Lessons Learned in the Implementation Phase of a Large Ada™ Project

Carolyn E. Brophy[1], Sara Godfrey[2],
William W. Agresti[3], and Victor R. Basili[1]

1 Department of Computer Science
University of Maryland
College Park, MD. 20742

2 Goddard Space Flight Center
Code 552.1
Greenbelt, MD. 20771

3 Computer Sciences Corporation
System Sciences Division
8728 Colesville Road
Silver Spring, MD. 20910

Abstract

We need to understand the effects that introducing Ada has on the software development environment. This paper is about the lessons learned from an ongoing Ada project in the Flight Dynamics division of the NASA Goddard Space Flight Center. It is part of a series of lessons learned documents being written for each development phase.

FORTRAN is the usual development language is this environment. This project is one of the first to use Ada in this environment. The experiment consists of the development of two spacecraft dynamics simulators. One is done in FORTRAN with the usual development techniques, and the other is done with Ada. The Ada simulator is 135,000 lines of code (LOC), and the FORTRAN simulator is 45,000 LOC.

We want to record the problems and successes which occurred during implementation. Topics which will be dealt with include (1) use of nesting vs. library units, (2) code reading, (3) unit testing, and (4) lessons learned using special Ada features.

It is important to remember that these results are derived from one specific environment; we must be very careful when extrapolating to other environments. However, we believe this is a good beginning to a better understanding of Ada use in production environments.

Ada incorporates many software development concepts; it is much more than "just another language". As such, we need to understand the effects of introducing Ada into the software development environment. This paper concentrates on the lessons learned from an ongoing Ada project in the Flight Dynamics Division of the NASA Goddard Space Flight Center (GSFC). The Ada project is sponsored by the GSFC Software Engineering Laboratory (SEL). It is part of a series of lessons learned documents being written for each development phase.

Environment

FORTRAN is the usual development language in this environment. The flight dynamics applications involve mission analysis and spacecraft orbit and attitude determination and control. Many of the software development projects are similar from mission to mission providing, for example, an attitude ground support system or an attitude dynamics simulator. This pattern of developing similar applications is important for domain expertise and for the legacy developed in this environment for code, designs, expectations and intuitions. The similarity between projects allows a high level of reuse of both design and code. Since the problems are basically familiar ones, the development methodologies which involve much iteration do not seem to be necessary. The waterfall development model is basically used here, and seems to work well in this case. Lessons learned from the initial uses of Ada do not include changing this basic methodology.

Project

The project was originally designed as a parallel study with two teams. Each would develop a spacecraft dynamics simulator, one with FORTRAN as the implementation language, and one with Ada as the implementation language. The specifications for each simulator were the same, supporting the upcoming Gamma Ray Observatory (GRO) mission. However, there are many

---

Ada is a trademark of the U.S. Department of Defense - Ada Joint Program Office.

Contact: Carolyn Brophy, Department. of Computer Science, University of Maryland, College Park, MD 20742, (301) 454-6154.

Support for this research provided by NASA grant NSG-5123 to the University of Maryland.

other differences between the projects which keep the study from being truly "parallel". The FORTRAN version was the production version, thus they had scheduling pressures the Ada team did not have. Without scheduling pressures, the Ada team made enhancements in their version not required by the specifications, which increased time spent on the project. This was also the first time any of these team members had done an Ada project, while the FORTRAN team was quite experienced with the use of FORTRAN. The Ada team required training in the language and development methodologies associated with Ada, while the FORTRAN team did things in the usual way [McGarry, Page et al. 83]. The Ada team also experimented with various design methodologies; this was necessary to find which ones would work better for this development environment. The FORTRAN team was working with a mature and stable environment. In switching to Ada, the legacy of reuse for design, code, intuitions and experience are gone, and will be rebuilt slowly in the new language.

The philosophies of development were quite different between the two projects. The Ada team consistently applied the ideas of data abstraction and information hiding to their design development. The FORTRAN development used structural decomposition methods.

Our goals with this project include:

(1) How is the use of Ada characterized in this environment?

(2) How should the existing development process be modified to best changeover from FORTRAN to Ada?

(3) What problems have been encountered in development? What ways have we found to deal with them?

Current Project Status

Both the FORTRAN and Ada teams started in January, 1985. The Ada team began with training in Ada, while the FORTRAN team immediately began requirements analysis. The FORTRAN team delivered its product (45K) after completing acceptance testing in June, 1987. The Ada team is scheduled to finish system testing its 135K product in February, 1988. Discussions of the product size differences and effort distributions are presented in [McGarry, Agresti 88].

