D3

N86-23317

A VIEW OF SOFTWARE DEVELOPMENT ENVIRONMENT ISSUES

Barry Boehm
TRW

C-2

OUTLINE

- Nature of the challenge
- Orange-Book issues (ref. 1)
  - Pros, cons, assessment
- Additional SDE issues
  - DOD coordination
  - Scope of SDE
  - Reuse support
- Summary

NATURE OF THE CHALLENGE

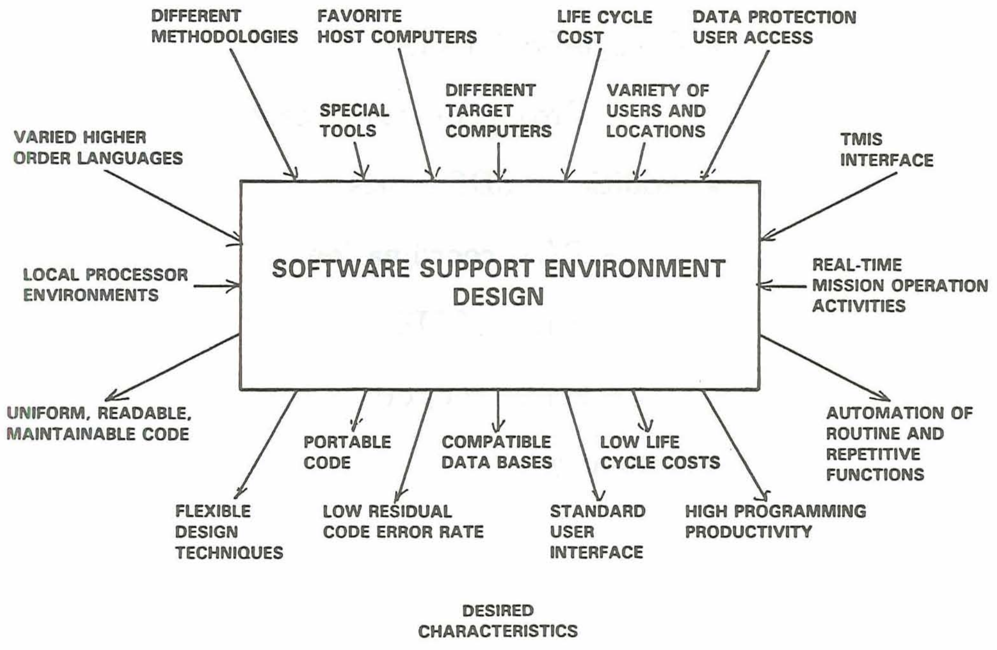
The SSDS is a mission support system for:

- Thousands of operators and decision makers
- At on-line terminals
- At many geographical locations
- Performing complex, interacting functions
- With imprecisely defined requirements
- In a dynamic, less-than-predictable environment
- Requiring essentially error-free performance

It is essential for coordinating the mission

It requires significant investments in time, dollars, talent

SDE Design Considerations


DIFFERENT METHODOLOGIES
FAVORITE HOST COMPUTERS
LIFE CYCLE COST
DATA PROTECTION USER ACCESS
SPECIAL TOOLS
DIFFERENT TARGET COMPUTERS
VARIETY OF USERS AND LOCATIONS
VARIED HIGHER ORDER LANGUAGES
TMIS INTERFACE
LOCAL PROCESSOR ENVIRONMENTS
SOFTWARE SUPPORT ENVIRONMENT DESIGN
REAL-TIME MISSION OPERATION ACTIVITIES
UNIFORM, READABLE, MAINTAINABLE CODE
PORTABLE CODE
COMPATIBLE DATA BASES
LOW LIFE CYCLE COSTS
AUTOMATION OF ROUTINE AND REPETITIVE FUNCTIONS
FLEXIBLE DESIGN TECHNIQUES
LOW RESIDUAL CODE ERROR RATE
STANDARD USER INTERFACE
HIGH PROGRAMMING PRODUCTIVITY
DESIRED CHARACTERISTICS

ISSUES 1,2,3: UNIFORM, NASA-FURNISHED, MANDATED SDE

PRO:

- Better software coordination
  - Fewer errors, interface problems
- Less duplication of effort
- Conceptual integrity
  - Reinforcement of management approach
  - SDE/user interface
- Controllability
  - Response to problems
  - Technology insertion
- Better life-cycle support
  - Ability to recompete maintenance

ISSUES 1,2,3: UNIFORM, NASA-FURNISHED, MANDATED SDE

CON:

- Contractor incompatibilities
  - Competitive bias
- Technology insertion
  - Disincentives to experiment
- Implied SDE warranty
- SDE size, development risk
- Breadth of user community
  - Centers, contractors, researchers
  - Levels of expertise
  - Special functions: simulation, test, etc.
  - Large up-front training cost

ISSUES 1,2,3: UNIFORM, NASA-FURNISHED, MANDATED SDE

PRO	CON
•Better s/w coordination	•Contractor incompatibilities
•Less duplication	•Technology insertion
•Conceptual integrity	•Implied SDE warranty
•Controllability	•SDE size, development risk
•Life-cycle support	•Breadth of user community

ASSESSMENT

- Go for it – in ways which minimize cons
  - Pre-delivery contractor option to use own SDE
  - SDE modularized for technology insertion
  - Establish levels of warranty
  - Incremental development to reduce risk

ISSUE 4: PRECEDE SDE DEVELOPMENT WITH DEVELOPMENT OF FUNCTIONAL CAPABILITIES, PROTOTYPE, DETAILED SPECS

PRO	CON
•Familiar acquisition approach	•Very high schedule risk
•Provides criteria for choosing developer	•Not clear more prototypes will add much information

ASSESSMENT

- Better to go for early initial capability
- Use DOD JSSEE Spec as basis for defining requirements
- Run competitive flyoff for production - engineered initial SDE capability

ISSUE 5 BUILD LAYERED SDE

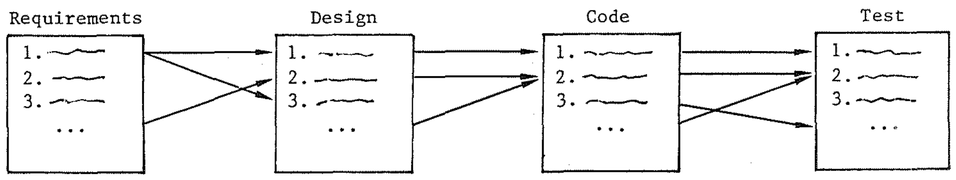
PRO	CON
•Accommodate change, growth, technology insertion	•Performance penalties
	•May pick wrong layers

ASSESSMENT

- Build layered SDE
  - Use info-hiding to modularize around major sources of change
    - Methodologies (requirements, design, management)
    - Mainframes, workstations
    - Networks, peripherals
    - Language, operating system?

