Programming Support Environment Issues in the Byron Programming Environment

Matthew J. Larsen
Intermetrics, Inc.
733 Concord Avenue
Cambridge, MA 02138

abstract: This paper discusses issues which programming support environments need to address in order to successfully support software engineering. These concerns are divided into two categories. The first category, issues of how software development is supported by an environment, includes support of the full life cycle, methodology flexibility and support of software reusability. The second category contains issues of how environments should operate, such as tool reusability and integration, user friendliness, networking and use of a central data base. This discussion is followed by an examination of Byron, an Ada based programming support environment developed at Intermetrics, focusing on the solutions Byron offers to these problems, including the support provided for software reusability and the test and maintenance phases of the life cycle. The use of Byron in project development is described briefly, and the paper concludes with some suggestions for future Byron tools and user written tools.

1. Introduction

Over the past two decades, producers and consumers alike of software products have become increasingly concerned with what has become known as the "software crisis". As computer hardware has evolved to enable the processing of more and more data at faster rates, the range of practically solvable problems has grown. Yet our ability to manage the growing capabilities of computer hardware, as Djikstra [1] has stated, has lagged. In order to combat the software crisis such weapons as design methodologies and software support tools have come into existence. Collections of these tools have become known as programming support environments, and there has been a gradual realization that such environments can be valuable. Ivie [2] identifies several benefits of such systems, including commonality of documentation, development of standards and enhanced programmer mobility and retrainability.

* Byron is a trademark of Intermetrics, Inc.

There is much disagreement concerning exactly which tasks a programming environment should support. The DoD has issued Stoneman [3], a document specifying the requirements an Ada* programming environment must meet, but Stoneman focuses primarily on how the tools are to work in general, not on the needs to be fulfilled by the tools. In this paper we shall first examine issues which programming environments, particularly Ada environments, must address. This will be followed by an examination of Byron, an Ada based programming environment developed at Intermetrics, and how Byron deals with these issues. We shall then examine how Byron might be applied to a project.

2. Programming Support Environment Issues

There are two sets of issues relating to programming environments. The first set focuses on how the environment supports software engineering. Included here are full life cycle support, support of software reusability and methodology flexibility. The second set is concerned with how the environment operates internally, including issues of environment integration, flexibility and user friendliness.

2.1 Software Engineering Issues

The purpose of a programming environment is to support software engineering. There are four concerns which must be addressed in order to do this effectively. First, the full software life cycle must be supported. Second, the user must be able to move freely from one life cycle phase to another. Third, the environment must not restrict the choice of methodologies available to the user, and finally, the environment must actively support the reuse of software.

2.1.1 Full Life Cycle Support

Frequently, the software life cycle is modeled as a discrete, linear process [4]. Initially requirements are drawn up, then a software system is specified, designed, implemented, tested and finally maintained. Each phase is treated separately, and is completed before the next phase begins. If revisions must be made, the process loops. For example, implementation might halt while the design is reworked, and then the implementation would be modified. The result of each phase is a document describing the results of that phase (in implementation this is the actual code). Note that these documents are often of vital importance to the following phases. For example, it is impossible to test a software system without knowing what it is required to do. Similarly, a design document may give a valuable overview of a system to the maintenence team.

* Ada is a trademark of the Department of Defense

In the past, automated support existed only for the implementation phase. Even now, research on programming environments is mostly directed toward the code-compile-debug cycle [5]. However, errors are cheaper and easier to fix if they are discovered earlier in the lifecycle. Also, if the computer is only usable for implementation, programmers will tend to concentrate on that phase. So the need for good tools which assist with earlier life cycle phases is paramount. As Gutz et al. [6] report, an environment must provide support throughout the life cycle.

2.1.2 Mobility Between Life Cycle Phases

Although the view of the life cycle as a discreet process is useful, it is not wholly adequate. Often an error is discovered which requires adjustments in an earlier phase. Because of deadline pressures, the corrections are usually made only in the current phase, which then bears some relation to the previous phases, but is not a direct descendant. Thus, the resulting implementation is based on an underlying design which evolved separately from the design document. The differences are likely to be subtle and difficult to understand, but are almost certainly important. If, however, there is a simple way to update the results of a previous phase (in this case the design document), the results of the phases are more likely to remain consistent with each other.

The essential problem, therefore, is to keep the documentation for the earlier phases consistent with the current phase. Naturally the previous phases are reflected in the current phase, although the information may be implicit rather than explicit. For instance, in Ada code some portions of the design are readily visible in the specifications of packages and the decisions concerning the grouping of subprograms into packages. Since the packages also contain information unimportant to the design, what is needed is a tool to distill the design out of the code. But in order to do this, the entire design must be explicitly stated, as must any other information we might want to use in creating reports. One way of providing easy mobility between life cycle phases is to introduce a programming language which permits explicit statement of information concerning all phases.