The lessons learned from major phases in the Ada development are being recorded in a series of SEL reports: Ada training [Murphy, Stark 85], design [Godfrey, Brophy 87], and implementation [in preparation]. This paper presents some of the main results from the implementation (code and unit test) lessons learned.

Lessons Learned

1. Nesting vs. Library Units

1.1 The flat structure produced by using library units has advantages over a heavily nested structure.

Nesting has many effects on the resulting product. The primary advantage of nesting is that it enforces the principle of information hiding structurally, because of the Ada visibility rules. Whereas with library units, the only way to avoid violations of information hiding is through self-discipline. In addition, the dot notation tells the package where a module is located.

There are quite a few disadvantages to nesting, however. Nesting makes reuse more difficult. A second dynamics simulator in Ada is now being developed which can reuse up to 40% of the Ada project's code. But in order to reuse it, the nested code has to be unnested, since the new application only needs some of the nested units. This is often a labor intensive operation. Nesting also increases the amount of recompilation required when changes are made, since Ada assumes dependencies between even sibling nested objects/procedures, even when the dependency is not really there. This requires more parts of the system to be recompiled than is necessary when more library units are used. It is also harder to trace problems back through nested levels than it is through levels of library units. There is no easy way to tell where a unit of code was called from, when it is nested. But library units have the "with" clauses to identify the source of a piece of code. For this reason it is now believed that over use of nesting at the expense of using more library units makes maintenance harder. This is contrary to the team's earlier expectations. The team had used nesting successfully before on a 5000 lines of code training project. However, this kind of approach does not scale-up well when developing large projects.

Library units seem to have a lot of advantages. Besides fewer recompilations when changes are made, and easier unit testing, every library unit can easily be made visible to any other library unit merely by use of the "with" clause. In nested units this visibility does not exist, and a debugger becomes essential to see what is happening at the deeper levels that are not within the scope of the test driver. Library units allow smaller components, smaller files, smaller compilation units, and less duplication of code. The system is more maintainable, since it is easier to find the unit desired. Reuse with library units is also easier, since the parts of the system are smaller. Configuration control is also easier with library units since more pieces are separate (i.e., the ratio of changes to code segments modified is closer to 1). The major disadvantage seems to be that a complicated library structure develops, which can lead to errors by the developers. However, if the Ada project were to be done over now, the team would use more library units, and nest less.

Advantages and Disadvantages of
Nesting vs. Library Units

NESTING

Advantages	Disadvantages
* information hiding	* enlarged code
* visibility control	* more recompilations
* type declarations in one place	* harder to trace problems through nested levels
	* can't easily tell where a unit of code called from
	* type declarations in one place means problems for reuse
	* harder maintenance
	* debugger required
	* larger unit sizes inhibit code reading
	* harder to reuse part of the system

LIBRARY UNITS

Advantages	Disadvantages
* fewer recompilations	* no information hiding
* easier unit testing	* complex library structure
* smaller components	
* smaller files	
* smaller compilation units	
* less code duplication	
* easier maintenance	
* "with" clauses show source of other code units used	
* easier reuse	
* easier configuration control	

1.2 The balance between nesting and library units is an important implementation issue, not a design issue.

The issue of whether to use library units or nested units first arises in the design phase. At least this is the case if it is assumed that the design documents reflect this aspect of implementation (i.e., the design documents indicate in some way when nesting is intended vs. when library units should be used). While it is appropriate for the design to show dependencies, these should not dictate implementation, as far as the library unit/nesting question is concerned. The team considered the decisions concerning nesting/library units to be an implementation issue.

The library units in the Ada project went down about 3 to 4 levels, while nesting went down many levels below that. Another view of the system shows the Ada project had 124 packages and 55 library units. During implementation most team members felt an appropriate balance had been reached between nesting levels and number of library units. However, in retrospect, several felt the nesting had been overdone.

1.3 It appears best to use library units at least down to the subsystem level, and nesting at lower levels where there is minimal interaction among a small number of modules.

Experiences with unit testing seem to indicate that library units should at least go down to the subsystem level. This makes testing easier. Below this level the benefits of nesting sometimes become too important to ignore. This is one heuristic which could be used to help determine when the transition from library units to nested units should occur.

An additional way to determine when the transition should occur is to examine the degree of interaction between pieces. For modules which interact heavily, library units are preferred. At the point where the interaction drops off, using nested units is preferable. Sections with nested code are easier to deal with when they are small.

