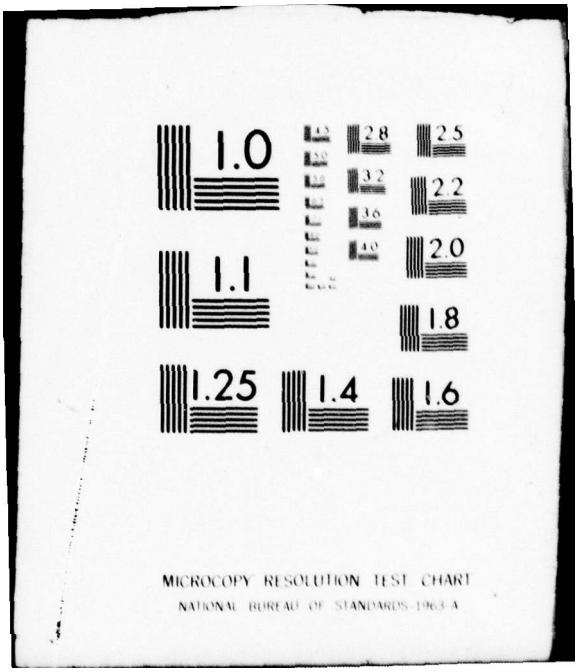
AD-A072 248 INSTITUTE FOR DEFENSE ANALYSES ARLINGTON VA SCIENCE A--ETC F/G 9/2
INITIAL THOUGHTS ON THE PEBBLEMAN PROCESS.(U)
JUN 79 D A FISHER, T A STANDISH MDA903-79-C-0202

UNCLASSIFIED IDA-P-1392 IDA/HQ-78-20933 NL

| OF |
AD
A072241

END
DATE
FILMED
9-79
DDC


1.0
1.1
1.25
1.4
1.6
2.8
3.2
3.6
4.0
2.5
2.2
2.0
1.8
MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

(12) LEVEL III

IDA PAPER P-1392

INITIAL THOUGHTS ON THE PEBBLEMAN PROCESS

David A. Fisher
Thomas A. Standish

June 1979

Prepared for
Defense Advanced Research Projects Agency

INSTITUTE FOR DEFENSE ANALYSES
SCIENCE AND TECHNOLOGY DIVISION

The work reported in this document was conducted under contract MDA 903 79 C 0302 for the Department of Defense. The publication of this IDA Paper does not indicate endorsement by the Department of Defense, nor should the contents be construed as reflecting the official position of that agency.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE	READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER	2. GOVT ACCESSION NO. 3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) Initial Thoughts on the PEBBLEMAN Process	5. TYPE OF REPORT & PERIOD COVERED FINAL Jan. 1978 - Dec. 1978
	6. PERFORMING ORG. REPORT NUMBER IDA Paper P-1392
7. AUTHOR(s) David A. Fisher Thomas A. Standish	8. CONTRACT OR GRANT NUMBER(s) MDA 903 79 C 0202
9. PERFORMING ORGANIZATION NAME AND ADDRESS INSTITUTE FOR DEFENSE ANALYSES 400 Army-Navy Drive Arlington, Virginia 22202	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS DARPA Assignment A-37
11. CONTROLLING OFFICE NAME AND ADDRESS Director, Information Processing Techniques Defense Advanced Research Projects Agency 1400 Wilson Blvd., Arlington, Virginia 22209	12. REPORT DATE June 1979
	13. NUMBER OF PAGES 71
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)	15. SECURITY CLASS. (of this report) UNCLASSIFIED
	15a. DECLASSIFICATION DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

DDC RECEIVED AUG 3 1979 B

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

None

18. SUPPLEMENTARY NOTES

N/A

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Ada, common language, software design, software maintenance, software environment, software quality, software life cycle, program development tools, language standards, language support and culture, software tools

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

The purpose of this paper is to stimulate thinking and participation in the planning and development of an effective software environment to accompany the DoD common high order programming language. It examines several views of programming environments from different perspectives, catalogues a number of issue areas, asks some as yet unanswered questions, and provides some initial analyses. The tentative conclusions encompass twelve goals that appear to be important for an effective software

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

20.

development and maintenance environment.

IDA PAPER P-1392

INITIAL THOUGHTS ON THE PEBBLEMAN PROCESS

David A. Fisher
Institute for Defense Analyses
Thomas A. Standish
University of California--Irvine

June 1979

INSTITUTE FOR DEFENSE ANALYSES
SCIENCE AND TECHNOLOGY DIVISION
400 Army-Navy Drive, Arlington, Virginia 22202

Contract MDA 903 79 C 0202
DARPA Assignment A-37

PREFACE

This paper was prepared for the Director, Information Processing Techniques, Defense Advanced Research Projects Agency, under IDA Task Order A-37. The work was part of a continuing study effort to provide technical support to the ARPA Software Technology Program, within which ARPA exercises technical direction for DoD technology base R&D in this area.

ACKNOWLEDGMENTS

The authors would like to thank the many individuals that have contributed directly and indirectly to this paper. These include those who have commented on the various iterations of the "PEBBLEMAN" document, the attendees at the Irvine Workshop on Alternatives for Environment, Certification and Control of the DoD Common High Order Language, and those who have commented on the 3 January 1979 draft of this paper. We would especially like to acknowledge the thorough and insightful reviews and comments provided by Peter Elzer and by Ian Pyle.

CONTENTS

EXECUTIVE SUMMARY

1. Purpose and Rationale

Our purpose is to stimulate thinking and participation in planning and developing an effective environment for *Ada*--the DoD common high order programming language. The design of *Ada* was chosen May 3, 1979 under the direction of the High Order Language Working Group (HOLWG) of DoD. The effectiveness of any programming language depends critically on the quality of the software development and maintenance environment in which it is used.

2. Scope

In attempting to stimulate reflection on environment issues, we do the following:

- We give eight views of environment requirements from different perspectives and we consider interactions between the views.
- We catalogue topical issue areas, in outline form, for an initial attempt at completeness.
- We ask a variety of questions that need to be answered.
- We give some initial analyses leading to tentative conclusions.

3. Tentative Conclusions

We have tentatively concluded that there are twelve important goals for the *Ada* software development and maintenance environment. Four of these are:

Commonality -- to leverage tool development for widespread use, commonality for the environment is no less important than commonality for the common language.

Efficiency -- to be satisfactory for production use, the environment and its tools must consume reasonable computer resources in relation to the services they provide.

Uniformity -- to assure appeal, acceptability, and survival, the environment and its tools must adhere to consistent interface conventions.

Open-endedness -- to avoid technological obsolescence, the environment must retain capacity to absorb new tools.

4. Observations

Timeliness, quality of analysis, depth of analysis, and completeness are critical issues in the environment planning process.

The planning process is both broad and deep. It requires expertise in diverse areas. Expertise can only be brought to bear deeply in all these areas in the limited time available by inviting a broad audience of experts to contribute the depth required, each in his own domain.

5. Appeal to the Reader

It is likely that the critical decisions on the *Ada* environment will be made during the next two years.

We ask for your help. We invite you to contribute to the analysis and planning by identifying a limited aspect of the effort and by furnishing in-depth criticism and suggestions.

- A manager might dwell on administrative and policy issues in Part I of PEBBLEMAN.

- A generalist might concentrate on the question of completeness of the issues being considered.
- Those interested in technical issues might develop a particular technical topic for the environment.

What you the readers contribute might critically impact the quality and success of the effort.

1. INTRODUCTION

This document gives some thoughts concerning the process for determining the characteristics that the common language environment should have. The term "requirements" will be used to refer to the system of characteristics, goals, and constraints that will result from such a process.

The PEBBLEMAN process has been underway for over eight months and already has resulted in considerable creative ferment. Our initial thoughts are designed to help the maturing of considerations already well begun and to add to the results of much previous work and activity. As this paper is written, the PEBBLEMAN document is already in its third iteration, and each time it has been reissued it has shown marked improvement.

However, the PEBBLEMAN document is a "requirements statement" document, not an "analysis and rationale" document. As such, it does not attempt to provide background, rationale, or justification for the choices it makes implicitly. We feel there is a need in the PEBBLEMAN process for a supplement to the "requirements statement" document--a supplement which spells out the justification for PEBBLEMAN's choices and encourages widespread scrutiny and participation in the processes of careful analysis of alternatives and well-reasoned choice. We hope this document provides an initial background for such a rationale and justification document and that it will provide useful source material for further consideration.

We view the environment quite broadly. For the purpose of this paper, we define the Common Language Environment (CLE) to

be the set of policies, procedures, agencies, and tools that: (1) control the common language standard and validate common language translators, and (2) provide supporting activities and aids to develop programs for all applications of the common language--small, medium, and large.

Devising good environment requirements is a complex process because environments themselves are complex and serve many purposes. In fact, as we shall see, the scope of the issues to be addressed in the environment effort is enormous, and many separate facets of the environment interact with each other. In order to come to grips with the issues, we shall try to "look at the environment from many angles", so to speak.

Thus, our method of exploring environment requirements issues will follow a course in which we shall adopt a succession of intellectual vantage points, each of which leads us to consider a particular view of common language environment requirements. After examining the various views separately, we then attempt to raise issues about how the views interact.

Specifically, our course of action will be to consider the common language environment requirements from the following eight vantage points:

1. Urgency of the Task
2. Software Quality
3. Software Life Cycles
4. Management Disciplines
5. Program Development Tools
6. Maintenance and Enhancement
7. Standardization and Control
8. Language Support and Culture

When we view common language environment requirements from these eight different vantage points, different aspects of the environment requirements are revealed. As these various aspects

are revealed, we attempt to write down tentative requirements statements that capture the observations being made in a form suitable for stimulating further debate and reflection.

Section 3 of the paper explicitly treats the interactions between the separate views in Section 2.

Section 4 presents an outline of issues to consider with an eye toward completeness in the requirements process. In Section 5, we throw out for consideration some experimental observations and general principles in the hope that they might prove helpful in the search for good environment requirements.

In Section 6, we give suggestions for tasks that need doing in order to advance the thinking process about the common language environment.

2. VIEWING ENVIRONMENT REQUIREMENTS FROM EIGHT VANTAGE POINTS

2.1 URGENCY OF THE TASK

It is highly important that timely, complete, well-thought-our requirements for the Common Language Environment (CLE) appear in the near future, so that by the time the final common language design is selected and contracts are let to produce translators, tools, and libraries, the environment issues will have been sufficiently explored and the environment requirements sufficiently specified to assure the success of at least an initial working environment supporting common language processors.