MODULARIZING AROUND SOURCES OF CHANGE IN SOFTWARE METHODOLOGY

- Make minimal assumptions on nature of elements (requirements, design, code, test, management)
  - Resolvable into separately identifiable items
- Develop traceability tool to track relations between items



ISSUE 6: AFFILIATE WITH DOD ENVIRONMENT

PRO	CON
•Technical synergy	•Not clear which one
•Less contractor confusion	•Schedule mismatches
	•Control; coordination

ASSESSMENT

- Propose coordinated, potentially joint SDE
- Volunteer to develop a pre-1990 initial SDE capability based on JSSEE spec

ISSUE 7: FURNISH FULL-UP SDE:

SOFTWARE, CPU, WORKSTATION, LAN

PRO	CON
•Fewer coordination problems	•Expensive to furnish •Technology insertion problems •Contractor SDE incompatibilities

ASSESSMENT

- Build SDE on standard, portable operating systems
- Support recommended hardware subset(s)
- Allow use of equivalent capabilities

ISSUE 8: SUPPORT LIBRARY OF REUSABLE COMPONENTS

PRO	CON
•Major source of future s/w cost savings	•Added investment •Hidden incompatibilities •Component warranties •Version control •Component pollution

ASSESSMENT

- Go for it – in ways which minimize cons
  - Levels of warranty
  - Strong documentation, CM
  - Selective incorporation

SUMMARY

- Building an SDE is in the same ballpark as building SSDS
    - Very complex, but essential
- Worth going for uniform, NASA-furnished, mandated SDE
    - In ways which minimize risks
- Value of further SDE prototyping unclear
    - Several de facto prototypes exist
    - Very high schedule risk
- Worth coordinating with DOD
    - JSSEE spec a useful starting point
- Furnish SDE as standard s/w on portable operating system
    - Support but not mandate CPU, LAN, workstation

KEY ISSUES ADDRESSED

PROGRAMMATICS

Uniform, NASA-Furnished, Mandated SDE

The issue raised addresses the realization of SDE capabilities. Should NASA provide and require the use of a standardized SDE for Space Station software acquisition? Examination of this issue reveals considerations which require focused attention.

Uniformity will yield fewer interface and coordination problems and will provide conceptual integrity. These benefits, however, are at the expense of multi-contractor incompatibilities and their combined strengths for technological development.

A mandated, government-furnished SDE provides direct control by NASA for problem solutions, evolutionary as opposed to revolutionary growth (mature expansion), and more opportunity for SDE-related cost containments. However, any GFE item bears an implied warranty. This needs to be addressed by defining levels of warranty for components of the SDE. Another issue is how a government-furnished SDE would be sized to efficiently service the wide breadth of the anticipated user community. Here, the SDE needs to be organized to be easily subsetable to specialized user communities, host computers (maxis versus work stations), or user expertise levels.

SDE Operations Concept

The scope of SDE application is indeed broad. Each of the major workpackage contractors is likely to have unique, embedded software development methodologies and supporting facilities. In turn, their subsystem development organization and/or subcontractors will have established computer system development tools, experience, and expectations. Further, the ultimate users of the Space Station will include a significant portion of small groups or individuals interested only in their experiment or production package and not in any required supporting software. Effectively scoping the range of SDE requirements requires the near-term definition of how all users--big and small, sophisticated or naive, experienced or novice--may use the system. An Operations Concept, addressing how all users expect to use the system during its entire lifecycle, has been found extremely useful in establishing a basis for subsequent hardware/software requirements specification.

The conclusion reached gave an affirmative answer to the issue: NASA should provide and mandate the use of a uniform SDE. The government furnished SDE should be effected in a manner which mitigates benefits and risks, specifically by establishing a widely accepted SDE Operations Concept.

Incrementally Developed

If the SDE is constructed as a set of functional modules enclosed by a communications structure, the modules can be acquired, inserted, and replaced on an incremental schedule. The general driving requirement for module acquisition and insertion is at the communications interface. Initial priorities should be established by NASA so that incremental implementation will support program requirements as they become needed. Some, indeed, are needed now.

The SDE must be subsetable, modularized, and concentrically layered to assist all mission, management, and communication requirements. This form of structural detail seems most likely to be able to achieve the desired flexibility and versatility over the range of specific SDE instances.

A strategy for incremental development is recommended which minimizes the dependence of the SDE development schedule on requirements to be derived by Space Station Phase B contractors:

Increment 1:	OS, DBMS, utilities, basic CM, office automation, and management functions
Increment 2:	Basic requirements and design specification, planning and analysis support
Increment 3:	Basic code, unit test, integration and test support
Increment 4:	Basic real-time OS, DBMS, and utilities for flight and ground target computers
Increment 5, 6,...:	User-prioritized additions and extensions to the above

This strategy allows NASA to get an early start on the portions of the SDE needed for initial Space Station program development support.

SDE SCOPE

Focus on Products

No clearly superior methodology for software design refinement has emerged, yet many have proven useful for unique or particular application arenas. For all methodologies, certain intermediate products or design representations are recognized. Focusing upon these products, as distinct from the methodology or process employed in establishing these products, permits considerable methodological flexibility and allows for future technology insertion. Where a generally agreed upon management model can be established, the SDE may support the process directly. We conclude that the SDE shall be nonprescriptive of a specific requirement or design methodology.

Supporting Software Reuse

Complete rebuilding of large software systems is no longer economically feasible. Full advantage must be taken of viable existing elements. Suitable reusable components may be commercially available off the shelf (COTS), may reside at one or more NASA centers, or may be adaptable from past contractor efforts. Making use of such elements requires careful initial attention to the framework or architecture of the SDE, including the definition of appropriate interfaces and the levels in the hierarchy. Clearly, multiple source languages and/or object code bodies should be accommodated in many instances. Certainly, the desired SDE subsetability considerations relate to the kind of structure promoting reuse described here.

We conclude that the SDE interface and architectural definitions should foster software reuse.

SDE STRUCTURE

Furnished as Portable Software Package

The SDE should consist of device-independent (loosely coupled hardware dependencies) functions such that changes in hardware do not have an effect on software functionality. Hardware availability should not drive the software requirements, but some well defined, vendor dependent elements may facilitate widespread use of currently available components. In some areas, such as target machine support, requirements may dictate a hardware component of the SDE.

Virtualized Operating System

The operating system which supports the SDE should be device and vendor independent insofar as possible. As a present starting point, UNIX appears to be the only candidate that meets this requirement and should be selected. Prevailing personal computer operating systems meet the spirit but not the large machine scope of this requirement. For the future, the SDE can implement other hardware-independent operating systems (e.g., CAIS or MAPSE for Ada) as they become available.

