difference in this approach to one in which naturally "symbolic applications" are being attempted with "inherently numeric tools".

To complement Cornish, another, and equally forceful, review of LISP was undertaken in David Touretsky's article, "How LISP Has Changed: After 30 years, LISP has evolved to be both versatile and powerful" [Touretsky (1988), pp. 229-234]

Touretsky comments on the current proposal before the ANSI Committee X3J13 to standardize Common LISP. The lack of a standard version -- one that was fully documented -- has been perceived as a drawback for many implementations. The 1989 edition of Winston and Horn is based on the updated version of LISP before the ANSI Committee and includes proposed alterations that are not being considered by the Committee, such as CLOS (Common LISP Object System). Thus, it, at last, appears that a standard definition will be achieved by one of the computer programming languages most widely subjected to adaptations through "versions".

A final point in favor of LISP above other languages is the fact that it is inherently recursive, that is to say, it possesses the power of "self-creation". Hofstadter draws the charming, as well as pertinent analogy, of the student of a foreign human language. At a certain point, beyond the elementary level of investigation, it is far more helpful for a student to receive explanations of French and German expressions in the language which gave them life. [Hofstadter (1985), p.442] notion/analogy goes a long way to reinforcing

Winston and Horn's claims for the usefulness of LISP not only as a computer language teaching tool, but also as a multidisciplinary educational machine, whose informational levels of exchange can be regulated to match the educational level of the student.

Pratt in a comparative study which includes FORTRAN 77, COBOL, PL/1, Pascal, Ada, SNOBOL4 and APL, notes that:

> LISP is the only example in [this] book of a functional programming language, a language in which expressions composed of function calls, rather than statements, are constructed. LISP programs run in an interactive environment... and as a result, a main program does not exist in the usual form. Instead, the user at a terminal enters the "main program" as a sequence of expressions to be evaluated. The LISP system evaluates each expression as it is entered, printing the result automatically at the terminal. Ordinarily some of the expressions entered are function definitions. Other expressions contain calls on these defined functions with particular arguments. There is no block structure or other complex syntactic organization. The only interactions between different functions occur through calls during execution.
> LISP functions are defined entirely as expressions... Various special constructs...make this pure expression syntax appear somewhat like the ordinary sequence of statements syntax, but the expression form remains basic.
> Data in LISP are... restricted [to] literal atoms (symbols) and numeric atoms (numbers)... Linked lists and property structures form the basic data structures. LISP provides a wide variety of primitives for the creation, destruction, and modification of lists (including property lists). Basic primitives for arithmetic are provided.
> LISP control structures are relatively simple. The expressions used to construct programs are written in strict Cambridge Polish form and may include conditional branching. The PROG feature provides a simple structure for writing expressions in a sequence with provision for labels and gotos. Generator primitives (called functionals) are also provided for generating elements of lists in sequence. Recursive function calls are heavily emphasized in most LISP programming.
> [Pratt (1984), p. 498]

Pratt's "brief overview of the language" is a testament to the simplicity of LISP. That LISP is powerful is evidenced by his further assertions that:

> LISP is different form most other languages in a number of aspects. Most striking is the equivalence of form between programs and data in the language which allows data structures to be executed as programs and programs to be modified as data. Another striking feature is the heavy reliance on recursion as a control structure, rather than the iteration (looping) which is common in most programming languages. A third key feature is the use of linked lists as the basic data structure together with operations for general list modification. List processing is the basis of most LISP algorithms, although numbers and characters may also be manipulated to a limited extent. The important storage management technique of garbage collection was also first introduced in LISP.
> [Pratt (1984), p.497]

4. DISCUSSION: THE ATTRACTION OF LISP

Thus far we have noted that LISP is a symbolic programming language, that it is universal when taken at the core level of pure LISP, and that it is functional, powerful and simple. An additional factor should be noted before directing our attention to the conventional programming languages of C and Ada. LISP is fun. The attractiveness and (ease of) "availability" of a language to its users and implementers would not appear at first sight to be a major criterion in selecting a particular programming language for computer implementation, unless, of course, one's task is to manage the programmers and implementers. In that case, human behavior in organizations would dictate that the more enjoyable and challenging the task assigned, the more effective and careful the worker input into the finished products involved -- in this case, hopefully, software that possesses a high initial correctness (verity) and that is highly accessible for maintenance (verification) purposes.

Programming LISP within its own environment holds a deep degree of attractiveness to the most inventive and imaginative minds. As early as 1983, Hofstadter wrote a series of articles on LISP in Scientific American. Quick to laud the language, he summarized his points in favor of LISP programming with six points:

> (1) LISP is elegant and simple.
> (2) LISP is centered on the idea of lists and their manipulation -- and, lists are extremely flexible, fluid data structures.
> (3) LISP code can easily be manufactured in LISP, and run.
> (4) Interpreters for new languages can easily be built and experimented with in LISP.
> (5) "Swimming" in a LISP-like environment feels natural for many people.
> (6) The "recursive spirit" permeates LISP.
> [Hofstadter(1985), pp.xiii and 444]

While all of the above points are important to advocates of the LISP language for software design and implementation, the fifth point may be of greatest importance, where accuracy and maintenance are key factors. The fun inherent in LISP can be the key factor in the most crucial and effective user interfaces. That "fun-factor", along with the ability to interpret [what Pratt would call the universality or ability to model theoretically] new languages and the ability to translate existing languages of concrete implementations take LISP to the horizons of computer technology. Sui generis, artificially intelligent, and possessed of an unequaled mathematical precision to reason as well as compute with symbolic logic, LISP is at the forefront of the Fifth Generation of computers. As McCarthy said, "Fans of other

programming languages are challenged to write a program to concatenate lists and prove that the operation is associative."

CHAPTER 3

C: HISTORY AND PHILOSOPHY

0. INTRODUCTION

The C programming language has become in the last few years the programming language of choice for many developers. Both government projects such as NASA's CLIPS and commercial products such as Aldus's Pagemaker have been coded in C. The fascination of C can be found in its speed and portability. When C is coupled with UNIX, its natural operating system, a powerful environment for networks and programming results

1. HISTORICAL BACKGROUND

In the last chapter the notions of power, simplicity, and universality were examined for LISP. These notions are so central to the comparison and appraisal of computer languages and applications that it is not the least curious that Kelley and Pohl begin their work on the C language with these notions in mind.

> A book on C must convey an appreciation for both the elegant simplicity and the power of this general purpose programming language.
> [Kelly and Pohl (1984), p. v]

That C and LISP are "powerful" and that both are "elegantly simple" is undeniable, but there is a vast difference in the philosophies and functions implemented by the designers of the two languages. This difference is what makes the comparison and appraisal of the languages both difficult and interesting. The difference lies in the fact that LISP is intended to be a symbolic language that operates in its own "intelligent" environment, while C is a conventional language intended to allow the skilled programmer to devise and control algorithms that are portable and allow the resources of the platform to be put to their most effective use.

LISP programs often embody principles of how to manipulate knowledge. This is very different from the conventional programming tasks at which C excels.The conventional process has been described rather well by Brinch Hansen [quoted in Mateti (1979), p. 52]:

> Programming is the art of writing essays in a crystal-clear prose and making them executable.

While it might be said of LISP programs that they too are intended to be crystal clear, there is an important distinction to be drawn as to the object of that clarity. In LISP it is often desirable to make the program clear to the programmer or those who will maintain the program. In this sense it is better to create functions that are human-like. The clarity that results is the sort of clarity addressed directly to human factors. On the other hand the clarity of C is machine-directed. Terse and unambiguous constructions that clearly indicate what the program will do when run are often thought to be desirable in C programming. Thus, the clarity of C can be said to be machined directed, while the clarity of LISP can be said to human directed.

In examining the creation of C three factors should be kept in mind. First, interactive computers were at an experimental stage. Second, physical batch processing of cards for input was a norm. Finally, the C language arose as an implementation device following the development of the UNIX operating system, which resolved the first two factors.

In the 1960's, Dennis Ritchie and Kenneth Thompson were engaged at Bell Laboratories in a cooperative project with GE and MIT called Multics [multiplexed information and computing service]. This project focused on the then innovative concept of multi-user time-sharing systems for programming. For both monetary and conceptual reasons, Bell Labs withdrew from the project. However, the experience of working within an interactive environment, one in which an on-screen interface allowed the user to see immediate programming results, had"spoiled"both Ritchie and Thompson. The batch processing of cards, demanding ninety per cent of effective time consumption, presented an untenable alternative to interactive programming.

Given Bell Lab's negative experience with Multics, Ritchie and Thompson set out on their own to develop an operating system that could be realistically implemented.The key was simplification:Multics was a research device whose essential features Ritchie and Thompson reduced to the first stage of UNIX. This device saw its first practical implementation as a(disguised)wordprocessor for the patent office at Bell Labs. From 1969 through 1979, when Bell's first serious marketing efforts on behalf of UNIX began, a process of experimentation and development that bridged both the corporate and academic

sectors occurred.The result was that when UNIX hit the marketplace, it did so as a finely tuned and very attractive instrument for the computer programmer.

The first version of UNIX was written in assembly language, but the notion of maintaining UNIX at the machine level render it unmarketable to the more general user.In other words, UNIX in assembly language was not portable -- a key feature in its future appeal.Ritchie and Thompson designed a new high level language [with the efficiency of an assembly language] they called C.It possessed features of both high and low level languages and was, therefore, quite versatile for systems programmers, who could address memory as input/output channels directly without assembler code.C combined powerful logical instruction with an ability to manipulate individual bits and characters, and it was significantly easier for a programmer to use than assembly language.Only when UNIX was rewritten for C did it become truly portable.C became popular as an implementation language or systems programming language in large measure because it was the language in which UNIX has been written. [Slater (1987), pp. 280-281]

2. THE C LANGUAGE

C was invented by Ritchie in 1972 and its nomenclature derives from the B language that Thompson wrote for Bell in 1970 for the first UNIX on the DEC PDP-11.[Ritchie et al (1978), p. 458] BCPL and B are "typeless" languages.By contrast, C provides a variety of data types.The fundamental types are characters, and integers and floating-point numbers of several sizes.In addition, there is a hierarchy of derived data types created with pointers, arrays, structures, and unions.Expressions are formed from operators and operands; any expression, including an assignment or a function call, can be a statement.Pointers provide for machine-independent address arithmetic.

> C provides the fundamental control-flow constructions required for well-structured programs: statement grouping, decision making (if - else), selecting one of a set of possible cases (switch), looping with the termination test at the top (while, for) or at the bottom (do), and early loop exit (break).
> Functions may return values of basic types, structures, union, or pointers. Any function may be called recursively. Local variables are typically "automatic," or created anew with each invocation.Function definitions may not be nested but variables may be declared in a block-structured fashion.The functions of a C program may exist in separate source files that are compiled separately.
> Variables may be internal to a function, external but known only within a single source file, or visible to the entire program.
> A preprocessing step performs macro substitution on program text, inclusion of other source files, and conditional compilation.
>
> [Kernighan and Ritchie (1988), p. 1]

The definition of C above is taken from the second edition of The C Programming Language [1988] by Brian W. Kernighan and Dennis M. Ritchie. This is the third "generation" [1978, 1983] of what its designers feel is a "general purpose programming language featuring economy of expression, modern control flow and data structure capabilities, and a rich set of operators and data types".[Ritchie, et al (1978), p.458]

3. C ISSUES

As with any programming language, many factors about C have generated both criticism and advocacy.

One of the less interesting factors concerns whether C is really a general purpose programming language.As we noted earlier, the universality of programming languages in the Turing sense can be applied to most programming languages, and, therefore, it can be said that most programming languages are "general purpose". In this sense C is no exception; it is a general purpose language.

We have also noted that the key factor often associated with the idea of a general purpose language that is more important is whether or not the language is qualitatively more effective, efficient and generally smoother than some other language [cf., Pratt above].In the case of C, Feuer and Gehani of Bell Labs,assess the relative merits of C and a more strongly typed language, Pascal, over a variety of domains of applications.

It is often argued that the more strongly typed a conventional language is, the more the language forces the programmer to construct a correct and verifiable algorithm.Feuer and Gehani require a strongly typed language to be such that:

> 1.Every object in the language belongs to exactly one type.
> 2.Type conversion occurs by converting a value from one type to another.Conversion does not occur by viewing the representation of a value as a different type.(In FORTRAN, by contrast, viewing the representation of a value as a different type is possible by means of the equivalence statement.)
>
> [Feuer and Gehani (1982),p.6]

We have noted that unlike its predecessors, B and BCPL, C is a typed language, but it is not a strongly typed language.The flexibility [that is, the close interaction between programmer and machine] that makes C an ideal language from the systems programming

viewpoint is precisely the factor that robs it of effectiveness when it comes to program verification.

The freedom of C is purchased at the price of concentrated accuracy; C is, therefore, not particularly desirable in business programming.For scientific programming a key factor involves the lack of built-in math functions, though these are available in the standard library. In both areas of application, Feuer and Gehani contend that the strongly typed character of PASCAL and its more "built in" mathematical functions make it, and not C, more desirable. [Feuer and Gehani (1982), p.27-29] This is a rather strong negative assessment of C when one remembers that both men are engaged by the firm producing UNIX.

Feuer and Gehani correctly extol the virtues inherent in C's bitwise control as an application for programming operating systems and systems utilities.These two application domains were, however, the central focus of C's designers.

In addition to the question of how good a general purpose language C really is, the question of the size of UNIX and C often inspires both the critics and the advocates.

Much has been said of C's size in relationship to other languages.A number of issues have to be addressed at this point. When considering the "size" of a language, one must also consider what design factors of the language can be isolated into units that allow comparison with other languages.One of the most helpful approaches from the evolving science of comparative program languages is the concept of isolating a "core" for each language to be considered.Pratt [above] has done this for LISP. As we noted in the first chapter Shaw et al [(1981), pp. 1-51] have developed a paradigm for such comparisons.

Shaw and her co-authors consider a single example program to be implemented in each of the four languages of their study: FORTRAN, COBOL, JOVIAL and IRONMAN, the proposed DoD standard that existed in 1981. From the outset it would be hopeless to include C in the comparison in any direct way. C, which is so famously "terse" [Kelly and Pohl (1984), p. 2], requires the addition of a standard library in order to handle input/output of data, and other common operations.This factor adds considerably to the "size" of a language, since one would have to access functions of the standard library in order to reach a "core" language definition of C in line with Shaw's constraints.Feuer and Gehani [1984, p. 1] note that while"C is also smaller than Ada,... when it is considered in

conjunction with the UNIX operating system, C also addresses many of the same issues as Ada".This is certainly a reflection on the size of C's scope, if not directly on C's semantic size. Thus, one may come to the conclusions that the size of C bears little relevance to an evaluation, and that there are important relations between the notions of size, portability and terseness.

Yet another issue advanced in considerations of C is the idea of functional minimality. Kelly and Pohl put this idea well:

> C gets its power by carefully including the right control structures and data types, and allowing their uses to be nearly unrestricted where meaningfully used.

They suggest that this makes C a vehicle of easy of access by programming students. However, they omit the notion that in order to learn to program in C "easily", one must first know how to interface with UNIX via the standard library.Arguably, C is portable; but, the cost of its portability is a sparseness that can negatively affect both the learnability of the language and programming verifiability.This is also reflected in Feuer and Gehani's negative assessment of C as a language for scientific and business purposes. Further, it should be noted that it is often the case that positive assessments of C are weighed heavily on the C environment. In particular the tremendous value of the UNIX operating system when programming in C, is often the source of positive assessment. As in the case of LISP this makes it difficult to distinguish between the language and the environment.

4. DISCUSSION: THE C PROGRAMMER

The final issue of interest centers on the image of the programmer. Though this may initially seem a bit odd, it is really not so at all. For example, in a somewhat poetic way, Peters and Sallam [1984, pp. xi-xii] refer to the "gentle art [of C] as having to do with noble or excellent things" in the Oxford English Dictionary's sense of the word "art". This echos much of the approach that Kernighan and Ritchie adopt in their various discussions of C over the years. The C programmer is presented as the paradigm of discipline,intelligence and efficiency. While this may, indeed, be true of the C programmer, it may also be true of the programmer in any language. Further, the more a programmer knows of a language, the more he may deviate from the paradigm in order to increase productivity, the speed of an application, or worst of all, his own position within the system. It is in this light that the following claims by Kernighan and Ritchie are interesting:

> A programmer can reasonably expect to know and understand and indeed regularly use the entire language.
>
> [Kernighan and Ritchie (1988), p. 2]

> Pointers have been lumped with the goto statement as a marvelous way to create impossible-to-understand programs.This is certainly true when they are used carelessly, and it is easy to create pointers that point somewhere unexpected.With discipline, however, pointers can also be used to achieve clarity and simplicity.
>
> [Kernighan and Ritchie (1988), p. 93]

It is at this point that "the axiom of Flon", quoted by Mateti, is an interesting and relevant response to Kernighan and Ritchie's assumptions about the prowess and discipline of the C programmer.

> There does not now, nor will there ever, exist a programming language in which it is the least bit hard to write bad programs.
>
> [Mateti (1979), p.34]

C offers an ardent programmer an almost unique access to his system and levels of functional control that are denied the devotees of many other programming languages. However, this access also allows the C programmer to deviate from the paradigmatic disciplined programmer. In the research and development areas of a government agency or corporation, there may be a great deal of unparadigmatic programming. This might especially be the case in those areas set aside and protected from outside interference by the standard devices of an orchestrator to handle administrative functions and an idea champion to control development flow. One might even begin to think that the approach is similar to that of one whose "hobby" is programming. The dedicated hobbyist may well emphasize the functionality of his program without attending to the intelligibility of his code. This "hobby" approach to which Ritchie freely admits [Slater (1987), p. 282] is not the stuff on which enforced teamwork, cohesive design efforts, and, ultimately, verifiability and maintain ability are built.

All of the criticisms of C, however, must be weighed against the tremendous advantage of C's portability. This, and the commercial success of UNIX as its parent operating system, account for many C implementations.In understanding the virtues of portability, one needs also to examine the terseness of C. This terseness surrenders a great deal, but is needed to insure that the language can be sufficiently portable to operate in a common way on many different hardware platforms. C, removed from UNIX and its own standard library, is so "terse" that within and of itself,one finds it difficult to arrive at a core definition of the language, in the sense of Shaw, in which programs could actually be written. In this sense C is a partial language, dependent on the operating system and libraries in order to function

well. By itself this factor may not seem important. However, in a comparison with LISP it argues for the claim that it is the overall programming environment and not the language by itself that should be taken into consideration when adopting a language for a project.

At any rate, the particular kinds of "elegant simplicity" and "power" that Kelley and Pohl claimed for C in our initial paragraph are both present in C and a cause of concern about C. There is undeniably a sense of power, elegance and simplicity that characterize C. C is undeniably portable. But, and this is an important consideration, the maintainability of C programs, as with other imperative languages, is still a problem. It follows for neither, LISP nor C that their virtues include the capacity to foreclose the possibility of bad programming. Significantly, it also seems to be the case for both LISP and C, that many of the virtues of the languages are intimately entwined with the environments and tools with which the programmer works.

CHAPTER 4

ADA: HISTORY AND PHILOSOPHY

0. INTRODUCTION

Ada is a language designed by committee to meet the needs of governmental agencies that require software programs to be built and maintained for embedded systems. Ada can trace its roots to Niklaus Wirth's Pascal and shares Pascal's structured approach to programming. Although all programming languages have their critics and advocates, Ada has had to endure an unusual level of criticism. It should, however, be remembered that Ada should be judged in terms of its intended applications area.

1. THE HISTORICAL BACKGROUND

"The development of the Ada language followed a pattern unique in the history of major programming languages." [Pratt (1984), p. 457] The motivation for this development arose from a "crisis" within the Department of Defense of the United States government [hereafter, DoD].In 1973, a study done by the DoD revealed that of $3 billion spent on software some 56 per cent had gone into developing and maintaining embedded systems.COBOL and FORTRAN were effectively handling data management and scientific programming functions [Barnes (1980), p. 426], but the languages in use for embedded systems were a confusion of variants that created severe problems of cost effectiveness.Obviously, translators were having to be developed across the "200 models of computers and over 450 general-purpose programming languages and dialects... used for embedded computer systems."[Shumate (1984), p. 10] The criticality of the situation stemmed from technical as well as financial perspectives.Pratt notes that in an embedded computer system, where the device is resident within a nuclear submarine or power plant, in monitoring devices at chemical manufactories and hospitals, or any critical environment, "the computer system has become an integral part of the larger system, and the failure of the computer system usually means failure of the larger system as well."[Pratt (1984), p. 333]

It is worth noting here, because of its association with the Department of Defense, that COBOL was developed in the early 1960's by a committee.The size of that effort and the time frame involved bear no resemblance to the massive, multinational effort that resulted in Ada. However, there may still be qualitative similarities in the effects of the program.

The repercussions of the crucial nature of embedded systems and the spread of the systems over many languages were that each new batch of very specialized software had to be tested, that failure was costly, and that even the testing of successful designs was inefficient.Maintenance sky-rocketed for specialized systems which were understood only by the developing contractor. Finally, retraining of government personnel to handle specialized embedded systems was a tremendous burden in time (lost man-hours) as well as financial allocations.

To answer these problems the DoD established the High-Order Language Working Group [HOLWG] in 1975. HOLWG's mission was to examine currently available languages that would be suitable for overall implementation within DoD of embedded functions or to establish some alternative language that could be developed based on a hard outline of requirements [Shumate (1984), pp. 10-11].