The time frame for this process is short compared to the time frame over which the requirements for the common language were evolved. There is insufficient time to go through as many iterations (STRAWMAN, WOODENMAN, TINMAN, IRONMAN, STEELMAN) as were performed in the careful evolution of the common language requirements. Yet, environments are perhaps less well understood than programming languages, and the feasible technologies for the CLE are perhaps more diverse than those for the common language itself.

It is highly critical, therefore, to achieve high quality in short time through collection, integration, and packaging of good ideas about the CLE. The successful outcome of the common language effort may hinge critically on the quality and timeliness of the environment requirements effort.

The CLE requirements analysis effort must blend cooperatively with applicable parts of the 5000.29 directive. We see

no difficulty with this, as will be explained subsequently in Section 2.4.

PEBBLEMAN must receive careful evaluation in the near future, and its contents must be justified in terms of the goals the CLE should achieve. This document attempts to help make progress in this direction by providing a framework of ideas and technical perceptions against which PEBBLEMAN may be evaluated and which can assist in its orderly, careful, rapid evolution.

In pursuing timely results, it may be valuable to attempt to partition the overall environment effort into subareas that can be tackled in parallel in order to help produce the answers we need on time.

The division into parallel areas does not imply that the areas are independent. In fact, in some instances, strong dependencies obviously exist. The first principle used to produce the division into subareas was the expertise that must be brought to bear to derive the requirements for each subarea.

For instance, Section I, below (from PEBBLEMAN), is concerned with administrative, management, and policy issues, while Sections II and III deal with technical issues. This illustrates a cleavage along managerial and technical lines.

Sections II and IV represent those things of concern to those who have responsibility for development and maintenance of applications, while Sections I and III are of concern to those who must enforce standards or manage application projects.

Another partitioning relates to time-critical decisions. Those are reflected in Sections I and IV.

I. Administrative, Policy, & Management Issues

These issues include administration, policy, and management for common language standards and control; for translator

validation and control; for development, maintenance, and distribution of software tools and standard libraries; for support of user communities; for software life-cycle management; and for software procurement.

II. Software Development & Maintenance Tools Environment

These involve the technical issues relating to the structure of the environment; the portability of the environment; the user interface; common characteristics of all tools; software analysis, query, and display tools; translation and optimization tools; software test and evaluation tools; software documentation tools; and execution tools.

III. Computer-Based Tools and Aids to Support Administration, Policy and Management

These are technical issues relating to tools and aids to support both the common language effort and software applications. Included are translator validation tools and technology, translator performance evaluation tools, configuration control tools, life-cycle management tools, and technology for language standards.

IV. Initial Critical Tools and Aids

These include the least general core of tools that must be in place to support use of the common language at the date it is added to the approved list and released for production use. The identification of this core of tools is time-critical. In addition, the selection of the core will establish conventions on how the environment grows and develops and will determine how open-ended the environment is. It will be an important determinant of the acceptability of the language and environment. Therefore, time-criticality is not the only important critical issue to be considered here.

2.2 THE SOFTWARE QUALITY VIEW

The overall goal we are trying to achieve by implementing and using the common language and its environment is to improve the quality of defense-embedded system software in the context of changing requirements and long maintenance lifetimes.

Software quality has a number of general dimensions such as (1) reliability and correctness, (2) efficiency, (3) unit cost, (4) maintainability, (5) transferability, (6) responsiveness to user needs, and (7) timeliness of delivery.

In addition to these general software quality properties, defense-enbedded system software has particular important quality goals of its own, including quality in meeting requirements for real-time constraints, parallel processing, fault-tolerance, self-diagnosis, and modifiability, i.e., capacity to sustain variation to meet changing embedded system requirements.

Both the common language and its environment must be designed to help produce embedded-system software that meets both these general software quality goals and these special defense system goals.

Pursuit of software quality along these various dimensions may involve tradeoffs. For instance, increasing efficiency may trade off against low unit cost. On the other hand, pursuit of some goals may simultaneously help in the pursuit of others. For instance, enhancing maintainability during pre-release development may lower overall life-cycle unit costs, and production of software that is initially efficient, correct, and responsive to its requirements increases chances that it will be delivered on time.

In general, in considering software quality goals, no single goal is to be pursued at the expense of the others and no single optimization criterion can be established. There is no point in maximizing one dimension of quality at great expense

when so doing reduces some other dimension of quality below a required threshold level.

Thus, we should view software quality goals as simultaneous constraints to be satisfied rather than as measures to optimize independently of one another. (Here, to avoid confusion, we should note that software quality in the applications written in the common language is the primary emphasis.)

Some required properties of the environment may involve indirect rather than direct support of the primary software quality goals. For example, if we look at the property of "simplicity", either in the common language or in the environment, we can see that "simplicity" is not an end in itself. Rather, it is a goal, which if achieved, supports other software quality goals such as implementability, ease of learning, low risk, and maintainability.

Thus, when we give environment requirements, we may attempt to justify them either by direct appeal to the primary software quality goals they help to achieve, or by indirect reasoning chains that show how they help achieve the primary goals by achieving one or more intermediate goals. Intermediate goals such as simplicity, generality, open-endedness, and ease of learning, are examples of goals whose achievement supports the achievement of the primary software quality goals indirectly.

In summary, we need to have a framework for justifying required properties of environments using goal chains, showing how required environment properties and features support overall software quality for embedded systems and help embedded software system projects to satisfy systems of simultaneous software quality constraints.

To illustrate this process, let us consider one specific technical subarea--that of software development and maintenance tools. By analogy with the approach used in STEELMAN (STEELMAN

1978), we have devised a tentative set of general characteristics desirable for development and maintenance tools. We intend to use these as an operational set in our reasoning about lower-level properties that the tools might have.

In honesty, we admit that the analysis underlying the choice of these high-level characteristics has not yet been completed, and it may be necessary to modify the set in light of new evidence or in response to well-reasoned challenge. We have selected this set based on our own current thinking and experience, and we provide later only a minimum of support for it in the form of a few paragraphs of illustrative reasoning.

2.2.1 General Characteristics for Tools Environment

We present the following set of general characteristics for the development and maintenance environment. In this set, we will use the term "Initial Environment" to refer to the core of essential tools and properties that will be released at the time of language approval and that will provide the basis for subsequent growth of the environment. We will use the term "Environment" to refer to the invariant characteristics the environment should have throughout its lifetime. Both terms refer to the host environment for embedded computer applications.

1. Simplicity

   The Environment should have a simple overall structure that involves a minimum number of concepts. The Environment should be composed of constituents that are individually as simple as possible consistent with the services they are required to provide. The rules of combination should be few in number, simple in nature, and consistent in form. The interface conventions between users and tools and among tools should be simple.

2. Low Risk

There should be an Initial Environment that provides minimally-necessary development and maintenance functions and is composed from proven low-risk technology. The Initial Environment should have a high probability of working as intended and of absorbing later useful extensions in a consistent manner. It should be devoid of novel, untested features, and should not depend on high-order interactions between features not previously used successfully in combination.

3. Least Generality

The Initial Environment should be no more general than necessary to support both development and maintenance of software in embedded computer applications and in the support environment.

4. Supportiveness

The Environment should provide a smoothly coordinated, complete set of useful and adequate software development and maintenance tools and a flexible software data base. The Initial Environment should establish a minimally necessary core of generally useful development and maintenance tools that have minimum risk, immediate payoff, and conspicuous benefits to potential users.

5. Open-Endedness

The design of the Initial Environment should be sufficiently flexible to ensure that specialized, computer-based tools for management, documentation, training, testing, and early life-cycle design and analysis disciplines, as well as software design and maintenance, can easily be incorporated. As the Environment is modified in response to new technology it should retain the flexibility and open-endedness of the Initial Environment.

6. Implementability

The Environment should be implemented entirely in the common language and should be designed to exist on a wide variety of substantially different computers.

7. Commonality

There should be a Standard Environment that is common to all users of the common language and that is maintained and distributed as part of the standard library of the language. (Note: Success in deriving the benefits of commonality depends critically on there being a method for maintaining and enforcing standards for the Environment and the library as well as for the common language.)

8. Efficiency

The Environment should execute efficiently on a variety of machines. Individual tools should run efficiently and should consume reasonable computer resources in relation to the services they provide.

9. Uniformity

The interface between users and tools should be uniform across tools, consistent with the services those tools provide. In addition, the same concepts, syntax, and semantics should be used in both the language and the Environment, and the conventions and mechanisms for transactions between environmental components and between the language and the Environment should be the same as those for transactions between language features. (Note: The latter characteristic addresses only the form of interaction between components of the Environment and does not necessarily imply that the syntax of the common language is appropriate for the linguistic interface between the user and the Environment tools.)

10. Descriptive Equality

The design of the Environment shall assure the ability to develop, test, and maintain Environment tools written in the common language. That is, the environment design shall not forbid users of the common language from writing and using tools drawing on any capability used by other tools in the Environment. (Note: Such usage might be precluded, however, by policies, procedures, or protection systems in specific situations.)

11. Ease of Learning and Use

It should be easy to learn and use the Environment. (Note: Simplicity and uniformity of the user interface support this characteristic.)

12. Style of Use

The tools Environment must be appropriate for both "batch" and "interactive" use.

2.2.2 Examples

We now provide limited examples of some analysis showing why some of the above requirements are justified.

For instance, let us begin by reasoning about "Risk". If the status quo technology is satisfactory to solve the problems at hand, it can be used with minimum risk. However, if the status quo technology is insufficient to solve current problems, then change is required, and change always implies risk. Since the use of the common language involves change to solve problems better, we must address the issue of minimizing risk associated with the change.

However, the risk associated with the common language Environment involves both short-term risk and long-term risk. The short-term risk is that the Initial Environment will not work properly and will jeopardize the common language effort. The

long-term risk is that the Environment will be insufficiently flexible to absorb new tool technology to keep pace with improvements, leading to weakened competitive posture. Both such risks must be minimized.

We see, in the "Low-Risk" requirement above, an attempt to address the short-term risk. We also see, in the "Open-Endedness" requirement above, an attempt to address the long-term risk.

The requirement for "Least Generality" helps achieve low-risk in the Initial Environment by limiting the objectives and scope of the Initial Environment. Such a limitation of the initial objectives also increases chances of meeting the time-critical constraints for Initial Environment release.

The requirement for "Commonality" can be justified by reasoning about economic and other benefits. If commonality can be justified in compelling terms for the common language (Fisher 1978, STEELMAN), it can be justified in no less compelling terms for the Environment.