2.1.3 Methodology Flexibility

There are many different software engineering methodologies, and new ones appear with frequency. Even such basic concepts as the life cycle are called into question [7] and revised regularly. It has become clear [8] that environments must be flexible enough to permit a variety of methodologies and the evolution of new methodologies, since different problems require different methods of solution. In order to provide this flexibility, environments must permit the expression of many different kinds of information and also the categorization of this information in many different ways. The environment must then provide access to this information, as we will see below. The importance of this flexibility cannot be overstated.

2.1.4 Software Reusability

The software industry has recognized the need to avoid continuously rewriting various pieces of software. One of the major goals of Ada is the proliferation of large libraries of reusable packages, in order to address this need. However, there is a very real danger, even in small environments, that one programmer might not know what other programmers have already done. Even if code is known to exist, it may be difficult to determine whether the package actually does what is necessary (and whether it has side effects), or whether it can be easily modified. If the only way to identify the functionality and effects of a package is to read the code, much of the advantage of reusing the code may be lost.

The suitability of a given package for a given task is best evaluated by examining the design of the package, if that information is accurate. Therefore we see that the design information should be explicitly stated, and extractable from the code. Furthermore, this information must be in a concise and standard format, so users will be able to quickly sift through the available packages to find what they need. It is important for the environment to support the act of finding software which could be reused.

2.2 Environment Operation Issues

Although the support of software engineering is the primary goal of programming environments, issues concerning the operation of the environment are also important. If the tools are too clumsy to use, the environment will not be useful. Osterweil [9] identifies five characteristics essential to programming environments: breadth of scope and applicability, user friendliness, reusability of components, integration, and use of a central data base. The first of these includes the issues we identified above as methodology flexibility and life cycle coverage. The rest we shall consider below.

User friendliness is a broad term, including many fairly obvious points. User interfaces should be consistent; help should be on-line and easily accessible; tools should perform obvious functions and be free from contradictory and confusing options. A less obvious aspect of this issue is that tools should not overlap in function, which will tend to confuse the users in choosing which tool is best suited to a specific task. Also, a user who needs to perform a specific task should be able to find the tool which does that task without intimately knowing all the tools.

In order to provide a flexible methodology as discussed above, the component toolset must itself be flexible. This toolset can then be the basis for new tools tailored to fit the project. Bergland and Gordon [10] comment that "if the tools come first, too often the design and development methods end up accommodating the tools instead of vice versa." This implies, among other things, that a facility for combining tools must exist. The power of this approach is well known from experiences with the Unix* programming

environment. It is, however, not well understood which tools should comprise the toolset. Presumably, the next few years of research will begin to identify the essential tools.

It is also important that the tools be well integrated, that is, they should work together to provide an abstraction which assists the user in working within a particular development methodology, and shields the user from the details of the environment. Thus the interaction of the tools should be controlled in order to avoid hidden side effects, yet reuse operations where possible. Since we have already acknowledged the fact that the user is expected to augment the environment with additional tools, this goal can only be partially achieved. However, the set of reusable elementary tools should certainly abide by these rules.

The idea of a central data base which contains all the information relevant to a project is one of the most widely accepted concepts concerning programming environments [11]. We identified a need earlier for a language which can be used to express all the information concerning a project. It is even more important that all this information be stored in one place. This can then be used to maintain various versions of a project, structure and retrieve information in manageable pieces and most importantly, maintain a single set of documents which describe the state of the project at any given moment.

3. The Byron Programming Support Environment

We will now review the Byron Programming Support Environment, and examine how it addresses the issues identified in the previous section. The three important aspects of the environment are how the data enters the environment, how it is stored, and what tools are available to access the stored information. The primary means of expressing information to be entered into the Byron environment is the Byron program development language (PDL). The PDL text is analyzed and stored in a structured data base (the program library). Once stored, the information is available to the various tools which comprise the Byron programming support environment.

3.1 The Byron PDL

The Byron programming support environment is centered around an Ada-based program development language (Byron/Ada PDL). Byron is compatible to Ada since any legal Ada program is also legal Byron, and vice versa. Byron provides a consistent way of entering information into the environment throughout the software life cycle, and thus smooths the transition from one phase to another.

