ADA FROM THE VIEWPOINT OF SOFTWARE ADAPTABILITY AND MAINTAINABILITY

By Paul E. McMahon
Link Flight Simulation Division of The Singer Co.
Binghamton, New York

Summary

A major goal of the Ada language is to decrease the cost impact of software modifications resulting from requirements changes. This paper shows how Ada has the potential to achieve that goal through examples of modifications to the F-16 Trainer Flight Simulator. Ada features not available in Fortran are discussed as they would be applied to the F-16 TFS examples with the potential modification in mind. From the Fortran examples provided, the rationale for many of Ada's features can be seen. The point is made that Fortran is a programming language, while Ada is a design, documentation, and implementation system. Ada addresses all stages of the software development life cycle and is not limited to the coding stage. Many of the tools which Ada provides must be initially applied during design activities if Ada's full benefits are to be realized in reducing the cost impact of software modifications.

Introduction

The field of computer programming is a young discipline with many complex applications growing faster than the required development methodologies. As a result, today it is more costly to maintain or modify software than it is to develop it. Barry Boehm, in his article entitled "The High Cost of Software," estimated that by 1985 software costs would be 90% of a project, with more than 50% of this being required for maintenance[1].

This paper shows how Ada has the potential to decrease the cost impact of this maintenance activity through examples of modifications to the F-16 Trainer Flight Simulator (TFS). The examples are first presented within the framework of the current F-16 TFS Fortran programming environment and then analyzed in terms of how the modification problems encountered can be avoided with Ada.

Definitions

Software adaption is defined as the activity required to preserve the specifications of a software component within a changing environment. The change to the environment may be a hardware change, such as a rehost to a new computer, or a software change, such as the installation of a new compiler. Software changes affecting the specification will be referred to as a software modification.

Software maintenance is not as easily defined as software adaption and software modification. Software maintenance activity is frequently required during both the adaption and modification process, but is usually not a planned activity. As stated by Glass and Noiseux[2], "Maintenance is the enigma of software: Enormous amounts of dollars are spent on it. Little research or management attention is given to it. And, in fact, it is not even a well-defined concept!" Glass and Noiseux define maintenance as "the act of taking a software product that has already been delivered to a customer and is in use by him, and keeping it functioning in a satisfactory way."

In this paper I plan to show that software maintenance difficulties are partially a result of programming languages which have not addressed the total life cycle of the software product. Fortran, in particular, does not provide the tools required to effectively support the modification phase. Frequently, this results in unplanned maintenance activity. In contrast, Ada's features are oriented to the potential modification, providing the mechanisms to develop software in a more rigorous fashion with the modification process in mind.

The F-16 TFS Environment

The computer system hosting the F-16 TFS consists of a 16-bit virtual machine, the Nord 10, controlling four 32-bit slaves, the Nord 50's. The Nord 10, produced by Norsk Data in Oslo, Norway, is controlled by an interactive multitasking operating system known as Sintran. All system I/O is through this computer. The Nord 50's are used as single task computers which offload the Nord 10 from CPU-intensive functions. A shared memory system provides communication. All software development is accomplished in the Nord 10, while the Nord 50's are dedicated to the simulation task. Over 90 percent of the F-16 software is written in Fortran, with the executive software and I/O handlers written in assembler. The support software had originally been rehosted from another hardware environment.

The examples discussed in this paper are taken from two block updates to the F-16 TFS. The Block 10/15 update included the addition of a fifth Nord 50, the addition of a block of shared memory, the incorporation of automated tools, and the installation of the latest vendor software. The Block 25 modification included a software rehost to the vendor's state-of-the-art equipment: a Nord 100/500 system. The ND100 replaced the ND10 and the ND500 replaced the ND50.

Computer Vendor Software

Both block updates required installation of the vendor's latest software products. This included compilers, loaders, and operating system. This activity caused unanticipated memory limitation problems to surface. The 16-bit Nord 10 has a limited logical addressing range of 128KB. On the Block 10/15 update, the vendor operating system enhancements required memory which was previously available for the real-time simulation device drivers. As a result, software which was not planned to be changed required adaption.