Single, Subsetable SDE Host

The central issue of the SDE structure is architecture. Associated subissues (incremental development, choice of modular or layered, ease of user accommodation) are facets of the SDE architecture issue perceived functionally as requirements.

Selection of the subsetable functions and interfaces is the most critical. A primary capability is to allow for support of multiple host targets. These subsetable functions must also support, by interface management, fully generalized and specific functions within the layered architecture. A major objective is to maximize commonality of widely used functions. There is a potential, as the SDE evolves over time, to yield unmanageable interface/function diversification. The result is that interfaces could multiply and become deeply nested, thus driving incremental mainframe costs of ownership for certain levels of capability.

The definitions of subsetable SDE elements, interface specification, communications/tasking network definition, and management provide the baseline from which to proceed. Plugability as to function, via the suitable interfaces, will result in achieving, integrating, and managing associated issues of portability, user interfaces, and mission requirements.

Instrumented for Self-Diagnosis

A rational basis for extension or improvement of the SDE can only come from an understanding of its strengths and deficiencies. Knowing how the SDE elements are employed by the spectrum of users throughout the life cycle of each particular software deliverable is a vital part of this understanding. We conclude that the SDE should automatically collect data that characterizes its use throughout the entire development process.

LANGUAGE PANEL SUMMARY

This panel was charged with making recommendations on the various language issues involved in the development of Space Station. This charge included the full set of development and user languages covering the entire life cycle of development and all types of user applications.

The selection and standardization of languages and interfaces for the Space Station program are critical needs to insure the success of this predominately engineering activity. While the Language Panel recognizes that the project life cycle will require a family of languages for the various classes of users and developers, it is crucial to begin making decisions which will focus planning efforts by limiting the range of possible selections. Requirements for the Space Station information system long-term maintenance and evolution will make it imperative that a high-order development language be utilized. It is recommended that the primary high-order language for source code development be Ada. (Ada is a registered trademark of the Department of Defense, Ada Joint Program Office.) Issues related to the utilization of Ada should be addressed as soon as possible. These include developing a transition strategy, providing education, accommodating the utilization of software already in existence, and developing fall-back options for high risk areas. One high-risk area is satisfying the requirements for run-time support for target systems, especially when the targets are distributed. Requirements for design specification languages or interfaces that complement Ada should be determined.

During its discussions, the panel operated under the basic assumption that Space Station is an engineering activity. Therefore, where appropriate, selection and standardization of languages and interfaces should begin constraining the degrees of freedom. The selection of languages and interfaces impacts the construction of a Software Development Environment (SDE), which is a substantially more critical component of Space Station software.

Although there were panels to discuss management, standards, environments, and languages, no panel was specifically charged with methodology issues. This is of real concern, and the language panel tried to address this issue whenever it was appropriate. The panel also felt that methodology should be discussed in any future meetings on software.

The panel was able by consensus to arrive at a total of 11 recommendations. These recommendations were discussed in the open forum, and there was felt to be reasonable agreement of the attendees at the open meeting.

These recommendations fall into 5 categories. Recommendation 1 deals with an important aspect of the whole software development process. Recommendations 2, 3, 4, and 5 deal with the choice of the software development language. Recommendations 6, 7 and 8 deal with languages at early phases of the life cycle. Recommendations 9 and 10 deal with user languages. The last recommendation says that NASA must track language technology in the future.

RECOMMENDATIONS

1. NASA should avoid premature commitment to hardware implementation decisions. System and software architecture should be defined first.

2. NASA should declare Ada now as the preferred high-order language for source code development and address the following issues as quickly as possible:

- transition strategy
- procurement issues
- interfaces to existing NASA software
- development of guidelines for applying Ada to various application areas
- development of appropriate run-time support environments for NASA applications
- education
- a liaison to DoD
- a seat on the Ada board
- benchmarks for performance
- prototyping
- development of appropriate tools to partition and allocate Ada entities across distributed applications
- introduction and utilization of reusable components
- investigation of fallback position options for high risk areas

3. The commitment to Ada requires an education program in software engineering methodologies with Ada, which should begin as soon as possible. The education includes the study of relevant examples. It should cover multiple levels of management, application programmers, etc.

4. NASA must define its requirements for the run-time support library and kernel for the target systems, including distributed targets.

5. NASA needs to define the requirements for the interface to the run-time system.

6. The first version of the SDE should not be constrained to have a single requirements language, AI expert systems language, or prototyping language.

7. NASA should determine the requirements for and select or develop requirements and design specification languages or interfaces that complement the SDE and Ada.

8. The design language should be syntactically and semantically consistent with the development language and should have on-line support for interface checks, etc.

9. For all levels of user interfaces, there should be a set of standards to provide commonality across all phases of the Space Station life cycle.

10. NASA should identify all categories of users and user interfaces, and quickly proceed with rapid prototyping to determine the real requirements.

11. Since Space Station software will evolve over 30 years, NASA should track language technology and act appropriately.

LANGUAGE ISSUES FOR SPACE STATION

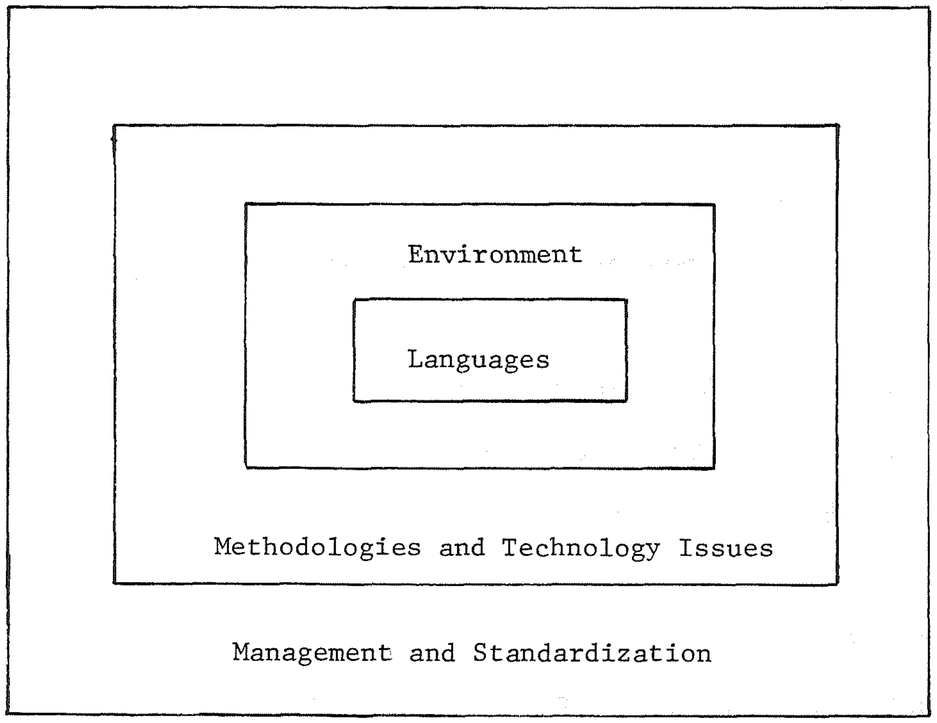
Professor Victor Basili began by reviewing the essential considerations and initial recommendations of 1984 workshop (ref. 1) given in the next three figures. He commented that language was to be considered as a notation and tool for supporting

- application domains
- phases of the life cycle
- methods

in such a way that it satisfies criteria of ease of use, readability, efficiency, modifiability, portability, low cost, etc.