For example, we know that the reason many programs written in a given language fail to be transferable is that they are written to depend on many features of the object environment. If a program is to be transferred to a new object environment with different features (for, say, device types, file structures, I/O conventions, and so forth), then it is the interconnections to these object environment features that have to be rewritten to perform substantially the same functions before the program can be transferred. Thus, we conclude it is important to have commonality in the object environment whenever we require portability of application programs.

Let us look at a particular application, namely, the application in which we develop the tools for the host environment of the common language.

In this application, there are many potential users, i.e., all users developing applications which use the common language. Consequently, it is extremely important that the object environment for the tools be portable. In fact, the major economic benefits of the common language effort are expected to derive from the portability of these tools. We therefore conclude that commonality is an essential requirement for the object environment of the tools.

We also observe, however, that the object environment for the tools is, by definition, the host environment for the common language.

Thus, the requirement for "Commonality" of the object environment of the tools is a requirement for "Commonality" of the host environment of the language.

Another approach for reasoning about the need for "Commonality" in the host Environment is by analogy to the need for a common language. The need for commonality in the language derived from increased leverage in training, economic incentives for tool development and application libraries, reduction in sole-source enhancement contracting, increased transferability of personnel across projects, and so forth.

These needs apply just as well to commonality of the host environment, as is revealed by the reactions of a number of users of Ken Bowles' Pascal Environment--this being a working example of the successful practice of host-environment commonality. Some of his users have commented on the convenience of being able to shift their software development activities from one machine to another without having to change a thing. Bowles' PASCAL environment is implemented on a variety of machines, including small "smart terminal" desk-top minis, and big systems development machines. Since the language and its environment are standardized across these implementations, the user can shift from machine to machine without having to change job

control languages, file system structure and naming conventions, control character usage conventions, and so on.

Being able to transact in a standardized program development environment with standardized file name conventions and job control conventions makes it easy for programmers to change implementation development machines, and makes program development tools genuinely transferable.

The National Software Works (NSW) illustrates another attempt at standardizing development environment characteristics across varying machines and host operating systems. Aspects of the NSW philosophy and design may well be a good model applicable to the CLE requirements effort. There may well be immediate chances for technology transfer of design concepts explored in the NSW.

Earlier in reasoning about commonality, we discovered a requirement for portability of the software tools. The common language, however, is a vehicle for implementing software that is portable. Thus, the requirement for portability of the tools can be satisfied by implementing the tools in the common language. Such reasoning leads to the general goal of the "Implementability" statement given above.

We hope that these limited examples have illustrated the flavor of the reasoning needed to justify the choice of general goals for the development and maintenance tool environment.

2.3 THE SOFTWARE LIFE-CYCLE VIEW

Let us reflect for a moment on how the common language and its environment fit into the overall picture of the software life cycle.

As a prelude to further discussion, let us take a brief look at the activities that go on in each of several life-cycle stages. Since many different breakdowns of activities and many different systems of titles for life-cycle phases are in use in various parts of the literature, we have chosen to follow the life-cycle phases and titles in use in DoD directive 5000.29. (Note: Wherever there are inconsistencies between the usage that follows and the usage in 5000.29, the differences are to be resolved in favor of 5000.29.)

2.3.1 Requirements Analysis

In requirements analysis we are trying to achieve a high-quality statement of the true capabilities and properties of the system that are needed by its users and operators. We are trying to conceive of everything that is relevant and nothing more. We attempt to generate a set of requirement statements with at least the following properties: (1) completeness, (2) consistency, (3) unambiguity, (4) correctness, and (5) legibility. Good requirements do not overconstrain the set of valid implementations. Good requirements analysis is a keystone activity in the software life cycle because ill-structured, unfathomable requirements preclude validation and verification at subsequent project stages.

2.3.2 Specification

Specification is the enumeration of specific, quantified behavioral constraints a system must satisfy in order to meet its requirements. Such specific measurable properties are necessary to provide operational tests during validation and verification in order to determine whether or not designs and

implementations produced at subsequent project stages satisfy the requirements. Good specifications are: (1) complete, (2) unambiguous, (3) minimal, (4) legible, and (5) sufficiently specific to be testable whether or not they are satisfied. Specification is a keystone activity in the software life cycle because, in the absence of specific measurable properties a system must exhibit to meet its requirements, it may not be testable whether a system satisfies the requirements.

2.3.3 Design

A design is a representation of an artifact or system. Designing is the art of constructing and evaluating designs to meet constraints and satisfy purposes.

In the software life cycle, we try to produce designs: (1) to satisfy the specifications and requirements, (2) to prescribe further implementation activities, and (3) to exhibit certain essential properties (namely, completeness, consistency, legibility, technical feasibility, unambiguity, and susceptibility to analysis and evaluation).

We use designs to produce subsequent tasking schedules and communication disciplines for subsequent project stages--such as implementation, testing, integration, and validation. We also try to produce designs that are susceptible to such forms of subsequent analysis as cost estimation, performance estimation, and determination of physical system characteristics.

Good design is a keystone activity in the software life cycle since: it provides a means to organize subsequent project stages and activities, it permits system synthesis errors to be caught at early stages where correction is relatively inexpensive in comparison to the cost of repair downstream, and it enables task schedules, communication disciplines, and resource performance estimation to take place.

2.3.4 Implementation

Implementation is the construction and debugging of a working executable representation of the system, which is a concrete realization of the design. Implementation is a project phase most intimately and directly related to the common language and its environment. During implementation, we not only try to produce a concrete design realization that satisfies the specifications and meets the requirements, we also attempt to produce an artifact that enhances the ease of subsequent maintenance and modification. Thus, we attempt to construct the artifact in such a fashion that its pieces can be readily understood and which has the capacity to sustain future variation in response to changing requirements.

2.3.5 Module Testing and Integration

During module testing and integration, we attempt to assure that the modules have correct behavior, that they satisfy performance specifications, and that they cooperate correctly together to achieve acceptable overall system performance. Testing and integration is a keystone phase of the software life cycle because it assures software quality goals are met prior to system release.

2.3.6 Maintenance and Enhancement

During Maintenance and Enhancement we attempt to alter the system to meet new behavioral requirements and to remedy defective properties revealed during system usage experience. The activities involved during maintenance may recapitulate any and all of the previous life cycle activities. Thus, attempting to modify the system to meet a new requirement may entail reenactment of requirements analysis activities, which, in turn, can lead to respecification, redesign, reimplementation of parts of the system, and new test and integration activities. On the other hand, fixing an implementation bug may not entail redesign, but only reimplementation, retesting and integration.

Typically, when a life-cycle decision is remade, it causes all successor life-cycle activities to be reenacted, whereas none of the predecessor activities need be reconsidered.

Maintenance activities are enormously varied in scope and a wide range of skills come into play. In the software maintenance phase, there is frequently emphasis on trouble shooting, configuration control issues, control of new software releases, software change requests, management of trouble reports, training, and documentation. As we will see later in the "maintenance view", special tools and properties of the environment may be needed to manage the special activities that tend to occur during maintenance.

Because we live in an imperfect world, where requirements are never likely to be complete or accurate, designs are never likely to be correct, and implementations are never likely to satisfy the requirements and reflect the design intentions perfectly, we must resort to special measures in order to improve quality progressively.

Thus, test and integration, on the one hand, and maintenance, on the other hand, occupy prominent roles because of imperfection in the earlier stages.

At a deeper level however, there are feedback loops between the activities in the life cycle that help us incrementally to improve the understanding and quality achieved at each stage. Thus, we may only really begin to understand the true requirements when we are exposed to the behavior of an implementation. Cyclical exposure to the behavior of the artifacts we build may be necessary to achieve understanding of the true requirements, especially for a system we are attempting to synthesize for the first time.

In the next view (from the perspective of management disciplines), we reflect on quality assurance practices necessary

to detect and remedy flaws in the quality of products of each of the life-cycle phases as they participate in feedback learning and improvement loops. These considerations will be seen actually to derive from the interaction of the life cycle and management views.

When we look at the environment from the life cycle point of view, we see the environment must furnish a framework in which the visible products of each of the life-cycle stages can be produced and incrementally improved in feedback loops. When the life-cycle view interacts with the management discipline view, we see that management disciplines must provide overall guidance and communication methods to schedule activities, configure resources, engage in quality assurance monitoring, and to adjust incremental effort to achieve the required quality in each of the visible life-cycle stage products.

If we anticipate that requirements documents and design documents will change in response to feedback learning from later life-cycle discoveries, then it profits us to consider the desirability of trying to capture requirements and design documents in machine-manipulable form, and to have editing tools available with which to keep them up to date as the project evolves.

In summary, because each life-cycle stage takes place in a context of imperfect predecessors, we are led to consider feedback loops which lead to incremental improvement of the products of the earlier stages. These interplays have impacts both on the tools we need in the environment and on management disciplines needed to manage activities at each of the life-cycle stages. A lot of serious problems would go away if we lived in an idealized world, but we see that part of the complexity comes from the "boundary conditions" in dealing with imperfect and incomplete information.

2.4 THE MANAGEMENT DISCIPLINE VIEW

Good management disciplines may be the single most important factor in assuring the success of software projects of substantial size.

It is important to emphasize that we are concerned with large-scale activities involving substantial numbers of different project personnel in many different specialties over a substantial number of years.

Thus, we are considering "programming in the large" as opposed to "programming in the small".

In situations where there is little or no personnel turnover, where the maintainers are the same as the implementers, and where the total system size is modest, we may have little true need for careful documentation, validation checks by independent teams, training materials, and so forth.

But, as a project increases in size and scope from a few people over a few years to many people over many years, and as project difficulty and novelty increase, the nature of the management disciplines required for success may have to change shape dramatically.

We have just seen, in considering the life-cycle view of the environment, that the visible products of each life-cycle stage must feed and control the activities in the next, and that imperfection in the quality of each stage leads to a need for management disciplines that provide quality assurance monitoring, and manage feedback loops aimed at incremental improvement of earlier life-cycle products.

Appropriate management disciplines for situations where we face "programming in the large" are beginning to be understood (Boehm 1973, DeRoze & Nyman 1978), and experience is accumulating with software engineering practices that show substantial promise.

Fortunately, the thinking about software management in DoD appears to be in excellent shape. Specifically, DoD Directive 5000.29 establishes policy for DoD software management and control, and a number of impressive initiatives have been taken (DeRoze & Nyman 1978) in the direction of making software policy, procedure, and practices into a true engineering discipline.