Among the lessons learned during the software rehost effort on the Block 25 update was an understanding of what the phrase "upward compatible" does and does not mean. Although the Nord 100 is viewed as 100% upward compatible from the Nord 10, extensive software adaption was required. This was due to the fact that "upward compatible" means that the CPU instructions and Operating System services are still supported. It does not refer to space and time requirements. For example, during the Block 25 update the F-16 TFS software development environment was rehosted on the 32-bit Nord 500. This effort was required because of the increased overhead in code generated by the new ANSI standard compiler. The Nord 100, with its limited logical addressing range, could not support this increase for the development software.

Recycle Automation

During the 10/15 update, software to simplify and standardize the user interface to compilation and loading was developed. This enhancement was desired because of the large number of user names, passwords, compilers, assemblers, loaders, preprocessors, and postprocessors required by the system. To meet this need, an interactive program, known as Cycle, was developed. Based on user requests, Cycle determines task requirements from a single database and appends to batch the appropriate job control stream. The design goal was to modify only the user interface, with Cycle generating automatically what the users previously input manually.

Each compilation or assembly is carried out in one of two library modes, known as LIB and DEF. At load time, LIB entries are only loaded if previously referenced, while DEF entries are always loaded. The loader also provides a REF command to allow "forced" loading of LIB entries not previously referenced. During the Block 10/15 update all software was recompiled and linked through Cycle in the LIB mode to ensure that only referenced software was loaded in memory.

During test it was discovered that many of the component system load files were dependent upon the compilation mode. Knowing the inner workings of the loader, programmers had tailored their load streams, with commands such as REF, and compiled some components in the LIB mode and other components in the DEF mode. On investigation, to determine if this process could be automated into Cycle, no simple rule could be determined. The lack of a rigorous standard for compilation and loading had resulted in different levels of programmer usage of the compiler and loader capabilities.

Although earlier recognition of this particular fact would not have changed the design approach, the overall cost of Cycle could have been reduced. Discovering the problem late in test proved more costly. This is due to the indirect manner in which problems are found at this time. A load stream failure due to the LIB/DEF condition may result in a "soft" error not detectable by the loader. The control language set up by the programmer may be acceptable to the loader, but no longer accomplish the original intent. This type of problem is frequently the most costly of all to track. The symptoms during execution may be subtle. Since a program such as Cycle touches all software, anticipation of where such problems may occur is difficult.

If during the design phase all load and recycle files were reviewed closely, the potential problem could have been anticipated up front and thus resolved before system test. More ideally, if the users had not been exposed to the details of Loader/Library facilities, the job streams could not have been individually tailored.

Auto Test Guide

The Auto Test Guide (ATG) software package was developed to automate software testing by controlling the inputs to the simulation task while monitoring and reporting the outputs. The system was designed to run in real time with the flexibility to allow the user to run predefined tests with any subset of the real-time modules enabled. The implementation was accomplished by porting the control software from a functional ATG system developed to test single modules in a batch or background mode. Following a period required to integrate the software into the real-time environment, the system was successfully tested and released to the users.

Shortly after release, various system failures were reported. One user reported that the first test case was ignored, while another reported that end conditions were randomly skipped. Another common report included system crashes resulting in a nonrecoverable condition; rebooting from disk failed to clear the problem. Initial investigations proved fruitless. The reports did not point to a particular area of code and the failures could not be reproduced. Since the users were reporting no difficulty in causing the problem, time was scheduled with both user and designer present to trap the problem. During the first few sessions the system functioned as advertised and the effort was abandoned. However, the problem soon returned. After a number of such troubleshooting efforts, enough information was gathered to solve the problem.

What would cause a reliable system to become so unpredictable in a modified environment? The clue to the solution was in the reports of a system crash and the change of the environment from background to real-time. All reports were eventually tied to an initial system crash. It was found that test data was driving the simulation software outside design tolerances, causing an exception to halt the system. After rebooting, the ATG system would be degraded and operate unpredictably.

In order to provide real-time common memory communication between the ATG system and the real-time environment, the ATG system had to be linked under the real-time loader environment in the Nord 10. Software running under the real-time loader can be thought of as one level closer to the Operating System and the virtual segment swapping environment. In the real-time segment environment, following a system reboot, each segment contains the information from the last time it was swapped out to disk. The Operating System contains no data on its own real-time segments, and is therefore not adversely affected by this swapping process. This was not the case for the ATG system.

The ATG system was found to contain various first-pass and "in progress" flags. Since part of the system's data was within RT Common which was reinitialized on system reboot, after a crash, the system would contain inconsistent data and perform erratically.