DoD defined an embedded system as a system with both hardware and software components whose purpose is other than computation.Examples include avionics, command and control, artillery fire control, digital communications networks, and so on.Such systems are in extensive use throughout DoD. [Evans (1984), p. 68, note.]

It was found early on that no extant language was suitable for DoD's purposes, and, therefore, a series of criteria were established which formed the basis for an international design competition. Four from among 17 proposals were chosen for further exploration; all four were based on adaptations and developments of Pascal, thus adhering to a conventional language base [Pratt (1984), p. 457].

HOLWG developed an on-going series of requirement documents over the course of 1975-1978.These ranged from the initial Strawman to Woodenman, Tinman, Ironman, and Revised Ironman to the final Steelman of June, 1978. The final specifications were exhaustive. Steelman clearly indicated the seriousness of purpose of DoD. The intention of the specifications was to establish a programming language that would satisfy the following demands:

> Generality.The language [should] provide generality only to the extent necessary to satisfy the needs of embedded computer applications.
>
> Reliability.The language should aid the design and development of reliable programs; be designed to avoid error prone features; maximize automatic detection of programming errors; require some redundant, but not duplicative specifications in programs.
>
> Maintainability.The language should promote ease of program maintenance; emphasize program readability (that is, clarity, understandability, and modifiability of programs); encourage user documentation of programs; require explicit specification of programmer decisions.
>
> Efficiency.The language design should aid the production of efficient object programs.
>
> Simplicity.The language should not contain unnecessary complexity; should have a consistent semantic structure that minimizes the number of underlying concepts; should be as small as possible consistent with the needs of the intended application; should have few special cases and should be composed from features that are individually simple in their semantics.
>
> Implementability.The language [should] be composed from features that are understood and can be implemented.
>
> Machine independence.The design of the language should strive for machine independence; shall attempt to avoid features whose semantics depend on characteristics of the object machine or of the object machine operating system; there shall be a facility for defining those portions of programs that are dependent on the object machine configuration.
>
> Complete definition.The language shall be completely and unambiguously defined.
> [Shumate (1984), p.13]

The required strict definition of Ada was perceived as a tremendous benefit.The standard reference appeared as ANSI/MIL-STD-1815A on January 22, 1983.Ada was named for Lady Augusta Ada, the Countess of Lovelace, daughter of Lord Byron, and mathematician and friend of Charles Babbage.It was through her efforts that some public recognition of Babbage's "Difference Engine" came to pass. Since she "programmed" some of its functions [Slater (1987), p. 11], she became the world's first computer "linguist".

2. THE ADA LANGUAGE

Shaw defines the core philosophy of Ironman — which has held (largely) for Ada — as follows:

> The attitudes which emerge are dominated by two practical considerations;
> 1.The intended application to embedded systems leads to (restricted) generality and specific machine-relevant considerations.
> 2.The importance of maintainability leads to a spirit of clarity, simplicity, and understandability.
> The dominant characteristics of the Ironman language seem to be:

> 1.Simplicity of language:This is explicit in the goals, and it is supported by lexical, syntactic, and semantic considerations in the specific requirements.The simplicity of the language is intended to arise from the similarity of related constructs, and from uniform assumptions about the interpretation of program statements. The simplicity is further supported by a number of requirements for compile-time checking.
> 2.Rich data structuring: Data structuring facilities include not only scalar values and structures which are aggregates of scalars, but also complex record structures and mechanism for defining application-specific data organizations.
> 3.Programmer-defined types:A powerful abstraction facility is intended to allow definitions of data structures and related operations to be encapsulated and protected. Abstractions defined by this mechanism are to have same status as the primitive data abstractions (types) of the languages.
> 4.Access to underlying hardware:Explicit provisions are made for providing access to all the features of the underlying hardware and for interposing as little language overhead as possible. In this way the programmer can obtain very detailed control over efficiency of both code and data representations.
> 5.Conscious restraint:A number of facilities have been omitted and the power of others has been restricted in order to retain leanness and safety in the language.In addition, the requirements show considerable sensitivity to the usability of the language.
> [But then,] we note that the view projected by the Ironman requirements is not totally self-consistent, and it may not be possible for any language to simultaneously satisfy all the goals.In particular, there seems to be a conflict between a desire for readability and safety on the one hand and detailed control over access to the underlying hardware on the other.
>
> [Shaw (1981), p. 21-22]

Shaw's point about conflicts between the dominant characteristics is also addressed by Evans.

3. ADA ISSUES

Evans [(1984), p. 68] notes that early in the design of Ada there was an attempt to rule out the possibility of side effects.Later this goal was dropped, since it impacted badly on the language.While safety is a desirable goal, one can object strongly to stripping the language of useful power in an attempt to protect from all possible consequences of a programmer's own folly. Evans cites other factors that had a negative impact on the ANSI Standard of Ada. In particular he notes that between Ironman and Steelman, a great many features were forced on the language by the international committees of designers and reviewers (those who were responsible for critiquing the on-going design efforts under Jean Ichbiah of France).These factors are perceived to have created a language whose sheer size impacts negatively on the programmer because (1) the vast quantity of material to be assimilated before approaching real world implementations of Ada is daunting; (2) the "combinatorial complexity" with which the features of Ada interact confuses both programmer and machine compiler; and, (3) most importantly, the size of the compiler that can effectively handle Ada is both an inordinate expense and is hard to debug. Given the initial impetus to

create a maintainable and verifiable programming language, the final committee-produced result is a rather sorry failure in Evans's view.[Evans (1984), pp. 89-91]

At the 1988 Ada Expo and Special Interest Group convention in Boston, James R. Ambrose, speaking as Under Secretary of the Army, addressed the claimed superiority of Ada:

> Ada is beset with too much hyperbole for its own good... I am not impressed, have not been, and I remain willing to be impressed, with quantitative measurements of the superiority or productivity or the virtues of anything such as Ada.What I have seen thus far has been largely rhetoric.
> [Hughes (1988), p. 60]

There were, of course, responses. David E. Quigley, the VAX Ada product manager for Digital Equipment Corporation, noted that, "Ada's only handicap is that it claimed to be a savior for all applications." Hughes replied to Evans' 1984 criticisms of Ada compilers. Hughes delineated efforts to rectify the situation and was able to state that "many high quality compilers [are now] available that are validated by the Defense Dept.'s stringent tests". [Hughes (1988), p. 60]

The complexities of these issues and others, however, do not seem to have greatly affected the practical decision of adopting the language. Wilson confirmed in the July 1987 issue of Computer Design [pp.90-91] that Ada is being accepted by DoD contractors, even if for no better reason than that Ada is an inherent part of many product and design specifications.Articles in Datamation [Carlyle (1986), pp.30-31] and Aviation Week & Space Technology [Nordwall (1987), pp. 91-93] indicate a fairly heavy government interface with private industry in order to promote the use of Ada for more general data processing functions than those at the Defense Department.Computer Corporation of America has built a massive database management system (DBMS) called Adaplex with a half-million lines of source code to handle general purpose functions.The effort was made in expectation of a DoD expenditure on DBMS of over a billion dollars annually by 1990.

Ada has had very powerful champions in DoD. [Ambrose retired in March of 1988]. It should be remembered that much of the initial acceptance of COBOL within the corporate community was based on DoD's support for its own creation. It is particularly worth noting the following:

> The Pentagon mandated on Mar. 31, 1987, that, effective immediately, all software development for new weapon systems be performed in Ada as the single, common, high-order programming language.Deputy Defense Secretary William Taft, 4th, broadened that directive in April, 1987 by specifying that Ada must be used on all Defense Dept. computer resources, with few other languages permitted in certain cases. These reaffirmations of the Defense Dept.'s commitment to Ada are expected to curb the number of waivers requested and the number granted to develop programs in a language other than Ada.
> The [Ada Expo in Boston] was sponsored in part by Software Valley Corp., a nonprofit organization founded by U.S. Senate Majority Leader Robert C. Byrd (D-W.Va.) to promote high technology industry in West Virginia. Software Valley has held other conferences to promote Ada and to attract related companies to the state.
> Furthermore, the use of Ada is spreading into the commercial marketplace, particularly in Europe, where telecommunications firms are using it.The European defense ministries and NATO are also advocating the use of Ada, but are not moving as quickly in this area as the Pentagon.
> The real test for the language is expected [by industry leaders] to be how it performs in weapon systems programs that are employing it for full-scale development. How the language performs in key programs like the ATF [Advanced Tactical Fighter:7,000,000 line of code planned]... will go a long way in determining its ultimate success.
> One worry expressed by many at the Ada conference... is that any budget cuts that trim or eliminate new weapon system programs in which Ada is being used will slow the acceptance of the language by the defense community and reduce the demand for Ada software support products.
>
> [Hughes (1980), pp. 60-61]

Nonetheless, such corporations as Boeing, Lockheed and IBM are heavily involved in both Ada research and development and in producing Ada software and software-related systems. And, it is estimated that by the year 1992, 50 per cent of the United States aerospace industry personnel will be working in the Ada market (up from 3 per cent in 1986 and 12 per cent in 1989) [Hughes (1980), p. 69].

Ultimately, there will be some sort of balancing out of those who perceive Ada negatively and those among the DoD who actively encourage its implementation. The essential fact to remember is that even though COBOL was pushed into initial acceptance and success by DoD, that particular language gained its wide usage because of its general applicative excellence in business situations.In the main, full systems applications of Ada are in an experimental stage at the moment.We cannot know what the outcome of NASA's fiat to Lockheed for 750,000 lines of code for the Space Station Software Support Environment until many years from now.

More voices than Evans's have risen in protest over the size of Ada.C.A.R. Hoare recalled his successful experiences with Algol 60,and the failure of the more complex Algol 68, and drew a parallel between Pascal and its larger descendent, Ada.Speaking of another language [PL/1] of which he had some knowledge, Hoare outlined his frustration in

attempting to standardize that language, and noted that some projects are "doomed to success," if enough money is involved. He continues by relating his past experience to the Ada project.

> The mistakes which [were] made in the last twenty years are being repeated today on an even grander scale.I refer to the language design project which has generated documents entitled Strawman, Woodenman, Tinman, Ironman, Steelman, Green and finally now Ada.This project has been initiated and sponsored by one of the world's most powerful organizations, the United States Department of Defense. Thus it is ensured of an influence and attention quite independent of its technical merits and its faults and deficiencies threaten us with far greater dangers. For none of the evidence we have so far can inspire confidence that this language has avoided any of the problems that have afflicted other complex language projects of the past.
> The original objectives of the language included reliability, readability of programs, formality of language definition, and even simplicity.Gradually these objectives have been sacrificed in favor of power, supposedly achieved by a plethora of features and notational conventions, many of them unnecessary and some of them, like exception handling, even dangerous. We relive the history of the design of the motor car. Gadgets and glitter prevail over fundamental concerns of safety and economy.
> ...by careful pruning of [Ada] it is still possible to select a very powerful subset that would be reliable and efficient in implementation and safe and economic in use.The sponsors of the language have declared unequivocally, however, that there shall be no subsets.This is the strangest paradox of the whole strange project. If you want a language with no subset, you must make it small.
> You include only those features which you know to be needed for every single application of the language and which you know to be appropriate for every single hardware configuration on which the language is implemented.Then extensions can be specially designed where necessary for particular hardware devices and for particular applications.That is the great strength of Pascal, that there are so few unnecessary features and almost no need for subsets.That is why the language is strong enough to support specialized extensions [for concurrency, discrete event simulation, and microprocessor work]. If only we could learn the right lessons from the successes of the past, we would not need to learn from our failures.
> And so, the best of my advice to the originators and designers of Ada has been ignored.In this last resort, I appeal to you, representatives of the programming profession in the United States, and citizens concerned with the welfare and safety of your own country and of mankind:Do not allow this language in its present state to be used in applications where reliability is critical, i.e., nuclear power stations, cruise missiles, early warning systems, anti-ballistic missile defense systems.The next rocket to go astray as a result of a programming language error may not be an exploratory space rocket on a harmless trip to Venus:It may be a nuclear warhead exploding over one of our own cities.
> An unreliable programming language generating unreliable programs constitutes a far greater risk to our environment and to our society than unsafe cars, toxic pesticides, or accidents at nuclear power stations.Be vigilant to reduce that risk, not to increase it.
>
> [Hoare (1981), pp. 487-495]

Hoare's exhaustive appeal drew a large body of correspondence on both sides of the "Ada issue" to the ACM [Saib and Fritz (1983), p. 475], but there had been earlier attempts in the United States of achieve a more viable real-world presentation of the Ada language, using Ironman as the base [Evans (1984), p. 69].

4. DISCUSSION: ADA, THE SAVIOR?

Two of the more important attempts in Communications of the ACM were Brian A. Wichman's "Is Ada Too Big?A Designer Answers the Critics" and Jean E. Sammet's "Why Ada is Not Just Another Programming Language."Wichman praised the DoD for allowing Hoare to chair a discussion in 1975 and for generally being open to debates on] the subject of Ada. He notes that even if Ada is "a large language not liked by every one,.. Clearly, if the DoD is satisfied, the language is — at the most pragmatic level — a success" [Wichman (1984), p. 98]. Wichman concludes that the "advantages of a single, highly standardized language should far outweigh the inconvenience presented to individual users by Ada's complexity. [But] in ten years, we will know how Ada should have been designed.It is the decision we made in the original design phase,and the alterations we make today, though, that really count". [ibid., p. 103]

Both Hoare's and Wichman's arguments are to a certain degree subjective, but there is a difference. Hoare speaks from an already vast experience of the extensiveness of programming languages, while Wichman freely admits that the final effectiveness of Ada's design can only be judged ten years hence.Only future real-world applications can determine whether or not Ada is "a savior for all applications".

This is not to say that the DoD will implement Ada in untested critical-mission configurations, or that Mr. Wichman is not an authority in programming languages, and an intelligent and competent commentator in the on-going Ada debates.Shaw notes that Ada is being developed and implemented with some twenty years of experience in the design of programming languages to guide its inception:

> Thus [since the 1950's], both the specific application domain and a great deal of cultural context differs.The notions of program structuring, potentially dangerous languages constructs, and software engineering did not explicitly exist before the mid 1960's.Thus Ironman is responding to a range of concerns quite different from those facing the designers of other [earlier] languages.These responses can be seen in the restrictions on the goto, the restrictions on aliasing, and in the explicit inclusion of an encapsulation mechanism.
>
> [Shaw (1981), p. 28]

And, finally, Booch [1987] in his standard textural reference, Software Engineering with Ada, addresses the issues of maintainability and verifiability of software, the central motivation for developing a language such as Ada. The elements of management and engineering associated with large software projects, may, however, be common to any

programming language. It might also be the case that Ada is a language in which such tasks can be most readily accomplished. As we have already noted, it is often the environment as much as the language that is critical in the adoption of a language.

CHAPTER 5

PROGRAMMING PARADIGMS

0. INTRODUCTION

We hope that we have made it clear that the programming languages (LISP, C, Ada) that we have examined can each legitimately claim a special competence. In the case of LISP it is symbolic processing; in C, the combination of portability and speed; and in Ada, uniformity and maintainability. We did not intend our examinations of these languages to be either technical or exhaustive. Rather, we intended to examine the languages from what might be considered a managerial point of view. We were concerned to examine the languages from the perspective that a decision maker might take when assessing their strengths and weaknesses on information presented by advocates and critics.

From our examination, we believe that decisions about a programming language and a programming environment cannot be meaningfully separated and that they ought not to be separated. Whether one examines LISP or C, the languages with the longest histories in our study, it is clear that the advocates of these languages are not considering the languages in isolation. Rather they are considering LISP in a LISP environment, if not on a LISP machine, and C in a UNIX environment. We believe that this is as it should be. The art of programming is not simply the procedure of writing down code. The art of programming is to be found in the choice of programming paradigms and in the use of tools that make programming in that paradigm easy, safe, and effective. This perspective gives rise to the idea that it is not so much the language that encourages good programming, but the combination of language and environment. While a language might force the use of certain programming constructs, it is still possible to write bad code with those constructs. This would seem to be the case no matter how one cares to operationalize the notions of "good" and "bad". The availability of good programming tools in a good environment makes it easier to write good code than to write bad code. If this is correct, then it is clear why considerations of programming languages ought not to be separated from considerations of the programming environment.

Further, it appears that the combination of programming environment and programming language is intimately connected with the programming paradigm that is to be used in the construction of the program. A programming paradigm may be thought of as the style or manner in which a program is created. The notion of a paradigm in this context has strong connections to the notion of paradigm as it is used in the history, philosophy and sociology of science. A paradigm is a sort of template that is filled in to attack certain sorts of problems. Within one paradigm there may be many particular templates, but there is a sense in which each of these reduces back to some primitive template. Alternatively, one may view the paradigm as a primitive object from which the specific template inherits structures and properties. Under either sort of interpretation, it should be clear that a programming paradigm acts as a vehicle through which a programmer designs and builds specific programs.

1. PARADIGMS

Stroustrup sets out the relation between a programming paradigm and a programming language rather neatly.

> A language is said to support a style of programming if it provides facilities that make it convenient (reasonably easy, safe, and efficient) to use that style. A language does not support a technique if it takes exceptional effort or exceptional skill to write such programs... Support for a paradigm comes not only in the obvious form of language facilities that allow direct use of the paradigm, but also in the subtle form of compile-time and/or run-time checks against unintended deviations from the paradigm... Extra-linguistic facilities such as standard libraries and programming environments can also provide significant support for a paradigm.
>
> [Stroustrup (1988), p. 52]

The problem of selecting a paradigm is both art and science. It is an art insofar as it requires a subtle understanding of the programming craft, and is a science insofar as a set of decision rules can be established for the paradigm. Stroustrup investigate this second way, and identifies several paradigms for program construction. We will augment his decision principles to capture the intentional character of the decision.

PROCEDURAL PARADIGM

Principle

If

the intention of the program is primarily procedural

then

decide which procedures are to be programmed and use the best algorithm you can find.

Comment

The core of the principle is the link between procedural knowledge and algorithms. The paradigm focuses on those parts of knowledge that can be codified in a step by step way. The sense of procedure here is not the broad sense of procedure in which it may be said that a manager follows procedures, but the quite narrow sense of procedure in which one proscribes a set of stepwise instructions that guarantee that an answer will be found. Sorting programs are a good example of programs that conform to this paradigm. Programming in conventional languages often conforms to this paradigm.

DATA HIDING PARADIGM

Principle

If

the intention of the program can be divided into modules

then

decide the modules you want and partition the program so that the data in one module is hidden from the data in other modules.

Comment

The principle of data hiding expands on the procedural paradigm. The core of the data hiding paradigm is the connection between modules and partitions. If the objective of the program can be divided into tasks that can act as a module, then the program should be divided so that the data in any partition does not affect the data in any other partition. Thus a module is a collection of procedures that act upon data in such a way that the internal operations of the module do not lead to global effects. For example, stacks and queues might be implemented in a module.

ABSTRACT DATA TYPE PARADIGM

Principle

If

the intention of the program is to provide for more than one object of a given type

then

decide which types are needed and provide a full set of operations for each type.

Comment

The data hiding and the abstract data type paradigms are similar. They differ insofar as the abstract data type is more general; data hiding can be considered as an instance of the abstract data type paradigm restricted to a single object of the type. It should be noted that the phrase "abstract data type" can be misleading. The type itself is not abstract; it is as real as any other type. The difference between the abstract data type and the types in the language, might therefore be better captured with the phrase "user- defined type." Although such user-defined types allow for multiple objects of the defined type, the functions that deal with these types must know about each type. Thus, if a new type is added to a program, the code that contains the functions that operate on that type must be modified.

OBJECT ORIENTED PARADIGM

Principle

If

the intention of the program is to provide for classes of multiply related user-defined types

then

decide which classes are needed, provide a full set of operations for each class, and make commonality explicit by using inheritance.

Comment

If the general and the specific properties of a class can be differentiated then it is desirable to use the object oriented paradigm. This paradigm makes it clear that the classes of user defined objects can be made more and more specific and that the specific objects have instances. Further, within this paradigm there is the idea that at least some functions are generic; some functions apply to different types of object. The key idea of the object oriented paradigm is that classes of objects can be defined (say, animals) from which more specific classes (mammals) can be built. These more specific classes inherit the properties

of the more general class, and these more specific classes may themselves be used to define even more specific classes (dogs). Further, within this paradigm, it is possible to use two or more general classes (say water-craft and motorized-vehicle) to create a more specific class (motor-boat). The new class will inherit the general properties of both classes and will allow the generic methods for both to operate on this new class.

3. DISCUSSION

Although there are formal senses in which all sufficiently rich languages are equivalent, this equivalence is only logical or formal. Although any of the paradigms can be accomplished in any of the languages that we have examined this does not mean that it is either easy or reasonable to use any of the mixtures of paradigm and language that are possible. Indeed in the case of two of the languages that we have examined, LISP and C, specific packages of extensions have been generated in order to support the object oriented paradigm, Flavors and C++. At the moment it is not clear what packages of extensions will or will not be available for Ada. What is clear is that the first three paradigms are supported by all three languages, although it is somewhat odd to think about the procedural paradigm for LISP.

The issues of programming paradigms are intimately connected to issues of language choice, and by extension environment choice, as well as to the issues that surround the idea of software engineering. We believe that this nexus of issues should be brought together, perhaps under the rubric of software management. Within this conception software management would include software engineering. Software engineering would provide the tools and methodologies that are needed to construct good, reliable and maintainable code. These tools would of course have to be tightly tied to the programming language(s) and environment(s) in which the program is built.