For instance, the Defense System Software R&D Technology Plan establishes the role of Principal Technology Agent (PTA) to monitor projects, to keep abreast of the state-of-the-art, and to advise DoD technologists, users, and developers on software technology matters in each of a number of defined areas of responsibility.

For example, the Air Force is the PTA for requirements analysis, the Army is the PTA for life-cycle management planning, and the Air Force is PTA both for management control technology and for cost/quality data collection/analysis (DeRoze & Nyman, p. 314).

Thus, an important matter for the CLE effort is to assure that the environment requirements blend smoothly and cooperate with the overall 5000.29 effort.

Whichever management disciplines are selected to assure software quality, the environment must assist managers in enforcing selected disciplines, in monitoring both performance and progress, in adjusting schedules and in taking remedial actions, and in getting thorough, informative, and timely feedback and reports.

While some management disciplines are best carried out by people following policies and procedures, others quite properly belong to the domain of automated or semi-automated computerized support.

For instance, the program development portion of the environment can measure cumulative resources spent by various

people and teams during implementation, and it can prepare management reports contrasting resources used with estimates of resources required.

By means of exception reporting and critical path scheduling algorithms, it can inform management of problem areas needing attention and remedial action.

A computerized medium can also be useful for thorough, precise version control, and for reporting progress on validation checkpoint activities. Since these sorts of activities are highly important to effective management support, they should be thought about carefully in the context of the environment requirements process.

Thus, in viewing the environment requirements from a management viewpoint, we see that a computer-mediated environment must be designed in a way so as not to preclude the "hooks and handles" needed by accounting tools, critical path scheduling tools, version control and file access control tools, validation checkpoint monitoring and reporting tools, and other such computerized management aids.

While it may not be realistic (from the perspective of the short time for planning that remains) to require that the Initial Environment contain a complete battery of computerized management aids, it is certainly not unrealistic to require that environment designs not preclude extension along these dimensions.

One might even look for current environment models used in present practice, with file structures and accounting systems that are adequate to support such management aids, and one might require environment designs to allow for adequacy of extension in these directions.

2.5 THE PROGRAM DEVELOPMENT TOOL VIEW

We know that tools are necessary for program development and maintenance. Program text must be composed and modified during the development and debugging process. Program modules may need to be compiled, loaded and linked. Test data sets are needed against which to check out program performance. Program performance may need to be measured and program optimization and improvement may be required as a result.

Let us focus on the activities that take place in a software project. Most software projects utilize what might be called a "Project Data Base". Such a data base may involve both "off-line" entities, such as listings and card decks, and "on-line" entities such as disk files. In general, a Project Data Base contains objects that are manipulated by Tool activities. The data base is a communication medium that records objects both between and during project programming activity sessions. As the project evolves, the tools are used to create and act on entities in the project data base to evolve final representations of the deliverable system and its supporting documentation, and perhaps later to support maintenance and modification in the context of changing requirements.

In short, we can think of tools as agents that support activities that create and modify project data base entities, and of the project data base as a repository of progressive results of tool activities. The tools and the data base thus play a dual, supporting role in the evolution of a software project from birth to death.

Thus, in our discussion of the issues that follows, we divide considerations into those that deal with the properties that the project data base should have, and those that deal with properties the tools should have. We begin with consideration of the project data base.

2.5.1 Project Data Base Properties

A project data base may contain a number of useful kinds of data such as: (1) representations of programs in partial stages of completion, (2) test data sets, (3) text and documentation relating to task requirements, specifications, and design, (4) records of activity, progress charts, version control records, and project workbook data, (5) suspended environments from previous sessions awaiting further processing, and, in general, (6) any file of data (in any of a range of permissible file formats, such as text, formatted collections of records, etc.) required to support any task or activity related to program development and maintenance.

In many contemporary operating systems, such a collection of data can be stored in file system groupings related to individual programmers, programmer teams, system modules, programming tasks, and entire programming projects. File system groupings related to one of these entities usually have directories that index a collection of named files possessed and controlled by the entity.

It is important to have a file system design that can support, in a flexible fashion, the task management structure for an overall software project. The file groupings and hierarchies must span the gap from an individual programmer's private version sequences for a given programmer task, on up through the subgroups and subproject structure, and eventually to that supporting the top management structure and the overall software project library.

The file entities at the various levels may differ dramatically. For instance, a given individual programmer at the lowest level may wish to have a "user profile" which specifies switch settings, terminal characteristics, and other option selections suiting his programming style and individually tailored usage conventions. At the topmost level, managers may want to transact

with critical path scheduling data, and with quality assurance and cost estimation data. Again, project librarians may wish to transact with files of module version control data to assure traceability of module authorship and to enforce project test and validation procedures.

The file system must permit implementation and enforcement of various access and installation policies. Module version control demands version installation privileges must be associated with traceability controls. Denial of access rights must be implementable selectively for various file system entities. File protection and access control must have sufficient granularity to permit fine-tuned selective control over read-write access by each file group user class.

Here, by granularity, we mean the size of the entities that may have their access controls independently varied. Granularity must cover a sufficient range to permit practical tools to be built to implement file control policies. (By analogy, we can create a complete text editor that transacts only at the level of individual characters. However, unless the granularity of a text editor also includes entities such as lines and pages, the text editor is unlikely to be a practical tool. By the same token, if the granularity of manipulable text units is too coarse (say, pages only, but not lines or characters) the text editor again becomes impractical. Only when granularity covers a sufficient range of groupings of units into composites can a practical tool be specified. The same is true for the granularity of file system groupings.

If project management is to exercise a task management policy stressing high visibility, progress measures, careful validation, and early detection and correction of poor performance, there must be hooks and handles on file system groupings to report resource estimation data and data on actual resources spent. Management reporting tools must have access to these hooks and handles.

This leads us to a view of requirements for the file system needed to support the use of the common language on big software projects in the context of extended maintenance lifetimes and in the context of changing requirements.

A project data base file system must:

(1) Support a range of file formats, including:

(a) text for documents
(b) program representations used by tools
(c) test data sets
(d) program libraries
(e) module version control records
(f) management reporting data
(g) project and task control data.

(2) Support a range of file groupings sufficient to implement a task subdivision policy selected by management (e.g., groupings for programmer data bases, team data bases, project data bases, enterprise data bases, accounting data bases, management data bases, and the standard library for the common language).

(3) Provide capacity to implement file access control policies for reading and writing at sufficient granularity to be a practical tool.

(4) Be implementable by representations and programs in the common language.

Since we are searching for simple, general, efficient structures to use in implementing the CLE, the following note may be of interest:

As a philosophical simplification, it may be useful to talk abstractly about a CLE file system as a possibly hierarchical collection of file groupings in correspondence with different user entities, such that each file grouping has a directory of distinct named file entries, each file of which has a format

(text, program rep., data, ISAM record collection, etc.), and such that composite data units (e.g., records, named files, directories, file groupings) can each have attached a set of distinctly named attributes, where each named attribute can have an associated value. Attribute/value attachment must be permissible at a sufficient range of granularity in the file system to permit hook-ups with tools (management reporting and estimation tools, accounting tools, access policy tools, project workbook tools, etc.) but must not have such a fine granularity as to preclude efficient file system implementation.

Examples of attributes and values are: (1) for a file named my Program--Attribute: protection, value: (017 = nobody outside my project can write it or read it, only my teammates can read it, and I can both read and write it), Attribute: Size, Value: 12415 characters, (2) for a file group named Navigation Module: Attribute: computer money spent to date, value: $12,514, Attribute: author, value: Jones, S. jr., Attribute: release date, value: 02 SEPT 81, and (3) for a record named heads-up display progress status, Attribute: number of modules written, value: 16, Attribute: number of modules checked out, value: 2.

2.5.2 Program Development Activities

In order to have a view of the tools required to act on the project data base entities, we first need to discuss the general goals that tool users try to achieve during program development.

We first examine the abstract activities that must take place during implementation (including associated local program refinements). These include fault detection and repair, performance measurement and improvement, and attaining confidence that a program is ready for release.

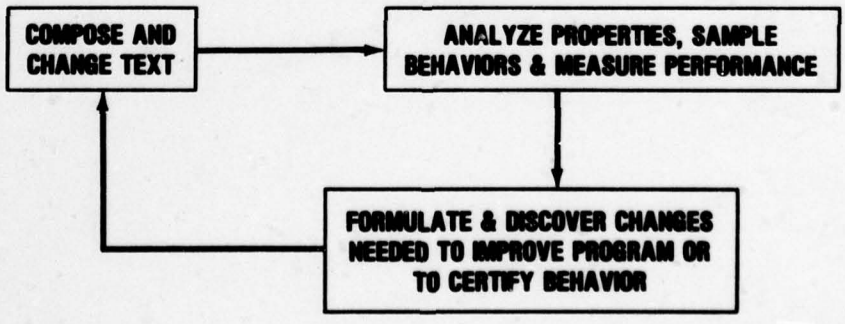
After describing the pattern of program development activities in the abstract, we look at categories of tools required to support concrete realizations of the abstract activities. These

concrete realizations tend to differ depending on the "style" of the environment under consideration--for example, one style is the "remote batch" style, and another is the "interactive style". Implications of these different styles on the tools and procedures appropriate to them are then discussed and an analysis of the differences is undertaken. This analysis is important in deciding what to require for the common language environment.

During our discussion of the life-cycle view, we chose to define implementation as the phase during which we produce a concrete executable representation of a program that realizes the design and satisfies the required performance properties. This implementation is created as a common language text.

It is rarely the case that an initially-created program text has all the properties we want it to have. Because imperfections in the text we first create are likely, we go through a set of activities to detect these imperfections and to reshape the text into a final product that meets the requirements and satisfies our need for evidence that the end product has the properties we wish and is ready for release.

The pattern of program development activities therefore tends to loop iteratively, alternating between episodes of human decision-making and computer execution in an interleaved fashion. The basic loop is somewhat like that pictured as follows:



6-1-78-3

We exit from this loop when we are satisfied that the program under development has the required properties and when we have increased our confidence in its behavior sufficiently that we are willing to release it.

A refinement of this basic loop that gives opportunities for exiting and that splits up the analysis and modification activities a bit further to aid the subsequent discussion is given as Fig. 1.

One sort of analysis, reflected in Fig. 1, is "static analysis". Static analysis may reveal places where the program is incomplete, places where programming language constructions have been used improperly, and even errors of a deeper nature.

Another sort of activity for detecting imperfections is that of trying to execute the program on some sample of data and of collecting and examining samples of its behavior. Such behavior samples may come in the form of printed intermediate results obtained during execution.