1.4 In mapping design to code, caution should be used in applying too rigorous a set of rules for visibility control.

In an attempt to control visibility, two features appear to have been too rigorously applied. The first feature is nesting. The design of the Ada project seemed to suggest a particular nesting implementation. But this created many objects within objects yielding a high degree of nesting. The second way to control visibility is through the use of many "call-through"s (a procedure whose only function is to call another routine). "Call-through"s were used to group appropriate pieces together exactly as represented in the design. They can be implemented via nesting or library units. Faithfulness to the design structure was maintained this way.

The design had non-primitive objects with specific operations. These objects were implemented as packages. To put the specific operations (subprograms) into the objects (packages) the team used "call-through"s. Thus a physical piece of code was created for every object in the design. "Call-through"s are one of the reasons for the expanded code in the Ada project when compared to the FORTRAN version. It is estimated that out of the 135K LOC making up the Ada system, 22K LOC (specifications and bodies) are because of "call-through"s. While "call-through"s provide a good way to collect things into subsystems, these should be limited to only two or three levels in the future.

If the implementation were to be done over now, many of the existing "call-through"s would be eliminated. Instead of creating actual code to correspond with every object in the design, some objects in the design would remain "logical objects". No actual packages would exist; instead, a logical object would be made up of a collection of lower level objects.

2. Code Reading

Code reading is generally done with unit testing. The developer doing the code reading is not the one who developed the code. Comments are returned to the original developer. After code reading and unit testing, the unit is put under configuration control.

2.1 Code reading helps in training people to use Ada.

Besides helping to find errors, code reading has the benefit of increasing the proficiency of team members in Ada. Individuals can see new ways to handle the algorithms being encoded. Code reading also allows another person besides the original developer to understand a given part of the project. This insight should help understanding and lead to better solutions of problems in the future.

2.2 Code reading helps isolate style and logic errors.

The most common errors found in code reading with Ada were style errors. The style errors involved adding or deleting comments, format changes, and changes to debug code (not left in the final product). Other types of errors found are initialization errors, and problems with incompatibilities between design and code. This can be due to either a design error or a coding error.

Because the Ada compiler exposes many errors not exposable by a FORTRAN compiler, code reading Ada has a different flavor than code reading FORTRAN. For example, the Ada compiler exposes such errors as (1) wrong data types, (2) call sequencing errors, (3) variable errors-- either the variable is declared and never used, or it is used without being declared. So, one seasoned FORTRAN developer working on the Ada project noted that code reading is more interesting in FORTRAN, since there were more interesting errors found in code reading FORTRAN, not found in reading Ada code. In general, logic errors are hard to find in this application domain, but enough logic errors are found to make code reading worthwhile.

Some of the difficulty of code reading with Ada on this project was due to the heavy nesting and the number of "call-through" units. Code reading would have been helped by a flatter implementation. The SEPARATE facility makes it necessary to look in many places at once to follow the code. However, code reading in Ada was easier than in FORTRAN because the code was more English-like, and hence, more readable. Often the reused FORTRAN code is an older variety without the structured constructs available in later versions.

Code reading tended to miss errors that spanned multiple units. This would be expected with any implementation language as well. One example was a problem where records were skipped when they were being output. The debugger actually found the problem.

Despite the implementation language, code reading appears to be important for highly algorithmic routines.

Groups of routines that are used only to call others may be checked to make sure the design's purity is maintained.

3. Unit Testing

3.1 Unit testing was found to be harder with Ada than with FORTRAN.

The FORTRAN units are already relatively isolated; this makes unit testing easy. Only the global COMMONs need to be added to do the unit tests. On the other hand, the Ada units require a lot of "with'd in" code, and are much more interdependent. Another very different Ada project had perhaps even more interdependence between its modules than the Ada project did. That team also found the interdependence made unit testing very difficult. More interdependence exists between Ada units because there are more relations to express in Ada. There are textual inclusion (nesting), with-ing in (library units), and invocation. FORTRAN only has invocation.

3.2 The introduction of Ada as the implementation language changed the unit testing methods dramatically.

Unit testing with Ada was done very differently. Since one unit depends on many others, it is usually hard to test a unit in isolation, so this was generally not done. The Ada pieces were integrated up to the package level, and then unit testing was done. Then testing was done with groups of units that logically fit together, rather than actual unit testing. The integrated units are tested, choosing only a subset of possible paths at a time. The debugger is used to look at a specific unit, since the test drivers cannot "see" the nested ones. With Ada projects a debugger becomes essential. This is in contrast to the usual development in FORTRAN where no integration occurs at all until after unit testing.