Symbol Dictionary

Example 1 - The F-16 maintains a Symbol Dictionary System to control the allocation of common data. Through the use of a source scan system the programmer is relieved from the task of generating his own Fortran type and common statements. The system was rehosted from a 32-bit environment to support both the Nord 10 and Nord 50. Because the Nord 50 does not admit to halfwords (16-bit), the 16/32-bit environment required special processing. In order to support both processors, common numbers were logically associated with memory type. This association was implemented through the use of hard-coded common checks. This programming technique caused the software to fail on reconfiguration of the commons due to the additional CPU.

Example 2 - During the 10/15 update, an unanticipated failure occurred within the symbol dictionary system. Following weeks of investigation the

problem was found to be twofold. By increasing the number of symbols for the block update, a maximum had been reached which was 4K below the documented maximum. This was found to be caused by a "hard-coded" check for 12K symbols. The system was designed and documented to support 16K symbols. The second problem centered upon the existence of an undocumented scratch file which needed to be increased in size.

Although the problems encountered in the symbol dictionary system were trivial to solve once found, the time required to track the problems was costly. The complexity of the system requires a lengthy learning curve preceding any level of troubleshooting. Barry Boehm states with respect to Air Force avionics software, "it costs something like $75 per instruction to develop the software, but the maintenance of the software has cost up to $4000 per instruction."[1] When it takes weeks to solve a problem, and the final solution affects only two instructions, the nature of the maintenance problem is better understood.

Debug Enhancements

During the F-16 TFS prototype effort it was realized that more advanced tools could significantly reduce the troubleshooting effort. Since the debug package required modification to support a fifth Nord 50 for Block 10/15, the enhancements were also planned. During this modification, the Nord 10 logical addressing limit of 128KB per segment was reached with the Debug System. This caused a need to restructure Debug into a multiple segment system. Following this activity, it was discovered that the Daily Readiness system was no longer functioning correctly. No changes had been made to this system. Upon investigation it was learned that the Daily Readiness system was tied to the segment structure of Debug. Thus the Debug restructure had in turn broken the Daily Readiness package. What initially appeared as a straightforward change was rippling through the system.

Simulator Initial Conditions

The Block 10/15 update did not specify any modifications to the Initial Conditions software. This system consists of a control module in each of the Nord 50's responsible for special processing required to properly initialize its CPU to the required conditions. To control the sequencing of the initialization process, a control variable, UCINT, in shared memory counts down from 50, with the values reflecting the current stage of initialization in process. For example, when UCINT equals 44, the control program in CPU 1, U101, executes specialized code to initialize the Nav/Comm Receivers to the required conditions.

By design, the initial conditions system contains interfaces to virtually every real-time software component. Because the system was highly visible and frequently utilized, many software anomalies unrelated to initialization required troubleshooting through the initialization process before the faulty simulation software could be identified. As a result, debugging of many simulation software problems required not only the simulation programmer, but also the training system initial conditions expert and a two-stage process.

The Role of a Computer Language

A computer language may be viewed as a tool to aid both the man-computer and man-man communication process. Fortran provided a significant communication improvement over assembly languages by allowing programmers to express their intent in terms more closely resembling the design problem. Statements such as IF (BRAKE .AND. WOW) THEN AIRSPEED = 0 are certainly more understandable than a sequence of instructions indicating various primitive arithmetic and logical operations on registers.

Fortran represented an initial step toward simplifying the communication process by recognizing that it was not necessary for the programmer to understand all of the details concerning how the CPU was solving the problem. This concept, known as abstraction, will be discussed further in the sections ahead.

Since the development of Fortran, numerous other high-level languages have addressed the communication problem by providing a rich assortment of powerful programming constructs, most of which are paid for in run-time efficiency. As a result, while most software designers have recognized the value of abstraction mechanisms in terms of communication and reusability of software, it has always been feared that the price would be too high, especially in real-time environments. As a result, while we no longer concern ourselves with the condition of CPU registers, the potential value of abstraction in reducing the cost of software maintenance has not been fully realized.

Ada/Fortran Comparison

The Fortran programming language features encompass four categories: Control, Execution, Input/Output, and Data. The following paragraphs compare Ada and Fortran within each of these categories, and are followed by an examination of Ada's extensions in support of maintainability.