Yet another sort of activity for detecting imperfections is "performance measurement", wherein we measure resources consumed by the program during a given execution episode. From an analysis of such performance measurements, we may decide that the program imperfectly fulfills the required performance properties, and that improvement of its performance is required.

After we have detected faults or poor performance, we modify the program, perhaps by again performing some text management activity manually, or perhaps by applying some automated performance improvement tool, such as an optimizer.

We repeatedly modify and analyze the program until we have raised our confidence in its properties sufficiently that we are ready to consider it available for release.

In order to support the above abstract activities, various tools are used. These tools tend to take on different characteristics depending on whether they are used in "batch" or "interactive"

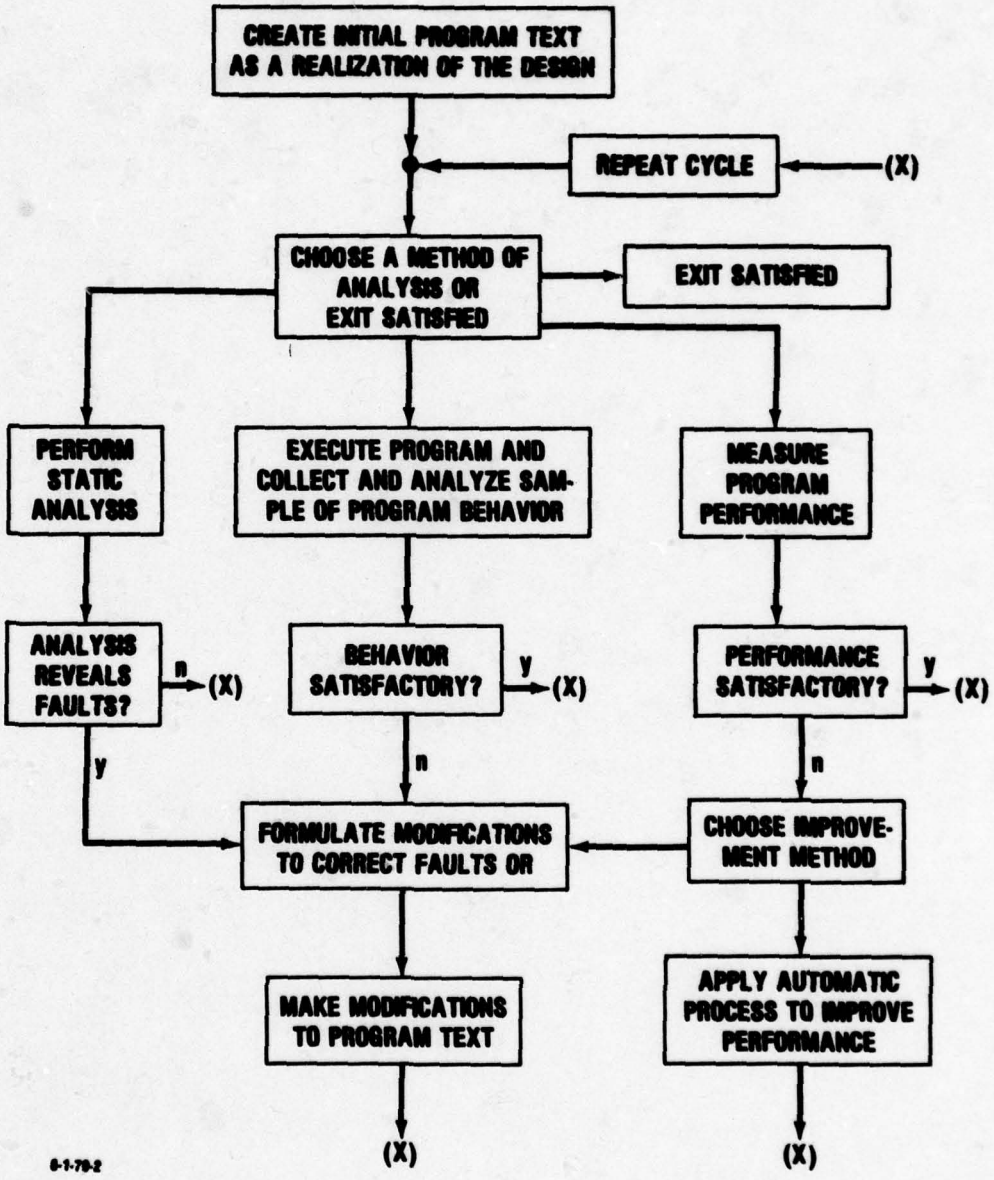


FIGURE 1. Abstract Program Development Activities

modes. What follows is an analysis and categorization of tools used in program development.

We define "tools" as the agents that carry out activities that create and modify project data base entities in a progression that evolves the entities into software systems.

2.5.3 Program Development Tools

Here, we enumerate, without analysis, categories of tools and their properties, and later we engage in a general analysis and discussion.

2.5.3.1 Text Management Tools

Text management tools permit entry, alteration, storage, retrieval, display, formatting, and manipulation of text. Generally, text editors are used for this purpose (though it is possible to manage text with card punches and boxes of cards, as we all know). There are many kinds of text editors. Remote batch editors are often line-oriented. One is permitted to replace, insert, and delete line groups and to move line groups to new locations. Usually, such tools provide listing features, such as obtaining a listing with lines newly renumbered in consecutive increments of some specified size. Some editors are passive and sit waiting for you to spill characters at them. Others are active, and prompt for input, correct spelling, check for syntax errors in program definitions, and so forth. Program formatting and "pretty printing" aids fall into the text management tool category, as do documentation aids such as text formatting tools (e.g., runoff). Some compilers perform text management and listing control functions by, for example, producing line-numbered listings of source programs: (1) before compilation and conditional expansion, (2) after compilation and conditional expansion, or (3) after library expansion. Line-numbered listings produced during this process can be used to cite the contexts of occurrence of errors and to index performance measures.

2.5.3.2 Program Query and Static Analysis Tools

This category of tools allows users to answer questions about properties of static program text, such as: (1) whom does procedure P call?, (2) who calls procedure P?, and (3) what variables does procedure P affect? Also included in this category are program fault detectors and static error analyzers. These latter sorts of tools may give advisories such as "Here it looks as if you have a possible subscript range error" or "Here you have used a programming construct poorly". Cross-reference listings are an example of the result of a static program analysis.

2.5.3.3 Program Instrumenting and Behavior Revealing Tools

These tools permit such activities as: (1) cost measurement --counting events such as executions of modules or measuring cumulative resources consumed, (2) tracing, (3) breaking execution and returning control to the user to permit inquiries and formatted printing of values at the point of suspension, (4) checking of assertions inserted in the program, (5) printing values at places print probes are inserted, and (6) suspending execution and asking for user input. Execution time profiles (Ingalls 1972) (which are essentially histograms showing resource consumption indexed by names of program units responsible for the consumption) might be built up from use of such measurement and probing tools, and these profiles might be used in later stages of program analysis and improvement.

2.5.3.4 Program Transformation and Optimization Tools

Program improvement tools include optimizers (which, for example, perform global data flow analysis to eliminate dead code, propagate constants, remove invariants from loops, remove useless assignments, etc.), and source-to-source program improvement tools (which may improve the efficiency or legibility of program forms, depending on their nature and their goals). Program transformers include compilers (which map source-like program

forms into machine language forms using code generators for one or more object machines), linkers, loaders, and module integration tools.

2.5.3.5 Program Execution and Interpretation Tools

This category includes interpreters that execute interpretable program forms, object environment simulators, execution tools that can run a mix of compiled and interpreted code, systems that run compiled code, and so forth. For batch environments, these tools may include JCL-type job specification languages that permit one to request resources, specify limits, and sequence several phases of a whole job. In the case of interpretation tools, hooks have to be available to cooperate with the interactive tools that converse with users. Job control tools for batch use may additionally be able to specify user interface device characteristics and may be able to take contingent action depending on the outcome of tests executed in the job control program. Also included are tools for multiple job control (such as attaching, detaching, background execution, etc.), and object program execution (running, starting, continuing, halting, reentering, running with DDT, etc.). In some sense, the initiation of an active user session via login, and its termination via logoff are also job control and execution tools. These might also be called user session management tools. In systems permitting both batch and interactive jobs, we may want to be able to allow (or control) a job whose status migrates back and forth between batch and interactive modes (as in, for example, asking about whether your batch job has been printed yet, and if not, being able to attach to it from a terminal, query it as to its execution status, and release it for continuation in background mode).

2.5.3.6 File Manipulation Tools

These tools permit files in a programmer (or other file group) directory to be copied, renamed, deleted, printed, typed, achieved, etc., and they permit the directory itself to be printed.

2.5.3.7 Resource Allocation Tools

These tools permit resources such as devices to be allocated to tasks, e.g., mounting tapes, dismounting tapes, setting device characteristics (terminals, printers, displays), and requesting time and space resources for job execution.

2.5.3.8 Job and System Information Query Tools

These permit users to find out time of day, date, system status, job numbers, terminal port numbers, etc.

2.5.4 Interactive Non-Interactive Program Development Styles

Whether we are using a "batch" or an "interactive" development environment, the basic cycle illustrated by Fig. 1 is the same. We alternate episodes of human decisionmaking with episodes of computer activity designed to manage text, analyze programs, sample program behavior, and improve programs.

For example, suppose we have a batch environment E. We may create program text by submitting a card deck or inputing lines of text to a remote-batch terminal. We may sample program behavior by executing the program in the presence of a file of input test data and by collecting traces, printouts, and memory dumps. We may also perform static analysis by, for example, computing a cross-reference listing and noting which subroutines have been called but not yet defined.

We may measure program performance by specifying the collection of run-time statistics in the form of an execution-time profile (Ingalls 1972). We may call for performance improvement by specifying the run of an optimizing compiler on modules indicated to be performance-critical by the execution-time profile.

On the other hand, suppose we have an interactive environment E′. We may begin a session by conversing with an executive program that permits us to alternate our human decisionmaking

with episodes of execution and analysis in a slightly finer grain of detail than in a batch system.

For instance, when talking to the executive, we may define a single function, assign values to its input variables, execute the function interpretively to obtain a behavior sample, and incrementally modify the definition.

In this small scenario, when we attempt to define the function, we converse with a text editor which accepts lines of the definition. This editor may attempt syntax checking on the spot, and it may inform us of syntax errors--allowing on-the-spot corrections during definition.