This shows that the biggest difference between the way FORTRAN and Ada projects are done at this point in development is incremental integration. This actually represents a change in the development lifecycle of an Ada product; integration and unit testing are alternately done rather than finishing unit testing before integration.

3.3 The library unit/nesting level issue directly affects the difficulty of unit testing.

The greater the nesting level, the more difficult unit testing is, since the lower level units in the subsystem are not in the scope of the test driver. This is the primary reason a debugger becomes a required testing aid with Ada projects. For this reason, more library units and less nesting would have made testing easier. Library units have to go down to a level in the design that makes testing more feasible. With the Ada project that would have meant taking library units down to a lower level in the design, if the project were to be done over.

Two other ways to deal with the nesting during unit test were tried and were not very successful. One solution pulls an inner package out, and includes the types and "with'd in" modules the outer package used in order to execute the inner one. This is difficult to do for each unit. The other solution is to modify the specifications of the outer package so that nested packages can be "seen" by the test driver. This solution requires lots of recompilation. With more library units, there would be less recompilation, because there would be fewer changes of specifications. Again however, the best way to test was to use the debugger on unaltered code.

3.4 The importance of unit testing seems to be more related to application area than to implementation language.

Whether the implementation is in FORTRAN or Ada, does not seem as important as whether the application has lots of calculations or has lots of data manipulations. Unit testing seemed more valuable with scientific applications; perhaps because calculation errors show up when only a small amount of localized code is executed. But data manipulation errors require more of the system to be operating before it is known if errors are present.

4. Use of Ada's Special Features

4.1 Separation of specifications and bodies is quite beneficial and easy to implement.

The team entered the specifications first, whenever possible, before the rest of the code. This gave a high level view of the system early in the development. Another benefit is that this helped clarify the interfaces early. Separating the specifications and bodies also reduces the amount of recompilation required when changes are made.

4.2 Generics were fairly easy to implement and they reduce the amount of code required.

The only problems encountered were with correct compilation of the generics in some cases, due to compiler bugs in an early version of the compiler, rather than incorrect code. As Ada matures, this will not be a problem at all.

4.3 Using too many types increases coding difficulty.

The strong typing was very difficult to get used to, when one is accustomed to weakly typed languages such as FORTRAN. It was easy to create too many new types as well.

Often a brand new type was created with a strict range appropriate for one portion of the application. Then in other areas where subtypes could have been used, the range on the original type was found to be too restrictive, so another brand new type was created instead to handle the new situation. Then a whole new

set of operations had to be created as well for the additional new type. Next time the team would recommend creating a more general new type, and using many different subtypes of the original type, rather than creating more new types. In this way operations can be reused and there are far fewer main types to keep track of. Designers need to spend time developing families of types that inherit properties from one another.

The strong typing presented some problems when testing units, though it prevents some kinds of errors, also. It was harder to write test drivers that could deal with all the types in the units being tested. It was also harder to do the I/O, since so many types had to be dealt with.

4.4 Tasking was difficult to code and test, however, this seems due to concurrency in general and not Ada specifically.

Tasks were used in the user interface part of the project. The user was given many options which made the interactions between the tasks of the subsystem very difficult to plan and execute correctly.

It was harder to code tasks from the design than it was to code other types of units. However, this is not really due to Ada, but rather it is the nature of concurrency problems. The language made the use of tasking easier, and encouraged the developers to use tasking more than they would have otherwise. The dynamic relationships of concurrency cannot be represented in the design (termination, rendezvous, multiple threads of control). Correctness was very difficult to assure, as is usual with these kinds of problems, and deadlock was hard to avoid. Functional testing was done, which is the usual type in this environment.

The major problem the developers had was with exceptions. These are no worse with tasking than they are with any other program unit, however.

4.5 Exception handlers have to be coded carefully.

The key problem with exceptions is deciding the best way to handle them. Errors and the sources of errors can be hard to find if the exception handlers are not coded carefully. Suppose a particular procedure will call another unit, expecting some function to be performed, and certain kinds of data to be returned. If an exception is raised and handled in the called unit, and it is non-specific for the problem raising the exception (e.g., "when others") , the caller gets control back without the required function being performed. But the exception was handled and data was returned, so the call looks successful. Yet as soon as the caller tried to use the data from the routine where the exception was raised and handled, it fails. Because of propagation, it can be very difficult to trace back the error to the original source of the problem.

Several members of the team would recommend incorporating the way exceptions are to be handled into the design, rather than leaving this until implementation. Put into the design (1) what exception would be raised, (2) where it will be handled, and (3) what should happen.