Fortran program control constructs include an unconditional, computed, and assigned GOTO, an IF-THEN-ELSEIF-ELSE-ENDIF construct, and a DO-ENDO construct. While Ada does not support assigned GOTO's and does not recommend the use of unconditional GOTO's, the control constructs otherwise contain essentially the same capabilities, with the IF-THEN construct being identical and the Ada case statement serving the same purpose as Fortran's computed GOTO. In terms of execution, the formulation of arithmetic and logical operations and subsequent assignment are once again almost identical between the two languages, with minor syntax and lexical element differences such as the requirement in Ada for a colon preceding the equal sign in assignment statements. Fortran's Subroutine Call and Function Statement are similar to Ada's Procedure Call and Function capability.

While Fortran provides Read, Write, Open, Close, and Format statements to process I/O, Ada provides no comparable facilities. In Ada, I/O is not an issue because it is not viewed as part of the language. Ada views I/O as an application to be implemented by the user.

The final category, Data, falls somewhere between the well-defined control and execution category and I/O. While Ada does provide predefined data types similar to Fortran's Real and Integer types, to a large extent data typing is also seen as an application problem and left to the user. In order to better understand how such features as I/O and data types can be treated in this manner, we must investigate Ada's extensions to the four categories of the Fortran environment.

Introduction to Ada Extensions

As stated earlier, a computer language is a tool to facilitate communication. While the immediate need during prototype software development is man-computer communication, and languages such as Fortran admirably support this need, it is the man-man variety which ultimately drives the software life-cycle cost. As seen with the symbol dictionary problem on the F-16, weeks were spent understanding how the program functioned, while the ensuing code change was trivial.

While it has long been realized that sound software engineering principles represent a solution, a practical method to ensure that rigorous methodologies are employed has not been available. Languages which have attempted to solve this issue by providing a richer collection of constructs have failed to lower the cost of software maintenance. High-powered language constructs ease the communication with the computer far more than the communication with a new programmer who is unfamiliar with the requirements and espoused methodologies.

Many have charged that Ada is doomed to failure because it is a complex language which is too difficult to learn. As seen in the previous section, the basic control and execution constructs of Ada are straightforward. Strictly viewed as a programming tool for man-computer communication, Ada is no more difficult to learn than Fortran. What is viewed as complex within the language are the tools which Ada provides to address the difficult, costly problems we have seen in the later stages of the software life cycle: the man-man communication problem.

Loose Coupling

The Loosely Coupled Concept - One of the major problems in maintaining a complex system is the potential ripple effect of any change. While the Fortran common memory system provides the environment, it is not the cause of the ripple effect. The ripple effect is caused by a lack of localization. We define "coupling" as "a measure of the strength of interconnection" of modules, and "cohesion" as "how tightly bound or related its internal elements are to one another."[3] A maintainable design requires a "strongly cohesive" and "loosely coupled" system. The ripple effect described in the Debug Enhancement example was caused by a strongly coupled system.

Development Methodology Effects on Loose Coupling - The terms "top-down" and "decomposition" are known to software developers as sound software engineering practices. Yet, it is not uncommon for products of such sound development methodologies to become costly to maintain during the modification phase. This situation was seen in the example on the F-16 TFS with the Debug and Initial Conditions systems. A partial explanation may rest within our application of this methodology.

During the initial stage of product development, functional software requirements are determined. These requirements are then allocated to computer program components, thereby initiating the top-down decomposition process. Traditionally, the hierarchical tree thus formed has come to represent not only a functional decomposition, but a physical one as well. Unfortunately, the optimum decomposition in support of the modification phase frequently differs from this functional representation.

Our experience on the F-16 TFS has shown the need to minimize interfaces in order to localize and simplify change impact and maintenance activity. In the example of the Initial Conditions (IC) system, by providing all logic within one physical component, any anomaly occurring during an IC required a training system IC programmer to wade through extensive unrelated initialization code to determine the area of concern, followed by an analysis of the state of the real-time simulation component exhibiting the anomaly. The latter stage required a second programmer knowledgeable with the simulation component software.

An alternative design approach, in support of the potential modification, would be a simplified IC control program with one function in life: accept initialization requests from the instructor and command the simulation components to perform their own initialization. Special initialization logic would be localized within the associated physical simulation software. When an anomaly occurs or a modification is required, the software impact will be localized, minimizing change activity and personnel required for technical investigations.