* Unix is a trademark of Bell Laboratories

Byron constructs are included with the Ada code in the form of annotated comments. (see [12] and [13] for more detail). The Byron PDL is designed to augment Ada's design language abilities by formally and efficiently expressing information produced in the course of engineering a large software system which cannot be expressed in Ada. Ada has many features which assist and improve design; however, it has been recognized that there is information which is not required by or even expressible in modern programming languages, including Ada, but which is nevertheless important and valuable [14], [15], [16]. This information is mostly semantic in nature, concerning the use or purpose of data items or subprograms. Consider the following Ada subprogram specification

function CopyLinkedList (List : in ListPtr) returns ListPtr;

This is sufficient for compilation; however, in order to use the function there are details one needs to know, such as whether a physical copy of each list element is made, or merely a copy of the pointer to the list. Byron permits the methodical inclusion and retrieval of such information.

Information is expressed in Byron either as Ada code or as Byron annotations. Annotations are formed with the prefix "--|" followed by text. In general, the text of an annotation is associated with the Ada construct that precedes the annotation. An annotation may also contain a keyword which categorizes the information. This permits the user to tailor the Byron PDL to suit many different tasks. For example, the effect of the CopyLinkedList subprogram might be described with the effects keyword, e.g.

```
--|Effects: Creates an exact copy of the list passed in. A copy
--|of each element is made, so the copied list shares no elements
--|with the original.
```

The user may specify what keywords may be used and what Ada context they are to be expected in. This permits the user to define a specific development methodology, giving the user the methodology flexibility discussed above. For instance, a methodology might require that every use clause that is placed in code be followed by a Byron annotation justifying the presence of the use clause. A Byron keyword "Use_Justification" could be used to enforce this requirement.

One problem with methodologies is that it is sometimes difficult to get programmers to adhere to them. Byron attempts to alleviate this problem through the mechanism of "phase checking." The user specifies what development phase a given keyword should be used at (keywords may also be optional). Program source is then categorized within the program library according to what phase has been reached based on what keyword annotations are present. The analyzer will warn a user who indicates that code has reached a phase which it has not; tools may also be written to report what phase any portion of code is currently in. Thus, the "Use Justification" keyword described above could be required at implementation phase. Warning messages would

indicate the absence of Use_Justification annotations when code was analyzed with a phase of implementation.

3.2 The Byron Data Base

The Byron system provides a database called the Ada program library, which provides a central repository for all the information concerning a project. This information is primarily stored in the intermediate form known as DIANA, including both the regular Ada syntactic and semantic information, and the Byron PDL information. Tools may be written which access either kind of information, and may be independent of life cycle phase or not. The PDL code enters the program library through the Byron analyzer. This program is the front end of an Ada compiler, and provides full syntactic and semantic checking, as well as the checking specified for Byron annotations.

The program library permits Ada programs to be broken down into any number of separate catalogs containing compilation units, which are linked together to form the program library which comprises a program. Catalogs may be either read-only resource catalogs, which contain a specific release of a set of compilation units, or modifiable primary catalogs which generaly represent a new revision under construction. Configuration management is assisted by the use of different revisions of a resource catalog. Thus, two projects might be using different revisions of the same resource catalog, so that the project using the older revision could avoid recompilations or regressions in the newer revision.

3.3 The Byron Tools

Tools are an essential part of a programming support environment but it is the selection of tools and the relationship between them that characterizes the working details of a truly integrated system. As we saw in section two, there are many factors to be considered when examining a programming environment. The Byron tools have been designed with an eye toward flexibility and smooth dissemination of the information concerning systems under development.

As we noted before, a programming support environment must include tools to support each phase of the software life cycle, and must smooth the transition between phases. The Byron tools fall into two broad categories: first, tools which assist with methodology and the life cycle phases, and second, tools which assist with programming tasks without regard to a specific discipline or life cycle phase. Methodology and life cycle tools include an Ada based PDL, described earlier, configuration manager for source and documentation, design requirements traceability package, data dictionary system and more. Of the second type of tool, Byron provides a variety of technical programming tools for static analysis. These include an Ada compiler, linker, recompilation manager, global cross-referencer, source formater, program lister and others. Another way of categorizing Byron tools is by the form of data they operate on. Many of the tools, including the global cross-referencer, the data dictionary

generator and the generalized document generator, operate on the data available in the program library. Other tools, such as the pretty printer, statement profiler and the compile order generator, operate directly on Ada source code.