When we execute interpretively, the attempt to reference unassigned variables or to call undefined subroutines may break the execution sequence, allowing us to assign values to the variables or to define the unimplemented subroutines. Thus, we observe a pattern of very rapid alternation between light analysis and human rectification of error, and we observe a corresponding rapid shift between the interactive tools needed to support our requests for computation--one moment we are text editing, the next we are executing, the next we are evaluating expressions or printing values of variables, and the next we are back to text editing.

By contrast, in a batch system, if we have to wait for turnaround for some non-trivial length of time, we may be penalized disproportionately for a trivial error--a single syntax error may prevent program execution, and a single trivial semantic error (such as referencing an uninitialized variable) may abort a whole run. In general, the interactive environment provides smaller penalties for trivial errors than the batch environment because of more immediate fault detection in programming activity. This leads to less waste of computer and human resources per unit of program development progress.

One may argue that fixing trivial errors constitutes only a small portion of the programming process and that performing trivial error-fixing even orders of magnitude better may not therefore improve the overall programming process very much. (It would be interesting if those who take this position could produce the data to demonstrate its truth.)

However, a much more serious level of activity is debugging at a "deep" level. We know that debugging can take up to fifty percent of the program development time (Boehm 1973). Hence, debugging is known to be decidedly non-trivial as far as the resources it requires.

In debugging, interactive environments are likely to be superior for the following reason. In debugging, we often face a "signal-to-noise" problem. When we choose to extract information from program behavior, we may be successful or unsuccessful in preplanning for the extraction. If we preplan artistically we will choose ahead of time to examine (or print) only information relevant for locating a bug, and we will never plan to examine information irrelevant for locating a bug.

Since we cannot plan our information extraction activities with perfection, we run into the difficult tradeoff between completeness and relevance.

The more complete we are in our information extraction policy, the less likely it is that the average information we extract will be relevant for locating a bug. The less complete we are in our information extraction policy, the higher the likelihood of missing crucial information needed to locate the bug.

In batch systems, we typically extract information from program behavior through selective printout of values at strategically placed checkpoints, through tracing critical sections of program code, and through memory snapshots and dumps at

strategic points. However, since we must preplan an entire batch run and specify it by some sort of job control program together with insertions for information extraction sprinkled throughout the program text, we tend to aim for completeness of the extraction--especially if there are long turnaround times and if we have lots of time to examine the output before the next opportunity for another run. By aiming for completeness, we frequently deluge ourselves with an overload of printed information that is expensive to search by hand since it contains little relevant information per unit extracted (i.e., its "signal-to-noise" ratio is small).

By contrast, in an interactive environment, we can set breakpoints coarsely at first, and we can check coarsely-set interface constraints (by printing or assertion checking) to locate a bracket where an error occurs. Once the error is bracketed, we can set the detection controls more finely.

By a progression of rapid alternation of program behavior sampling in small units with remaking of information extraction decisions (with the human in the decisionmaking loop), we can home in on the error much in the fashion of a binary search for the root of a polynomial--wherein our next information extraction activity is contingent on the outcome of a previous modest sample of program behavior, and is mediated by the programmer making decisions at a fine level of granularity. This is in stark contrast to the course grain of the episodes of program information extraction typical of batch system usage.

In the case of the interactive system, the control over the fineness of the grain of the interaction cycle by the programmer permits higher signal-to-noise debugging activity, since the human can supply much more rich decision interactions about what to look for next than is profitable in the batch environment.

Thus, while both the batch and interactive environments provide support for the basic alternation between human decisionmaking and information extraction, the interactive environment

provides richer opportunities for control over the granularity of the alternation between making decisions and extracting behavioral information, leading to an increased range of choice and opportunity in program development activities.

As usual, of course, riches can be abused. One form of abuse of the riches of an interactive environment is to waste computer resources flagrantly by attempting to substitute poorly planned computer execution for careful reflection and analysis.

There is another phenomenon which makes batch systems potentially disadvantageous in comparison to interactive systems. If a batch system has slow turnaround times, the behavior samples and analysis produced by prescribed runs may not satisfy the programmer's appetite for information to enable him to take the next step in program modification and improvement. If this happens, programmers often start developing several parallel strands of implementation by feeding the machine enough batch jobs on the independent strands to satisfy their appetite for useful information. This situation tends to produce the ironic consequence that it is the programmer who time-shares his attention between jobs, rather than the computer. This can be expensive for the programmer if the mental shift between parallel jobs involves costly mental swapping of complex program details. In coping with complexity during "programming in the large", perhaps it is better for the machine to time-share and swap core loads, than for the programmer to time-share and to swap large mental contexts.

On the other hand, some programmers may prefer to work on several parallel development strands for reasons of temperament and personal style. They may, in fact, be more effective working in batch mode than in interactive mode. Thus, we argue that the CLE should support both batch and interactive tools.

Interactive tools to support program development activities tend to be qualitatively different than batch tools because of

the increased fineness of granularity of activity. For instance, we need to be able to compose small episodes of tool activities one after the other to obtain the flexibility and granularity we need. This leads to our considering a set of "laws of conversation", which implies that interactive tools must meet certain requirements above and beyond those required for the batch environment.

The analysis of these laws of conversation and of required properties the environment must have to support them is given in the next subsection.

2.5.5 Laws of Conversation

When we have an interactive environment for program development, we have to pay very careful attention to the manner in which the tools listed in the previous subsection can call on and interact both with each other and with the user. Good collections of tools are well-integrated and cooperate well together.

In the following paragraphs, we mention some abstract properties tools must have in order to be well-coordinated.

2.5.5.1

Given that a program P is executing, either the user or another parallel monitoring process must be able to interrupt and suspend P. For example, this includes the case of a user interrupting and suspending a program that has gotten into an endless loop.

2.5.5.2 At the Point of Suspension of an Active Program:

(a) The system must be able to respond to the question: "Where are You?", and the answer must be given in a form that clearly and easily relates to the visible structure of the program being executed.

(b) The user must be able to interrogate and reset variables.

(c) The user must be able to evaluate expressions in the context of suspension.

(d) The user must be able to perform program queries and definitions.

(e) The user must be able to print formatted values.

(f) The user must be able to save, store as a file, retrieve, and resume the state of a suspended session at the point of suspension.

(g) The user must be able to resume program execution at the point of suspension, or, at his option, at any other valid point of control resumption that execution of a programming language command could result in. The latter should include sufficient granularity and choice to resume execution at enclosing program levels, and, most importantly, to reset control to the topmost level (clearing the execution stack, for instance, if there is one).

2.5.5.3 Well-Nestedness of Tools

Tools used at a point of suspension inside each other should be able to perform their activities, leave an intended effect on the environment of suspension, and allow resumption of activity in the context of the tool call. Tools called within tools in a nested fashion should be able similarly to leave an intended effect and clean up, permitting resumption of the calling tool. This is what is meant by well-nestedness. For instance, at a point of suspension, one should be able to call any tool that can be called at the top level. This means being able to install new system levels at the point of suspension, or to invoke the conversational executive at the point of suspension. For example, say that program P is executing and it encounters an execution error and suspends the program execution after printing an appropriate error message. In a well-nestable set of tools,

one might be able to: (1) print a file directory, (2) print part of a program definition file, (3) discover a missing definition, (4) call a text editor to excise the relevant definition from the text, (5) install the text of the definition in the suspended program environment, (6) reset some values of variables, and (7) resume execution of the suspended program with the error now fixed.

The technology for coordinating tools in a well-nested fashion is here today. This technology seems to rest on at least two important principles of tool organization.

2.5.6 Principle A

If the environment of suspension or execution is a data structure in the language, then one can write tools in the language. (Here, by the environment of suspension, it is meant that when the interpretive form of a program is suspended, the set of symbol tables, execution stacks, state indicators, file pointers, etc., are data structures in the common language and so can be manipulated, queried, stored, retrieved, and executed by an interpreter, which itself is a common language program.)

2.5.7 Principle B

Any conversational command that can be given to the interpreter for immediate execution can also be given in program text and can be executed as part of a program. Thus, any system function can be performed by a program as well as a user.

> Question: Is this list of principles complete? Are there other requirements for well-nestedness of tools not given here? What about allowing the system to respond to error messages and resume in place of the user. How is this to be accomplished unless the user is essentially called as a subroutine, and can be replaced by a subroutine call that can read the latest increment of text to an output file and can substitute

its own response for the user's command string. How is this to be arranged? Must one also be able to insert layers of control immediately inside a function call or before a printed response? Who has a handle on what a complete set of requirements for well-nesting might be?

2.6 THE MAINTENANCE AND ENHANCEMENT VIEW

Improving the quality and reducing the cost of "maintenance" may well be the name of the game for dramatically improving software quality.

Maintenance cost studies have revealed that from 50 to 90 percent of the total life-cycle cost can be spent in maintenance. Perhaps this is not surprising if we consider that maintenance lifetimes occupy from 15 to 25 years for some embedded systems. On the other hand, it comes as a shock to discover that the number of instruction changes per year can equal or exceed the number of original instructions in the originally released system.

Maintenance may well be the least understood and the most costly activity in the system life cycle. Hence, any adequate consideration of environment requirements must consider those special requirements derived from viewing the environment from the perspective of maintenance. Thus, while maintenance is part of the life cycle, this section is presented separately from the life-cycle view to give maintenance the special emphasis it deserves.

We know that during the maintenance phase of the life cycle certain special activities receive emphasis that are of a distinct or more active nature than the activities that occur prior to release. These activities are: (1) trouble shooting and fault diagnosis and repair, (2) control of new software releases, (3) configuration control (sometimes for multiple configurations), (4) management of and response to software change requests, (5)

management of and response to software trouble reports, (6) training, and (7) documentation.

During initial pre-release system development, we progressively evolved and refined programs, documents, and test data in the project data base into a refined collection of visible products that constituted the version of the system released. During maintenance and enhancement, we must continue the refining and evolving process, and we may need to refine the same entities further using the same tools for measurement and modification and for keeping the documentation up to date.

In addition, enhancement of the system to meet a new requirement may involve the replay of earlier life-cycle activities. We may have to perform incremental requirements analysis (e.g., to determine whether a new requirement is consistent with the old ones), incremental redesign, incremental reimplementation of old modules or implementation of new ones, and incremental retesting and reintegration.

Thus, this view implies that we must transfer intact to the maintenance environment the visible end products of the pre-release life-cycle stages, together with all the tools needed to support further system evolution.

We do not want to discard all the program development and refinement tools at the instant a system is released for operational use and is transferred to its user organization--leaving the user organization with a binary deck and a hard printed program logic manual.

Thus, our analysis leads to the following environment requirement statement.