Loose Coupling Through the Ada Package - In the previous section we looked at the role of loose coupling in attaining a maintainable software system. This discussion was independent of the implementation language and, indeed, loosely coupled products can be achieved in any language, including Fortran and Assembly. However, Fortran and Assembly do not provide features to assist or monitor progress towards the attainment of this characteristic. In fact, in most software environments, including Fortran, the strength of interconnection of a system is largely determined prior to the use of any language features. Ada, on the other hand, is a design language with tools addressing the early development stages.

Through packages, Ada provides the mechanism to strongly support the attainment of a loosely coupled design and final product. In general, packages are used to group logically related entities, such as data objects, or subprograms. However, its major value during the design phase is in its structure, which consists of two distinct parts: a package specification and a package body. Through the package specification, a complete system interface may be defined and compiled separately from the user of the interface, as well as separately from details required to implement the interface. The language will also enforce the interface by not allowing accessing units to compile correctly if the specification is not maintained.

Example - Figure 1 presents an example of how Ada packages can be used during design to control ripple. Data is segmented into a logical hierarchy based on visibility requirements only. Each block represents an Ada data package. As you move up in the tree, data visibility and potential ripple impact increase. To achieve a loosely coupled final product, the design goal is to minimize the size of higher-level packages. By using such a system, design approaches could be given a tangible maintainability measure based on impact to the tree. Alternate design approaches could be compared in the light of this measure. This would encourage designs which minimize impact to higher level packages and could be used during the early design review process to assist the evaluation of alternatives.

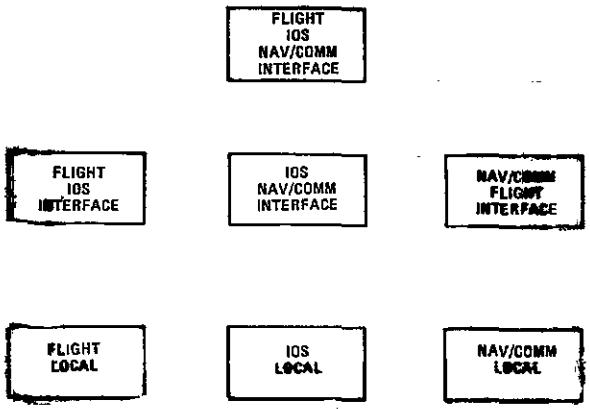

Figure 1 LOGICAL HIERARCHY OF DATA PACKAGES

The package specification is a feature of Ada which supports verification and enforcement of interface designs. It is a tool which focuses on the early precode software development stage and is not meant to encourage premature coding of design details. Used properly, interface package specifications can provide the visibility of system interconnections needed to control potential impact of change activity.

Abstraction

To assist in reducing software cost, one of Ada's goals is the development of reusable and easily modified software components. Easily modified components can be achieved through a separation of details and functional requirements. This compartmentalization of details serves two purposes. First, understandability is enhanced when top-level functions are easily determined without a need to study lower-level details. Second, by localizing the details, the dependency of other software on these details can be minimized. The software can be more readily modified without unforeseen impact. Ada refers to this concept as data hiding, or encapsulation.

The term abstraction refers to "the process by which we distinguish the functional characteristics of a facility from the implementation of that facility."[4] Various levels of abstractions occur naturally within the software development process. For example, the symbol dictionary system on the F-16 may be viewed as having one high-level functional characteristic: to process symbols. *This characteristic* may be decomposed into the three functions of adding symbols, deleting symbols, and reporting status. This refinement process continues until the functions identified are expressible within the constructs of the programming language.

This iterative process of moving from the abstract problem domain to the details of implementation represents an essential ingredient in the software development process. Unfortunately, in many development environments this process is not captured within the final product. The abstraction process frequently occurs only in the mind of the programmer. Traditionally, the computer language represents the final phase of translating the lowest-level details into terms understandable to the underlying machine. As a result, the code provides only the implementation details of the software development process.

During the modification process, visibility of all development stages is necessary. In the example of the symbol dictionary on the F-16, the top-level functions were not clearly visible. The code contained a flat structure and did not reflect the abstraction process which led to its development. As a result, it took weeks to determine which detail, among many, was relevant to the current problem.