In an earlier section of this paper, we pointed out that an environment must permit user constructed tools. We have seen that the Byron PDL permits the user to store arbitrary information in the program library. Tools are also provided which the user may use to extract that information from the program library, as well as Ada syntactic and semantic information. The first of these tools is a generalized document generator, which creates documents based on user written specifications. These specifications are written using an interpreted language, BDOC, which permits the extraction of information from the program library and the output of that information in a formated form. The second tool is the program library access package (PLAP), a set of Ada subprograms which provide a window into the library. The user can extract information about his program, as it evolves, without being concerned with the internal structure of the library. Ada programs can be written which utilize the PLAP to query the program library and output individually tailored reports. Using this package, it is possible to construct complex tools such as a hierarchy chart drawer or a program interconnectivity matrix. These two tools provide the elementary tools spoken of in 2.2.

Also pinpointed earlier was the need to support reusability. It is not unusual for a programmer to duplicate the work of an associate simply because no one knows that the work has been done before. This problem is especially pronounced in a distributed computing environment. Even if a piece of code is available which does a similar task, it may be nearly as difficult to modify as to write from scratch, since the programmer must first understand how the existing code works. The userman tool provided with Byron can assist in relieving this problem. The document created by userman is a description of the purpose and use of package or subprogram, and is intended to be a document of the design of a package or subprogram following the design methodology suggested by Liskov [14]. Other documents supporting other design methodologies could be produced. One can then envision a design library stored on a computer to which programmers could refer when looking for a package to do a specific job. Another way to encourage reusability would be to create a user defined keyword "keywords." This keyword would be permitted on all library units, and the text following it would be a list of keywords describing the functionality the unit provides. A simple program could be written using the Plap which would extract the keyword list from each unit in the program library. Each element in the list would be compared to a string which the user of the program would supply, and if they matched, an overview of the unit would be printed. This would assist users in sifting through large librarys of software to find appropriate tools.

Documents of this nature also help support the software life cycle, by helping to show the design as it currently exists, rather than as it is intended to exist or as it used to exist. Byron also offers a tool to support the tracking of requirements, to ensure that the final result of the project does in fact fulfill the needs it was intended to. The userman and requirements tracking tools help Byron to support transitions from one phase to another, as does the fact that many of the tools are useful in multiple phases.

4. Project Use of the Byron Programming Environment

A comprehensive development system, such as Byron, is difficult to visualize at work. An operational view is necessary to appreciate the ability of such a large number of tools to function together usefully. The following scenario is presented as a brief illustration of how a hypothetical project might evolve using this system.

First of all, Byron provides methods and tools to assist management with organization, planning, tracking and reviewing of this project throughout its entire life cycle. Since the user is permitted to decide what information is to be stored in Byron annotations, valuable project information such as names of implementors and/or designers, project progress information, project statistics may be easily stored and accessed. Tools for computing the Halstead and McCabe complexity metrics are included, which assist software management in several ways, including estimating the number of outstanding bugs and the time needed to complete pieces of software.

The requirements phase of this project defines the problem to be solved, defines a system design to solve the problem, and allocates the requirements of that design to hardware and software. The design requirements traceability tool provides the facility to relate requirements to design elements and modules. This is especially useful in later phases where the impact of change may be quickly traced.

During specification and design work shifts from functionality to decomposition. Program architecture and data structures may be developed using Byron PDL. The result of this process will be a set of heavily annotated Ada specifications, from which the document generator can create design documents which may be reviewed. The template driven document generator can produce documents as complicated as MIL standard C5 specifications, as well as other specialized documents a user might need. Sufficient flexibility is available to support many different design methodologies, by the careful selection of Byron annotations.

The transition from design to implementation is a smooth one since an Ada based PDL was used as the program design language. It is a small step from Ada specifications to subprogram bodies which contain nothing more than a few lines of comments describing what that routine is to do, in fact, a tool could be written using the program library access package to automaticly generate simple

bodies on the basis of annotated specifications. As implementation begins in earnest, the user can take full advantage of the PDL aspects of Ada, and where Ada is deemed inappropriate, Byron annotations may be defined to fill the gap. A user defined Byron annotation "TBD" or "to-be-determined" might be used as a general purpose annotation to mark these comments, and permit their extraction and inclusion in documents. Two such documents might be a report on which modules are not yet fully implemented and what work needs to be done on them, or perhaps a design document of a more detailed nature than that produced by the userman tool. Coding and debugging are assisted by frequent reports such as cross-references and compilation listings. When implementation is complete enough, the Ada compiler will generate object code (also stored in the program library) which may be linked for testing. Implementation and testing are further assisted by a symbolic debugger and performance analysis tools.

Once the system begins to work, it must be carefully tested. Software which has not been adequately tested cannot be considered reliable. The Byron tools assist testing in several ways. First, the design requirements tracer shows which modules implement which requirements, helping to focus testing efforts. Second, when a module is designed it's purpose is well understood, and the test which should be applied are often more obvious than after implementation. The functionality which a module is intended to provide should be tested, not a specific implementation. Byron annotations provide a means for expressing this information at whatever point in the life cycle it can best be specified.