2.6.1 Requirement for Retention at Release Time

At the time a newly developed system is released and begins its maintenance lifetime, the maintenance environment should retain the last versions of the visible products and tools used

during development, including up-to-date requirements, design, and implementation documents, and up-to-date test data and procedures.

In addition, we see that there is an advantage to having computerized support for the products to be evolved and refined during maintenance to provide incentive and ease in keeping them up to date. This leads to the following additional requirements statement.

2.6.2 Requirement for Computerized Support

To the extent consistent with methodologies used for requirements analysis and design (some of which may not lead to easily computerized end products), machine manipulable, up-to-date documents specifying current requirements, design, and implementation should be utilized, and appropriate document management and display tools should be present in the environment to ensure the the success of document currency maintenance.

As the system changes in response to fault repairs and enhancement, machine manipulable versions of the documents should be updated to keep pace with the changes.

Unfortunately, the nature of hardware and software errors differs in at least one fundamental characteristic--hardware deteriorates because of lack of maintenance whereas software deteriorates because of the presence of maintenance.

There is a problem of very deep and insidious consequences for the activity of maintenance. Some errors in software consist not of trivial diagnosable and treatable faults at the programming level, but rather of subtle "logic" errors in the design or implementation.

The problem is that a "logic" error may occur at any of a number of levels of program representation spanning the gap from the problem domain to the concrete implementation domain.

To make this situation vivid, consider a navigation module on a supersonic aircraft. Let us suppose that the navigation module is supposed to provide the correct position of the aircraft to within 10 meters anywhere in the atmosphere of the earth. The module obtains input from gyros, accelerometers, clocks, doppler radars, and navigation signal receptors that can listen to satellite and ground station signals. Suppose it has been determined (perhaps by exercise of self-diagnosing interface monitoring procedures and execution of fault-detection decision trees) that none of the input devices is malfunctioning. But suppose that the results produced by the module are consistently in error.

Let us further suppose that the actual error is a superimposition of errors from three separate sources: (1) a simple programming error involving unintentional clobbering of the contents of a global variable by a local procedure that incorrectly assumes that the global variable is local, (2) the decay in numerical accuracy of a certain class of computations through inadequate numerical analysis of error propagation, and (3) failure to design the module to take account of coriolis force, leading to systematic errors on north-south trajectories at high mach numbers.

Each of these three error sources might fall within the province of distinct skills at the command of distinct trained specialists. Only a physicist familiar with the laws of kinematics and dynamics might be expected to realize and correct the coriolis force error. Only a numerical analyst familiar with the laws of numerical error propagation might be expected to discover and correct the error of numerical accuracy decay. And only a programmer trained in the use of nomenclature scope rules in the programming language used to implement the module might be expected to discover and correct the error of unintentional information clobbering.

If the actual error is a superimposition of these three sorts of errors at these three sorts of levels of program logic, it is doubtful that a maintainer, trained only in one of the three relevant skills, could succeed in untangling the superimposed errors, in isolating their sources, and in making appropriate corrections.

In a similar vein, if the system is being enhanced to meet new requirements, the skills of requirement analysts and designers may be required to modify the requirements and the design incrementally and to bring the requirements and design documents up to date consistent with the enhancement. In fact, because of the presence of more constraints, incremental reanalysis and redesign might be more difficult than original analysis and design. It may not be enough for the maintainer skilled only in the implementation, test, and integration phases of the software life cycle to perform acts of enhancement that call for the reply of skills exercised by teams of skilled specialists at earlier life cycle phases--teams now disbanded and unavailable. This is particularly likely to be true if the requirements and design levels of the system being enhanced demand skilled thinking in application domains widely separate from programming.

But we know that maintenance and enhancement may tend to occur under circumstances under which the original teams that performed the high-level logic analysis and design (and that used special application domain skills remote from programming skills) have long since disbanded, leaving maintenance and enhancement tasks to those unskilled in the higher logic levels of the system. Such maintenance circumstances are unpropitious unless techniques can be found to determine when to call in or recongregate teams of skilled specialists needed for fault detection, repair, or enhancement.

The raising of the maintenance problems and issues may suggest fruitful avenues for research. However, unless progress

is made soon and tested technologies are made available for solving some of these problems, it is unlikely that the common language and its environment could be in a posture to help substantially with improvement of maintenance quality and reduction of maintenance expense in connection with problems of the sort just discussed.

Some of these problems of maintenance are thus deep, vexing, and expensive. The use of any current proven technology or the meeting of any envisaged environment requirement may not, in fact, make substantial inroads on some of these difficult problems. It is thus important that the environment requirements document not make unjustified promises or raise false hopes in connection with these difficult aspects of maintenance.

2.7 THE STANDARDIZATION AND CONTROL VIEW

As we pointed out earlier in our discussion of the view from the vantage point of software quality, "commonality" in embedded system programming languages and environments is a goal that, if achieved, can have substantial benefits both economically and in supporting pursuit of other software quality goals.

In order to achieve commonality, the proliferation of variants must be suppressed. In turn, in order to control and oppose the proliferation of variants, standards must be defined precisely, they must be maintained, and implementations of language processors and environments must be controlled to ascertain that they meet the standard.

The use of standards and controls are thus techniques that support commonality, which, in turn, supports yet more important software quality goals.

When we view the environment from the vantage point of standards and certification, we take ourselves out of the arena of computerized environment support agents and data bases and

we enter the realm of policies, procedures, and agencies that are assigned judicial and certification duties.

Standards maintenance can be performed by agencies responsible for the custody and modification of standards documents, and for adjudicating disputes and furnishing judicial rulings.

Control can be mediated by certification authorities that use testing procedures to assure that proposed implementations meet current standards (insofar as testing can do this feasibly and completely). Enforcement of standards can be accomplished for government use by following government contract policies requiring contractors to use standard, certified language processors and environments on embedded system software projects for which the government is paying. Support can be lent to this effort in the form of an economic incentive by providing GFE language processors, environments, and training aids that have been certified to meet the standards, and by making them available to contractors.

2.8 THE LANGUAGE SUPPORT AND CULTURE VIEW

A flourishing language needs a flourishing language culture. Such a culture might embrace user organizations centered about particular applications, compilers, or computers. It might embrace an extensive use of the language in military training, industrial training, and educational institutions. It might embrace standard libraries of useful programs maintained and distributed by a support agency. It might embrace serious study of the language and its environment by research organizations. It might embrace the development of software support products by computer manufacturers to enhance the marketability of their hardware. It might embrace the publication of books and training systems by commercial publishers convinced of the market potential of publishing common language books for profit.

A flourishing culture has many mutually reinforcing aspects. If we look at the cultures surrounding popular programming languages such as Fortran, COBOL, APL, PL/I, and Basic, we can discern many aspects of flourishing language cultures.

It is clear that there are enormous benefits to the government that result from having a flourishing language culture, since free enterprise in the private sector automatically provides rich support and improved skills and training in the use of any language with a flourishing culture. Reliance on such benefits improves software quality without increased cost to the government.

There is a question of where to draw the line on common language support activities that constitute legitimate pursuit by the government of its own legitimate interests. In fact, the government now has an opportunity to influence and improve the state of the available engineering technology because of its unusual dominance at this time in the marketplace. This opportunity will likely pass. It may not be around in another ten years.

3. INTERACTIONS BETWEEN THE VIEWS

Some of the eight views we have given interact strongly. It is important to study these view interactions and to reason about the impact of the interactions on the environment requirements.

We have seen that the view of the urgency of the task interacts with the view of software quality by making desirable a requirement for use of low-risk, proven technology in the initial environment. Software quality and timeliness of delivery of the environment are best served by limiting objectives and using low-risk technology initially.

The view of the urgency of the task also interacts with the management discipline view. There we suggest abandoning the objective of putting in complete, initial batteries of computerized management aids in favor of leaving the design of the initial environment open-ended so it could later grow and absorb management support tools.

The life-cycle view interacts with the management discipline view by emphasizing that each life-cycle stage may be carried out in the context of imperfect and incomplete information and that the visible products of each life-cycle stage may be subject to incremental improvement and refinement in the context of later life-cycle experience. This led to the consideration of management disciplines for quality assurance focusing on use of independent validation check point and testing activities and of careful monitoring and exception reporting for management.

Management policies and disciplines attempt to configure resources and to use tools over some portion of the life cycle

in a way that attempts to meet the software quality goals. Thus, the management discipline view and the software quality view interact strongly, since the objective of management disciplines is to meet simultaneous software quality constraints.

One highly visible locus for interaction between the views is the project data base. The project data base is a communication medium and a repository of visible products of each of the project stages. It contains the entities that are incrementally improved and refined into the end product of the project effort. The project data base can be measured and manipulated by management tools to provide management reports. Version control and file access policies are implemented using tools that interact with the project data base. Hence, the project data base is the focus for interaction between the management discipline view and the program development tool view. Also, a number of visible end products and tools extracted from the project data base may be handed off to the maintenance phase of the life cycle.

The project data base is thus a connecting tissue between pre-release activities and maintenance, between programmers and managers, between sub-projects at different life-cycle stages, and between successive incremental steps by individual implementers and testers (since it is a repository for successively improved versions of each person's work).

It becomes very challenging to design (or select) a file system design that will support all of these interactions gracefully. File-naming conventions, directory and indexing conventions, access privilege controls, resource accounting methods, and file format representations must be chosen with the totality of environment support missions in mind.

4. OUTLINE OF ISSUES TO CONSIDER

In this section we attempt to enumerate as many relevant issue areas as possible. The primary purpose is to ensure completeness in the requirements analysis process.

Completeness can be achieved only if we are able to catalogue all the issues that must be considered. The current list is a very limited start. The reader is invited to suggest additions.

The list can be used to ask which issues are critical and which are not, and to ask which should be considered in the initial environment and which can be put off until later. It can be used as a guide in organizing the subsequent analyses. It can also be used to mark those issues that need not be considered, thereby avoiding the possibility of repeatedly deciding not to consider the same point.

One danger we should attempt to avoid, in using such a catalogue of issues, is the tendency to assume that we must provide something in the common language environment to address each topic on the list. Instead, we advocate using it as a guide for ruthless simplification, and would hope that for the majority of topics the decision is to take no action whatsoever.