Therefore we need to (1) categorize applications, e.g. flight software, support systems, and operations, (2) categorize phases of the life cycle, e.g. requirements, design, code, and test, and (3) delineate methodologies and recommend languages or criteria for selecting a family or set of languages for use in Space Station.

One of the concerns in choosing languages is that because they are an integral part of the software development environment, the decisions on languages cannot be made independent of the decisions about that environment. In turn, the environment will and should be influenced and constrained by the methodological and technological issues decided upon for Space Station. These methodological issues will certainly be influenced by the management and standardization issues.


Environment
Languages
Methodologies and Technology Issues
Management and Standardization

ESSENTIAL CONSIDERATIONS

1. REQUIREMENTS

   HAVE DEFINED CANDIDATE LANGUAGES FOR OPERATION

   NEED STUDY FOR DEVELOPMENT

2. USE OF LANGUAGES

COBOL, FORTRAN, HAL/S	PRIMARY
C, PASCAL, PL/1	SOME GAINS

3. SOFTWARE HERITAGE AND REUSABILITY

   LONG LIFE OF SPACE STATION-SPACE TRANSPORTATION SYSTEM

4. EVOLUTION OF LANGUAGES

   STRATEGY FOR CHANGING LANGUAGES OVER TIME

5. GENERAL AND SPECIAL PURPOSE LANGUAGES

   HOW MANY LANGUAGES ARE NECESSARY?

   HOW DO WE HANDLE A MULTIPLICITY OF LANGUAGES?

6. STANDARDIZATION

   SHOULD THE LANGUAGE DEFINITION BE STANDARDIZED?

ESSENTIAL CONSIDERATIONS

7. ASSEMBLY LANGUAGE

   HOW MUCH, IF ANY, ASSEMBLY LANGUAGE SHOULD BE ALLOWED?

8. TOOLS

   WHAT IS THE EFFECT OF LANGUAGE SELECTION ON AN INTEGRATED SET OF TOOLS?

9. MULTI-LINGUAL ENVIRONMENTS

   HOW ARE LANGUAGES CHOSEN TO BE COMPATIBLE WITH EACH OTHER AND THE SOFTWARE (HARDWARE) NETWORK ARCHITECTURES TO BE USED?

10. DISTRIBUTED PROCESSING

    HOW WILL THE LANGUAGE SUPPORT DISTRIBUTED PROCESSING?

11. TRANSPORTABILITY

    HOW WILL THE LANGUAGE ADDRESS TRANSPORTABILITY CONCERNS?

12. LESSONS LEARNED

    HOW DO WE MAKE USE OF THE DATA ON LESSONS LEARNED ABOUT SOFTWARE MANAGEMENT?

INITIAL RECOMMENDATIONS FOR PANEL CONSIDERATION

1. REVISIT "HIGH ORDER LANGUAGE" WHITE PAPER (AUDREY DOROFEE)
2. USE ANSI STANDARDS
3. COLLECT DATA ABOUT DEVELOPMENT TO DETERMINE EVOLUTIONARY APPLICATIONS
4. ESTABLISH GENERIC REQUIREMENTS OF TOOLS
5. STANDARDIZE ON LANGUAGE - STUDY ADA
6. USE OF ASSEMBLY LANGUAGE SHOULD BE MINIMIZED
7. EVALUATE ADVANTAGES AND DISADVANTAGES OF A CANDIDATE SET OF LANGUAGES
8. EVALUATE DISTRIBUTED PROCESSING MACHINES WITH RESPECT TO LANGUAGES AND TOOLS
9. EVALUATE LANGUAGES FOR REQUIREMENTS AND SPECIFICATION, DESIGN, AND SPECIAL APPLICATIONS

Basili proposed that the panel proceed by (1) generating a set of goals based upon the requirements for Space Station, (2) refining (and defining) those goals for the various languages into a set of technology questions that should be answered, and (3) selecting languages or giving selection criteria based upon the answers to these questions.

Sample goal areas include theoretical, technical, methodological, political, management, and application oriented issues. Sample questions in these areas (adapted from questions posed by Susan Gerhart on Prolog) are:

Theory:

- Is the language well defined?
- What are the functional capabilities of the language and its limitations?

Technology:

- How stable is the technology behind the language design, the compiler design?
- Are there production quality compilers or interpreters?
- Are there performance issues that need to be addressed?
- Are there adequate development environments?
- How does the technology behind the language compare with the technology behind other languages in its class?
- What kinds of tools exist?
- Is there control of the definition of the language?

Methodology:

- What methodologies does the language support?
- Can the language be combined or interfaced with other languages and systems?
- Will the programs in the language make use of existing software in other languages?
- How are the usual desirable properties of programs, such as correctness, robustness, efficiency, modifiability, etc., addressed in the language?
- Can the language be integrated with other phase languages across the entire life cycle?

- How are other technologies supported by the language, e.g. transportability, distributed processing, prototyping, etc.

Applications:

- What application areas does the language address?

- What application libraries exist?

- What application areas have used the language?

Management:

- How does one manage (plan, control, direct) projects in the language?

- Can modern software engineering practices be brought to bear on projects in the language?

- What is involved in the training of personnel in the language?

Evaluation:

- Are there marketing and technical projections for the language?

- How does one become proficient in the language?

- What evaluations or case studies have been done, and what are the concerns and benefits they point out?

Social, Political, Historical:

- Is the language politically sound?

- What controversies has it gone through?

- What is the extent of its use?

RATIONALE FOR RECOMMENDATIONS

1. Recommendation:

NASA should avoid premature commitment to hardware implementation decisions. System and software architecture should be defined first.

