Future Ada* environments

by SABINA H. SAIB

*General Research Corporation
Santa Barbara, California

ABSTRACT

The current Ada environments are oriented toward traditional code production tools such as editors, compilers, loaders, and program library managers. Future Ada environments will add to the initial capabilities to provide support from the initiation of requirements to the enhancement of existing operational software. In addition to software development facilities, future Ada environments will support management activities. The future will also see applications of current tools and techniques across the entire life cycle.

*Ada is a trademark of the Department of Defense.
INTRODUCTION

Ideally all phases of the software development cycle from requirements to maintenance or enhancement should be supported by a software environment. Many current software tools contain capabilities for supporting one phase of the life cycle. These tools ignore what has gone before in the life cycle or what will happen in the next phase. Often different languages that are oriented towards a portion of the life cycle are used, so that the tool is totally incompatible with all other tools in use. In the use of such tools, much effort is expended in transferring incompatible data from one tool to another. This effort could be avoided by implementing an integrated support environment. During all phases of software development, management needs to be able to see what the status of a project is. Today this is often done without the use of tools that can look at the actual state of the software. As a result, the management view is often incorrect and management is unable to address problems in a timely manner.

LIFE CYCLE CONSIDERATIONS

The traditional approach to identifying the capabilities needed in a software development environment has been to examine each phase in the software development life cycle as a separate activity. Occasionally its relationship to immediately preceding and following phases is also considered. Recently emphasis has been placed on the ability to trace requirements through specifications and design to the implemented code elements and acceptance tests. Another approach to developing an environment for life cycle support is to examine the needs of the maintenance or enhancement phase. The impact of a change in the requirements must be traceable out to the affected software components, and the proposed changes to the software must be traceable back to the unaffected requirements.

GOALS FOR AN ENVIRONMENT

Any future Ada environment must include a wide variety of capabilities in order to support the development of software during all phases of the life cycle. It must be easy to use, and assist the user not only in such detailed activities as interactive debugging, but also in organizing and directing the effort. The environment must be extensible to allow the addition of capabilities as new tools and techniques emerge, and it must be standardized across machine architectures, operating systems, and file access methods. The capabilities for an environment can be implemented as a set of cooperating development and management tools. These tools can be clustered into three major groups: multipurpose, software production, and management.

Multipurpose tools find use in numerous phases of the software life cycle. These tools for examining and updating text files and the generation of reports must be designed with general purpose capabilities to reflect their universal usage. The most important multipurpose tool that can be developed for an environment is the database manager. The Ada environment requires a single database for all environment activities. Such a database will automatically provide a history of a project and avoid duplication of database functions among tools.

Software production tools that work together can form a comprehensive package of support for a user. Some specific tools that should be developed for the future Ada environments are requirements processor, specification processor, design analyzer, coding assistant, standards checker, compiler, static analyzer, linking loader, configuration manager, test assister, and verifier.

Management tools fill needs that differ from those of the software production staff. Activities that should be supported by tools are planning, staffing, controlling, directing, organizing, and status reporting.

From a designer's viewpoint, the goals imply that the Ada environment must be both extensible and modifiable. Experience has shown that nearly all truly useful systems grow and change over time as new needs are developed and old ones become obsolete. Extensibility and modifiability are enhanced by the use of a single, uniform, functionally oriented command language. Tools will be contributed to the Ada environment from numerous sources. The sheer number of tools that should be provided makes it impossible for one group to be the sole suppliers to the environment. Furthermore, the particular tools within the environment will be changing. A tool should rely less on the physical format of information than on its logical structure. A uniform database system and database manager provide such a capability. The virtual memory manager in the Ada Integrated Environment appears to offer this capability. The design of a new tool can ignore which other tools in the environment create the information it needs; it only needs to know that the data will be created and will be accessible though a standardized format that stresses the logical rather than the physical characteristics of the data.

SOFTWARE DEVELOPMENT DATABASE

For each software project, a common information storage and retrieval system should provide a repository to consolidate all
relevant project data. The database also serves to unify the tools in the environment by providing common access to project data. The project database must, therefore, be common to all tools in the environment. Each tool should use the database access facilities to retrieve the data it requires and to store the information it derives. The use of such a database is shown in Figure 1.