Software systems spend the majority of their life cycle in maintenance phase. The cost in terms of both time and money of repairs and enhancements can be greatly reduced by the availability of accurate documents which describe a system at various levels, from requirements down to detailed design. If an engineer must read code to understand a system, it may be a considerable amount of time before changes can be made to the system. Comments, when they exist, tend to be vague and incomplete. Byron provides a mechanism for specifying what structures should be commented and what type of information the comments should include. One major purpose of Byron is to provide a series of documents which describe the system, providing insight at several levels of complexity.

5. Conclusion

We have identified a number of concerns which programming environments must address, and seen how Byron addresses them. Of these issues, perhaps the most important are the full coverage of the life cycle, including transitions between phases, and flexibility, both with respect to the methodology supported and the tools available. It is expected that successful environments will support a wide coverage of the life cycle and permit great flexibility.

Byron provides a great deal of flexibility, both in what information is to be stored and what tools can access that information. Although there are limits to

how this information can be expressed, such as the Ada framework surrounding the information, these limitations serve to focus the user's attention on the purpose of the environment: to assist the creation of Ada programs. Thus, the environment supports primarily the act of producing programs, not the act of using that product. This is accomplished by helping the user organize the information which would otherwise be in some possibly out of date design document, or as a series of random comments, or perhaps not at all. The user may then use this organization to extract only as much of the information as is necessary for a specific tool to do its work, or to answer specific questions concerning the software system.

Acknowledgements

This paper was reviewed and commented on by Michael Gordon, David Ortmeyer and Haynes Turkle. Their suggestions and support are greatly appreciated.

References

[1] Dijkstra, E.W. "The Humble Programmer" (Turing Award Lecture). *Communications of the ACM*. vol. 15, No. 10 (October 1972) : 859-866.

[2] Ivie, E.L. "The Programmer's Workbench - A Machine for Software Development." *Communications of the ACM*. Vol. 20 No. 10 (October, 1977) : 746-753.

[3] Department of Defence. *Requirements for Ada Programming Support Environments, 'STONEMAN'*. (Washington : US Department of Defence, 1980).

[4] Yourdon, E. and Constantien, L.L. *Structured Design*. (New York : Yourdon, Inc., 1978) : 3-9.

[5] Henderson, P., ed. *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*. New York: The Association for Computing Machinery, Inc., 1984.

[6] Gutz, S., Wasserman, A.I., and Spier, M.J. "Personal Development Systems for the Professional Programmer." *IEEE Computer*. Vol. 14 No. 4, (April 1981) : 45-53.

[7] McCracken, D.D., and Jackson, M.A. "Life-Cycle Concept Considered Harmful." *Software Engineering Notes*. Vol. 7 No. 2, (April 1982) : 29-32.

[8] Hoffnagle, G.F., and Beregi, W.E. "Automating the Software Development

Process." *IBM Systems Journal.* (vol. 24, No. 2, 1985) : 102-120.

[9] Osterweil, L. "Software Development Environment Research: Directions for the Next Five Years." *IEEE Computer.* Vol. 14 No. 4, (April 1981) : 35-43.

[10] Bergland, G.D. and Gordon, R.D. "Software Development Environments." *Tutorial - Software Design Stratagies.* 2nd ed. New York : IEEE, 1981 : 347-353.

[11] Wasserman, A.I. "Automated Development Environments." *IEEE Computer.* Vol. 14 No. 4, (April 1981) : 7-10.

[12] Larsen, M.J., Ortmeyer, D.O., Turkle, H., and Gordon, M. "The Byron1100 Program Support Environment." *Proceedings of Use, Inc. Fall Conference, vol. 1.* Anaheim, CA, (November, 1985) : 119-134.

[13] *Byron Program Development Language and Document Generator.* Cambridge : Intermetrics Inc., 1985.

[14] Liskov, B. *Modular Program Construction Using Abstartions.* MIT Computation Structures Group Memo 184. September 1979.

[15] von Henke, F.W., Luckham, D., Krieg-Brueckner, B., and Owe, O. "Semantic Specification of Ada Packages." *Ada in Use: Proceedings of the Ada International Conference.* Cambridge: Cambridge University Press, 1985 : 185-196.

[16] Luckham, D.C., von Henke, F.W., Krieg-Brueckner, B., Owe, O. *Anna: A Language for Annotating Ada Programs.* Computer Systems Laboratory Technical Report 84-261. Stanford University. July, 1984.