Rationale:

A recurring problem with large systems, particularly those with a long development cycle, is that the hardware is selected (or mandated) before the system architecture is designed. As a result, the software architecture is overconstrained, memory and performance become serious constraints as the requirements evolve, and the hardware is obsolete before the system is operational.

By delaying selection of the hardware until the system and software architecture is understood, NASA can make intelligent engineering trade-offs between hardware and software. System and software architecture should allow

- early prototyping using available hardware or emulation,
- use of the most advanced hardware available when it is time to commit, and
- replacement of this hardware later with minimum impact.

This recommendation complements the SDE panel recommendation that the SDE support multiple targets. It does not conflict with the aggressive adoption of standards; rather, it serves to focus on adoption of standards at the appropriate level (e.g., bus standards and protocols). It is also consistent with the choice of Ada as the implementation language, provided that portability guidelines are developed and stressed.

2. Recommendation:

NASA should declare Ada now as the preferred high-order language for source code development and address the following issues as quickly as possible:

- transition strategy
- procurement issues
- interfaces to existing NASA software
- development of guidelines for applying Ada to various application areas
- development of appropriate run-time support environments for NASA applications
- education
- a liaison to DoD

- a seat on the Ada board
- benchmarks for performance
- prototyping
- development of appropriate tools to partition and allocate Ada entities across distributed applications
- introduction and utilization of reusable components
- investigation of fallback position options for high risk areas

Rationale:

Many aspects of Space Station software would be simpler if it were all written in a single programming language: compilers, support tools, training, software reusability, maintenance. Such uniformity is of course not completely realizable, for no single language would be appropriate in every case, and NASA already has software in several languages. Nevertheless, selecting one high-order language as the preferred language for new software and supporting this choice with the SDE and training would focus the Space Station software effort and foster the aforementioned benefits of commonality. Calling this selection a preference instead of a requirement would leave room for NASA to allow the use of other languages when it is more appropriate, while firmly establishing the direction of NASA's economic and organizational support.

If a single high-order language is to be preferred, it should be evaluated according to several criteria outlined elsewhere. One of these criteria is support for modern software engineering methods. It would certainly be a mistake for NASA to prefer a language that did not support these methods, for such a language would inevitably tend to impede their use. A high-order language supporting abstraction, information hiding, communicating sequential processes, and similar concepts would be a welcome improvement over older languages that do not adequately support these methods.

After reviewing the alternatives, the panel concluded that Ada is the language showing the greatest potential in this regard. Ada's strong data typing, packages, generics, and overloading support abstraction and information hiding. The exception handling capability supports the reporting and handling of errors and unlikely situations in a manner consistent with abstraction. Tasking supports communicating sequential processes at a higher level (analogous to procedure call) than other synchronization mechanisms, such as semaphores. Arithmetic is well defined and supports efficient fixed-point operations. Representation clauses support interrupt handling, hardware input-output interfaces, and similar implementation-dependent matters. Separate compilation supports efficient software development and distribution. Although Ada is a large and complex language, its features are useful.

Unlike most languages supporting modern software engineering methods, Ada is not a product of the academic community, with informal support and uncontrolled changes; nor is it a proprietary language with limited availability. Ada is a government and ANSI standard, and as such it is stable and supported. This support is rapidly growing. More and more compilers and programming environments for various host and target machines are coming onto the market. Applications are also being written in Ada. (The company of one panel member has already generated more than one million lines of Ada code.) Resources supporting Ada application developments are already in the range of one half billion dollars per year. The research community has taken a great interest in Ada and distributed systems, program design, program validation, and other areas applicable to Space Station. By selecting Ada, NASA can capitalize on this substantial investment and begin to influence the course of future Ada work.

If NASA is to choose Ada, it should do so now, so that activities dependent on this choice can begin. NASA and contractors need time for education, planning, and the specification of Ada-related requirements. Ada vendors need time to become aware of the new market provided by Space Station and to adapt compilers and run-time support packages to Space Station requirements. NASA must also address the series of issues enumerated in this recommendation, which are discussed below.

The first thing NASA must do is to formulate a strategy for the transition to Ada. Naturally, these plans will involve the management, standards, and SDE issues considered by the other panels. In particular, the SDE must include a full set of software development tools compatible with Ada.

Procurement issues must be addressed, including

- development of Ada compilers and run-time packages for new environments

- contractual obligation to use Ada and the SDE -- who will bear the risks?

- contractual obligation to use Ada properly -- how can the use of appropriate software engineering methods be guaranteed?

- waivers -- when is another language preferable for new software?

- procurement of off-the-shelf software -- should it too be written in Ada, in case NASA should have to take over its maintenance? How would this affect its cost and availability?

NASA must decide how to apply its large base of existing software to Space Station:

- NASA could continue to use stand-alone software, as long as maintenance costs were not excessive.

- Other software could be used directly within an Ada environment, if suitable implementations of the "interface" pragma existed in that environment. NASA would probably have to fund the development of Ada interfaces to HAL/S and any other NASA-specific languages. Perhaps it would be better to rewrite such software in Ada: this would often be straightforward, the resulting Ada code would be much more portable, and it might even be economical if the software had to be changed anyway.

- Software that is not directly reusable may contain the only existing documentation for algorithms applicable to Space Station. Important algorithms that would be difficult to re-derive should not be lost; Ada or an Ada-based PDL would be an ideal medium for preserving and documenting them, as well as using them.

A caveat is in order, however: Most older software was developed without benefit of concepts that enhance reusability and ease of change, such as abstraction, information hiding, and even good documentation. Consequently, the strategies noted above may prove difficult. Old software should be evaluated and adapted using the same criteria applied to new software; to do otherwise would defeat much of the purpose of using Ada and would prolong reusability, portability, and maintainability problems into the 21st century.

Training in the proper use of Ada is of such importance that the panel made a separate recommendation in this area (see recommendation 3).

Any major Ada user should have close ties with the Ada community at large. Consequently, NASA should establish a liaison with DoD and the Ada Joint Program Office. Furthermore, any agency committing such an important and visible project to Ada deserves a voice in Ada's future development. Therefore, NASA should seek a seat on the Ada board.

Currently, Ada compilers are validated by the DoD with respect to correctness only; they do not have to pass any performance benchmarks. Since performance will be a major issue in many Space Station applications, NASA should initiate or jointly sponsor a benchmarking activity for evaluation of Ada compilers and support libraries. It should test the performance of Ada programs in distributed systems and high-speed real-time systems as well as in more routine contexts. Such benchmarks will also help to identify high-risk areas needing attention.