CHAPTER 6

SOFTWARE ENGINEERING

0. INTRODUCTION

To this point we have briefly examined the histories and philosophies of three programming languages, LISP, C and Ada. These three languages have their adherents and critics, their strengths and weaknesses. From our point of view it does not appear that any one of these languages can at the moment claim to have (1) resolved all of the issues surrounding the adoption of a programming language, (2) demonstrated itself to be clearly superior in all programming situations, (3) demonstrated itself to be uniquely qualified for a particular programming task, (4) gained sufficient adherents to make the other languages begin to disappear, or (5) lost sufficient adherents so as to make its survival unlikely. From our point of view each of the languages can legitimately claim certain virtues and strengths, however. Now there would be little evil in having a polyglot programming community, if it were not for certain external features of the programming enterprise. It is these features that lead to a certain sort of crisis.

The crisis is not one that threatens small or medium scale projects. It is not a crisis that directly threatens commercial computer software corporations. At the moment, it is only a crisis for those projects that are very large, that are very critical, and that have a nondisposable nature.

Such projects as the multiplicity of programming languages would affect are those that the government is likely to undertake or has undertaken. The prime examples of such projects are those connected with space exploration and defense. Both NASA and DoD are aware of the problem. As we have already noted a response to this crisis is the Ada programming language. Before hazarding an opinion on that language as a solution to the problem, we will first investigate a bit of the the history and philosophy that surrounds another effort to address the crisis, software engineering. We do not intend to portray software engineering and the adoption of a single language as competitive solutions; they are not. However, we

do wish to examine whether there is a sense in which the independence of the two ideas can shed light on the crisis and its resolution.

1. INITIAL DEFINITIONS

In a minitutorial at the September, 1987, IEEE Conference on Software Maintenance in Austin, Texas, Peercy, Tomlin and Horlbeck, offered a group of definitions which are helpful in approaching the task at hand. SOFTWARE MANAGEMENT can be described as the collection of policies, methodologies, procedures, and guidelines that are applied in a software environment to the software development and maintenance activities. SOFTWARE MAINTENANCE is comprised of those actions required for the correction, enhancement and conversion of software. Correction is the removal of software faults, enhancement is the addition or deletion of features, and conversion is the modification of the software because of environmental (data or hardware) changes. SOFTWARE SUPPORTABILITY is a measure of the adequacy of products, resources, and procedures to facilitate the modification and installation of the software on various hardware platforms, the establishment and operation of software baselines, and the degree to which user requirements are met. Finally, software management in the most pertinent sense requires a close linking of software engineering, validation and maintenance with the notion of RISK MANAGEMENT, defined as "the total process of identifying, controlling, and minimizing uncertain events". [Peercy et al (1987), p. 73]

By system, we refer specifically to software (vs. hardware), or programs which execute in a computer. Maintenance, then, which may at first appear to be a simplistic notion of the physical "up-keep" of a program, actually is a complex discipline, falling across the entire SOFTWARE LIFE CYCLE which can be defined as:

> The period of time that starts when a software product is conceived and ends when the product is no longer available for use, including project planning (system/subsystem requirements analysis phase), task planning (system/subsystem requirements analysis phase), task planning (system/subsystem functional design phase), software requirements analysis phase; implementation and acceptance test phase; system integration, test and delivery phase, and the operation and maintenance phase.
> [Evans and Marciniak (1987), p. 307]

In this chapter we will consider software engineering to be the discipline in which policies, procedures, and tools are used in an effort to produce good programming code, especially in terms of maintenance and support. Software management will be considered as a broader

discipline that must engage the various constituencies of managers, both laterally and hierarchically. In this chapter our focus will be on software engineering. In later chapters we will address the issue of software management.

2. An Overview of Software Engineering

Howden, of the University of California at San Diego, examines the factors that determine the complexity of the software engineer's tasks and investigates the factors that differentiate one level of task from another. [Howden (1982), pp. 318-329] His models range from the very simple, Environment I, to "a complete one including many automated tools and built around a software engineering database." The parameters of Environment IV encapsulate those of the smaller environments in such a way that one may interpret Environment IV as a composition of several instances of software programs from the less complex environments, complicated by the necessary interfaces and [presumed] developing technologies that must be programmatically captured as a given of the on-going systems effort.

Howden establishes a framework for a software engineering technology which can handle the overwhelming intricacies of large software projects without surrendering to the lack of control engendered by the demands of time and communication. The demands for control are vast, and the lack of answering these demands denies the criticality of current engineering products. The basic characteristics of a large software product include: a development time in excess of three years; a budget of over $20 million, expected systems lifetime of ten years; large staffing requirements, development of new technologies by external contractors and subcontractors, engineering of embedded concurrent systems, interactive real-time use, and, critical reliability. [Howden (1982), p. 319] These characteristics are analogous to those found in the effort to land a man on the moon.

In the model for an Environment IV project, Howden envisions a requirements phase which includes implementation of a software engineering data base containing firm details to forecast expected product behavior, costs and design criteria.

> As many tools and methods as possible store the objects which they generate and manipulate in the software engineering database. Tools that do not use the database must be compatible. Compatible tools should be able to call one another.
> [Howden (1982), p. 323]

This implies that the programming language or languages used for the project must interface with other programs and languages and that a programming language must be able to "interpret" other languages used in the project. Thus, the selection of programming language appears as a critical decision at the very outset of a software engineering project. It should be noted that this issue is even more complex since it is possible to write a language for programming some specific operations in another language. This opens the possibility for standardization to occur at several different levels.

The requirements phase employs the following tools and technologies:

The use of an automated requirements specifications tool [such as SADT, Softech Approach to System Development of Softech, the Software Technology Co.] is assumed. Both the requirements definition and specification methodologies must incorporate procedures for incremental construction and review. Formal, systematic requirements reviews are considered essential.

> The need for incremental build and review procedures in large-scale systems is partially due to the size of the requirements documents. It is not feasible to construct and deliver a 500-page specification to a customer and then ask for an opinion. Nor is it acceptable to deliver a hierarchically organized document, one chapter at a time, after the entire document has been completed. Examining the first few levels of specifications may reveal a problem that could, in the long run, render the rest of the document obsolete.
> [Howden (1982), pp. 324-325]

Here, again, is a critical point affecting the choice of a programming language. If the software is to be built incrementally and if the time span of the project is large, then a language with a broad base of government, industry and academic support would seem to be desirable. Further, the language should not only be effective, but easily teachable, and it should possess an inherent ability -- within the parameters of its environment -- to "interpret" other languages. These factors would seem to be essential elements in this phase of decision making since in a large project there will be a large number of programmers, and these programmers will be familiar with the environments and languages that they currently use.

Clearly, at this point, a "librarian" and the process of systems validation should begin. Zelkowitz [(1978), pp. 203-204] strongly argues for an early instance of documentation. In the average computer software lifecycle, maintenance is responsible for as much as seventy per cent of total costs [Zelkowitz, Shaw and Gannon (1979), p. 9], and this is the phase where documentation is key. If a librarian is established initially, the design processes are

captured from the outset, preventing duplicative design implementations, and all design implementations, as well as the ultimate product, are available to the maintenance staff. Boehm [cited in Zelkowitz (1978), p. 207] draws attention to a TRW study that revealed fixing programs in the coding phase cost twice what it does to handle the same tasks during the design process.

Requirements definition is followed by the design, programming and verification phases. Each is marked by on-going testing, increasing use of the data base to handle new implementations, and increasingly "harder" documentation of the ultimate software product, its performance and units analyses, and its verification data. [Zelkowitz (1982), pp. 321-325] Because many of the tasks inherent in a large Environment IV project are handled by contractors, it is quite easy to perceive the exponentially increasing difficulties embedded in the communications network, if anything like timely updates to systems changes are to be maintained. This gives rise to a third key factor in programming projects: a language that has proved itself able to correctly and verifiably handle complex networking is essential.

It should be remembered that software engineering is a rather new discipline. In fact the term was first used at NATO sponsored meetings in 1968-1969 [Wasserman and Belady (1980), p. 84]. Wasserman and Belady trace the development of software engineering over its first ten years in "Software Engineering: The Turning Point." Generally speaking, and unfortunately, there has always been a "software crisis". A resolution of the crisis in its current terms is one of the most engaging and difficult problems facing the United States today, if it is to maintain its pre-eminence as the leading technological and economic power in the free world.

Macro and Buxton extended Bauer's 1972 definition of SOFTWARE ENGINEERING to reflect the issues of the current crisis and the opportunities currently available to resolve it. They define software engineering as:

> practice, and the evolution of applicable tools and methods and their use as appropriate, in order to obtain -- within known but adequate resource limitations -- software that is of high quality in an explicitly defined sense.
>
> ...[And, they continue,] at a less general level of definition, software engineering is:
>
> 1. Making and maintaining software systems. The activity generally understood to include requirement specification; software system definition; software design, coding, and program testing; software system integration testing, and verification/ validation/

> certification; and software maintenance and emendation for new versions. Subsumed within this definition are the essential issues of documentation and testing strategies.
> 2. Devising tools and methods for making and maintaining software systems, and using them appropriately. For example (inter alia), special approaches and methods for specification; notations and languages for representing design ideas; software tools and utilities -- language/debug facilities, editors and the like -- to aid coding and testing; configuration management systems to assist in documentation and version control.
> 3. Managing the whole process, within economic constraints of cost and timescale. For the process of software engineering to be manageable it must be, so far as possible, a visible set of recognized activities.
>
> For this state of affairs to occur, both managers and software engineers who are party to the process must be competent in their fields of endeavor; this includes sufficient understanding of software engineering on both sides, and the possibility to communicate that understanding.
>
> [Macro and Buxton (1982), pp. 14-15]

Documentation, testing strategies, communication between software engineer and user during the design and implementation process, and maintainability are critical issues.

3. DISCUSSION

In his Principles of Software Management, Gilb formulates two principles that relate the crisis factor to software products:

> 1. If your tools can't operate in all critical dimensions, then your problems will.
>
> 2. Dynamic environments require thinking tools instead of unthinking dogmas.
>
> [Gilb (1988), pp. 19-21]

Software products must come to encapsulate factors of time and communication as well as validation and maintenance, if they are to be effective management tools. They must allow for the real situations that are developing in business. Peters in Thriving on Chaos [1987] and Naisbitt and Aburdene in Re-inventing the Corporation [1985] point out a growing trend towards entrepreneurship in the technical industry. This implies that the DoD will have to take in consideration the ability of small companies, who may be producing at state-of-art levels, to capitalize computer systems. These companies will have radically reduced amounts of money to invest in hardware and software in comparison with giants like Boeing, and these reductions will affect their ability to educate and re-educate employees in programming languages and compilers and interpreters.

It is a given that some sort of standardization of computer programming language by the DoD is inevitable and desirable in order to capture reliability and security factors in software. It is obvious that maintenance is a critical issue in conjunction with time and

technological factors for cost control. However, it is also important to examine the ways in which software engineering stands to benefit from not only the choice of a programming language, but also the choice of a management structure.

CHAPTER 7

THE MANAGEMENT CONTEXT

0. INTRODUCTION

In the previous chapters we have examined programming languages, programming paradigms, and software engineering. The selection and support of decisions made in each of these arenas is an important factor in the success of a software project. But what of the organization and effort needed to make the selections and provide the support? In this chapter we will examine a particular paradigm of management structure, matrix management, that we believe can provide a model for the management structure of large software projects. In our view it is the role of management to attend to the needs of the project, provide for effective communications between parts of the project, and monitor and control the monetary and manpower requirements of the project through the structures and policies it sets in place.

1. MANAGEMENT AND THE LUNAR LANDING

Certainly, within the history of our own century no complex technological undertaking has so captured the imagination as that of man's first landing on the moon. Fortuitously, Dr. Eberhard Rees, Director, George C. Marshall Space Flight Center, recorded the management structure that he adopted to effect the landing, in a paper given before the CIOS World Management Conference, Munich, 1972. It is precisely the sort of case needed, for as Rees points out:

Basically, project and systems management is nothing new. It is axiomatic that since the dawn of history there have been combines of human beings trying to achieve a common goal within a certain time span and with available resources... In modern times we call the executional approach to such an undertaking "Project and Systems Management." Large projects of scientific and technical nature generally involve:

a. a multitude of government agencies, industrial firms and other organizations sometimes on an international basis
b. funds in the multi-million to billion dollar category

> c. complex technology sometimes reaching beyond the state-of-art
> d. large forces of scientists, engineers, technicians and administrative personnel
> e. construction of extensive and highly specialized facilities.
>
> This type of problem became more and more common in this century and especially in recent decades to solve problems of national and worldwide importance or to pursue scientific, large scale endeavors or to meet the needs created by a rapidly expanding world population and to achieve other goals. It soon became evident that such projects, of great magnitude and complexity, had to be considered under the overall systems point of view continuously during execution. The alternative of such a concept leads inevitably to non-optimal technical solutions, cost overruns and schedule slippages which would occur to the embarrassment of the responsible country, agency or firm. Therefore, terms like... "Systems Engineering"... were introduced to describe the systems aspects that had been emphasized as an inescapable necessity.
>
> [Rees (1972), unpaginated]

The organizational form described by Rees was an innovative structure, based on general developments within DoD during the 1960's, and known within the management science community as the project matrix. [Kerzner (1984), pp. 109-126]

2. MATRIX MANAGEMENT

Essentially, a matrix structure serves as a linkage of functional teams involved in engineering, manufacturing and other technical tasks, with the various individual programs or projects that rely on technical support for implementation. A matrix structure allows scarce resources of technical knowledge to be shared over every area of an individual project as well as over the larger program itself. Rees, for example, cites the Saturn launch vehicle as an example of a "project" within the larger context of the Apollo "program". The greatest benefit of the matrix structure in project management, and DoD's seminal contribution to management science, is the evolution of a formal way for technically complex organizations to respond quickly to an equally complex and volatile environment. In the case of Apollo, the internal structure was complex because much of the technology was changing or being developed as an on-going function of meeting project specifications. The environment was complex because it was inherently subject to political fluctuations that affected budgetary allocations, and to the ability of contractors to deliver product designs within time and budget constraints.

The type of matrix that Rees describes is multi-dimensional and typifies what Kerzner [(1984), p. 126] refers to as "matrix layering... the creation of one matrix within a second matrix". At the lowest layer of the matrix there were design matrices within contractors, augmented by a "resident manager" from NASA representing Apollo's interests and

delegated the legal authority to act as NASA's agent in accepting design changes. The next layer of the matrix included particular projects at one of the three space centers where contractors and government came together to produce one unit of the Apollo program. The third layer included the individual efforts of the Marshall Space Flight Center in Huntsville [MSFC], the Manned Spacecraft Center in Houston [Houston], and the Kennedy Space Center at Cape Canaveral, Florida [the Cape]. These were tied together by a fourth layer, the Management Council, which met monthly and was chaired by the Associate Administrator for Manned Space Flight [Rees], who reported to the NASA Administrator. Thus, there was a highly complex organizational structure existing over several dimensions, and the success of the Apollo program depended on a unifying leader.

Within this structure great amounts of responsibility and authority were delegated from the top down. However, Rees recognized the importance of other elements.

> ...the assignment of all responsibility [Since "responsibility" was delegated even to the lowest level of the design matrix in NASA's resident managers within contractors, Rees is referring directly to the whole of the program "control" here.] to single organizational management structures pyramiding into a single strong personality.
> ...This prevents fragmenting that ["control"] among numerous individuals with attendant loss in time, money, manpower, and technical progress. Of course, with the ["control"], the manager must have commensurate authority to resolve technical, financial, production and other problems that otherwise require coordination and approval in separate channels at different echelons. And he must have clear, concise channels of communications flowing in all directions.
> With these tools, program management can bring to bear all the capabilities, technological, sociological, economic or whatever upon any project and systems problem however large or complex it might be.
> [Rees (1972), unpaginated]

The emphasis that Rees places on two factors inherent in matrix structure are hallmarks for the development of current software engineering and management. The first is the timely capturing of technological developments and the second is the fluidity of communications as a function integral to program management.

The first factor, time, can be understood by considering the product life cycle of current new technologically-based mechanisms. What the twentieth century has been forced to accept is a shortening of product life cycles that is quite unprecedented in the history of man. Consider transportation. For centuries man continued to develop and improve wind-powered vessels for ocean voyages, and for perhaps a century steam-powered vessels represented a marked improvement in both time and reliability in traveling from Europe to North America. But, Lindberg shortened the time dramatically, and within 10 years of his

initial flight, transatlantic air travel was relatively commonplace. The key point is that the product, a ship, that was a traditional norm for thousands of years was replaced with another, similar product for only a hundred years. The second product had a life cycle that was shortened by technological advances of subsuming attractiveness in terms of time and speed. Today, of course, travel aboard the Concorde reduces transatlantic travel from months to four or five hours. And, all of this has been done in something approaching a half century: it took roughly 50 years from Lindberg's landing in Paris to put man on the moon.

The second factor that Rees stresses is communication. If products are not to be redundant, and if the product life cycle is shortening, then even before products roll off the assembly line, management must address the issue of constructing reliable, and inviolable, networks of complex communication which cross multidimensional matrices that exist in layers and cross great geographic spans. It is of paramount importance that the controlling leadership of critical programs have at hand the latest technological information for each level of submatrix answering to its centralized authority and that the leadership at each level is able to communicate freely, accurately and quickly with other leadership centers.

3. THE ANALOGY

The management of complex software projects bears a striking analogy to the project which Rees supervised, and we believe that the management of that project is suggestive for the development of software management and software engineering.

The specific analogy can be seen more clearly by focusing on the five characteristics that Rees' identified as characteristics of large projects. Let us consider, as an example, the software that might be constructed for a project to send a team of scientists and engineers to Mars. It will surely be the case that the software on the vehicle will be complex and require for its design and production the cooperation of many agencies and corporations. Political factors could complicate the situation even more, if the mission involved several nations, Japan and the Soviet Union, for example. Clearly the amount of money spent on software will be large. The size of the project will demand large forces of scientists, engineers, technicians and administrative personnel. All of these, more or less, physical considerations would be common to both projects. The important issues, however, concern how the project would push programming technology beyond the state of the art and how extensive and specialized the support facilities must be.

It would seem reasonable to claim that a project such as a Mars mission would push the state of the programming art in several ways. At the design level, decisions would have to be made about which functions and facilities would be placed on board and which would remain on Earth. Given the length of the mission, the length of transmissions, and the possibility that the craft might have to "go it alone," the decisions of where to place certain functions and facilities will be critical. One can consider any space mission to be an extremely large-scale computer network. Our ideas about such networks and the distributed processing that they may support would be challenged by a Mars project. It is reasonable to expect that if the problems are solved, they will either push or alter the state of the programming art.

Further, the use of "intelligent systems" may well be a necessity for such a project. It may not be the case that all of the software that may be needed will be able to operate in a conventional way. Of course, much of the software will operate in a conventional way. Much of the software needed to control particular devices and analyze and communicate information may be more or less conventional, though here again it is reasonable to suppose that the project will push the state of the programming art. However, the software that will be used for mission support, for revising schedules, for helping to take care of emergencies, and for helping to examine and analyze the unexpected will not be of a conventional character. Such programming will drive the state of the programming art and place great demands upon the creativity, inventiveness and intellectual abilities of the teams of scientists, engineers, and programmers who design, build and test such software systems.

The support facilities for the required programming will be extensive. Even if one applies the rule of thumb that "the fewer programmers working on a project the better," there will still be a great many programmers because of the great number of projects. These programmers will need to share information. They will need to share information about requirements and protocols; they will need to share a library of code; they will need to share ideas and algorithms; they will need to share their creativity.

Thus, the analogy between the two projects seems strong at this level. The further characteristics of the timely capturing of technological developments and the fluidity of communication are equally strong.

The timely capture of technological developments can be viewed in two ways. The first way concerns the ability to use new technologies outside of the project. In this case one can easily imagine that advances in programming art and science will continue to be fueled by other efforts and by intellectual curiosity. Advances outside of the project could be the fuel needed to improve or complete some project within the Mars effort. Secondly, one might reasonably expect the project's programming efforts to feed on themselves. As such a project moves toward completion, more of the completed subprojects will be known. As they become known, they may well provide the keys to the success and completion of other projects. In this sense, an inverted "waterfall" would arise.

The fluidity of communication on a large software project is also an essential element in the success of the project. For example, particular programming tasks may be similar even though the subprojects to which they are attached are different. The programming required for understanding the outputs of groups of sensors may bear marked and important similarities even though the specific sensors to which the program responds are connected to vastly different parts of the physical project. Thus, the programs used to understand sensor readings from medical sensors may bear important similarities to the programs used to understand sensor readings from environmental sensors. In such a case the programmers should be able to easily communicate with each other in an effort to program efficiently and well.

The creative and technological advances achieved in the Apollo program were accomplished with the aid of computers that would be considered highly primitive today. Although today's computer are faster and more powerful and it is possible to build more intelligence into programming tools, the imperatives of constrained time and effective communications networking remain.

While these factors are not included in most formal definitions of software engineering, they must be considered in software management. It is worth noting that the term "engineering" in connection with software concepts can be misleading. Although engineering is one of the oldest professions, not all parts of engineering are equally old. This difference in age can give rise to other significant differences. Zelkowitz, Shaw and Gannon [1979, pp. 1-2] point out that when the Verrazano Narrows Bridge was completed in 1965, the project came in on time and at projected cost after a six year period of construction, yet the Alaskan oil pipeline jumped from its $900 million budget to $9 billion when it opened in 1977 [p.20]. A significant difference between the two projects is that the

technology for building a suspension bridge has been well tried by experience, while that of the pipeline was relatively experimental. Software engineering is even newer.