Ada Features*

	implementation ease	benefit
tasking	-	+
generics	+	++
strong typing	0	0
exception handling	0	+
nesting	+	-
separate specs/bodies	++	++

* This figure represents a subjective assessment based on team member interviews

Summary

We have learned several important things about four major areas in implementation. There are many advantages to using library units, though nesting can have its usefulness at some point below the subsystem level. Code reading helps train people in Ada, and helps to isolate style and logic errors. Unit testing was substantially changed by using Ada: the first stages of integration often began before unit testing proceeded. Some Ada features are quite powerful and should be carefully used.

It is important to remember that these results are derived from one specific environment. We must be very careful when extrapolating to other environments. There are also many questions still left to be answered. Studies of this project will continue, and other Ada projects are being started. These will help us evaluate the effects on longer term issues such as reuse and maintainability of the Ada projects. We believe this project is a good beginning to a better understanding of Ada use in production environments.

Acknowledgements

The Ada experiment is managed by Frank McGarry of NASA/GSFC. The authors would like to thank him and the Ada team for their cooperation and assistance.

References

[Agresti 85]
Agresti W., "Ada Experiment: Lessons Learned (Training/Requirements Analysis Phase)", Goddard Space Flight Center, Greenbelt, MD 20771, August 1985.

[Godfrey, Brophy 87]
SEL-87-004, "Assessing the Ada Design Process and Its Implications: A Case Study", Godfrey S., and Brophy C., Goddard Space Flight Center, Greenbelt, MD 20771, July 1987.

[McGarry, Agresti 88]
"Measuring Ada for Software Development in the Software Engineering Laboratory", Hawaii International Conference on Systems Science, January, 1988.

[McGarry, Nelson 85]
McGarry F., and Nelson R., "An Experiment with Ada -- The GRO Dynamics Simulator Project Plan," Goddard Space Flight Center, Greenbelt, MD 20771, April 1985.

[McGarry, Page et al. 83]
SEL-81-205, "Recommended Approach to Software Development", McGarry F., Page J., Eslinger S., Church V., and Merwarth P., Goddard Space Flight Center, Greenbelt, MD 20771, April 1983.

[Murphy, Stark 85]
SEL-85-002, "Ada Training Evaluation and Recommendations from the Gamma Ray Observatory Ada Development Team", Murphy R., and Stark M., Goddard Space Flight Center, Greenbelt, MD 20771, October 1985.

Biographies

Carolyn E. Brophy is a graduate research assistant at the University of Maryland, College Park. Her research interests are in software engineering, and she is working with the NASA Goddard Software Engineering Laboratory. Ms. Brophy received a B.S. degree from the University of Pittsburgh in biology and pharmacy. She is a member of ACM.

Sara H. Godfrey is with Goddard Space Flight Center in Greenbelt, Maryland, where she has been working with the NASA Goddard Software Engineering Laboratory. She received a B.S. degree from the University of Maryland in mathematics. (picture missing)

William W. Agresti is with Computer Sciences Corporation in Silver Spring, Maryland. His applied research and development projects support the Software Engineering Laboratory at NASA's Goddard Space Flight Center. His research interests are in software process engineering, and he recently completed the tutorial text, *New Paradigms for Software Development*, for the IEEE Computer Society. From 1973-83 he held various faculty and administrative positions at the University of Michigan-Dearborn. He received the B.S. degree from Case Western Reserve University, the M.S. and Ph.D. from New York University.

Victor R. Basili is Professor and Chairman of the Computer Science Department at the University of Maryland, College Park, Maryland. He was involved in the design and development of several software projects, including the SIMPL family of programming languages. He is currently measuring and evaluating software development in industrial and government settings and has consulted with many agencies and organizations, including IBM, GE, CSC, GTE, MCC, AT&T, Motorola, HP, NRL, NSWC, and NASA.

He is one of the founders and principals in the Software Engineering Laboratory, a joint venture between NASA Goddard Space Flight Center, the University of Maryland and Computer Sciences Corporation, established in 1976. He has been working on the development of quantitative approaches for software management, engineering and quality assurance by developing models and metrics for the software development process and product.

Dr. Basili has authored over 90 papers. In 1982, he received the Outstanding Paper Award from the IEEE Transactions on Software Engineering for his paper on the evaluation of methodologies.

He was Program Chairman for several conferences including the 6th International Conference on Software Engineering. He serves on the editorial boards of the *Journal of Systems and Software* and the IEEE *Transactions on Software Engineering* and is currently Editor-in-Chief of *TSE*. He is a member of the Board of Governors of the IEEE Computer Society.