Ada's support for abstraction and information hiding makes it especially good for rapid prototyping. Once a design has been produced in the form of a collection of Ada package specifications (with associated semantics), the component packages can be implemented in parallel, each without regard for how the others are implemented. Such a prototype can then be transformed into a finished product by independently changing the implementations of each of its components. With the interface pragma or a special interface package, the SDE might also support the rapid implementation of an Ada package using a separate program, perhaps in a very high level language (e.g., Prolog). NASA should use early prototyping to investigate application areas such as fault-tolerant and distributed systems. This would help determine how well Ada supports these applications and would consequently reduce the present uncertainty in this regard.

The use of Ada in distributed systems, including the need for tools to allocate Ada entities across such systems, is addressed further in recommendations 4 and 5.

Space Station applications should share the same software wherever possible. Reusable software can reduce the cost of software requirements specification, decomposition, and design (because it is often easier to recognize what is needed than to define it), coding and testing (because neither is needed in order to use an existing, tested implementation), and maintenance (because changes to one reusable module are cheaper than changes to several nearly identical ones). Ada is an excellent tool for supporting reusability, since reusability is directly related to abstraction and information hiding. However, it is no trivial matter to design abstractions that are amenable to reuse. To support reusable software, NASA should

- develop or adopt a taxonomy of software abstractions,

- identify specific reusable abstractions,

- develop a library of Ada package specifications for these abstractions, catalogued according to the aforementioned taxonomy (so that projects can find packages useful to them)

- develop a prototype package body for each library package (so that projects can test code that uses these packages),

- publicize the library and encourage -- perhaps even reward -- the use of its packages,

- develop efficient package bodies for each library package (so that projects can test their products for performance and release them), and

- devise a plan for adding to this software library.

The SDE should support the use of reusable components from this library and the search of the library catalog for components of interest. In addition, it should allow the library to contain more than one implementation version of a single Ada package, so that users can select from implementations optimized in different ways (e.g., execution speed versus memory required).

The choice of Ada is not without risk, although much of it is in areas that will be risky whether Ada is used or not. In particular, in some quarters there is uncertainty about (1) the applicability of Ada to distributed, fault-tolerant, and hard real-time systems, (2) the efficiency of Ada run-time support environments and of code generated by Ada compilers, especially for tasking in real-time and distributed systems, and (3) the development of good Ada implementations for the particular machine architectures that might be used for Space Station.

Prototyping, benchmarking, and work on run-time support environments should resolve the first two issues. Postponement of hardware selection and the eventual use of off-the-shelf machine architectures should minimize the last problem, by reducing the chance that an unexpectedly difficult architecture will be selected with insufficient time to produce a good implementation for it. Nevertheless, at least until these problems have been put to rest, fallback policies should be established in each of these problem areas.

3. Recommendation:

The commitment to Ada requires an education program in software engineering methodologies with Ada, which should begin as soon as possible. The education includes the study of relevant examples. It should cover multiple levels of management, application programmers, etc.

Rationale:

The rationale behind this recommendation may be perceived from three perspectives: systems engineering, methodology, and language.

From a systems engineering perspective, Space Station software is just one important part of a complex system. Software management, development, acquisition, and evolution are all subordinate to a total systems engineering activity requiring management and technology trade-offs. These trade-offs are constrained by practices, obligations, and requirements at the project, systems, subsystems, and institutional levels. To make intelligent decisions at all levels of management and engineering, NASA personnel need to understand, to different degrees and from different perspectives, the programming and engineering capabilities and limitations of Ada, and the management implications of using Ada.

For example, if it went uncorrected, the myth that Ada is inherently inefficient could distort evaluations of trade-offs between hardware and software, or between Ada and some other programming language. As a more positive example, an understanding of how Ada and the technique of information hiding can support abstract interfaces to hardware would make the strategy of postponing hardware selection appear much more practical.

From the methodological perspective, Ada is more than a mere programming language. It embodies and supports modern software engineering concepts, such as rich data structures, data abstraction, information hiding, modular packaging, exception handling, and communicating sequential processes. It has features that enforce disciplined engineering, such as strong typing. It (or a derivative) can be used as a high-level program and system design language. It is to be used in conjunction with an Ada Programming Support Environment comprising powerful tools for software development. Together, these form a system supporting modern software engineering methods. To ensure that developers and contractors take full advantage of these methods and Ada's support for them, NASA personnel must themselves understand them.

Although the aforementioned software engineering concepts are well known in the academic and research communities, they have not penetrated the software community at large to any great degree. Consequently, many software professionals will come to the Space Station project without experience in applying these concepts, and sometimes without even a basic understanding of them. Therefore, NASA will need a training program that provides

- good definitions of these concepts,
- examples of their use, and
- practice in applying them to program design and implementation with Ada.

The shortage of professionals trained in these methods extends to the education and training community itself, so NASA should establish a quality assurance program to guide and audit this training.

For example, the important concepts of abstraction, information hiding, and communicating sequential processes can be briefly defined and related to Ada as follows:

- Abstraction supports the orderly decomposition of a software system into components that can be understood solely by reference to their interface specifications, which include black-box descriptions of the associated behavior; implementation details are suppressed. In addition to facilitating the program design process, this enhances software reusability, since each abstraction is a potentially reusable design. Each of Ada's compilation units (package, task, subprogram, and generic) supports a kind of abstraction.

- Information hiding emphasizes the importance of concealing the details of the implementation of an abstraction. Because these details are hidden, users of the abstraction cannot make unwarranted assumptions about the implementation; this makes it easier to change the implementation without affecting the software that uses it. Information hiding involves designing the abstractions used to build a system so that each aspect of the system that is judged likely to change is hidden behind a single abstraction; by anticipating changes, it makes those changes easier. Ada's packages, visibility rules, and private types support information hiding.

- Communicating sequential processes (CSP's) allow the decomposition of a system into tasks that logically run in parallel, occasionally communicating with one another. Complex real-time systems can be built using CSP's, and distributed systems can be implemented by assigning CSP's to different processors; however, many more mundane problems also have natural solutions involving CSP's. Ada's tasks support communicating sequential processes.

Viewed simply as a rich language, Ada can either be applied properly to solve complex problems, or it can be misused to complicate solutions. A programmer experienced with conventional languages may be tempted to use Ada as a conventional language with new syntax. This mode of application would be most unfortunate, for it would defeat the fundamental purpose of Ada's existence, which is to foster the use of methods mentioned above. To fully exploit Ada's many features, programmers (both NASA personnel and contractors) need training on its proper usage.

The study of relevant examples will be an important part of all this training. Obviously, examples of Ada programs will be relevant in this case. However, bad Ada programs should not be used as examples -- other than examples of what not to do. Unfortunately, there is a real danger here: some books on Ada utterly fail to address the software engineering principles that Ada was developed to support, and instead teach little more than mechanical translation of bad programs in other languages into bad programs in Ada.