The advantages of a common database are many. Information required by more than one tool can be computed and stored once, avoiding duplicate data files and extra processing. Also, information can be conveniently passed from one tool to another, communicating through the database. Information managed within a database is more reliable than data scattered in separated files. Information is less likely to become inconsistent, because the database can impose a number of consistency constraints. For example, analysis data may be inconsistent if a program has been modified since the analysis was done. Such analyses can be marked as obsolete when a program has been modified, so that reanalysis can be scheduled at a convenient point in the development.

Figure 1—Future Ada environments

Requirements and Specification Tools

The requirements state what a computer system should do from the user's viewpoint. The environment should aid the user or systems analyst who must enter the requirements in machine readable form and must aid the analyst who must convert the requirements into a system design.

There are five properties that a well-written requirements document should have; it should be

- **Complete**: say everything the implementor needs to know.
- **Consistent**: not contradict itself.
- **Testable**: implementor can objectively determine when the job has been done correctly.
- **Unambiguous**: implementor can interpret requirements in only one way.
- **Concise**: not ramble on.

To a certain extent the environment can help an analyst judge a set of requirements with respect to these criteria. In addition, it is possible to construct a requirements definition system that prompts the user for information so as to encourage requirements with these desirable properties and also to enhance the environment's ability to detect flaws.

Requirements are complete when all system inputs and outputs are fully characterized, system level error policy is stated, all documentation and deliverables are specified, and the functional relationships between inputs and outputs are stated. For inputs and outputs this requires specifying the data type, the value range, a prose description, a mnemonic name for reference, the source for the input and destination for the output, and its format. The environment can prompt the user for such information about each proposed system data item. This will help ensure that the requirements are complete. Furthermore, since the environment obtains the requirements through an interactive dialogue, the information will necessarily be in machine-readable and machine-analyzable form.

For system errors the user should decide on the error handling capabilities of the system. Error processing is too important to be left in the hands of systems analysts. The analyst’s job is to determine the feasibility and cost of the desired error processing capabilities. For each error condition the requirements tool should prompt the user for a description of the nature of the error, what the user would like to see displayed to indicate detection, such as a lit-up panel or a sounding alarm, and what recovery action to take, such as turning off a sensor.
The functional relationships between input and output should be specified in a moderately nonprocedural way, since the user who will supply them cannot be assumed to be a systems analyst. SADT, a manual system from Softech for requirements definition, is a nonprocedural graphical approach to system decomposition. Graphical support for requirements definition seems highly advantageous because a clear drawing can offer a better perspective on logical relationships than simple words can. An automated version of SADT that is merged with a prompting system as just described would be a valuable requirements tool.

In addition, the requirements tool should request a schedule of activities, deliverables, tolerances whenever approximate answers are possible, timing constraints, physical constraints, budgetary information, equipment to be used, applicable standards, testing practices, and acceptance procedures. By having the tool explicitly request this information, which will be primarily prose, hence not very analyzable, the system database will be able to index the requirements automatically for each of these major categories. Later, if the user wishes to request the set of applicable standards for the project, the database manager will be aware of which part of the requirements dealt with this subject. With a less structured format for entering requirements, such queries would be more difficult, if not impossible, for the database manager to successfully respond to.

If the requirements are entered in the fashion just described, it may be possible to perform limited tests for consistency. RSL, the TRW requirements specification language, has such a capability. With the limited information about the system available in a requirements document, it is still able to check, for example, that all input items are used in the computation of some output value, and that no output item is also treated as an input item at a different point in the system.

Another advantage of using an automated tool to enter requirements is that it simplifies tracing requirements into the design and code. The Requirements Tracing Tool developed by Logicon tries to achieve this by forcing requirements to be written in a format in which each identifiable requirement is tagged with an indexing number that is explicitly written into the design and code implementing the requirement. With the requirements tool described here, the tracing of requirements to design is automatic because the design will be generated automatically from the requirements. The system can report the relationships between design elements and those aspects of the requirements from which they were generated.

SOFTWARE DESIGN

A program design should not only be useful in the stage of the life cycle between the requirements phase and the implementation phase of a project, it should also aid the designer in expressing the design, aid a reviewer in checking the design, aid an implementer in developing the design into a product, aid a tester in validating the resulting software, and aid a maintainer in changing the design.