The anlaogy between the sort of project management used for NASA's lunar mission and the the sort of management needed for the production of large software systems is strong. It should also be noted that the demands of creativity and the economic rewards of technology transfer provide a further impetus to examine more closely the demands of software management.

4. AN ECONOMIC ISSUE

Wysocki [1988, pp. 1-46], writing in a special report for The Wall Street Journal of November 14, 1988, paints a disturbing picture of the current state of U.S. technology vs. that of Japan.

Japanese technology tactics will look familiar to anybody who remembers the U.S. Apollo moon-shot project of the 1960's. The overall Japanese effort resembles hundreds of miniature Apollo projects, with highly focused targets, severe timetables, heavy financial investment and unwavering support from the Japanese government.
[Wysocki (1988), p. R5]

Among the currently stated goals of Japan's long-range technology are heavy investments in developing Computer Aided Design to make computer chips with more than one million gates by 1994; and, by 1999, to develop AI systems to control aircraft, including collision prevention devices. The Japanese have, in point of fact, even posited a cure for cancer by the year 2005. [Wysocki (1988), p. R12]

Despite the enormous significance of these projects and the debilitating imbalance of trade concurrent with America's increasing posture as a debtor nation, this country still holds the lead in research. [Wysocki (1988), pp. 21-23] But, for how long? Erich Bloch, Director of the National Science Foundation, clearly points to Japan as a military ally and an economic adversary. He cites America's loss of basic manufacturing skills as the principal reason that devaluing the dollar has no effect on the trade deficit; there is simply nothing that America has left to sell overseas. The movement of the Japanese into more basic research will probably have a great long-term effect on whether or not the United States ultimately loses the technological war. [Wysocki (1988), pp. R32-33] Although the United States currently

holds an advantage in software, and this may in a sense be our most important export, that advantage may give way. It would, in the end, be most odd if the management experience gained in U.S. space exploration, and "exported" to Japan, would be a significant factor in undermining the competitive position of the software industry upon which further space exploration depends.

Paul Kennedy, a military historian at Yale, examined The Rise and Fall of the Great Powers from 1500 to 2000 and came to the following conclusions:

> 1. There exists a dynamic for change, driven chiefly by economic and technological developments, which then impact upon social structures, political systems, military power, and the position of individual states and empires... The intellectual breakthroughs from the time of the Renaissance onward, boosted by the coming of the "exact sciences" during the Enlightenment and the Industrial Revolution, simply meant that the dynamics of change would be increasingly more powerful and self-sustaining than before.
>
> 2. This uneven pace of economic growth has had crucial long-term impacts upon the relative military power and strategic position of the members of the states system... The world did not need to wait until Engels's time to learn that "nothing is more dependent on economic conditions than precisely the army and the navy." It was as clear to a Renaissance prince as it is to the Pentagon today that military power rests upon adequate supplies of wealth, which in turn derive from a flourishing productive base, from healthy finances, and from superior technology... the fact remains that all of the major shifts in the world's military-power balances have followed alterations in the productive balances.
>
> [Kennedy (1987), p. 439]

5. DISCUSSION

If the current facts of the American economy are somber, it is well to consider most carefully the current American ascendancy in both basic research and advanced applied development in the fields of artificial intelligence, computer design, computer integrated manufacturing, computer software and telecommunications networks. In these areas may be found the factors that will lead to the next breakthrough and perhaps the preservation of the United States' economic position.

The Secretary of Defense of the United States, Frank Carlucci, speaking at American University on November 19, 1988 [CSPAN Broadcast] remarked that the first supercomputer, which went into operation just twelve years ago, is already a "museum" piece, with current models operating a hundred times faster. The deep implication of this is that time is far too short for dawdling and experimentation in areas where the technologies have died. It may be dangerous in the long run to support conventional to the exclusion of applicative languages. If this danger presents itself as an actual debilitating state of affairs,

it may follow that its effect on the economic/ productive sector will lead to detrimental effects on the military/ power sector. The next generation of computers may well be constructed for languages devoted to artificial intelligence and knowledge systems technology. The U.S. currently leads the world in AI and knowledge based technology; that lead should be exploited.

If the matrix organizational structure is currently the most rational method for industries to function productively and, if computerized knowledge based systems are to be the instruments that provide the motive power for the technical production of matrix products, then a programming language that effectively and securely handles multi- dimensional organizational communication of ever increasing knowledge content over ever-shortening time spans must be found.

Unlike the traditional or hierarchical methods of organization, the matrix in a rather peculiar way, with its inherent and modular building blocks from simple cells [or "nodes"] to complex organisms, mirrors nature. Military organizations are a paradigmatic example of hierarchical structures, yet even here the rigidity of the hierarchy must give way to creativity in the extreme and critical circumstances that confront a fighting force in battle. When a general falls, a subordinate takes command and so on down the line. The human body with its diversity of cells is the paradigmatic example of a complex natural organism. In the case of command continuing on in battle, the cognitive units of the natural organism are overcoming the deficient unit of the hierarchical organism, by providing leadership where that has been removed by death. In the case of the human body, the circulatory system (among others) provides a natural unifying mechanism that reduces complexity into a manageable framework. Matrices are designed and modeled to behave in similar ways. [Smith (1978), pp.922-926]

Gleich in his overview of CHAOS quotes Stephen Hawking from a 1980 lecture entitled, "Is the End in Sight for Theoretical Physics?":

> We already know the physical laws that govern everything we experience in everyday life... It is a tribute to how far we have come in theoretical physics that it now takes enormous machines and a great deal of money to perform an experiment whose results we cannot predict.
>
> [Gleich (1987), p. 7]

Gleich continues that having discovered so very much about "how" the physical world functions, man is currently moving toward the exploration of new areas of knowledge

[chaos] that represent the scientific examination of the way in which man perceives the world. That is, "why" things are the way they are.

The effectiveness of the computer society will be determined in large part by its ability to construct software that will be able to reason "heuristically", to think as intuitively as the human organism does. After all, the subordinate upon whom sudden command falls in heated battle does not stop to ask "how" he will lead: this is something that he has been prepared for [or "programmed" to accomplish, if you will]. The new commander simply responds because he understands intuitively the "why", the key and really important issue in crisis decision making.

The ability to construct the sort of software that will be needed by various governmental and corporate projects will without doubt be complex and be the stock on which America's economic future depends. The success of specific projects demands some account of the management structure appropriate to such projects. Software engineering is a field too newly born to be savior. Project management, though young, has had a chance to grow and develop. We suggest that software management ought to become a focus of intellectual and practical effort in the building of large software systems. Further, we suggest that the ideas of matrix management so successfully employed in the United States lunar missions, should be used as a model for such management systems. Finally, we want to suggest that such an enterprise will bear fruit not just in the specific arena of the project in which it is used, but in the larger economic arena as well.

CHAPTER 8

SUGGESTIONS

0. INTRODUCTION

In this chapter several tentative suggestions will be made. These suggestions reflect the fact that this work is only at an initial stage and the fact that the domain itself is in flux. This latter point is in some ways unavoidable. Computers and the art and science of their programming is in constant evolution. The suggestions will focus, from the top down, on three topics: software management, software engineering, and software languages and environments.

1. SOFTWARE MANAGEMENT

Obviously, software management is a complex and many faceted task. However, that very complexity suggests that the model for such management should be taken from projects of similar complexity. In Chapter 7 we surveyed such a project, NASA's lunar project, and suggested that there is a strong analogy between it and the management of large software projects. In particular, we suggested that the strength of the analogy rested on both considerations of the size and complexity of the projects, and the needs for absorbing technological innovation rapidly and fluid lines of communication.

The basic idea of matrix management is to establish a grid-like (possibly multidimensional) authority structure in which decision making authority is pushed as far down into the organization as possible. The grid is established by looking at intersecting interests. In this case the intersecting interests would be the particular projects and the software for those project. These interests differ insofar as they respond to differing pressures, demands, and higher order interests. Let us assume that the critical resources that need to be distributed are the personnel and equipment required to produce a piece of software. Many differing projects may make demands on this resource. For example, projects concerning the medical and environmental aspects of a space mission may both make demands on the pool of software development specialists.

Within the matrix the scarce resource of the software developer is shared. However, within the group of software developers there may be several differing target projects. To continue the example, there may be program developers who are concerned with the user interface, with machine interfaces, with sensor processing, with data base construction, with communications, etc. Ideally each of these projects would be organized so that a uniformity of software can be preserved across essentially similar functions. Thus, the basic code for monitoring sensors should be essentially the same across different sensor groups, and inference procedures should be the same across several different knowledge based systems. Of course, differing systems will differ in specifics, and these specifics should be things added to common elements and should not be the focus of the work.

Within the matrix the scarce resource of program developers answer to two bosses. On the one hand, the program developer must answer to the project manager, say the environmental systems manager. On the other hand, the software developer must also answer to the function manager, say the sensor interpretation group. Further, the software developer must maintain effective communications with the developers of other related modules. What is encouraged, therefore, is the effective identification of difficulties and the development of solutions amongst those who need to know and who have the knowledge to complete the task.

The matrix system imposes certain demands on the general management of the operation. First, the general management must agree to share power and responsibility, both downward and laterally. Second, and perhaps more importantly, the general management must keep a sharp eye for commonalties. Rather than focusing on the specifics of a task, general management must focus on the commonalties and the interactions of the parts. Further, to accomplish this, the general management must operate at a somewhat abstract level, at least one level of abstraction higher than the specific task. The convergence of the search for commonality and the sense of abstraction, lead to effective use of the available resources by allowing lower level resources to come together in a cooperative and productive manner.

The matrix model differs markedly from a hierarchical model. There are three major differences. (1) In a hierarchical model specific demands are transmitted down the organization. In the matrix model abstract demands are transmitted down and cooperative groups respond. (2) In a hierarchical model each project needs its own team of software

developers, and those developers may or may not be in communication with other related teams. In the matrix models the teams are distributed across the organization, and, therefore, similar projects are responded to by similar teams who themselves are in effective communication. (3) In a hierarchical system an innovation at a lower level must be passed up one organizational line and down another before it reaches its counterpart group in another project. In a matrix system the innovation can be shared quickly since it can be spread across differing projects by individuals in the same group

To have effective matrix management two requirements must be carefully attended to. First, there must be effective and efficient communication between the various elements. Timely and relevant information must be gotten to those who need to know, and those who do not need to know must be spared the useless and confusing deluge of information. This calls for intelligent communication networks. In point of fact, such systems already to some degree exist. Second, those in charge of projects and cooperative teams must be able to understand and use the available information. This, of course, is a task for which there is no "technological fix." However, even though there is no straight forward "fix," it is possible to supply the tools that will improve the ability of the managers and team leaders to use such information. Efforts to provide such tools are just beginning. By combining some of the features of software engineering with the a computer network that supports non-linear text bases (hypertext), the ability of the mangers to understand and use the information at their disposal may be improved. Further research in computer supported cooperative work (CSCW) should also prove valuable.

With the foregoing considerations in mind, we offer the following suggestions:

SUGGESTION 1

> The matrix model of management ought to be more thoroughly investigated as a model for software management.

SUGGESTION 2

> The communications network ought to be thoroughly investigated with the intent of fostering effective communication among cooperating individuals, teams, managers and projects.

SUGGESTION 3

> The tools available for managers ought to be investigated with the intent of generating better tools that will allow for a more flexible access to information that will improve the manager's understanding of the information which he or she has available.

SUGGESTION 4

> The resources and requirements needed for effective computer supported cooperative work should be investigated with the intent of providing the tools needed for effective cooperative work.

These are suggestion for further research are predicated on the view that a matrix system of management is desirable. As we develop our suggestions in this chapter, we will attempt to show how the various pieces of the software development puzzle can be assembled in a consistent and productive way within this model.

2. SOFTWARE ENGINEERING

Within the matrix perspective on software management, what is the role of software engineering? We believe that software engineering ought to be viewed as the micromanagement of a software project. This differs form the current conception of software engineering in that discussions of this topic often view software engineering as a complete management account. For example, Fairley (1985) defines software engineering as the "technological discipline concerned with the systematic production and maintenance of software products that are developed and modified on time and within cost estimates," and claims that software engineering is "a new technological discipline distinct from, but based on the foundations of, computer science, management science, economics, communication skills, and the engineering approach to problem solving." While we do not disagree with Fairley's position our interpretation of software engineering stresses its role within a management context, rather than stressing its status as a quasiautonomous technological discipline. In brief, software engineering is one of the tools that an effective software manager should use.

Rather than investigating the details and techniques of software engineering, we will discuss two related and important aspects of software engineering, principles and tools, and their relations to the management context.

Boehm identifies seven basic principles in software engineering. These are:

1. Manage using a phased life-cycle plan,
2. Perform continuous validation,
3. Maintain disciplined product control,
4. Use modern programming practices,
5. Maintain clear accountability for results,
6. Use better and fewer people,
7. Maintain a commitment to improve the process.

[Boehm (1983)]

Of the principles identified by Boehm two require special attention in our context. These principles are the injunctions to use modern programming practices and to use better and fewer people.

The injunction to use modern programming practices is closely allied to the material in Chapter 5. Programming paradigms are at the root of modern programming practices. As Boehm notes, "The use of modern programming practices (MPP), including top-down structured programming (TDSP) and other practices such as information hiding, helps to get a good deal more visibility into the software development process, contributes greatly to getting errors out early, produces understandable and maintainable code, and makes many other software jobs easier, like integration and testing." [Boehm (1983), p. 13] At issue, of course, is what counts as a modern programming practice. We assume that modern programming practices are not fixed, that such practices are the outgrowths of programming paradigms, and that the paradigms are responses to the practical needs of computer software developers and the intellectual demands of computer scientists. We will treat the object-oriented paradigm as the most modern of such practices.

If the object-oriented paradigm is a significant advance over the other paradigms, then it would seem to follow from Boehm's fourth principle that it should be put into use. A good case from the management perspective can be advanced for this. An object can be considered as a body of code that is encapsulated in such a way that it can be addressed by other objects, can address other objects, and encapsulates its own processing. Data which are wholly internal to the object are not publicly known, and the messages that are passed between objects allow objects to exert control while promoting modularization of the overall software project. In many ways object oriented programming can be seen as an ideal of encapsulation. Each object is, in a sense, its own program. The object can be judged to be correct, accurate, or valid in its own right. Further, objects are a natural way of thinking about the world, and, therefore, are more closely tied to the design process. If

an object is needed, it can be created without risking the sorts of clashes that can cause great difficulties in other paradigms. Objects can be continually refined and, therefore, the notion of rapid prototyping can be merged with the waterfall, iterative, and incremental development models.

From the matrix management perspective the object oriented paradigm makes good sense. First, objects seem to correspond to a natural way of reasoning about problems. Where we might say about physical things "Get an object of kind K that can do task T when it encounters conditions C," we might equally say about the software "Get an object of kind K that does task T when it gets message M." Further, where we might say of ordinary physical things, "I want an object O2 that is like object O1, but has the addition feature F," we might equally say about software "I want object O2 to inherit the features of O1 with the additional feature F." Second, objects can inherit properties from other objects and this supports the reuse of those objects in similar tasks. For example, the general characteristics of sensors may, at an abstract level, be common to many sensors. A specific type of sensor could inherit those general properties and have added the properties specific to its kind. Thus, sensor objects could be reused in many different software contexts. Third, the notion of message passing encourages the development of a standard set of protocols across objects and projects. In this sense the integration and, perhaps, maintenance aspects of software development are built in at the beginning of the project. Finally, the object oriented metaphor promotes good documentation. All one needs to know about the object at one level is what messages it sends, what messages it receives and, in general, what it does. These specifications are common to all objects and might, therefore, facilitate the manager's process of acquiring, understanding, and using the information to which he or she has access.

If object oriented programming is a good paradigm, why is it not employed in all large software development tasks? There are several answers to this question. First, some programming languages do not support the object oriented paradigm. As we noted earlier although there are object oriented extensions for LISP and C, there does not appear such an extension for Ada. Second, object oriented languages are either nonstandard or are not sufficiently general. Examples here would include SMALLTALK and SIMULA. Third, there is the perception that object oriented programming would impose unacceptable overheads. Finally, object oriented programming is relatively new and has not yet established a strong following. These reasons are interconnected. For example, the paradigm may not prove its worth if it does not attract a strong following, and it might not

attract a strong following if there is not sufficient support in an acceptable language. This last point could be critical, if one considers the role of government institutions in generating technological innovation, and the fact that the government supports the imperative Ada language which does not directly offer support for the object oriented paradigm.

Thus, Boehm's principle that modern programming practices ought to be used is a bit odd. What it might really might mean, however, is not that any modern programming practices should be used, but that the modern programming practices for imperative, conventional, languages that are used for large software projects and can be handled within the current discipline of software engineering should be used. Neither LISP nor object oriented programming can satisfy those demands. However, Ada, which does not support the object oriented paradigm, comes near to being the ideal language from the point of view of software engineering with conventional languages. This points out the difficult in generating a set of principles to guide software engineering. As has been noted the the analogy of software engineering to the rest of the engineering field begins to break as one attends to the nonphysical character of software. For example, when building a bridge or a pipeline, the standard elements of the construction remain static. Bridges will have beams and pipelines will have pipes. The materials and techniques may change, but the basic elements remain. Unconfined by such physical characteristics, the elements of software construction can change. Subroutines, subprograms, libraries, modules, packages, units, functions, objects and many other elements are available to the software programmer. New elements may be added. All of this adds to the complexity of choosing and using modern programming practices, and points to the important role of the software manager even within the software engineering discipline.

The second of Boehm's principles that needs special attention is the injunction to use better and fewer people. There are three distinct aspects to this injunction. [Boehm (1983), pp.19-21] The first is the most obvious; personnel vary in their talents. In many cases, the Marine Corps attitude is the one to employ, "We are looking for a few good programmers." The second feature is the overhead of interpersonal communications. In this case the fewer the programmers, the less the overhead. Finally, there is the use of automated aids. Tools can increase the productivity of a programmer, and, thereby, reduce the number of programmers and the communication overhead.

While there is clearly a great deal that is right about this injunction, there is also something disturbing. While it is clear that communications overhead can be a significant factor in the

loss of productivity, it is not at all clear why this must be so. If the communication only serves administrative purposes, or if the communication takes longer because of the organizational structure, then it seems that it is the organization itself that causes the lack of productivity and not the number of programmers. Further, one must look at what might be lost by having fewer programmers. With fewer programmers errors, over-designing, under-designing, and missed opportunities could more easily arise. Why should programmers communicate with other programmers? Perhaps to heighten their objectivity and spark new criticism, in much the same way that these things happen in a scientific community. While it may be good to eliminate unnecessary communication overhead, there may be better ways than limiting the number of programmers. It is at this point that the software manager must decide whether a programmer is more akin to an assembly line worker or a professional scientist.

We have already taken a brief look at the idea of automated aids or tools in Chapter 6. The use of such tools to manage the software coding process including the generation of source code in a target language raises another interesting issue. If the tools are good tools and if the code they generate is good code, then what is the programmer doing? In a primary sense he is running the tool; in a secondary sense he or she is programming in some language. There is a sense in which if the tools are very well done the "programmer" need not even know the language in which he or she is programming, and, indeed, need not even know in what language the code is being generated. As Howden has noted:

> The manufacture of software is perhaps one of the most logically complicated tasks. The intellectual depth of software systems development coupled with the lack of physical restrictions imposed by properties of the product make software manufacturing intriguing in its possibilities for highly automated, sophisticated manufacturing environments. Research has begun on environments containing their own conceptual models of general programming knowledge... It has been speculated that in the future software engineers will be able to describe application programs to a system capable of automatically generating specifications and code.
>
> [Howden (1982), pp. 327-328]

An intriguing possibility! Taken together the two injunctions upon which we have focused our attention can lead to a radical re- interpretation of the software programmer If good software engineering advises that the best available tools be used, if the best available tools embody the object oriented paradigm, and if the best available tools embody the ability to automatically generate appropriate and valid application code, then the role of the programmer in software development is radically changed. The programmer is no longer a crafter of code, but an expert user of a tool. The connection between the programming

language and the programmer is, in a sense, severed. The programmer with such tools may, therefore, function at a higher, more natural, level of abstraction without needing to attend to the syntactic complexities of the language in which the application is finally coded. This is not, however, surprising. It represents simply another moment in the evolution toward higher level languages. Rather than a traditional higher level language being used with a compiler to generate the low level instructions to the processor, a new generation of tools may operate at an even higher level and a translator may then convert the tool's specifications into a higher level language which in turn may be compiled.

With the foregoing considerations in mind, we offer the following suggestions:

SUGGESTION 5

The object oriented paradigm of programming ought to be fully explored.

SUGGESTION 6

The use of computer programming tools ought to be investigated with the intent of establishing clear relations to both the object oriented paradigm and the ability to automatically generate code in a higher order language.

SUGGESTION 7

The roles of the software designer and the programmer ought to be investigated with the intent of isolating common concerns and interests.

SUGGESTION 8

The role of communication among programmers ought to be investigated with the intent of identifying those sorts of communications which are productive and those which are unproductive.

SUGGESTION 9

The relations between the principles and practices of the disciplines of software engineering and project management ought to be carefully investigated

SUGGESTION 10

The needs for creativity and productivity in programmers ought to be critically examined.

These suggestions for further research are predicated on the idea that what is right about the idea of software engineering is the emphasis upon using modern techniques and tools. We

believe, however, that the discipline does not go far enough in this direction. The suggestions that we offer push both of these ideas within the matrix management context.