On the other hand, some of the best and most relevant examples may not even use Ada. Examples of good software engineering methods are rare, and fully worked out examples of systems of reasonable size are rarer still. Some of these may use other languages, but they will nevertheless be worthy of study by those involved in software design, for it is the method of decomposing software into modules and defining the interfaces of those modules -- the software architecture of the system -- that is the most important aspect of an example. A good architecture will be valid regardless of the implementation language, and it will be easy to map into Ada.

NASA should search the literature for examples of good software design applicable to the use of Ada before trying to develop them in house or under contract. Even if an example is not fully implemented, it may still contain useful material.

At this time the pool of trained Ada professionals, particularly lead designers, is very small. The typical training time for a lead designer may be as much as a year. NASA must rapidly select or develop training methods that will ensure a sufficient supply of trained professionals for the Space Station program. Training may prove to be the largest startup cost of the transition to Ada. The duration and success of this training will have a strong effect on the long software life cycle projected for Space Station.

4. Recommendation:

NASA must define its requirements for the run-time support library and kernel for the target systems, including distributed targets.

Rationale:

To derive the maximum benefits from the choice of Ada as the preferred high order language for source code development, NASA should move quickly to determine and catalog its requirements for the run-time support environment of target processors to be embedded within the applications needed for the Space Station program (e.g., highly data-driven applications versus critical, real-time applications). Although such requirements are not unique to NASA, the panel feels that

- the development schedule for the Space Station program plus

- the lack of an appropriate catalog of requirements for the run-time support environment of processors embedded in large, complex, distributed applications

should cause NASA to quickly take a leading role in defining such requirements.

The run-time support environment (RTSE) provides resource management and other services to the object code modules of the application programs. This support is typically provided by a run-time kernel, which separates both the application modules and the run-time library modules from the bare target processor. The kernel contains a minimal set of functions that are used frequently and must be executed sequentially.

The run-time library may be divided into a basic library set and an extended library set. The basic library contains modules that provide services to the object code modules produced by a host Ada Programming Support Environment (APSE) for a broad class of applications. The full set of basic library modules need not be present on all target processors. For example, if the application program objects assigned to a given target processor do not require Ada's tasking or heap management, then the basic library modules responsible for those facilities may be omitted from the run-time environment.

The extended library contains modules that may be used to support APSE-produced object code in specific applications having requirements beyond those addressed in the Ada Language Reference Manual (ref. 3). For example, many applications would benefit from a run-time "monitor" that gathers and reports performance statistics and facilitates remote diagnostics and reconfiguration. Other modules might support multi-level security and access control, or transactions with nested atomic actions. All such modules could be transparent at the Ada source code level and thus facilitate the cost effective utilization of reusable components across a broader spectrum of applications.

Clearly, regardless of the efficiency and reliability of the object code produced by a host APSE, the performance and reliability of the executing program are dependent on the run-time kernel and library.

Another important reason why NASA should begin quickly to define its requirements is the complexity spectrum of implementing RTSE's shown below:

- Single "stand-alone" embedded processor to support
    - subsets of Ada
    - full Ada
- Multiprocessor applications, which support the partitioning and allocation of objects within the Ada programs for execution in target environments implemented with
    - shared memory
    - shared bus
    - "n level" redundancy
    - combinations of the preceding
- Distributed network applications, which support the partitioning and allocation of objects within the Ada programs among geographically separate processing resources for execution. Such implementations may include
    - Local area networks composed of single processor nodes and multiprocessor clusters
    - Remote area networks of local area networks, single processor nodes and multiprocessor clusters.

Ada was designed to serve as a "common language for programming large scale and real time systems" (Foreword, ref. 3). The objects of an Ada program can be distributed "whenever an implementation can detect that the same effect can be guaranteed" as for execution by a single processor (Section 9, ref. 3). However, the current implementations of Ada compilers and environments respond only to the requirements for a Minimal tool set (MAPSE). Those requirements address a single, stand-alone target processor, and therefore only the simplest RTSE on the complexity scale. Specifically, the MAPSE does not require the tools needed for

- allowing the software engineer to scan the Ada source code and identify which program objects should be allocated to which target resources and then
- building the load modules of application code and, possibly, run-time library modules to be exported to the various target processors.

(It should also be noted that such tools have not been created in the HAL/S environment or in other environments that were not designed to support large, complex distributed applications.) The construction of such tools as a necessary part of the Space Station program's Software Development Environment is dependent upon an understanding of NASA's requirements for a catalog of features and options for the run-time kernel and run-time library.

5. Recommendation:

NASA needs to define the requirements for the interface to the run-time system.

Rationale:

Whereas recommendation 4 addressed the need for NASA to begin defining its spectrum of requirements for the functionality, performance, and reliability of the run-time support environments needed for the Space Station program, this recommendation focuses specifically on the requirements for the interface of the object code of the application programs to the run-time kernel and run-time library.

A major goal of the Space Station program is to support technology transparency. The economics of thirty or more years of Space Station evolution, operation, and maintenance will require that diverse instruction set architectures (ISA's) coexist in the target environment. Some of these ISA's will participate in subsystem activities that provide an integrated, end-to-end information system from earth stations, through entities in various earth orbits, to a permanent presence on the moon. Some of them will participate in subsystems that must operate continuously during diagnostics, repair, expansion, reconfiguration, software and hardware updates, and other system activities. Thus, the ability to map the object code modules of applications programs to an interface model of a virtual Ada machine is highly desirable.

Hiding machine dependencies as much as possible (consistent with NASA's requirements for RTSE functionality, performance, and reliability) and encapsulating code that must be machine dependent will enhance the transportability, reusability, and interoperability of Ada source code modules and thus help control the costs of software ownership and incremental development.

Organized, international working groups are now addressing these interface issues. NASA should take a leading role in advancing this work.

6. Recommendation:

The first version of the SDE should not be constrained to have a single requirements language, AI expert systems language, or prototyping language.

Rationale:

There are a number of requirements methodologies, languages, and tools that might be of use for Space Station software development. The panel considered whether NASA should select a preferred or standard set of requirements languages, to facilitate communication among space station participants and contractors. However, the panel decided not to recommend this because

- Space Station needs in this regard are not yet well defined;
- requirements methodologies, languages, and tools have not yet reached the degree of maturity required for selecting standards; and
- it is not clear that any of the currently available items is adequate for Space Station needs.

Similar considerations make it premature to select other specialized languages, such as expert system languages and prototyping languages.

However, because all these types of development aids have potential for improving the productivity of the software and system life cycle, their use should be explored. For this purpose, the SDE should initially offer a selection of languages of each type. Many of the criteria for language selection given elsewhere should be applied to the evaluation of these languages. [See also recommendation 7.]