There are at least two levels of detail that a design language should support: programming in the large and programming in the small. In programming in the large, the user should be prevented from detailed design. It should be possible to use the requirements database to develop automatically a skeletal system design that contains the highest level modules written in pseudocode. Module inputs and outputs, a pseudocode description of purpose, the attributes of the inputs and outputs, and the interfaces between modules can all be automatically generated from the information gathered through the dialogue described earlier. The analyst can restructure this skeletal design and enhance it with the further details that are inappropriate for the requirements.

In the design stage, a user should be able to input fragments of a design and receive information on what has been entered, consistency checks on what has been entered, and if desired a measure of the completeness and complexity of the design description. Analysis of a design should proceed in stages such that the user is not inundated with information on the entire design while the design is incomplete. Analysis reports should be interactive, with the designer able to quickly alter the design, reanalyze that part of the design, and view the results on an interactive basis. The design analyzer should produce consistency reports at various levels of detailed analysis, documentation reports, a detailed design skeleton that an implementor can use, a test plan outline that a tester can use, a change history that a maintainer can use, and a project history that can track the design progress.

At the design stage there is much more room for analysis tools than at the requirements stage. One aspect of an Ada design that must be carefully analyzed at the design stage before much effort is spent in implementation is the package organization. In Ada, when a package is recompiled, all the programs using that package must be recompiled. Ideally, the module-package relationships will be simple, without a great deal of interdependencies between them. Tools showing the dependencies between packages and compilation units will be very useful at the design stage to minimize compilations when changes are made and to minimize communications between packages and multiple modules. Whether the design is written in syntactically correct Ada or in a mixture of legal Ada and prose that uses Ada keywords, the design tools should be able to automatically generate a syntactically legal Ada program turning the prose into comments. This program skeleton, which is automatically generated from the design description as the design skeleton was generated from the requirements, cannot help but be traceable to and consistent with the design.

TESTING SUPPORT

In addition to the debugging facilities provided by the compiler, there are a number of formal testing techniques that the environment should support with test tools. Formal testing should be supported at both the single-module and system (after integration) levels. The environment should maintain a record of the test description, test data sets used, modules tested, and test results. The testing information is useful not only in accounting for the test performed but also in determining the retesting requirements for the maintenance and enhancement phase. In formal testing there is the need to
generate appropriate tests to demonstrate that the software performs correctly. There are several approaches to testing, each of which can be supported by a test tool that assists in test data generation. The test data generation tools provide data sets for exercising the software in a particular way. Tools performing boundary testing, symbolic execution of loop constructs, checking of assertions from requirements standpoint, stress testing, and path testing should be provided.

Test harnesses to assist the user in exercising single or multiple modules in a simulated system environment should be easily fabricated. A general test harness that provides hooks to program-defined data can save effort and result in more thorough test cases. Tools should also evaluate testing thoroughness. Criteria that can be used are

1. Showing that the complete input space for small modules is exercised
2. Deriving the response function for a module to compare it to the sampled ideal function
3. Checking that all combinations of paths for a small module, and all path segments for a program, have been exercised
4. Demonstrating that all functional requirement paragraphs have test cases that have been used

For the purpose of demonstrating that optimal performance has been achieved, the places where most of the time is being spent should be determined. It has been shown that only 10% of most code needs to be optimized for maximum performance. Test tools should identify these areas for possible redesign.

SOFTWARE MAINTENANCE AND ENHANCEMENT

Pennington has estimated that software maintenance consumes about 60% to 85% of the total software life cycle costs. Since maintenance is the most expensive part of computer software, the environment must provide maximum support for the maintenance effort. The tools discussed in preceding sections will help generate high-quality software and documentation. These tools will reduce many of the current software maintenance problems. However, even the most reliable software can contain errors, and all software is eventually modified.

Software maintenance can be divided into two categories:

1. Correction of faulty programs
2. Modification or enhancement of processing capabilities

To aid in the correction of faulty programs, tools will be required for establishing a test environment to check reported errors. Diagnostic facilities will also be necessary to aid in tracking error symptoms to their source.

Corrected software must be retested to ensure that no new errors were introduced. The test conditions that demonstrated the error must be incorporated into the test plan. The changes must be documented, and change notices must be distributed. Major changes to a software product, whether corrections, modifications, or enhancements, will require additional support from software tools. Changes in the user’s software requirements can necessitate dramatic changes in the design of existing code. Tools to provide assistance for these worst case conditions will be required in any comprehensive software support environment.