3. Programming Languages and Environments

The account of software management that we have been developing points to a novel way of thinking about programming languages and environments. If the programmer is thought of as similar to a scientist, and if the programmer is a scarce resource in a matrix system, and if programming aids and tools are well developed, then the choice of a programming language and environment becomes much less important. This is not to say that the choice of a programming language is not important. However, the choice of the language is placed in an environment where other considerations may be equally, if not more, important.

Let us begin this discussion by examining why there are so many different programming languages. One can readily point to four factors: improvements in processors, improvements in paradigms, improvements in environments and tools, and the specialization of programming tasks.

It is readily apparent the processors have improved greatly over the past two decades. Increased speed, increased word size, augmented capabilities, decreased power consumption, and decreased cost are readily apparent. All of these factors combine to allow those who design and build languages and environments to implement more easily and effectively ideas and constructs which with less capable processors would remain dream and desire. One need only to recall what it was like to run LISP on a PDP-11 under RSTS and look at a Symbolics or Texas Instruments LISP machine to recognize the difference. Similarly, C in its own UNIX environment has come to be recognized as a powerful system, and has led to the evolution and development of the computer workstation. The future holds even more promise. Even as physical limitations begin to affect the development of better processors, new architectures begin to evolve. Multiprocessor machines, parallel processor machines, and other objects of wonder and splendor open new vistas to the language crafter. Although languages like LISP and C will probably move into these new environments, their form and function will probably be much different. Equally probably new languages may emerge. In any case, the point we believe is clear. Languages are not static. Language development responds to the state of the processor art. As long as processor development continues, it is reasonable to expect programming languages to develop.

It should also be clear that programming paradigms change over time. The changes of paradigm reflect both the intellectual development of computer programming and the ability of the language crafters to build support for a paradigm into a language. BASIC was a wonderful language. It was criticized for not supporting a structured programming paradigm. New BASIC arose in response to that criticism. Classic LISP did not support object-oriented programming. LISP with FLAVORS is a virtually seamless environment in which such programming is supported and encouraged. C did not support the object-oriented paradigm; C++ is the response. If it were not for the government's involvement with Ada, one might well think that OO-Ada (Object Oriented Ada) would soon appear. There is, of course, no reason to think that the story ends with the object-oriented paradigm. New paradigms, perhaps tailored to particular classes of problems, may well arise. As they do old languages may evolve to support them, and new languages may arise to enforce them in much the way that PASCAL and MODULA-2 enforce structured programming and SMALLTALK enforces object-oriented programming.

Perhaps the most dramatic changes of language will occur with the improvement and development of programming environments and tools. As we noted in the chapters on LISP and C, it is often the environment that captures the programmer. The facilities of the LISP and C environments allow the programmer to concentrate on the task at hand and quickly and efficiently produce the needed code. This is especially true of a LISP environment on a LISP machine. The programmer can build his own tools and tailor the environment to his needs and preferences. More importantly, the language environment and machine function in harmony to allow the programmer to build a language in which problems can be solved. By allowing a measure of abstraction, generality and efficiency can be gained. All of these things taken together point out that the development environment is an important factor in selecting a language.

As programming tools and aids evolve the direct contact with the programming language may begin to disappear. Such tools may allow the programmer to either break the programming task down into parts that are sufficiently small and standard that existing libraries of routines can be employed, or may allow the programmer to build the program specifications in such a way that a translator will be able to translate the specification into the target language. Both approaches currently have their problems. In the former the programmer is left at some point to grapple with the language itself, and in the latter the programmer might find the translated code for the target language indecipherable. Although

these are serious problems, they may not be insurmountable. If they can be overcome, the contact of the programmer with the programming language will be stretched thinner and thinner.

The continued improvement of programming tools and environments, may lead the manager to base his or her decision of which programming language to use on the presence or absence of certain features in the tools and environments more than on the characteristics of the languages. The decision, of course, is still affected by external factors. Ada will be used on the Space Station. However, much might be learned by examining the environments and tools for other languages such as LISP and C in an effort to build better tools for Ada. After all, given that Ada is a sufficiently universal language, it can be made to look like other languages.

The final point to be made about programming languages and environments is a very simple one: there is no language that allows the production of code to be easy within every paradigm for every application. Programming languages, like all other human artifacts, are imperfect and limited. Thus, it should not be surprising that some programming languages perform better at some tasks than others. As was noted in the chapter on Ada, Ada is not the savior. Likewise neither LISP nor C can fulfill that role. It is reasonable to expect that with the increasing specialization of programming tasks new languages and extensions to old languages will evolve that will handle such tasks more easily and efficiently.

With the foregoing considerations in mind, we offer the following suggestions:

SUGGESTION 11

> Various programming languages ought to be investigated with the intent of identifying the features of the languages where each language has its greatest strength.

SUGGESTION 12

> Various programming environments and tools ought to be examined with the intent of identifying those features that contribute to both productivity and efficient management.

SUGGESTION 13

> Various programming languages should be investigated with the intent of identifying the tasks to which those languages are particularly well suited.

SUGGESTION 14

The concepts and requirements for good programming environments should be studied with the intent of identifying common features which any good programming environment should have.

SUGGESTION 15

The concepts and requirements for good programming tools should be studied with the intent of identifying common features which any good programming tool should have.

SUGGESTION 16

The automatic generation of program code from higher level descriptions and specifications should be carefully and critically examined.

SUGGESTION 17

The role and importance of decisions about programming languages should be carefully and critically examined.

SUGGESTION 18

The conditions under which new programming languages arise and the conditions under which old languages are modified should be carefully studied.

These suggestions are predicated on our belief that decisions about programming languages ought not to drive the software development process. Rather we believe that it would be reasonable to place other software development factors ahead of that decision and then select the best language, environment and tools for the job.

4. CONCLUDING COMMENTS

In this brief preliminary report on the languages and management of software development, we have attempted to highlight some of the factors that should be examined, if good judgments are to be made. The list of suggestions is in no way exhaustive. Many more suggestions could have been advanced. For example we have not explicitly considered the problem of knowledge acquisition. This is a crucial area in the development of any viable software. Another omission is the potential conflict between AI efforts and more conventional programming efforts. This issue is of great significance when a decision must be made about whether the AI components of a project like the Space Station should be exempt from the Ada edict. More omissions can be added to the list.

We have attempted to present the results of our inquiries fairly, but recognize that our own biases may show through. However, we have also attempted to be clear that the three languages with which we were concerned each have their strengths and weaknesses.

A final note on the evolution of this report is in order. The report began life as an effort to compare LISP, C, and Ada. However, we quickly realized that no such comparison can be meaningful without a context for the comparison. For example, one might set out to compare the logical features of the languages, or their programming features, or their size, or... We decided to approach the comparison from the point of view of a manager who is attempting to produce some rather large and complex piece of software. We believe this is a rather novel approach that can be developed. This leads to a final suggestion.

SUGGESTION 19

> The needs and demands on programming languages, environments, and tools ought to be carefully and critically investigated with the intent of determining the guidelines by which the goodness of such objects can be judged.

CHAPTER 9

CONCLUSIONS ABOUT PROGRAMMING CULTURES

0. INTRODUCTION

This chapter brings together the examinations of the previous chapters by focusing on the idea of a programming culture. It will be argued that the centrality of the notion of a programming paradigm within the programming culture is an important part of understanding and managing the development of software.

1. PROGRAMMING CULTURES

A programming culture is a collection of practices and attitudes that surround programming languages and paradigms. The culture is the tangible means by which programming skill and knowledge is transmitted. The transmission of such skill and knowledge may take place either in the culture or between cultures. Transmission within a culture allows new members to be added to the culture, and transmission between cultures allows ideas to pass from one culture to another. The differentiation of a culture into a practice / attitude component and a language / paradigm component allows for two methods of comparing cultures.

The first way in which cultures may be compared is in terms of internal strength. The internal strength of a culture is given in the degree to which practices and attitudes are shared by the members of the culture and the degree to which individuals outside of the culture recognize practices and attitudes as being associated with a specific culture. In general the idea is that stronger cultures will have common practices and attitudes that outsiders will quickly and accurately recognize as characteristic of that culture. The second way in which cultures can be compared is in terms of their content as specified in both the programming language and programming paradigm. This second means of comparison allows for a concept of family resemblance among cultures, and indicates ways in which the transmission of cultural ideas can take place. The idea here is that two cultures may be similar either because they share a common language or because they share a common

paradigm. Thus commonality of language may encourage members of one culture to explore the paradigms of another, and commonality of paradigm may encourage exploration of new languages. This report has focused on the second means of comparison, although many of the suggestions of the previous chapter concern the first means. It should be clear that the second way requires a conceptual analysis, while the first way requires a sociological analysis.

A hypothetical example may help to clarify the ways a culture operates. Consider a group (G) of programmers that are at work in organization (O). The G in O may or may not be a culture. For example, G may may be disparate in term of attitudes, practices, languages, and paradigms. In this case all that would hold G together is the fact that they work as programmers in O, and this would be so even if the members of G were working on the same task. Now suppose that the members of G are working in Pascal and within the data hiding paradigm. In this case a programming culture would exist within O such that a commonality of practice and attitude should be associated with the members and such that new members could be brought into G's culture (C).

The relation between the practice / attitude component of C and the language / paradigm component of C may or may not be causal and if causal may be such that either component is the cause of the other. These are matters for more detailed empirical inquiry. All that is required in this conceptual analysis is that there is an association of one with the other. The internal strength of this C in O is specified in terms of the strength of the practice and attitude component of C. The language and paradigm component of C points in another direction. Each aspect of this component links the members of this culture (G1 in O1) to other groups in O (G2 in O1) and to other groups in other organizations (G3 in O2). These links are important. In the hypothetical example the C of G in O is linked to other Pascal cultures and other data hiding cultures. These links provide passage ways for new ideas that may ultimately be embodied in new practices. Thus the existence of the culture makes it possible for the members of the culture to leap the boundaries of the organization. These boundaries may either be in terms of the parts of a single organization or may be between organizations. In either case the culture creates a situation in which there is a dual allegiance to both C and O. This further allows for the infusion of new ideas into O. For example, given the culture associated with Pascal programmers it would be possible for the hypothetical C in O to become interested in the object oriented paradigm and to restructure itself to become an object oriented Pascal culture. Alternatively, the interest in data hiding may lead to a closer allegiance with the cultures that surround Modula-2 or Ada. The paths

of such links and transitions are important from both a managerial and conceptual point of view. A quick conclusion that can be drawn from this examination is that some cultures will be unlikely to make certain transitions. For example, it would be highly unlikely that an internally strong Fortran culture would quickly move to an object oriented C culture, or that the internally strong BASIC culture of the programmers of commercial programmable controllers would quickly embrace a data hiding LISP culture.

This report has discursively and implicitly developed the theses that:

- the programming paradigm is the central, but not exclusive, element of a programming culture
- the object-oriented paradigm facilitates the production of programs in which diverse participants are required
- the issue of the language in which programming takes place is important but secondary.

Each of these these will now be briefly examined.

2. The Centrality of the Programming Paradigm

Two general arguments support the centrality of the programming paradigm in both the understanding of software development and its management.

The first argument is wholly conceptual. A programming language is a language in which instructions are given to a mechanical device to perform a set of specified operations. The higher the level of the programming language the more the language departs from the explicit instructions about the direct operations of the mechanical device. Languages such as Ada, C, and LISP are higher level languages. However, there are special cases. For example, on a LISP machine only LISP exists as a programming language. In general, however, higher level programming languages are translated (interpreted or compiled) into lower level languages until the instructions are in a form that the mechanical device can execute directly. With this in mind, higher level languages do two things. First, they cluster together lower level operations into more humanly meaningful and commonly used instructions. Second, the languages provide constructs or templates that the programmer uses to create a program. The ability to do the first task is a precondition of being able to do the second. In this sense, it is a necessary condition for the existence of higher level programming language. However, the differentiation of programming languages and their evolutionary development is most tightly associated with the second task. The former task

requires that an upper level instruction have the ability to become a specified cluster of lower level operations. Nothing new in the way of conceptual structure needs to be added, and all added instructions are of the same type. Each added instruction is such that the higher level symbol simply and directly stands for a collection of lower level operations. The second task, however, alters the situation by providing new structures. These new structures are, of course, linked to collections of operations, but they also create programming structures that are distinct from those which are only names for collections of lower level operations. This difference points to the centrality of the programming paradigm in the development, implementation, and design of higher level languages.

The second argument for the primacy of paradigms rests on the way in which programs are designed and coded. The paradigm establishes in a more or less definitive manner the way in which the intention of the program is implemented in code, the restrictions on coding, and the possibilities of coding for the program. The paradigm acts as a template through which the programmer breaks the intention of the program into smaller and smaller units. Thus, the original intention, specified as the functions and operations that the program should perform as well as some of the ways in which those functions and operations should be performed, are broken into smaller intentions which themselves are to be decomposed. At the termination of this process the units as coded are combined. This classical Aristotelian methodology of analysis and synthesis will be accepted in the following discussion.

In the analysis phase, a top-down approach to program design and code is desirable. It can be fairly assumed that the top includes the specifications of the intent of the program — what the program is to do — as well as some notion of the process of the program — inputs, processes, outputs. It is important for both the general process of analysis and the determination of the termination of analysis to specify the kinds of units into which the decomposition will occur. If such units are not specified, the decomposition may become chaotic and the analysis phase of program development unmanageable. This would be so since the lack of a specific bottom level unit would make the determination of the terminal point of analysis undefined, and the higher level decompositions either unstructured or unprincipled. The higher level decompositions are unstructured if the decomposed units do not have a close resemblance, and is unprincipled if there is a resemblance among some of the units but the resemblances are accidental. In any case, the lack of definition, structure, or principle will create problems in both the understanding of how the intention of the program is being satisfied, and the understanding of the processes in the program.

Programming paradigms provide a way of avoiding these difficulties; they provide both the means for analysis and the target for decomposition. The means are specified in terms of the intention of each level of analysis such that each level of analysis can terminate in some form of pseudocode that resembles the structure of the lowest level of analysis. For example, in an algorithmic paradigm each level would terminate in the specification of an algorithm such that each step in the algorithm is either a primitive of the program or points to another algorithm. Alternatively, in an object-oriented paradigm each level would consist of a collection of types of objects each of which was either a primitive type of object or was a type of object that can be derived from other objects. The process of analysis stops in the first case when the final algorithm is specified and in the second case when all of the object types are specified. In each case it is the existence of the paradigm that makes the process of analysis intelligible and manageable. It should be noted that in any program the analysis phase may require the use of several paradigms. This does not argue against the centrality of programming paradigms, but does emphasize the fact that the paradigm must be matched to the intent of the program or program unit.

In synthesis the procedure is a bit different, but is still highly dependent on the programming paradigms used. In synthesis the units identified in analysis and coded in the target language by the programmer are assembled. The culture's practices are most clearly apparent in the coding process. Those practices, however, are closely connected to the way in which the paradigm is implemented in the desired language. Hence, the programming paradigm has a central place in synthesis.

Both the conceptual argument and the design argument support the idea that programming paradigms play a central role in the programming culture. These arguments indicate both the need for paradigms and the consequences of their use or disuse. In brief, programming paradigms are central to the programming culture and to the actual construction of programs because they supply the conceptual tools through which the design and development of computer programs becomes defined, structured, and principled.

3. The Value of the Object-Oriented Paradigm

The object-oriented paradigm provides a modern conception of programming within which it is possible to make the good use of the human experts who are a party to the development of the computer program. The interaction of multiple experts in the design and development of a sophisticate computer program is essential. The program, in one way or another, must

encode expertise. The expertise may be either directly encoded, as in knowledge based programming, or indirectly encoded, as in other traditional programs. In any case the program must have the same content as is expressed in the the original intention of the program. The computer program does not exist for its own sake, but for the sake of something else. The program might provide a way of checking sensors, guiding a ship, taking notes, or processing a check list. Whatever it is that the program does makes the program connect to some sort of human expertise. Whether the expertise is about the way in which Newton's methods can be used to find roots, or the expertise about how water is to be purified on Space Station Freedom, the same point is clear; all useful computer programs imbed human expertise.

Once it is acknowledged that it is human expertise that drives the need for computer programs and drives the specification of the intentions of the program, it becomes much more important to evaluate the ways in which the task of programing approaches the fields of specific expertise. The recent past has clearly shown that the more difficult a program is to understand and the less that it does what the expert in the field wants done, the less it will be used. The beauty of a well coded program may indeed be an important value consideration, but more important is the participation of the field experts in creating a program which can actually be used for the problem that the field expert wants to use it for.

This need to address the concerns of the field expert means that some medium of exchange must be developed that allows the field expert and the programmers to interact. While it may be true that both the field expert and the computer scientist are both scientists, it also appears to be equally true that the paradigms with which they operate are very different. In order to provide a successful exchange between the two parties some means of communication must be developed. One of the most significant way in which this can occur is through a shared paradigm. It should be noticed that although programming paradigms are implemented in programming languages, the programming paradigm by itself is a template that can be filled in in different ways. In analysis, for example, the decompositions are filled in with some form of pseudocode that resembles real code constructed in the manner of the programming paradigm. Ideally the pseudocode should be as intelligible to the field expert as it is to the programmer. As the analysis process continues and as decomposition proceeds this relation often becomes more imbalanced. The decomposition leads to the actual code that will be the instructions of the program and it is this with which the field expert may be unfamiliar. However, if the paradigm of the programmer and the field expert represents a shared cognitive structure, then the

communication at even the lower levels can be facilitated by virtue of that common structure.

Of the programming paradigms examined in this report the one that comes closest to providing a common structure is the object oriented paradigm. Within this paradigm it is possible for the field expert to clearly establish the kinds of objects with which the program should deal, the kinds of data such objects will need, and the kinds of operations that objects will need to perform in order to make the resemblance between the programmed objects and the real world objects as close as possible. From the programming point of view, the object-oriented paradigm, if well supported in a language, provides a mechanism whereby code can be naturally structured, changes can be incrementally made, code can be easily reused through inheritance, and libraries of validated objects can be made available for reuse. The advantages for both the field expert and the programmer are clear. From the manager's perspective there are even more dramatic advantages. Since there is a common cognitive structure and since the division of labor between programmer and field expert is clear, the manager is better able to allocate both of these scarce resources. The confusion that can easily exist between the specifications for the program and the coding of the program can be alleviated. The confusion can be alleviated because the manger will be better able to determine whether or not a problem is the result of the lack of field expertise in object type or because of some problem connected with the program itself. If it is former, then the field expert must provide the clarifications. If it is the latter, the programmer must revise the program.

Further, the object-oriented paradigm points the way in which the development of programming can evolve. By concentrating on common cognitive structures, the field expert will be able to select the structure and fill in the template with the explicit knowledge and expertise that is needed to make the program useful. On the other hand, the programmer will not need to guess about what the field expert means or wants; the filled in template is a guide. Further, the programmer will be free to create new structures and templates. These structures and templates will in fact be the articulation of new paradigms and these paradigms will draw on the computer scientist's expertise in the creation of computer languages and computable paradigms. These are desirable consequences from a managerial point of view since there is both a clear division of labor and a clear medium of exchange. These two ideas are central components of the matrix management theory and practice that have proved their worth in space exploration and in the operation of NASA.

The extension of these sorts of principles to the programming craft should provide similar benefits.

The object-oriented paradigm is a first step toward building computational environments that allow for shared paradigms between programmer and field expert. As such the paradigm is the most desirable of the modern paradigms, but is also one that will surely given way to others. The object-oriented paradigm is the first step toward making the process of building a program a tool driven activity in which the field expert runs the tool built by the computer scientist.

4. The Programming Language Issue

The issues that once surrounded the debates about programming languages will begin to fade as new programming tools are developed. The essential question will no longer be "In what language is this proposed program to be implemented?" In its place two new questions will become important. The first is, "What are the tools that can be used to fabricate this program?" This question will be the one that the knowledge centers of field expertise will need to have answered. The second question will be "What are the languages with which better tools can be constructed?" This second question is, in a sense, a question about what programing languages are good languages for making new languages. Each new programming tool that is developed is a new programming language, a programming language that operates at a level higher than the language in which it is programmed. Just as higher level programming languages in the past have evolved new constructs, so shall the languages embodied in programing tools. Rules, frames, scripts, objects, and scenes may all be new structures layered onto an existing programming language, or may be constructs that built into tools that can generate code in a desired programming language.

Programming languages may, therefore, begin to be understood in a much different way as the tools for programming evolve. The tools will become the prime means for field experts to interact with the computing machine and the programming language that actually produces the code will become more and more hidden. Further, the programmer's work will become less oriented toward the coding of the field expert's knowledge and skill and more strongly oriented to the production of tools through which the expert will be able to express that knowledge in a machine computable manner.

This emphasis on tools and diminished emphasis on languages is speculative. However, it is well grounded on the historical development of programming languages and environments. From BASIC and LISP to Ada and C, programming languages have been generated to make certain kinds of operations easier for the programmer. The evolution of these languages have continually added programmer tools such as structures, libraries, modules, objects, editors, debuggers, and analyzers. Each of these has made the language an even more efficient tool for producing code. The evolution of tools for the field expert is simply an extension of that evolution.

5. CONCLUDING COMMENTS

The languages Ada, LISP, and C all exhibit advantages and disadvantages. Each of these languages is part of a programing culture in which practices and attitudes develop around the languages and the paradigms that they support. It has been argued that it is the programming paradigm that is both central to the programming culture and to the construction of good and useful programs that are well informed by field experts. It has been suggested that the evolution of the programming craft will be in the direction of the development of tools that facilitate the transmission of knowledge and skill from the field expert to the programming culture and that this will be a function of the identification of the cognitive paradigms needed to facilitate that transfer. With this as a background, the question of which language is the best language, or which language is the language that should be used becomes secondary. The new questions should focus on which languages best support the coding of the intentions of the program and which languages best support the tools needed to accomplish both the transfer of expertise and production of code.