7. Recommendation:

NASA should determine the requirements for, and select or develop requirements and design specification languages or interfaces that complement the SDE and Ada.

Rationale:

The specification of both system and software requirements and designs for a system as complex as the Space Station is a major undertaking that is crucial to system success or failure. Previous programs at NASA and elsewhere have identified requirements specification in particular as an extremely difficult activity in the system life cycle. It often has been characterized as a chaotic decision-making process exacerbated by a lack of adequate methods, languages, notations, and tools. Research and development efforts over a decade or more have resulted in a number of approaches and tools, some of which have merit for the Space Station effort.

The panel considered whether NASA should simply rely on existing languages and tools to meet Space Station needs. The panel did recommend that several of these aids should be part of the initial SDE [see recommendation 6].

However, the magnitude of the Space Station undertaking and the benefits of good requirements and design specification aids argue strongly for a focused, early effort to define and then acquire a set of tools tailored to meet the specific needs of the Space Station program. The panel expects that many of these tools will be commercially available, but some may have to be developed. NASA's goal should be a set of standardizable requirements languages and interfaces that can be used to facilitate communication among all Space Station participants.

If Ada is to be the primary software implementation language, then any requirements and design methods eventually adopted should be consistent with the use of Ada. (Recommendation 8 addresses program design languages in more detail.) Similarly, SDE support for these methods is crucial if they are to be used efficiently and in a disciplined manner.

8. Recommendation:

The design language should be syntactically and semantically consistent with the development language and should have on-line support for interface checks, etc.

Rationale:

The use of a program design language (PDL) is a recognized component of good software engineering practice. A common excuse for avoiding the practice is that, as the software evolves, the PDL is an added cost and often becomes inconsistent with the code.

These difficulties can be overcome if the PDL is consistent with the development language because the PDL is embedded in the implementation. As such a software structure evolves, the PDL is maintained naturally. Further, designs using such a PDL can be checked for semantic consistency.

Given Ada's facilities for structuring software, the use of an Ada-compatible PDL would allow semantic consistency to be maintained throughout the software implementation. The SDE should therefore support the use of an Ada-based PDL. The IEEE is currently completing a standard for the use of Ada as a PDL; NASA should investigate whether this standard is appropriate.

In cases where Ada is not used as an implementation language, an explicit decision should be made whether to use Ada as a PDL or to use a PDL consistent with the implementation language. In fact, this decision should be considered in the trade-off analysis leading to selection of a language other than Ada for a particular application.

9. Recommendation:

For all levels of user interfaces, there should be a set of standards to provide commonality across all phases of the Space Station life cycle.

Rationale:

The need for a set of standards for user interfaces is driven by the following considerations:

- the long life cycle of the Space Station and its support systems and environments,
- the constantly changing and growing set of users,
- the use of common or government furnished support systems and environments,
- the need to minimize program costs, including software, training, and customer costs,
- the high degree of commonality in the functions performed by various types and groups of users, and
- the high degree of coordination and integration of activities and products required throughout the program.

A set of standards for user interfaces (i.e., methods and languages) will

- permit users to migrate among sites and across support systems and environments without the need for extensive retraining,
- provide a greater degree of portability and reusability of user generated procedures and programs,

- decrease communications, coordination, and data exchange problems among user groups,

- provide a central core to which unique user interface requirements can be added, and

- minimize the amount and cost of user interface software, documents, tools, and training.

10. Recommendation:

NASA should identify all categories of users and user interfaces, and quickly proceed with rapid prototyping to determine the real requirements.

Rationale:

User interfaces are an essential part of any support system, environment, or tool. The definition and design of user interfaces come early in the life cycle of support systems, environments, and tools. If there is to be a set of standards for Space Station user interfaces (as in recommendation 9), all user categories must be identified, and their interface requirements must be defined and analyzed to derive that set of standards.

To be of maximum benefit to the program, these standards must be ready in time to be applied to the work that has already begun on common and government furnished support systems and environments. These systems and environments will not only have their own user interfaces, they will also support the development of software, tools, and systems having still more user interfaces. It is therefore imperative that users and user interface requirements be identified as soon as possible.

Rapid prototyping would probably be the most viable method leading to the definition of interface requirements and the derivation of standards.

11. Recommendation:

Since Space Station software will evolve over 30 years, NASA should track language technology and act appropriately.

Rationale:

Thirty years is an unprecedented lifetime for software. No project of such duration should ignore the advance of relevant technology. Developments in software technology over the past thirty years -- especially those of the past decade -- presage even greater changes during the next thirty.

Some phases of the software life cycle do not have good language support at this time. The requirements definition phase is a case in point; should better language support emerge for requirements definition, NASA and the Space Station project would surely benefit from it. Similar reasoning applies to aspects of software outside the traditional life cycle, such as prototyping.

On the other hand, obstraction and information hiding will in any event continue to be fundamental principles for structuring software. This generality is important, because it supports the decomposition of a software engineering problem into subproblems that can be implemented independently, each in the most appropriate language. For instance, it should eventually be a straightforward matter to implement an Ada package specification as a program in a fifth-generation artificial intelligence language. This sort of flexibility should also be a goal of this SDE.

Languages evolve to support software technology and consequently serve as indicators of the state of that technology. NASA needs to track all software technology; tracking language technology is an important subset of such activity.

SOFTWARE STANDARDS PANEL SUMMARY

The unique and challenging nature of the Space Station Program requires that software standards be effectively used to control costs, facilitate enhancements and ensure safety. The Software Standards Panel identified and developed recommendations in four areas to help the Space Station Program achieve these objectives. The areas in which recommendations are offered are policy, organization, process and candidate software standards for the Space Station Program. The consensus process employed by the panel involved:

A. Initial survey of general software standards issues.

B. Analysis of the specific software standards issues stated in reference 1.

C. Restatement of issues and discussion in open panel session.

D. Consideration of alternate recommendations.

E. Development, presentation and discussion of specific recommendations in open panel session.

A list of the recommendations arrived at in the above manner is given in the following section. The panel did not attempt to recommend the selection of specific software standards, but did recommend that NASA move at once to act on the selection of standards in specific areas. A minority of the standards panel, as well as large number of audience participants, took the position that current software standards have grown into areas that are not consistent with the traditional concept of standards. In other words, the current definition (usage and implementation) has been bent far beyond a useful definition of "standards". A critical re-examination of standards, at this time, would be in order.

RECOMMENDATIONS

The Software Standards Panel recommends that the Space Station Program Office take the following actions:

1. Establish a Program policy supporting software standards.

2. Establish an organizational structure to support software standards at each level within the Space Station Program.

3. Capitalize on existing software standards to meet Program requirements.

4. Establish software standards early in specific candidate areas.