To simplify and minimize potential impact, the modification programmer needs to understand as much as possible concerning the original development process. Mechanisms which capture the abstraction process within the final product can ease this process.

Ada Support For Abstraction - The concept of abstraction is supported in Ada through the Package, Generic, and Data Typing mechanisms. The Package mechanism in Ada supports abstraction in two ways. First, by providing a capability to separate the Package Specification from the Package Body, the goal of separating functional requirements from implementation details is supported. The modification programmer can view a package specification quickly and does not need to study the implementation details to understand the effect of the package components. Second, through the Private and Limited Private parts of a specification, the logical interface to a package may be made visible, while the physical interface details are hidden. The Private part of a specification defines data which can be operated on by the user of a package, but does not allow the package user to see the details of how the data is constructed. The user cannot tailor application software to the structure of the data. This feature provides an enforcement tool to minimize the ripple effect as seen in the example of the Debug Enhancements. The Limited Private mechanism is similar to the Private, but also limits the operations on the data available to the user.

Data Abstractions In Ada - Capturing the abstraction process in the development of data structures is also possible in Ada. Once again, using the example of the F-16 TFS Symbol Dictionary, the system must operate on only one type of data, referred to as Symbols. The functions identified in the previous section, Add, Delete, and Report, operate on Symbols. Further analysis during the development process reveals a need to refine the type Symbol into two categories: 16-bit and 32-bit Symbols. This process has identified two valuable pieces of information to the modification programmer -- the abstract notion of a Symbol and the details of implementation requiring two different computer word sizes. In the case of the F-16, since Fortran does not provide mechanisms to support data abstraction, the code provided the details of implementation only, while the basic notion of a Symbol was lost.

As mentioned earlier, many high-order languages have addressed this problem by providing a richer assortment of data types. However, this approach leaves the programmer with the more difficult decision of choosing which data type contains the characteristics which most closely resemble the real-world problem domain. This approach provides no insight into the rationale for the programmer's translation from problem domain to solution domain.

Ada supports the capture of this abstraction process by providing a minimal collection of predefined types and a powerful collection of tools to allow the software designer to rigorously define his own data types to match the characteristics of the real world. In our example of the F-16 Symbol

Dictionary, suppose one of the requirements included an interactive program to store or display the initial value of six character symbolic variables. Furthermore, suppose the number of symbols from each of the Nord 10 and Nord 50 and the total number of symbols required to be supported was not specified. In Ada we could define our own type called Symbol_Name_Value and declare it in a Package as Limited Private. We could then define two operations on data of type Symbol_Name_Value: ADD and DISPLAY. Since we have defined type Symbol Name_Value to be Limited Private, Ada will not allow any other operations on this data type and will hide the structure of the data from the user of the package.

In the real world, since the structure of certain types of data may vary, it is desirable to have a mechanism to describe this variance within the data type. Ada provides the tool to accomplish this conditional data structure through the Discriminant and Variant components within Record types. The structure of the record may be varied (Variant part) based on the value of a parameter known as the Discriminant. Through the Variant Record capability, we can implement the details of the type Symbol_Name_Value in such a way that variables of this type would automatically take on desired characteristics as a function of a discriminant identifying the Nord 10 or Nord 50 computer. Furthermore, by use of Ada's Access type mechanism, the maximum number of symbols need not be constrained by the code. Access types allow for dynamic allocation of objects during run-time.

Although run-time overhead is required to allocate space for objects created dynamically, all constraints on the data structure must be established at allocation time. In simple terms, this means that Ada supports the need for data structures which vary under run-time conditions and grow with new requirements without a need to reprogram the software. Through the Package, Limited Private, Variant Record, and Access type features of Ada, the abstraction process, which previously existed only in the mind of the programmer, can be reflected in Ada code. See Figure 2 for an example of these features.

Generics in Ada - Another feature of Ada which addresses separation of the functional requirements from implementation is the Generic mechanism. The Generic is a facility within Ada which allows a programmer to implement an algorithm independent of the data on which the algorithm will be applied. Generics may be thought of as programming templates. The data to be operated on is filled in by a process known as instantiation, which creates a specific instance of the template for each data type needed. This feature of Ada effectively allows the same software to operate on different types of data without the software model being tailored to the specifics of that data.