BIBLIOGRAPHY

ABRAHAMS, P. (1978) LISP / Session IV: Discussant's Remarks. In WEXELBLAT (ed.) (1981), 191-194.

AMOROSA, S. and INGARGIOLA, G. (1985) ADA: An Introduction to Program Design and Cooling. Pitman, Boston.

BACKUS, J. (1978) "Can Programming be Liberated from the von Newman Style? A Functional Style and its Algebra Programs." In HOROWITZ (ed.) (1987), 174-201.

BARNES, J.G.P. (1980) "An Overview of Ada." In HOROWITZ (ed.) (1987), 426-427.

BERK, A.A. (1985) LISP: The Language of Artificial Intelligence. Van Nostrand Reinhold Company, New York.

BOEHM, B.W. (1983) "Seven Basic Principles of Software Engineering." The Journal of Systems and Software . 3,3-24,3-24.

BOOCH, G. (1987a) Software Components with ADA: Structures, Tools, and Subsystems. The Benjamin/Cummings Publishing Company, Inc., Menlo Park, California.

BOOCH, G. (1987b) Software Components with ADA 2nd ed. The Benjamin/Cummings Publishing Company, Inc., Menlo Park, California.

BISHOP, J. (1986) Data Abstraction in Programming Languages. Addison-Wesley Publishing Company, Inc., Menlo Park, CA.

BROMLEY, H. and R. LAMSON (1987) LISP Lore: A Guide to Programming the LISP Machine, 2d ed. Kluwer Academic Publishers, Boston.

BROOKS, R.A. (1985) Programming in Common LISP. John Wiley & Sons, New York.

BUNYARD, J.M., Maj. Gen., USA, Assisstant Deputy Chief of Staff for RD&A (1987) "Mission Critical: Computer Resources," Program Manager, May-June 1987, 30-34.

CARLYLE, R.E. (1986) "Putting Spurs to Ada," DATAMATION, 32,17,30-31.

CHAPIN, N. (1987) "The Job of Software Maintenance." P:CSM (1987), 4-12.

CHARNIAK, E. et al. (1987) Artificial Intelligence Programming, 2d ed. Lawrence Erlbaum Associates, Publishers, Hillsdale, NJ.

CHIRLIAN, P.M. (1984) Introduction to C. Matrix Publishers, Inc., Beaverton, Oregon.

CORNISH, M. (1987) "Rapture, Ecstasy and LISP," Computerworld, XXI, 43, October.

CROSS, F. (1987) "An Expert System Approach to a Program's Information / Maintenance System." P:CSM (1987), 120-126.

DALEY, E.B. (1979) "Organizing for Successful Software Development," Datamation, December, 1979,107-116.

ELIOT, L.B. (1987) "The Power of Ada," Journal of Information Systems Management, 4,4,27-29.

EVANS, J., Jr. (1981) "A Comparison of Programming Languages: Ada, Pascal, C." In FEUER & GEHANI (eds.) (1984), 66-94.

EVANS, M.W. and J. MARCINIAK (1987) Softeware Quality Assurance and Management. A Wiley-Interscience Publication. Basil Wiley & Sons, New York.

FAIRLEY, R.E. (1985) Software Engineering Concepts. McGraw-Hill Book Company, New York.

FEUER, A.R. (1982) The C Puzzle Book. Bell Laboratories, Incorporated, Prentice-Hall, Inc., Englewood Cliffs, NJ.

_____ (1984a) "Methodology for Comparing Programming Languages." In FEUER & GEHANI (eds.) (1984), 197-208.

FRENZEL, L.E., Jr. (1987) Understanding Expert Systems. Howard W. Sams & Company, Indianapolis, IN.

GEHANI, N.H. (1983) "An Early Assessment of the Ada Programming Language." In FEUER & GEHANI (eds.) (1984), 116-141.

GILB, T. (1988) Principles of Software Engineering Management. Addison-Wesley Publishing Company, Reading, Massachusetts.

GILDER, G. (1988) "You Ain't Seen Nothing Yet," Forbes, 141,7,89-93.

GLASER, G. (1984) "Managing Projects in the Computer Industry," Computer, October,1984, 45-53.

GLEICK, J. (1987) CHAOS: Making a New Science. Viking, New York.

HAWKING, S.W. (1988) A Brief History of Time: From the Big Bang to Black Holes. Bantam Books, Toronto.

HIBBARD, P., A. HISGEN, J. ROSENBERG, M. SHAW, and M. SHERMAN (1983) Studies in Ada Style 2d ed. Springer-Verlag, New York.

HOARE, C.A.R. (1981) "The Emperor's Old Clothes." In SAIB & FRITZ (eds.) (1983), 487-495.

HOFSTADTER, D.R. (1985) Metamagical Themes: Questing for the Essence of Mind and Pattern. Basic Books, Inc., Publishers, New York.

HOROWITZ, E. (ed.) (1987) Programming Languages: A Grand Tour, 3d ed. Computer Science Press, Rockville, MD.

HOWDEN, W.E. (1982) "Contempory Software Development Environments," Communications of the ACM, 25,5,318-329.

HOWES, N.R. "Managing Software Development Projects for Maximum Productivity," IEEE Transactions on Software Engineering, SE-10,1,27-35.

HUGHES, D. (1981) "Next-Generation Defense Programs Will Increase Use of Ada Language," Aviation Week & Space Technology, 128,13,60-75.

KAISLER, S.H. (1986) INTERLISP: The Language and Its Usage. John Wiley & Sons, New York.

KELLEY, A. and I. POHL (1984) A Book on C: An Introduction to Programming in C. The Benjamin/Cummings Publishing Company, Inc., Menlo Park, California.

KELLEY, A. and I. POHL (1988) TURBO C: The Essentials of C Programming. The Benjamin/Cummings Publishing Company, Inc., Menlo Park, California.

KENNEDY, P. (1987) The Rise and Fall of the Great Powers: Economic Change and Military Conflict from 1500 to 2000. Random House, New York.

KERNIGHAN, B.W. and D.M. RITCHIE (1988) The C Programming Language, 2d ed. Bell Telephone Laboratories, Incorporated: Prentice Hall Software Series, Prentice-Hall, Englewood Cliffs, NJ.

KERZNER, H. (1984) Project Management: A Systems Approach to Planning, Scheduling and Controlling, 2d ed. Van Nostrand Reinhold Company, New York.

LABICH, K. (1988) "The Seven Keys to Business Leadership," Fortune, 118,9,58-70.

LEDGARD, H. (1983) ADA: An Introduction, 2d ed. Springer-Verlag, New York.

LEDGARD, H.F. and A. SINGER (1982) "Scaling Down Ada (Or Towards a Standard Ada Subset)." In SAIB & FRITZ (eds.) (1983), 483-486.

LISKOV, B. (1978) Chairman: LISP / Session IV. In WEXELBLAT (ed.) (1981), 173-197.

MACRO, A. and J. BUXTON (1987) The Craft of Software Engineering. Addison-Wesley Publishing Company, Reading, Massachusetts.

MANUEL, T. (1986) "What's Holding Back Expert Systems?" Electronics, 59,28,59-65.

_____(1987) "TI's LISP Machines Boost AI Performance Fivefold," Electronics,60,23,110.

McCarthy, J. (1960) "Recursive Functions of Symbolic Expressions." In HOROWITZ (ed.) (1987), 203-214.

_____(1978) "PAPER: HISTORY OF LISP." Session IV. In WEXELBLAT (ed.) (1981), 173-185.

McCARTHY, J., P.W. ABRAHAMS, D.J. EDWARDS, T.P. HART and M. LEVIN (1965) LISP 1.5 PROGRAMMER'S MANUAL (Reprinted from MIT Press Cambridge, Mass. 1965.). In HOROWITZ (ed.) (1987), 187-211.

MUSA, J.D., A. IANNINO and K. OKUMOTO (1987) Software Reliability: Measurement, Prediction, Application. McGraw-Hill Series in Software Engineering and Technology. McGraw-Hill Book Company, New York.

NAISBITT, J. and P. ABURDENE (1985) Re-inventing the Corporation. Megatrends Ltd.: Warner Edition, New York.

NORDWALL, B.D. (1987) "Government Agencies Promote Business, Industry Access to Ada," Aviation Week & Space Technology, 127,46,91-93.

OSBORNE, W. (1987) "Building and Sustaining Software Maintainability." P:CSM (1987), 13-23.

PEERCY, D.E., Maj. E. Tomlin and Maj. G. Horlbeck (1987) "Assessing Software Supportability Risk: A Minitutorial." In P:CSM (1987), 72-80.

PETERS. J.F. III and H.M. SALLAM (1986) Compleat C. A Reston Book: Prentice-Hall, Englewood Cliffs, New Jersey.

PRATT, T.W. (1984) Programming Languages: Design and Implementation. Prentice-Hall, Inc., Englewood Cliffs, NJ.

PRESSMAN, R.S. (1982) Software Engineering: A Practitioner's Approach. McGraw-Hill Book Company, New York.

PROCEEDINGS: CONFERENCE ON SOFTWARE MAINTENANCE - 1987 (September 21-24, 1987). Computer Society Press of the IEEE, Washington, D.C. (Hereafter, P:CSM).

REES, E., Director, George C. Marshall Space Flight Center, NASA (1972) "Project and Systems Management," CIOS World Management Conference XVI, Munich, Section 6, Topic 4.3.

RICHARD, D.C. (unpublished paper) "Crosstraining from Conventional to LISP Programming." General Reasearch Corporation, Huntsville, Alabama

RITCHIE, D.M. (1984) "Turning Award Lecture: Reflections on Software Research," Communications of the ACM, 27,8,758-760.

SAIB, S.H. and R.E. FRITZ (1983) The Ada Programming Language: A Tutorial. IEEE Computer Society Press, New York. (Hereafter SAIB & FRITZ).

SAMMET, J.E. (1986) "Why Ada is Not Just Another Programming Language," Communications of the ACM, 29,8,722-732.

SCHNEIDER, H.J. (1984) Problem Oriented Programming Languages. John Wiley & Sons, New York.

SHAW, M, G.T. ALMES, J.M. NEWCOMER, B.K. REID, and W.A. WULF (1981) "A Comparison of Programming Languages for Software Engineering." In HOROWITZ (ed.) (1987), 209-225.

SHUMATE, K. (1984) Understanding Ada. Harper & Row, Publishers, New York.

SIMON, H.A. (1986) "Whether Software Engineering Needs to Be Artificially Intelligent," IEEE Transactions on Software Engineering, SE-12, 7 726- .

SLATER, K. (1984) Portraits in Silicon. The MIT Press, Cambridge, MA.

SMITH, H.R. (1978) "A Socio-Biological Look at Matrix," Academy of Management Review, 3, October, 1978, 922-926.

STEELE, G.L., Jr., with contributions by S.E. FAHLMAN, R.P. GABRIEL, D.A. MOON, and D.L. WEINREB (1984) COMMON LISP: The Language. Digital Press, Bedford, NY.

STROUSTRUP, B. (1987) "What is 'Object-Oriented Programming'?" ECOOP '87: European Conference on Object-Oriented Programming, Paris, France, June 15-17, 1987, Proceedings. (Bezivin, J., P. Cointe, J.-M. Hullot, and H. Lieberman (eds.)) Lecture Notes in Computer Science (276), Goos, G. and J. Hartmanis (eds.), Springer-Verlag, Berlin.

SUYDAM, W.E. Jr. (contrib. ed.) (1986) "AI Becomes the Soul of the Soul of the New Machines," Computer Design, 25,55-70.

THOMPSON, K. (1984) "Turning Award Lecture: Reflections on Trusting Trust," Communications of the ACM, 27,8,761-763.

TOURETZKY, D.S. (1988) "How LISP Has Changed," BYTE, 13,2,229-234.

UNITED STATES DEPARTMENT OF DEFENSE (1983) Reference Manual for the ADA Programming Language, ANSI/MIL-STD-1815A-1983. (Approved February 17, 1983: American National Standards Institute, Inc.) Springer-Verlag, New York.

WASSERMAN,A.I. and L.A. BELADY, with contributions from S.L. GERHART, E.F. MILLER, W. WAITE and W.A. WULF (1980) "Software Engineering: The Turning Point," Computer, 11, 9. Reprinted in Selected Reprints in Software (1980) The Institute of Electrical and Electronics Engineers, Inc., Computer Society Press, New York.

WEGNER, P. (1976) "Programming Languages--The First 25 Years." In HOROWITZ (ed.) (1987), 4-22.

WEXELBLAT, R.L. (ed.) (1981) History of Programming Languages. (From the ACM SIGPLAN History of Programming Languages Conference, June 1-3, 1978.) Academic Press, New York.

WICHMANN, B.A. (1984) "Is Ada Too Big? a Designer Answers the Critics," Communications of the ACM, 27,2,98-103.

WILSON, R. (sr. ed.) (1987) "Ada's Influence Spreads Through the Defense Community," Computer design, 26,13, 91-99.

WINSTON, P.H. (1984) Artificial intelligence, 2d ed. Addison-Wesley Publishing Company, Menlo Park, CA.

WINSTON, P.H. and B.K.P. HORN (1989) LISP, 3d ed. Addison-Wesley Publishing Company, Menlo Park, CA.

WIRTH, N. (1977) "Programming Languages: What to Demand and How to Assess Them." In FEUER & GEHANI (eds.) (1984), 245-261.

WYSOCKI, B. (1988) "The Wall Street Journal Reports. Technology: The Final Frontier. Japan Assaults the Last Bastion: America's Lead in Innovation, " The Wall Street Journal, November 14, 1988, Section 4, R1-R46.

YAZDANI, M. and A. NARAYANAN (eds.) (1984) Artificial Intelligence: Human Effects. Ellis Horwood Limited, Publishers, Chichester. Halsted Press: a division of John Wiley & Sons, New York.

ZELKOWITZ, M.V. (1978) "Perspectives on Software Engineering," Computing Surveys, 10,2,197-216.

ZELKOWITZ, M.V., A.C. SHAW and J.D. GANNON (1979) Principles of Software Engineering and Design. Prentice-Hall, Inc., Englewood Cliffs, New Jersey.

APPENDIX

THIRD INTERNATIONAL

Software for Strategic Systems Conference Proceedings

FEBRUARY 27-28, 1990

VON BRAUN CIVIC CENTER
HUNTSVILLE, ALABAMA

The University Of Alabama In Huntsville

Division of Continuing Education
Computer Science Department

An Affirmative Action/Equal Opportunity Institution

Huntsville Chapter
IEEE Computer Society

PROGRAMMING PARADIGMS: A MANAGERIAL PERSPECTIVE

E. Davis Howard, III
MSM (cand.)

Daniel Rochowiak
Research Scientist

Johnson Research Center
University of Alabama in Huntsville
Huntsville, AL 35899

ABSTRACT

Various programming paradigms are available to the software developer and the manager of a software project must make decisions about them. Focusing on the object-oriented p: adigm and its relations to the languages and environments of LISP, C, and Ada, we will examine the issues which confront the software management decision. In particular, we will suggest that the software manager will continue to make management, rather than engineering, decisions about software projects because it can be reasonably expected that paradigms, languages, and environments will continue to evolve.

INTRODUCTION

If one wants to generate a debate at a party for persons connected with computer programming, just ask "What is the best programming language?" The result is often an outpouring of praise, curses, hyperbole, and technical detail that will either quicken the pulse or induce tranquil repose. Programming languages are at times treated as matters of religious fervor, and at other times treated as mere notational convention. All of this would be fine were it not for the demands for "good" software and the increasing size, complexity and seriousness of software programming projects.

To be sure software is more than the code for a program. Software, in the sense in which we will consider it, includes all of the information that is: (1) structured with logical and functional properties, (2) created and maintained in various representations during its life-cycle, and (3) tailored for machine processing. This information in large projects is often used, developed, and maintained by many different persons who may not overlap in their roles as users, developers, and maintainers. In order to develop good software, one must explicitly determine user needs and constraints, design the software in light of these and in light of the needs and constraints of the implementers and maintainers, implement and test the source code, and provide supporting documentation.

The preceding dimensions and constraints on producing software can be looked at in two different ways. In one way they can be thought of as, perhaps sequential, modules. In another way they *might* be thought of as aspects of different moments in the software production process. We will adopt this latter, process-oriented, interpretation, and focus upon programming paradigms.

The programming languages LISP, C, Ada can each legitimately claim a special competence. In the case of LISP, it is symbolic processing; in C, the combination of portability and speed; and in Ada, uniformity and maintainability. We will adopt a managerial point of view in examining these languages and the paradigms they support.

From our examination of these languages, we believe that decisions about a programming language and a programming environment cannot be meaningfully separated and that they ought not to be separated. Whether one examines LISP or C, it is clear that the advocates of these languages are not considering the languages in isolation. Rather they are considering LISP in a LISP environment, if not on a LISP machine,

and C in a UNIX environment. We believe that this is as it should be. The art of programming is not simply the procedure of writing down code. The art of programming is to be found in the choice of programing paradigms and in the use of tools that make programming in that paradigm easy, safe, and effective. This perspective gives rise to the idea that it is not so much the language that encourages good programming, but the combination of language and environment. While a language might force the use of certain programming constructs, it is still possible to write bad code with those constructs. This would seem to be case no matter how one cares to operationalize the notions of "good" and "bad." The availability of good programming tools in a good environment makes it easier to write good code than to write bad code. If this is correct, then it is clear why considerations of programming languages ought not to be separated from considerations of the programming paradigms and environments.

Further, it appears that the combination of programming environment and programming language is intimately connected with the programming paradigm that is to be used in the construction of the program. A programming paradigm may be thought of as the style or manner in which a program is created. The notion of a paradigm in this context has strong connections to the notion of "paradigm" as it is used in the history, philosophy and sociology of science. A paradigm is a sort of template that is filled in to attack certain sorts of problems. Within one paradigm there may be many particular templates, but there is a sense in which each of these reduces back to some primitive template. Alternatively, one may view the paradigm as a primitive object from which the specific template inherits structures and properties. Under either sort of interpretation, it should be clear that a programming paradigm acts as a vehicle through which a programmer designs and builds specific programs.

PROGRAMMING PARADIGMS

Stroustrup (5) sets out the relation between a programming paradigm and a programming language rather neatly.

> A language is said to support a style of programming if it provides facilities that make it convenient (reasonably easy, safe, and efficient) to use that style.... Support for a paradigm comes not only in the obvious form of language facilities that allow direct use of the paradigm, but also in subtle forms of compile-time and/or run-time checks against unintended deviations from the paradigm... Extra-linguistic facilities such as standard libraries and programming environments can also provide significant support for a paradigm.

The problem of selecting a paradigm is both art and science. It is an art insofar as it requires a subtle understanding of the programming craft, and is a science insofar as a set of decision rules can be established for the paradigm. Stroustrup investigates this second way, and identifies several paradigms for program construction. We will augment his decision principles to capture the intentional character of the decision.

Procedural Paradigm

Principle

If

the intention of the program is primarily procedural

then

decide which procedures are to be programmed and use the best algorithm you can find.

Comment

The core of the principle is the link between procedural knowledge and algorithms. The paradigm focuses on those parts of knowledge that can be codified in a step by step way. The sense of procedure here is not the broad sense of procedure in which it may be said that a manager follows procedures, but the quite narrow sense of procedure in which one proscribes a set of stepwise instructions that guarantee that an answer will be found. Sorting programs are a good example of programs that conform to this paradigm. Programming in conventional languages often conforms to this paradigm.

Data Hiding Paradigm

Principle

If

the intention of the program can be divided into modules

then

decide the modules you want and partition the program so that the data in one module is hidden from the data in other modules.

Comment

The principle of data hiding expands on the procedural paradigm. The core of the data hiding paradigm is the connection between modules and partitions. If the objective of the program can be divided into tasks that can act as a module, then the program should be divided so that the data in any partition does not affect the data in any other partition. Thus a module is a collection of procedures that act upon data in such a way that the internal operations of the module do not lead to global effects. For example, stacks and queues might be implemented in a module.

Abstract Data Type Paradigm

Principle

If

the intention of the program is to provide for more than one object of a given type

then

decide which types are needed and provide a full set of operations for each.

Comment

The data hiding and the abstract data type paradigms are similar. They differ insofar as the abstract data type is more general; data hiding can be considered as an instance of the abstract data type paradigm restricted to a single object of the type. It should be noted that the phrase "abstract data type" can be misleading. The type itself is not abstract; it is as real as any other type. The difference between the abstract data type and the types in the language, might, therefore, be better captured with the phrase "user-defined type." Although such user-defined types allow for multiple objects of the defined type, the functions that deal with these types must know about each type. Thus, if a new type is added to a program, the code that contains the functions that operate on that type must be modified.

Object-Oriented Paradigm

Principle

If

the intention of the program is to provide for classes of multiply-related, user-defined types

then

decide which classes are needed, provide a full set of operations for each class, and make commonality explicit by using inheritance.