The appropriate place to start making changes in computer software is with the statement of the user’s requirements and the specifications for the software product. These documents describe the user’s needs and how the software is to support those needs. It is, therefore, most important to keep this documentation up to date to direct the modification effort and to allow for future maintenance efforts. Requirements and specification documents in machine-readable form can be easily reviewed and edited to reflect any changes to be made.

Machine-readable software specifications can be expressed in forms that can be analyzed automatically for completeness and consistency. Therefore, changes to the specifications can be checked for conflicts and missing information. Identifying such errors at the earliest possible time minimizes the cost of their correction. Automated analysis of specifications can also produce a skeletal design that aids in evaluating the extent of design changes precipitated by new requirements.

The aids described for generating test data and providing the necessary testing environment (test harness) will find use in the retesting phases. It is rarely sufficient to test just the changed part of a program. A complete check of all processing capabilities is necessary to verify that no adverse side effects from any modifications affect other parts of the program. Modifications that are necessary to correct faulty software may indicate a shortcoming in previous testing efforts. The criteria for acceptable test thoroughness can be reevaluated and changed to improve testing reliability. Performance improvements can be measured and verified during retesting.

Documentation of the testing activity, including descriptions of all tests and a log of all tests conducted, will be assisted by the testing tools. An inventory of all tests will be maintained, as well as a history of the tests performed on all software. Test histories will be maintained for individual-unit tests, subsystem and system level integration tests, and final system checkout.

The final item of responsibility of software maintenance is to keep track of various versions of modified programs. The software maintenance environment will aid in documenting program versions and storing distribution information such as release dates and names of recipients. The documentation will include installation instructions, user manuals, and system descriptions that can be compiled from the project database.

MANAGEMENT TOOLS

Management activities can be broken down into five basic categories:

1. Planning—preparation of schedules, budgets, resource estimates and other factors relevant to the execution of a project
2. Staffing—assignment of personnel to organizational positions
3. Controlling—enforcement of directives and management decisions
4. Directing—providing direction to project personnel to support management objectives
5. Organizing—establishment of the project structure, positions, and lines of authority

These activities are pervasive in software development, being necessary during all phases of the software life cycle. Up-to-date information regarding the project status and the staff activities must be available for effective management. The environment can play a large role in providing the necessary information in a timely fashion and in a format suited to someone removed from the technical aspects of the project.

The common project database will play a large role in supplying the needed data to the manager. The environment can automatically and accurately record as complete a history of the project’s development as management needs. This information can be reported to the manager, who plays a relatively passive role in the building of the database, by a series of management tools.

For planning activities, one of the most important reports would be one indicating the degree of completeness of the project. At the beginning of the project, the manager could enter into the database a proposed schedule of events, including recognizable milestones. The schedule could be updated as the project progresses and unforeseen events occur; however, it would also present the means for the manager to note which project tasks are falling behind schedule and which are going as expected. Exception reports can indicate trouble spots requiring special attention.

The manager should also be able to see at any time how well the actual figures match the projected costs by entering budgetary information into the database. The system can automatically record the computer resources used to date, together with a record of personnel active on the project. From this information, status reports on expenditures and indications of possible trouble spots where projections are far off can be automatically generated for the manager.

Once planning information is entered, special tools can analyze the schedule of tasks to determine those critical paths whose successful timely completion is vital to the overall successful completion of the project. Interactive scheduling programs can assist the manager in developing a schedule that tries to optimize staff and material resources. When a project runs into trouble and requires rescheduling, such programs can suggest new schedules consistent with the revised information.

Another aspect of control is accountability. The environment enhances accountability because all activity within it is recorded. If a module is erased from a library, the environment will indicate who issued the erasure command. If a document is modified, the environment will be able to report who the editor was. With this automatic recording and reporting capability, programmers will also be discouraged from mischief since their identity will likely be known.

SUMMARY

Future Ada environments will support the entire life cycle of a system, from helping to budget personnel to helping with software maintenance. The code development tools presently being implemented will be only a small part of future Ada environments.

ACKNOWLEDGMENTS

Many present and past GRC personnel have contributed to the ideas contained in this paper. In particular, Carolyn Gannon, Nancy Brooks, Reg Meeson, and Art Pyster made substantial contributions.

REFERENCES