Run-Time Expense

We have discussed the role of a computer language and drawn a comparison between Fortran and Ada language features. The four categories of Control, Execution, Input/Output, and Data were identified in the Fortran language. In Ada a new category encompassing Packages, Data Typing, and Generics is needed. This category will be referred to as Development Tools. Since the Fortran concepts of Input/Output and Data are developed in Ada through the use of our tools, these categories are no longer required. Furthermore, as the Control and Execution categories are similar, in comparison to Fortran, the run-time expense of Ada is dependent on the cost of the Development Tools category.

Packages and Generics are tools to assist in the abstraction process. These tools encourage sound development techniques and provide a vehicle to help in describing how the final product was developed. These tools are directed at the man-man communication problem discussed earlier. Packages and Generics allow us to capture more of the development process within the source code without the *expense of overhead at the object code level. They* are purely tools to assist in the development and documentation of software.

Although a major goal of the Ada Data Typing mechanism is also to ease the man-man communication process, there exists a potential for run-time expense. Three characteristics of data objects must be considered: creation, representation, and verification. Data objects are created through a process referred to as elaboration, which is based on extensive visibility and scope rules within the Ada language. Depending on how the software is structured, elaboration may cause run-time overhead. However, the extent of elaboration is controllable through the scope rules. Data defined within the scope of the main program will remain in existence and will not require the overhead of periodic *elaboration.*

One of the more costly features of other high-level languages is run-time determination of data representation. While Ada does provide facilities for dynamic allocation of data through access types and provides for declaring variable-sized objects of the same type, once an object has been declared and elaborated, its internal representation in the computer is fixed. In other words, there is no run-time overhead required in determining how the data should be stored.

The last category, verification, refers to run-time checks such as array bounds and limits placed on data during its type definition. Run-time overhead for these features is expected to be in the range of 10% over comparable Fortran code. While *it is not advocated by the language, a Pragma exists* to disable run-time checks, eliminating this overhead.

Operating System Interface

In order to achieve a loosely coupled final product, it is equally important to strive for clear, simple interfaces into the operating environment, as well as within the components of the application software. Ada's package specification can help in achieving this goal.

Example - The final example from the F-16 to be discussed in this paper is the vendor Operating System. The real-time device drivers encompass some of the most complex, time-critical software on the F-16 TFS. Due to response time requirements and *the architecture of the Nord 10, this software was* required to reside in the top 128KB of memory with direct interfaces into Sintran, the vendor Operating System. This requirement posed a significant potential impact to the device drivers with each new release of Sintran.

In order to clearly establish and control this interface, all entry points into Sintran and references to Sintran symbols were maintained in a single interface control file. To minimize poten-

```
with Tools,
     Text_io;
use Tools, Text_io;
procedure Main is

-- This Procedure can use Package Tools without knowing
-- details of data structures or logic of the procedures
-- within the package.

    type Response_Type is (ADD, DISPLAY, DONE);
    Response      : Response_Type;
    Previous_Link : Link_Type;
    package enumera_io is new Text_io.enumeration_io (Response_Type);
    use enumera_io;
begin
    loop
        Put_line (" ADD, DISPLAY, or DONE ? ");
        Get (Response);
        case Response is
            when ADD     =>  ADD (Previous_Link);
            when DISPLAY =>  DISPLAY (Previous_Link);
            when DONE    =>  exit;
        end case;
    end loop;
end Main;
```

```
package Tools is

--Enumeration Types Support Abstraction
    type Computer_Type is (Nord_10, Nord_50);
    Computer : Computer_Type;

--Limited Private Type hides implementation details of data structure
    type Link_Type is limited private;

--Only Procedures ADD and DISPLAY may be used on Types Link_Type
    procedure ADD     (Link : in out Link_Type);
    procedure DISPLAY (Link : Link_Type);

private

    type Symbol_Name_Value (Computer : Computer_Type);

-- Access Types allow dynamic allocation of objects and thus
-- support abstraction in design.  There is no need to embed
-- Max numbers, such as file size, within the code.  This results
-- in a more maintainable and product.
    type Link_Type is access Symbol_Name_Value;

-- By defining our own subtype, the values that objects of
-- a given type accept can be controlled.  Objects of Type
-- Nord_10_Type are limited to the usual 16 bit range.
    subtype Nord_10_Type is Integer range -32768 .. 32767;

-- The discriminated record type allows the structure of the record
-- to be varied based on the value of the discriminant.
    type Symbol_Name_Value (Computer : Computer_Type) is
        record
            Symbol_Name   : String (1 .. 6);
            Previous_Link : Link_Type;
            Computer      : Computer_Type;
            case Computer is
                when Nord_10 =>
                    Initial_Value_Nord_10 : Nord_10_Type;
                when Nord_50 =>
                    Initial_Value_Nord_50 : Integer;
            end case;
        end record;
end Tools;
```