Comment

If the general and the specific properties of a class can be differentiated, then it is desirable to use the object-oriented paradigm. This paradigm makes it clear that the classes of user-defined objects can be made more and more specific and that the specific objects have instances. Further, within this paradigm there is the idea that at least some functions are generic; some functions apply to different types of object. The key idea of the object-oriented paradigm is that classes of objects can be defined (say animals) from which more specific classes may themselves be used to define even more specific classes (dogs). Further, within this paradigm, it is possible to use two or more

general classes (say, water-craft and motorized-vehicle) to create a more specific class (motor-boat). The new class will inherit the general properties of both classes and will allow the generic methods for both to operate on this new class.

Although there are formal senses in which all sufficiently rich languages are equivalent, this equivalence is only logical or formal. Although any of the paradigms can be accomplished in any of the languages that we have examined, this does not mean that it is either easy or reasonable to use any of the mixtures of paradigm and language that are possible. Indeed in the case of two of the languages that we have examined, LISP and C, specific packages of extensions have been generated in order to support the object-oriented paradigm, Flavors (and now CLOS) and C++. At the moment it is not clear what packages of extensions will or will not be available for Ada. What is clear is that the first three paradigms are supported by all three languages, although it is somewhat odd to think about the procedural paradigm for LISP.

The issues of programming paradigms are intimately connected to issues of language choice, and by extension environment choice, as well as to the issues that surround the notion of software engineering. We believe that this nexus of issues should be brought together, perhaps under the rubric of software management. Within this conception software management would include software engineering. Software engineering would provide the tools and methodologies that are needed to construct good, reliable, and maintainable code. These tools would of course have to be tightly tied to the programming language(s) and environment(s) in which the program is built. Software management would focus on the decision to implement a concept or design with specific paradigms and tools.

Fairley (3), for example, defines software engineering as the "technological discipline concerned with the systematic production and maintenance of software products that are developed and modified on time and within cost estimates," and claims that software engineering is a "new technological discipline distinct from, but based on the foundations of, computer science, management science, economics, communication skills, and the engineering approach to problem solving." While we do not disagree with Fairley's position, our interpretation of software engineering stresses its role within a management context, rather than stressing its status as a quasi-autonomous technological discipline. In brief, the products of software engineering are tools that an effective software manager should use, but they exhaust neither the domain nor the tools that the manager must consider.

Boehm (1) identifies seven basic principles in software engineering. These are:

1. Manage using a phased life-cycle plan,
2. Perform continuous validation,
3. Maintain disciplined product control,
4. Use modern programming practices,
5. Maintain clear accountability for results,
6. Use better and fewer people,
7. Maintain a commitment to improve the process.

Of the principles identified by Boehm two require special attention in our context. These principles are the injunctions to use modern programming practices and to use better and fewer people.

Programming paradigms are at the root of modern programming practices. As Boehm (1) notes, "The use of modern programming practices (MPP), including top-down structured programming (TDSP) and other practices such as information hiding, helps to get a good deal more visibility into the software development process, contributes greatly to getting errors out early, produces understandable and maintainable code, and makes many other software jobs easier, like integration and testing." At issue, of course, is what counts as a modern programming practice. We assume that modern programming practices are not fixed, that such practices are the outgrowths of programming paradigms, and that the paradigms are responses to the practical needs of computer software developers and the intellectual demands of computer scientists. We will treat the object-oriented paradigm as the most modern of such practices.

If the object-oriented paradigm is a significant advance over the other paradigms, then it would seem to follow from Boehm's fourth principle that it should be put into use. A good case from the management

perspective can be advanced for this. An object can be considered as body of code that is encapsulated in such a way that it can be addressed by other objects, can address other objects, and encapsulates its own processing. Data which are wholly internal to the object are not publicly known, and the messages that are passed between objects allow objects to exert control while promoting modularization of the overall software project. In many ways object-oriented programming can be seen as an ideal of encapsulation. Each object is, in a sense, its own program. The object can be judged to be correct, accurate, or valid in its own right. Further, objects are a natural way of thinking about the world, and, therefore, are more closely tied to the design process. If an object is needed, it can be created without risking the sorts of clashes that can cause great difficulties in other paradigms. Objects can be continually refined and, therefore, the notion of rapid prototyping can be merged with the waterfall and incremental development models.

From a management perspective the object-oriented paradigm makes good sense. First, objects seem to correspond to a natural way of reasoning about problems. Where we might say about physical things, "Get an object of kind K that can do task T when it encounters condition C," we might equally say about the software, "Get an object of kind K that does task T when it gets message M." Further, where we might say of ordinary physical things, "I want an object O2 that is like object O1, but has the additional feature F." Second, objects can inherit properties from other objects and this supports the reuse of those objects in similar tasks. For example, the general characteristics of sensors may, at an abstract level, be common to many sensors. A specific type of sensor could inherit those general properties and have added the properties specific to its kind. Thus, sensor objects could be reused in many different software contexts, in a way similar to that in which the physical sensor is used in many different systems. Third, the notion of message passing encourages the development of a standard set of protocols across objects and projects. In this sense the integration and, perhaps, maintenance aspects of software development are built in at the beginning of the project. Finally, the object-oriented metaphor promotes good documentation. All one needs to know about the object at one level is what messages it sends, what messages it receives and, in general, what it does. These elements are common to all objects and might, therefore, facilitate the manager's process of acquiring, understanding, and using the information to which he or she has access.

ISSUES

If object-oriented programming is a good paradigm, why is it not employed in all large software development tasks? There are several answers to this question. First, some programming languages do not support the object-oriented paradigm. As we noted earlier although there are object-oriented extensions for LISP and C, they have not been generally available and tightly integrated until recently . Although there does not appear such an extension for Ada, perhaps owing to the restrictions on creating supersets of Ada, some of the characteristics of object-oriented programming can be incorporated into Ada. Booch (2) has attempted to illustrate the use of object-oriented programming techniques in Ada, but notes the difficulties in providing for inheritance in such an approach. Second, object-oriented languages are either nonstandard or have not been generally accepted. Examples here would include SMALLTALK and SIMULA. Third, there is the perception that object-oriented programming would impose unacceptable overheads. Finally, object-oriented programming is relatively new and has not yet established a strong following. These reasons are interconnected. For example, the paradigm may not prove its worth if it does not attract a strong following; and, it might not attract a strong following, if there is not sufficient support in an acceptable language. This last point could be critical, if one considers the role of government institutions in generating technological innovation, and the fact that the government supports the Ada language which does not directly offer support for the object-oriented paradigm.

Thus, Boehm's principle that modern programming practices ought to be used is a bit odd. What it might really mean, however, is not that any modern programming practices should be used, but that the modern programming practices for imperative, conventional languages that are used for large software projects and can be handled within the current discipline of software engineering should be used. Neither LISP nor object-oriented programming can satisfy those demands. However, Ada comes near to being the ideal language from the point of view of software engineering with conventional languages. This points out the difficulty in generating a set of principles to guide software engineering. The analogy of software engineering to the rest of the engineering field (6) begins to break as one attends to the nonphysical character of software. For example, when building a bridge or a pipeline, the standard elements of the construction remain static. Bridges will have beams and pipelines will have pipes. The materials and techniques may change, but the basic elements remain. Unconfined by such physical characteristics, the

elements of software construction can change. subroutines, subprograms, libraries, modules, package, units, function, objects and many other elements are available to the software programmer, and new as yet unthought of constructs might be added. All of this adds to the complexity of choosing and using modern programming practices, and points to the important role of the software manager even within the software engineering discipline.

The second of Boehm's principles that needs special attention is the injunction to use better and fewer people. There are three distinct aspects to this injunction. The first is the most obvious: personnel may vary in their talents. In many cases, the Marine Corps attitude is the one to employ: "We are looking for a few good programmers." The second feature is the overhead of interpersonal communications. In this case, the fewer the programmers, the less the overhead. Finally, there is the use of automated aids. Tools can increase the productivity of a programmer, and, thereby, reduce the number of programmers and the communication overhead.

While there is clearly a great deal that is right about this injunction, there is also something disturbing. While it is clear that communications overhead can be a significant factor in the loss of productivity, it is not at all clear why this must be so. If the communication only serves administrative purposes, or if the communication takes longer than it might because of the organizational structure, then it seems that it is the organization itself that causes the lack of productivity and not the number of programmers. Further, one must look at what might be lost by having fewer programmers. With fewer programmers, errors, over-designing, under-designing, and missed opportunities could easily arise. Why should programmers communicate with other programmers? Perhaps to heighten their objectivity and spark new criticism, in much the same way that these things happen in a scientific community. While it may be good to eliminate unnecessary communication overhead, there may be better ways than limiting the number of programmers. It is at this point that the software manager must decide whether a programmer is more akin to an assembly line worker or a professional scientist.

The use of automated tools to manage the software coding process, including the generation of source code in a target language, raises another interesting issue. If the tools are good tools and if the code they generate is good code, then what is the programmer doing? In a primary sense he is running the tool; in a secondary sense he or she is programming in some language. There is a sense in which if the tools are very well done the "programmer" need not even know the language in which he or she is programming, and, indeed, need not even know in what language the code is being generated. As Howden (4) has noted:

> The manufacture of software is perhaps one of the most logically complicated tasks. The intellectual depth of software systems development coupled with the lack of physical restrictions imposed by properties of the product make software manufacturing intriguing in its possibilities for highly automated, sophisticated manufacturing environments. Research has begun, on environments containing their own concept models of general programming knowledge... It has been speculated that in the future software engineers will be able to describe application programs to a system capable of automatically generating specifications and code.

An intriguing possibility! Taken together the two injunctions upon which we have focused our attention can lead to a radical re-interpretation of the software programmer. If good software engineering advises that the best available tools be used, if the best available tools embody the object-oriented paradigm, and if the best available tools embody the ability to automatically generate appropriate and valid application code, then the role of the programmer in software development is radically changed. The programmer is no longer a crafter of code, but an expert user of a tool. The connection between the programming language and the programmer is, in a sense, severed. The programmer with such tools may, therefore, function at a higher, more natural level of abstraction without needing to attend to the syntactic complexities of the language in which the application is finally coded. This is not, however, surprising. It represents simply another moment in the evolution toward higher level languages. Rather than a traditional higher level language being used with a compiler to generate the low level instructions to the processor, a new generation of tools may operate at an even higher level and a translator may then convert the tool's specifications into a higher level language which in turn may be compiled.

The account of software management that we have been developing points to a novel way of thinking about programming languages and environments. If the programmer is thought of as similar to a scientist, and if the programmer is a scarce resource, and if programming aids and tools are well developed, then the choice of a programming language and environment becomes much less important. This is not to say that the choice of a programming language is not important. However, the choice of the language is placed in an environment where other considerations may be of equal, if not greater, importance.

Let us end this discussion by examining why there are so many different programming languages. One can readily point to four factors: improvements in processors, improvements in paradigms, improvements in environments and tools, and the specialization of programming tasks.

It is clear that processors have improved greatly over the past two decades. Increased speed, increased word size, augmented capabilities, decreased power consumption, and decreased cost are readily apparent. All of these factors combine to allow those who design and build languages and environments to implement more easily and effectively ideas and constructs which with less capable processors would remain dream and desire. One need only to recall what it was like to run LISP on a PDP-11 under RSTS and look at a Symbolics or Texas Instruments LISP machine to recognize the difference. Similarly, C in its own UNIX environment has come to be recognized as a powerful system, and has led to the evolution and development of the computer workstation. The future holds even more promise. Even as physical limitations begin to affect the development of better processors, new architectures begin to evolve. Multiprocessor machines, parallel processor machines, and other objects of wonder and splendor open new vistas to the language crafter. Although languages like LISP and C will probably move into these new environments, their form and function will probably be much different. Equally probable is that new languages will emerge. In any case, the point we believe is clear. Languages are not static. Language development responds to the state of the processor art. As long as processor development continues it is reasonable to expect programming languages to develop.

It should also be clear that programming paradigms change over time. The changes of paradigm reflect both the intellectual development of computer programming and the ability of the language crafters to build support for a paradigm into a language. BASIC was a wonderful language. It was criticized for not supporting a structured programming paradigm. New BASIC arose in response to that criticism. Classic LISP did not support object-oriented programming. LISP with FLAVORS is a virtually seamless environment in which such programming is supported and encouraged. C did not support the object-oriented paradigm; C++ is a response as is Objective C. If it were not for the government's involvement with Ada, one might well think that OO-Ada (Object-Oriented Ada) might soon appear. There is, of course, no reason to think that the story ends with the object-oriented paradigm. New paradigms, perhaps tailored to particular classes of problems, may well arise. As they do, old languages may evolve to support them, and new languages may arise to enforce them in much the way that PASCAL and MODULA-2 enforce structured programming and SMALLTALK enforces object-oriented programming.

Perhaps the most dramatic changes of language will occur with the improvement and development of programming environments and tools. As we noted above, it is often the environment that captures the programmer. The facilities of the LISP and C environments allow the programmer to concentrate on the task at hand, and quickly and efficiently produce the needed code. This is especially true of a LISP environment on a LISP machine. The programmer can build his own tools and tailor the environment to his or her needs and preferences. More importantly, the environment and machine function in harmony to allow the programmer to build new languages in which problems can be solved. By allowing a measure of abstraction, generality and efficiency can be gained. All of these things taken together point out that the developmental environment is an important factor in selecting a languages.

As programming tools and aids evolve, the direct contact with the programming language may begin to disappear. Such tools may allow the programmer to either break the programming task down into parts that are sufficiently small and standard that existing libraries of routines can be employed, or may allow the programmer to build the program specifications in such a way that a translator will be able to translate the specification into the target language. Both approaches currently have their problems. In the former the programmer is left at some point to grapple with the language itself, and in the latter the programmer might find the translated code for the target language indecipherable. Although these are serious problems, they

may not be insurmountable. If they can be overcome, the contact of the programmer with the programming language will be stretched thinner and thinner.

The continued improvement of programming tools and environments, may lead the manager to base his or her decision on which programming language to use on the presence or absence of certain features in the tools and environments more than on the characteristics of the languages. The decision, of course, is still affected by external factors. Ada will be used on the Space Station. However, much might be learned by examining the environments and tools for other languages such as LISP and C in an effort to build better tools for Ada. After all, given that Ada is a sufficiently universal language, it can be made to look like other languages.

The final point to be made about programming languages and environments is a very simple one: there is no language that allows the production of code to be easy within every paradigm for every application. Programming languages, like all other human artifacts, are imperfect and limited. Thus, it should not be surprising that some programming languages perform better at some, tasks than others. It can well be noted that Ada is not of necessity, the savior. Likewise neither LISP nor C can fulfill that role. It is reasonable to expect that with the increasing specialization of programming tasks new languages and extensions to old languages will evolve that will handle such tasks more easily and efficiently.

CONCLUSION

In this brief discussion, we have attempted to highlight some of the factors that should be examined, if good judgments are to be made about software development and production. The list of suggestions is in no way exhaustive. Many more suggestions could have been advanced. For example we have not explicitly considered the problem of knowledge acquisition. This is a crucial area in the development of any viable software. Another omission is the potential for conflicts between artificial intelligence (AI) efforts and more conventional programming. This issue is of great significance when a decision must be made about whether the AI components of a project like the Space Station should be exempt from the Ada edict. More omissions can be added to the list. What we hope to have made clear is that decisions about programming are subject to a multitude of factors, and that the programming craft and science is changing and may well continue to change. If this is so, then managers of software projects will continue to make managerial decisions about which projects are to be done, how they are to be done, and with what they are to be done. Thus, it is reasonable to think that the debates at parties will continue.

ACKNOWLEDGEMENTS

This research has been partially funded under NAS8-36955 (Marshall Space Flight Center) D.O. 34 "Applications of Artificial Intelligence to Space Station."

REFERENCES

1. BOEHM, B.W. "Seven Basic Principles of Software Engineering," *The Journal of Systems and Software* 3, (1983), pp. 3-24.
2. BOOCH, G. *SOFTWARE COMPONENTS WITH ADA: Structures, Tools, and Subsystems.* The Benjamin/Cummings Publishing Company, Inc., Menlo Park, California, 1987.
3. FAIRLEY, R.E. *SOFTWARE ENGINEERING CONCEPTS.* McGraw-Hill Book Company, New York, 1985.
4. HOWDEN, W.E. "Contemporary Software Development Environments," *Communications of the ACM*, 25, 5 (1982), pp. 318-329.
5. STROUSTRUP, B. "What is 'Object-Oriented Programming'?," *ECOOP '87: European Conference on Object-Oriented Programming, Paris, France, June 15-17, 1987, Proceedings.* [Bézivin, J., P. Cointe, J.-M. Hullot, and H. Lieberman (Eds.)]. *Lecture Notes in Computer Science (276)*, Goos, G. and J. Hartmanis (Eds.), Springer-Verlag, Berlin, 1987.
6. ZELKOWITZ, M.V., A.C. SHAW, and J.D. GANNON. *Principles of Software Engineering and Design.* Prentice-Hall, Englewood Cliffs, New Jersey, 1979.

To be included in
Fifth Conference on Artificial Intelligence for Space Applications

ADA AS AN IMPLEMENTATION LANGUAGE FOR KNOWLEDGE BASED SYSTEMS

Daniel Rochowiak
Research Scientist

Johnson Research Center
University of Alabama in Huntsville
Huntsville, AL 35899

ABSTRACT

Debates about the selection of programming languages often produce cultural collisions that are not easily resolved. This is especially true in the case of Ada and knowledge based programming. The construction of programming tools provides a desirable alternative for resolving the conflict.

INTRODUCTION

If one wants to generate a debate at a party for persons connected with computer programming, just ask "What is the best programming language?" The result is often an outpouring of praise, curses, hyperbole, and technical detail that will either quicken the pulse or induce tranquil repose. Programming languages are at times treated as matters of religious fervor, and at other times treated as mere notational convention. All of this would be fine were it not for the demands for "good" software and the increasing size, complexity and seriousness of software programming projects. To be sure software is more than the code for a program. Software, in the sense includes all of the information that is: (1) structured with logical and functional properties, (2) created and maintained in various representations during its life-cycle, and (3) tailored for machine processing. This information in large projects is often used, developed, and maintained by many different persons who may not overlap in their roles as users, developers, and maintainers. In order to develop good software, one must explicitly determine user needs and constraints, design the software in light of these and in light of the needs and constraints of the implementers and maintainers, implement and test the source code, and provide supporting documentation. These dimensions and constraints on producing software can be looked at as aspects of different moments in the software production process..

The programming languages LISP and Ada can each legitimately claim a special competence. In the case of LISP, it is symbolic processing, and in Ada, uniformity and maintainability. In making a decision about a programming language, the programming language and its environment cannot be meaningfully separated. Whether one examines LISP or Ada, it is clear that the advocates of these languages are not considering the languages in isolation. The combination of programming environment and programming language is intimately connected with the programming paradigm that can be used in the construction of the program. A programming paradigm may be thought of as the style or manner in which a program is created. Within one paradigm there may be many particular templates, but there is a sense in which each of these reduces back to some primitive template. Alternatively, one may view the paradigm as a primitive object from which the specific template inherits structures and properties. Under either sort of interpretation, it should be clear that a programming paradigm acts as a vehicle through which a programmer designs and builds specific programs.

Certainly another way to generate debate is to ask, "What is the best representation of knowledge?" or "What is the best way to manipulate knowledge?" The list of answers will grow rapidly: logic, rules, frames, scripts, objects, trees, nets, inferences, associations, statistical inferencing, case based reasoning, analogy, and so on. All of these styles and techniques have valued uses. All have their strengths and weaknesses. Unless a person was very lucky, no consensus would be achieved at the party.

Behind both the questions about programming languages and the questions about knowledge is a common social structure. Programming and the construction of knowledge based systems occur in cultures. These cultures are the repository for tradition, tacit rules of procedure, and tacit rules of appraisal. A person's training is the way in which they are enculturated. Someone who is trained on a certain hardware, in a certain language, and in a certain style will carry the culture generated through that training onto his or her new works. As persons with their cultures collide differences of opinion, and difficulties in adjusting to the demands of another are sure to be produced. This collision of cultures is a central element of the issues surrounding the debates about knowledge based programming and Ada.

PARADIGMS AND CULTURES

Typically a culture has a core paradigm or set of paradigms that capture the core of the culture. The paradigms act as cognitive templates that are filled in when either trying to solve a problem or develop an object.

In programming there is an interaction between what may be considered a programming paradigm and a programming language. Stroustrup (8) sets out the relation between a programming paradigm and a programming language rather neatly.

> A language is said to support a style of programming if it provides facilities that make it convenient (reasonably easy, safe, and efficient) to use that style.... Support for a paradigm comes not only in the obvious form of language facilities that allow direct use of the paradigm, but also in subtle forms of compile-time and/or run-time checks against unintended deviations from the paradigm... Extra-linguistic facilities such as standard libraries and programming environments can also provide significant support for a paradigm.

The problem of selecting a paradigm is both art and science. It is art insofar as it requires a subtle understanding of the programming craft, and is science insofar as a set of decision rules can be established for the paradigm. The four typical paradigms are: procedural, data hiding, abstract data type, and the object-oriented Paradigm.

In examining Ada and LISP the idea of the programming paradigm can be usefully extended to the paradigmatic way in which programming languages and their environments are used. The question is whether one language offers tools that make it best suited to a particular task. Although there is a formal sense in which all sufficiently rich languages are equivalent, this equivalence is only logical or formal. Although any of the paradigms can be accomplished in either LISP or Ada , this does not mean that it is either easy or reasonable to use any of the mixtures of paradigm and language that are possible. Since LISP has been the chief language of artificial intelligence research, it is reasonable to investigate whether Ada can support the constructs of LISP. In this way the issue concerns whether Ada can implement the LISP paradigm.