4.1 COMMON LANGUAGE FRAMEWORK AND ENVIRONMENT--ISSUE AREAS

I. Administrative, Policy and Management Issues

1. Language standards and control
   - Configuration Control Board -- Charter
   - Configuration Control Board -- Organization
   - Language standards defining document

- Policy on DoD use of the language
- Configuration management of the language

2. Translator validation and certification

3. Development, maintenance and distribution of the software tool environment and standard application libraries

4. Supporting a user community

5. Software life-cycle management

6. Software procurement
   - Policy
   - Warranties
   - Royalties

7. Management plan for the initial development period

II. Software Development and Maintenance Tools Environment

1. Structure of the tools environment
   - The environment as a program data base
   - Concepts needed in the environment
   - Relation to language components
   - Node structure and attributes

2. Portability of the tools environment
   - Goals
   - Use of common language as source language
   - Use of standard library for machine dependent parts
   - Minimum required to port tools environment

3. User interface characteristics
   - Need for commonality among subsystems
   - Use of common language as JCL/exec. language
   - Batch interface characteristics
   - Terminal interface characteristics

4. Commonality of tool characteristics
   - User interface conventions

- Tools interface conventions
- Display format conventions
- Standard character set
- Separable I/O in tools
- Tools sourced in foreign languages
- Granule size for tool data

5. Host system management functions
   - User identification and context specification
   - Context protection and access rights
   - Accounting
   - Message passing
   - Version control
   - System query functions
   - File manipulation tools

6. Standard library entries
   - User level I/O
   - File system
   - Data formatting/deformatting
   - Mathematical routines
   - Dynamic resource allocating routines

7. Software analysis, query, and display tools
   - Static analysis tools
   - Dynamic analysis tools
   - Instrumenting
   - Batch display tools
   - Interactive query and display tools

8. Translation and optimization tools
   - Translators
   - Code generators
   - Optimizers
   - Linkers
   - Loaders

- Module integration tools
- Foreign to common language source-source translation

9. Software test and evaluation tools

10. Program documentation tools
    - Text editor
    - Text formatter
    - Program editor
    - Program formatter

11. Program execution and interpretation tools
    - Interpreter
    - Debugger
    - Object environment simulators
    - Object environment emulators

III. Computer-Based Tools for Administration, Policy and Management

1. Technology for language standards

2. Translator validation tools and technology

3. Translator performance evaluation tools

4. Configuration control tools

5. Software life-cycle management tools
    - Requirements specification tools

IV. Initial Critical Tools and System Characteristics

1. Measures of criticality
    - To provide the minimum needed utility of software development and maintenance
    - To provide consistent conventions
    - To ensure portability of the tool environment
    - To provide for initial bootstrapping of language/system
    - To ensure open-endedness of tool environment
    - To ensure complete implementation of the initial standard library

2. Tool environment characteristics

3. Specific initial tools

5. EXPERIMENTAL GENERAL PRINCIPLES AND OBSERVATIONS

In this section, we throw out for consideration a number of trial ideas aimed at making the design of the environment simple, general, and efficient. If the ideas are workable, they might evolve into environment requirements, since they support general environment goals of reliability, modifiability, portability, efficiency and minimal cost.

5.1 A PIECEWISE BASIS FOR BUILDING ENVIRONMENTS

We might want to investigate the possibility of specifying that the common language library should contain a tool kit of small pieces of operating system functions (akin to the UUOs on Tops-10 or the JSYS commands on TENEX). It might be possible to specify a set of such pieces users could use to write tools in the form of common language programs, much as Tops-10 or TENEX users can now write tools in MACRO-10, using UUOs or JSYS commands.

5.2 ATTRIBUTE--VALUE ATTACHMENT TO GRANULES IN THE PROJECT DATA BASE

In principle, for any granule (node) in the project data base (or file system), such as a granule corresponding to a task, programmer, project, module, etc., we should be able to attach (create) new variables (attributes) which can have values. We can set and display values of variables. We can write programs to access and set values of variables and to enumerate and process nodes in any composite structure consisting of nodes. This allows us to do things such as implementing version control and traceability by adding "version number" and "author"

attributes to nodes. We can write resource estimation, management reporting, and critical path scheduling programs using this framework. Each "entity" (e.g., device type parameter block, file group, job, etc.) in the system has values of attributes that can sustain variation. (Authorized) users and programs may set and interrogate values of attributes of entities. If this general principle is followed, very general classes of tools can be written as common language programs, and these can be stored in the standard library, enhancing portability. Note that by combining the advantages of Principles A and B, portability across machines is enhanced.

5.3 PERSONAL OBSERVATION

When we finished looking at the environment from as many angles as are given in this paper, we came to the conclusion that the scope of the environment for the common language is truly enormous. It was then that we began to get a queasy feeling in the pits of our stomachs (one which we hope you too now have if you have read this far).

How can we cope with all these environment issues in the short time we have? Can we avoid being overambitious and embarking on on effort of the scope, say, of OS/360 (which involved from 3 to 5 man-millennia, and which might sink the entire common language effort)?

The only way we know to cope with the enormous scope is to follow the advice of Tony Hoare (Hoare 1978):

"--- they must simplify!
Such a painful operation
--- almost as painful as thinking!"

It may well be that the success of the common language environment effort will rest on our ability to pursue simplification with something resembling religious fervor.

5.4 PRINCIPLE

Utilities and OS/commands are semi-interchangeable--any utility can become an OS/command and nearly any OS/command can become a utility. Therefore, nearly everything can become a utility and can become a library routine in the common language. Note, however, that some utilities are "more equal" than others. For example, if machine has multiple concurrent users, there must be a utility called a scheduler (to act as a traffic cop) that says who gets to do what when, and it calls another utility called a command interpreter (or executive) to talk to users and accept their requests for subsequent processing. An accounting utility could be written as a common language program called by the traffic cop (scheduler) to update entities with incremental usage charges. A reporting utility could be written to look at the entities updated by the accounting utility to query and report usage for entities.

5.5 PRINCIPLE

A capability of the system should not be tied to levels or distinct classes of entities (unless intolerable inefficiency results). For example, granting simultaneous read/write access to two different users should not be limited to files but should be possible on all levels of granularity in the project data base that have access/update control attributes associated.

5.6 EXPERIMENTAL REQUIREMENT

It shall be possible to express any job control program executable in the CLE as a program in the common language.

(Thus, job control language shall be subsumed by the common language.)

5.7 PRINCIPLE

There is no reason why a task cannot change states from interactive to (background + executing + detached-from-terminal),

or from (remote job entry batch) to interactive, etc. The difference between batch and interactive modes is that in interactive mode, an executing program can ask a user to supply a decision and can take contingent action--i.e., the user is a parallel process callable as a subroutine by a job.

[Job control languages allow one to make a query and to take contingent action on the outcome of the query to, e.g., schedule subsequent job phases (run, print, file).]

5.8 EXPERIMENTAL REQUIREMENT

The CLE shall provide a set of tools for analysis and stepwise refinement of program forms consisting of valid common language programs in which grammatically well-formed parts of the program have been deleted to form holes, and in which the holes have been filled by specially bracketed English descriptions. The English descriptions describe what the replaced pieces do, or, when used in program design and stepwise refinement, what they will do later when they are written in common language.

[Note: This provides a low-cost implementation of the notion that there should be a capability for supporting program forms at different levels of abstraction and at different stages of partial completion. It is close to the PDL (Program Design Language) of Caine and Gordon (Caine & Gordon 1975), a program design tool of proven effectiveness on large software projects. Further, once a programmer is trained in reading common language programs, he is also trained in reading the PDL abstract program forms at higher levels of refinement. This low cost choice of a PDL has several benefits.]

[Some easy tools to implement to support such a PDL are: syntax checkers, formatted program printers, line-numbered listing printers, cross-reference listers, query answerers, and completion analysis checkers (which print names of modules called, but not written, for instance).]

6. SUGGESTIONS FOR TASKS THAT NEED DOING

The following items are not findings, rather they are things you the reader might be interested in accomplishing as an individual. They are things someone should go off and do and for which results are needed as an input for all of us.

- Somebody should look at all the functions performed by the set of JSYS or UUO commands on the PDP-10, (or by any other such set of operating system service call routines for any other operating system or machine), to analyze what a complete set of them might look like, and whether or not the idea of having a basis set in the common language standard library with which to make up tools is both possible and a good idea. This idea might have considerable power.

- Somebody should take the list of tool properties in Section 2.5.3 and check them against the set of functions one can perform in each of several operating systems (with utilities included) such as UNIX, RSTS, TOPS-10, OS/370, OS/MVT, TENEX, the NSW, etc. Are the functions described complete? consistent? mininal?

- Somebody should look at a collection of good interactive environments (e.g., APL, InterLisp, NSW, etc.) to see if the laws of conversation, and the requirements for obtaining the property of well-nestedness of tools are complete, consistent, and minimal.

- Somebody should take a look at a good modern operating system of modest size (UNIX?) to see how it is structured on the basis of its collection of primitives, its file system, its

utility calling patterns, its executives, its schedulers, etc., to see if all of its design features could be the basis for a good program development system environment of reasonable generality, good efficiency, and modest size.

- Somebody should generate a (careful) scenario of tool use required to develop a program module, and should analyze how the tools are called and in what order. This scenario might ultimately draw on all the necessary tools for program development and maintenance. Things would be given by example, and things would be left out implicitly. This might be a better style of specification document than, say, the style of STEELMAN, since it would be pleasant to read and would leave things out implicitly. Its very concreteness might lead to good environment designs. Also, it would be a valuable document to have to perform a checkout of the completeness, consistency, well-coordination, and minimality of PEBBLEMAN, and any other requirements documents that may evolve in the CLE requirements analysis process.

  Question: What if the end product of PEBBLEMAN was based on a scenario of program development and maintenance?

REFERENCES

B. DeRoze, and T. Nyman, "The Software Life Cycle--A Management and Technological Challenge in the Department of Defense," IEEE TSE 4,4 (July 1978) 309-318.

S. Caine, and E.K. Gordon, "PDL--A Tool for Software Design," Proc. NCC, AFIPS Press, Montvale, N.J., (1975), 271-278.

W.A. Whitaker, Introductory Remarks Workshop on Environment, Certification, and Control of DoD Common High Order Language, UC Irvine, Irvine, California, (June 1978).

D.A. Fisher, "DoD's Common Programming Language Effort," Computer, (March 1978).

C.A.R. Hoare, personal correspondence (November 1978).

STEELMAN, Department of Defense Requirements for High Order Computer Programming Languages, (June 1978).

Barry W. Boehm, "Software and its Impact: A Quantitative Assessment," *Datamation* (1973).

D. Ingalls, "The Execution Time Profile as a Programming Tool," in *Design and Optimization of Compilers*, (R. Rustin, ed.), Prentice-Hall, Englewood Cliffs, N.J., (1972), 107-128.

INDEX