Figure 2 SAMPLE OF ADA FEATURES

```
with text_io;
use text_io;
package body Tools is
    package My_Integer is new text_io.integer_io (Integer);
    package My_enumera is new text_io.enumeration_io (Computer_Type);
    use My_Integer, My_enumera;

--Through the separation of the Package Body and Specification
--the details of the Procedures ADD and DISPLAY are hidden
    procedure ADD (Link : in out Link_Type) is

        Symbol   : String (1 .. 6);
        New_Link : Link_Type;



    begin

        put_line (" Nord_10 or Nord_50 ? ");
        get (Computer);
        put_line (" Input six character symbol ");
        get (Symbol);
-- Create a record for the propar computer
        New_Link := new Symbol_Name_Value (Computer);
-- Store symbol name
        New_Link.Symbol_Name := Symbol;
-- Store initial value
        put_line (" Input Initial Value ");
        if Computer = Nord_10 then
            get (New_Link.Initial_Value_Nord_10);
        else
            get (New_Link.Initial_Value_Nord_50);
        end if;
-- Store Computer Type in Record
        New_Link.Computer := Computer;
-- Link new record in the list
        New_Link.Previous_Link := Link;
-- Update last link
        Link := New_Link;
    end ADD;

    procedure DISPLAY (Link : Link_Type) is
        Symbol      : String (1 .. 6);
        Search_Link : Link_Type := Link;
    begin
        put_line (" Input six character symbol ");
        get (Symbol);
--Find record in List
        loop
            if Search_Link = null then
                put_line (" Symbol not found ");
                exit;
            elsif Search_Link.Symbol_Name = Symbol then
                put_line (Symbol);
                if Search_Link.Computer = Nord_10 then
                    put (Search_Link.Initial_Value_Nord_10);
                    exit;
                else
                    put (Search_Link.Initial_Value_Nord_50);
                    exit;
                end if;
            else
                Search_Link := Search_Link.Previous_Link;
            end if;
        end loop;
    end DISPLAY;
end Tools;
```

Figure 2 SAMPLE OF ADA FEATURES (CONT'D)

tial software impact, a strict protocol was maintained by enforcing the rule that each device driver use only the interface control file in communicating with Sintran. Although the enforcement of this interface was accomplished manually on the F-16, the philosophy is similar to the automatic enforcement mechanism provided by Ada's package specification. Despite the complexity of the device handlers, adaption requirements for each new release of Sintran were almost totally localized to the interface control file. As a result of this loosely connected, enforced design, the device drivers required minimal maintenance activity.

Conclusion

Our experience on the F-16 simulator indicates that the high cost of software modifications is largely the result of unnecessarily complex and unclear interfaces. Software which is loosely connected to its neighboring components, and easily understood in terms of functional characteristics, has not exhibited the increase in maintenance costs in the later life cycle stages.

Complex interfaces can frequently be simplified by decomposing systems based on interface requirements rather than functional requirements. Ada's package specification can be used early in the software development cycle to support this decomposition process. Used in this fashion, the package specification provides both early visibility of potentially high-cost modification areas and a vehicle to enforce design and documentation requirements.

Unclear interfaces result from incomplete documentation and designs which have been tailored to special-purpose applications. Ada supports the capture of the development process through its Generic, Data Typing, and Package mechanisms, and does so without costly run-time overhead.

References

1) Boehm, Barry W., "The High Cost of Software," Excerpted from Practical Strategies for Developing Large Software Systems, edited by E. Horowitz, Addison-Wesley, 1975.

2) Glass, Robert L., and Noiseux, Ronald A., Software Maintenance Guidebook, Prentice-Hall, 1981.

3) Booche, Grady, Software Engineering With Ada, Benjamin/Cummings, 1983, p. 29.

4) Brender, Ronald F., "What is Ada," Computer, June 1981, pp. 17-24.