Schwartz and Melliar-Smith (7) analyzed the Ada specification to determine its potential as an AI research language. Their conclusion is that Ada, as defined in the Preliminary Standard,

would not be suitable as a "mainstream research language." They proposed, however, that with some extensions it is plausible that a substantial portion of AI "algorithms" could be translated into Ada. This translation would not be easy, since it would be more of a "reimplementation" of the program, but the "complex heuristic algorithms that provide the artificial intelligence" could be retained.

Schwartz and Melliar-Smith's claim of Ada's unsuitability is fundamentally based on the determination to enforce a particular programming paradigm. One goal that was set forth in both the Ironman and Steelman Requirements, is to create "an environment encouraging good programming practices." Ada imposes a style of programming that is the result of many years of research on programming methodology. Ada is intended to impose a very disciplined style of programming that assists those who are developing large, complex projects that require teams of programmers. Furthermore, Ada is said to be 'readable and understandable rather than writeable' so as to minimize the cost of program maintenance. Thus, Ada's mandated programming style is beneficial for the targeted Ada community - a production community, especially a community that produces real-time embedded systems. In general, AI work does not occur in a production community, but a research and development community. This difference in orientation is a factor in making Ada unsuitable as a general AI research language. The constraints of production prevent the AI programmer from using the most natural method of expression for whatever system is being developed. The LISP programmer places greater value on code that is more easily writeable than readable. However, two things should be remembered. First, the readability of any code is a function of the enculturation of the reader. Second, the readability of the code is a function of the tools available with which to read it. This latter point is important when one considers LISP on a LISP machine. Within that environment the code may become very readable through the tools that are available for reading it.

Schwartz and Melliar-Smith contend that the utility of Ada for AI programs is confined to the reimplementation. This operation would be carried out by software teams by following the algorithms of an original program, but not necessarily its detailed code. Extensions to Ada are needed, however, if such reimplementation is to be carried out while preserving Ada's structure and modularity.

A typical AI task for a knowledge based system in LISP is to generate solutions to problems that have a very large number of alternatives. To attempt to solve such a problem by exhaustive search or "best fit" is not feasible even with a supercomputer. A heuristic based guess is used to prune branches from the decision tree so that the problem becomes tractable. In some "classic" systems, a breadth-first or depth-first search is used to consider candidate solutions. When it becomes apparent that an incorrect decision has been made, then the search resumes at the junction where that decision was made. Use of heuristics allows for systems to "learn" from their mistakes and refine their search techniques as more is "learned" about the problem domain. Several features of AI programs stand out. First, extensive use is made of the list structure and the processing of lists. Second, procedures are often used as values that can be stored in a data structure. This allows for the construction of a generic framework for the parameterized transformation of a given type of structure. An example of this would be the construction of a system to perform an arbitrary function on a tree or graph structure. If the procedures are values of a procedural data type, then the procedures could be passed as parameters to perform the desired manipulation of data. Third, LISP provides for a similar representation of both data and programs that allows for the creation of functional abstractions "on the fly." These abstractions, expressed by Lambda calculus list expressions, can be passed as parameters to other abstractions. Fourth, as each function or expression is defined, it becomes part of the system. Thus, the application program can examine its run-time environment, a fact which makes the program inseparable from its environment. Finally, the ability to use procedures as storable objects is essential to many AI programs. One use for this ability is as a method to express knowledge about a particular domain. Frequently, several different knowledge representations will be used

in one system. The particular representation used would depend on the availability of information.

PACKAGES FOR ADA

Much of the success of LISP as an AI language can be attributed to fact that it is extensible. It is possible, for instance, to construct rather easily interpreters for other high-level languages using LISP. This ability is facilitated by the manner in which LISP programs are represented: as lists. Ada, too is extensible. (2) However, Ada is more limited in its extension capabilities, with packages, generic procedures and tasks being all of the extension methods. Whether or not this extensibility is to limited for the needs of reimplementing AI programs remains to be seen. Ada provides data abstraction facilities that allow one to create extensions to the language by the defining of new data types and the operators that can be used to manipulate them. Through the use of a package containing a data abstraction, a programmer can write code as if the facilities provided by the package were provided by Ada. Thus the addition of packages may provide a way in which the typical features of an AI program written in LISP can be reimplemented in Ada. Such a package may, for example, supply the tools needed to handle lists, procedures, and garbage collection.

List processing is an important feature of AI research languages. Whereas Ada does provide the features needed to implement list processing, its garbage collection facilities leave much to be desired. No special considerations have been made for list processing, and consequently, the efficiency of such will likely be minimal. To implement lists in Ada, one could create a data structure as follows. Each list cell would be a record that has two list pointers: CAR and CDR. A list pointer would then be a record that has only a variant part. The discriminant of this variant part would have two possible values: ATOM and LIST. This would indicate whether the list pointer component is a list reference or an atom reference. There would need to be a LIST_REFERENCE and ATOM_REFERENCE access types for the dynamically allocated list cells and atom cells.

Although procedural variables cannot be readily added to Ada, it is conceivable that the ability to pass procedures as parameters could be added. The effect of the instruction part of a procedural value can be simulated through the use of a generic procedure. This method would avoid using a CASE statement as would be necessary if the indexing scheme were used. Generic procedures used in this fashion would carry the name of the "passed" procedure but would not have the closure or environment.

As most AI programs "run," they pursue a number of possible alternative paths of action. This attempt to find the best possible path usually succeeds in allocating a great deal of memory. Since the memory objects have a lifetime that is dependent on the duration of the utility of the data, and not the flow of control of the program, these objects must be allocated in a global heap. By using a heap, storage can remain at least as long as it is referenced anywhere else in the system. Consider a typical embedded system application. Here, the data that must have space in the heap is minimal. Thus, reclamation of heap space is not important, and in some cases, heap space is not reclaimed at all. This is yet another design philosophy contradiction, between Ada and AI languages. AI languages are designed with the philosophy that "no amount of initial heap allocation will be sufficient for the continued operation of many AI programs." It is not a question of if all of the heap space will become allocated, rather is is a question of when it will happen. Obviously, some strategy must be used to reclaim this storage space. The language specification for Ada does not preclude garbage collection capabilities, nor does it indicate these will be included. There is a mechanism, FOR-USE, which indicates the maximum number of objects of an access type that may be generated. Since the compiler knows the maximum size in advance, the necessary space can be allocated. This provides a sort of heap-type allocation with

automatic reclamation for objects that have a limited scope of use. Unfortunately, this method causes allocation/deallocation to be dependent on control flow or block entry and exit.

PATTERNS

Obviously, not everyone agrees with Schwartz and Melliar-Smith on Ada's place in AI. Larry Reeker, John Kreuter, and Kenneth Wauchope of Tulane University have done much work on pattern matching in Ada. In answer to the question of "will Artificial Intelligence be done in Ada?" they answer that "anything can be done in Ada," and attempt to show how Ada, when appropriately used, can facilitate the programming of Artificial Intelligence applications. (6)

Reeker has chosen to focus on a pattern-directed because "pattern-directed facilities provide the most effective means for creating complex programs for non-numerical applications." Further support for pattern matching can be found in the work of Warren, Pereira and Pereira. (9) They contend that pattern matching "is the preferable method for specifying operations on structured data, both from the user's and the implementer's point of view."

Reeker envisions the addition of AI oriented features to Ada through the use of packages. The list of features that are candidates for incorporation into Ada include:

- String definition and manipulation facilities more flexible than those built into Ada.
- List processing functions
- Pattern definition and matching functions for strings and lists
- A means of manipulating lists returned by the pattern matching functions

Ada's concurrency paradigms lead to a number of possible methods for pattern matching. One such method would be to use tasks as "coroutines" to match patterns. There are areas in AI which have made use of "quasi-parallel" processes previously. True parallel tasks executing on a true multiprocessor system would surely improve on those systems.

In his section of their paper, Kenneth Wauchope presents an Ada language implementation of a pattern-directed list processing facility. A set of SNOBOL-4 like primitives are used to construct lists that are equivalent to arbitrarily complex LISP-like data structures. Wauchope advocates the addition of packages to make AI feasible in Ada. In this paper he describes the operation of a package which provides basic list creation and manipulation functions similar to those in LISP. Wauchope then presents several applications of these new features, including: parsing a context free grammar and symbolic differentiation.

Krueter presents several algorithms for pattern matching in Ada The first of these is the recursive descent parsing method which is a common way to implement the backtracking strategy. Backtracking is based on the intuitive approach of trying every possibility for each pattern element. This generates every possible parse of the string but is rather costly in terms of time.

One particularly interesting possibility arises with the use of Ada for coding such algorithms. Since Ada allows concurrent tasks, the backtracking aspects of the algorithm could be achieved through the use of tasks that behave as coroutines. A task would start by examining each bead in the first set of alternatives. A new task is forked for each successful match. This new tasks will then examine the remainder of the string and the remaining sets of alternatives. After all alternatives have been examined, the task will pass back the matching substrings, or null in the case of no match, and terminate. Each successive parent tasks will then add its substring to the beginning of each tree on the list which has been passed to it. Then, this list is passed back, and so forth, until the master task is reached.

Combinatorially implosive algorithms (CIA's) are a class of parallel algorithms that employ two or more algorithms running concurrently such that they will solve a problem more quickly than one would by itself. Brintsenhoff, Christensen, Mangan, and Greco demonstrate a CIA coded in Ada in their paper, "The Use of Ada Concurrent Processing Features in an Implementation of Parallel Tree Searching Algorithms." (3) This study is interesting because the authors had access to a multiprocessor with run-time support for concurrent tasking. Their findings show the speed advantages of parallel algorithms written in Ada. Although the results were highly data dependent, the running of two algorithms concurrently proved to be more efficient than just one and thus proved the utility of CIA's in Ada. If such CIA's could be developed for pattern matching, it is reasonable to expect that pattern driven AI applications would prove to be very efficient in Ada.

OPTIONS

The two previous sections have indicated two ways in which the confrontation of Ada and AI might proceed. In the first way the differences of the two cultures are acknowledged and an effort is made through the addition of appropriate packages to provide the tools for a reimplementation of an AI program. The second option acknowledges the fact that in a sufficiently complete language it is possible to implement the idea of a program directly. Each approach has certain advantages and disadvantages. In the first approach the program does not have to be completely rethought and redesigned. This is a disadvantage of the second option since the ideas for the program have to implemented from scratch. In the second approach there are potential advantages to be gained by using the strengths of Ada. This is the disadvantage of the first option. The addition of the packages may in effect provide for a LISP interpreter that circumvents the natural strengths of Ada.

One way in which a decision between these options might be facilitated is by using the resources of software engineering. Fairley (4), for example, defines software engineering as the "technological discipline concerned with the systematic production and maintenance of software products that are developed and modified on time and within cost estimates," and claims that software engineering is a "new technological discipline distinct from, but based on the foundations of, computer science, management science, economics, communication skills, and the engineering approach to problem solving." Boehm (1) identifies seven basic principles in software engineering. These are:

1. Manage using a phased life-cycle plan,
2. Perform continuous validation,
3. Maintain disciplined product control,
4. Use modern programming practices,
5. Maintain clear accountability for results,
6. Use better and fewer people,
7. Maintain a commitment to improve the process.

Of these principles one requires special attention in this context, injunction to use modern programming practices.

Programming paradigms are at the root of modern programming practices. As Boehm (1) notes, "The use of modern programming practices (MPP), including top-down structured programming (TDSP) and other practices such as information hiding, helps to get a good deal more visibility into the software development process, contributes greatly to getting errors out early, produces understandable and maintainable code, and makes many other software jobs easier, like integration and testing." At issue, of course, is what counts as a modern programming practice. Interpreting this principle is complicated by the facts that modern programming practices are not fixed, that such practices are the outgrowths of programming paradigms, and

that the paradigms are responses to the practical needs of computer software developers and the intellectual demands of computer scientists.

Thus, Boehm's principle that modern programming practices ought to be used is a bit odd. What it might really mean, however, is not that any modern programming practices should be used, but that the modern programming practices for imperative, conventional languages that are used for large software projects and can be handled within the current discipline of software engineering should be used. In this sense the modern programming practices are those geared to the community and culture of production. Neither LISP nor object-oriented programming can satisfy those demands. However, Ada comes near to being the ideal language from the point of view of software engineering with conventional languages. This points out the difficulty in generating a set of principles to guide software engineering. The analogy of software engineering to the rest of the engineering field (10) begins to break as one attends to the nonphysical character of software. For example, when building a bridge or a pipeline, the standard elements of the construction remain static. Bridges will have beams and pipelines will have pipes. The materials and techniques may change, but the basic elements remain. Unconfined by such physical characteristics, the elements of software construction can change. subroutines, subprograms, libraries, modules, package, units, function, objects and many other elements are available to the software programmer, and new as yet unthought of constructs might be added. All of this adds to the complexity of choosing and using modern programming practices, and points to the important role of the software manager even within the software engineering discipline.

The decision as two which of the two options should be pursued any not therefore be decidable on the grounds of software engineering alone. If the other principles that Boehm isolates are essentially management principles then it is fair to assume that they can be satisfied with any language and any programming paradigm. In this sense they are transcultural. However, the injunction to use modern programming practices is what the collision of cultures is about. What is a modern programming practice? Each culture will defend itself as being the exemplar of modern programming practice. Given the existence of the colliding cultures, it does not appear that the principles of software engineering will be able to generate a clear choice.

Another way in which the choice might be made is to focus on a technological solution. In particular the development of software tools that allow for program development in a neutral environment, but can generate code in a target language. The use of automated tools to manage the software coding process, including the generation of source code in a target language, raises another interesting issue, however. If the tools are good tools and if the code they generate is good code, then what is the programmer doing? In a primary sense he or she is running the tool; in a secondary sense he or she is programming in some language. There is a sense in which if the tools are very well done, the "programmer" need not even know the language in which he or she is programming, and, indeed, need not even know in what language the code is being generated. As Howden (5) has noted:

> The manufacture of software is perhaps one of the most logically complicated tasks. The intellectual depth of software systems development coupled with the lack of physical restrictions imposed by properties of the product make software manufacturing intriguing in its possibilities for highly automated, sophisticated manufacturing environments. Research has begun, on environments containing their own concept models of general programming knowledge... It has been speculated that in the future software engineers will be able to describe application programs to a system capable of automatically generating specifications and code.

An intriguing possibility! The programmer is no longer a crafter of code, but an expert user of a tool. The connection between the programming language and the programmer is, in a sense, severed. The programmer with such tools may, therefore, function at a higher, more natural level of abstraction without needing to attend to the syntactic complexities of the language in which the application is finally coded. This is not, however, surprising. It represents simply another moment in the evolution toward higher level languages. Rather than a traditional higher level language being used with a compiler to generate the low level instructions to the processor, a new generation of tools may operate at an even higher level and a translator may then convert the tool's specifications into a higher level language which in turn may be compiled.

If this technological path is found desirable, then it suggests that the first option, the addition of packages and reimplementation of code, is on the right track. Further it suggests that the second options benefits might be incorporated into the tool. If for example the target hardware is a multiprocessor system, then a tool should be able to guide the tool user in creating code appropriate to the hardware. The creation of such tools is not, however, to thought of as revolutionary. Rather the emergence of such is another evolutionary step in generating higher level software facilities for programming more complicated hardware.

SPECULATION

It is clear, for example, that processors have improved greatly over the past two decades. Increased speed, increased word size, augmented capabilities, decreased power consumption, and decreased cost are readily apparent. All of these factors combine to allow those who design and build languages and environments to implement more easily and effectively ideas and constructs which with less capable processors would remain dream and desire. One need only to recall what it was like to run LISP on a PDP-11 under RSTS and look at a Symbolics or Texas Instruments LISP machine to recognize the difference. Similarly, C in its own UNIX environment has come to be recognized as a powerful system, and has led to the evolution and development of the computer workstation. The future holds even more promise. Even as physical limitations begin to affect the development of better processors, new architectures begin to evolve. Multiprocessor machines, parallel processor machines, and other objects of wonder and splendor open new vistas to the language crafter. Although languages like LISP and C will probably move into these new environments, their form and function will probably be much different. Equally probable is that new languages will emerge. In any case, one point is clear. Languages are not static. Language development responds to the state of the processor art. As long as processor development continues, it is reasonable to expect programming languages to develop.

It should also be clear that programming paradigms change over time. The changes of paradigm reflect both the intellectual development of computer programming and the ability of the language crafters to build support for a paradigm into a language. BASIC was a wonderful language. It was criticized for not supporting a structured programming paradigm. New BASIC arose in response to that criticism. Classic LISP did not support object-oriented programming. LISP with FLAVORS is a virtually seamless environment in which such programming is supported and encouraged. C did not support the object-oriented paradigm; C++ is a response as is Objective C. If it were not for the government's involvement with Ada, one might well think that OO-Ada (Object-Oriented Ada) might soon appear. There is, of course, no reason to think that the story ends with the object-oriented paradigm. New paradigms, perhaps tailored to particular classes of problems, may well arise. As they do, old languages may evolve to support them, and new languages may arise to enforce them in much the way that PASCAL and MODULA-2 enforce structured programming, SMALLTALK enforces object-oriented programming, and Ada enforces some software engineering practices.

Perhaps the most dramatic changes of language will occur with the improvement and development of programming environments and tools. It is often the environment that captures

the programmer. The facilities of the LISP and C environments allow the programmer to concentrate on the task at hand, and quickly and efficiently produce the needed code. This is especially true of a LISP environment on a LISP machine. The programmer can build his own tools and tailor the environment to his or her needs and preferences. More importantly, the environment and machine function in harmony to allow the programmer to build new languages in which problems can be solved. By allowing a measure of abstraction, generality and efficiency can be gained. All of these things taken together point out that the developmental environment is an important factor in selecting a languages.

As programming tools and aids evolve, the direct contact with the programming language may begin to disappear. Such tools may allow the programmer to either break the programming task down into parts that are sufficiently small and standard that existing libraries of routines can be employed, or may allow the programmer to build the program specifications in such a way that a translator will be able to translate the specification into the target language. Both approaches currently have their problems. In the former the programmer is left at some point to grapple with the language itself, and in the latter the programmer might find the translated code for the target language indecipherable. Although these are serious problems, they may not be insurmountable. If they can be overcome, the contact of the programmer with the programming language will be stretched thinner and thinner.

The continued improvement of programming tools and environments, may lead the manager to base his or her decision on which programming language to use on the presence or absence of certain features in the tools and environments more than on the characteristics of the languages. The decision, of course, is still affected by external factors. Ada will be used on the Space Station. However, much might be learned by examining the environments and tools for other languages such as LISP and C in an effort to build better tools for Ada. After all, given that Ada is a sufficiently universal language, it can be made to look like other languages.

CONCLUSION

It is difficult if not impossible to directly solve the cultural collisions that are bound to occur in the interaction of programming languages, and paradigms. Those cultural collisions will not be resolved by attempting to enforce a uniform programming language and culture. An alternative, however, is to build tools that remove the programmer from direct contact with the programming language. This removal can allow the tool user to overcome the cultural problems, while still allowing the production of code in a desired language. If and when such tools become available, the questions with which this essay started will be displaced with the question, "What can your tool do?"

ACKNOWLEDGEMENTS

This research has been partially funded under NAS8-36955 (Marshall Space Flight Center) D.O. 34 "Applications of Artificial Intelligence to Space Station."

REFERENCES

1. BOEHM, B.W. "Seven Basic Principles of Software Engineering," *The Journal of Systems and Software* 3, (1983), pp. 3-24.

2. BOOCH, G. *Software Components with Ada: Structures, Tools, and Subsystems*. The Benjamin/Cummings Publishing Company, Inc., Menlo Park, California, 1987.

3. Brintsenhoff, Alton; Christensen, Greco, Joe; Steve; Mangan, John. "The Use of Ada Concurrent Processing Features in an Implementation of Parallel Tree Searching Algorithms. *Proceedings of the Third Annual Conference on Artificial Intelligence and Ada*. George Mason University, October, 1987.

4. FAIRLEY, R.E. *Software Engineering Concepts*. McGraw-Hill Book Company, New York, 1985.

5. HOWDEN, W.E. "Contemporary Software Development Environments," *Communications of the ACM*, 25, 5 (1982), pp. 318-329.

6. REEKER, LARRY H.; KREUTER, JOHN; WAUCHOPE, KENNETH. "Artificial Intelligence: Pattern-Directed Processing." Final Report AFHRL-TR-85-12 Air Force Human Resources Laboratory, Lowry Air Force Base, Colorado. May 1985.

7. SCHWARTZ, RICHARD L AND MELLIAR-SMITH, P.M. "On the Suitability of Ada for Artificial Intelligence Applications." Final Report for Defence Advanced Research Projects Agency Contract DAAG29-79-C-0216. July 1980.

8. STROUSTRUP, B. "What is 'Object-Oriented Programming'?," *ECOOP '87: European Conference on Object-Oriented Programming, Paris, France, June 15-17, 1987, Proceedings*. [Bézivin, J., P. Cointe, J.-M. Hullot, and H. Lieberman (Eds.)]. *Lecture Notes in Computer Science (276)*, Goos, G. and J. Hartmanis (Eds.), Springer-Verlag, Berlin, 1987.

9. WARREN, D. H. D.; PEREIRA, L. M.; PEREIRA, F. "PROLOG - the language and its implementation compared with LISP." *Proceedings of the ACM Symposium on Artificial Intelligence and Programming Languages*, Rochester, New York, pp 109-115. 1977.

10. ZELKOWITZ, M.V., A.C. SHAW, and J.D. GANNON. *Principles of Software Engineering and Design*. Prentice-Hall, Englewood Cliffs, New Jersey, 1979.