Development of a Reusable Library of Ada Simulation Modules - Based on the User's Need

By Lynn D. Stuckey Jr. and Alan J. Hicks

Boeing Military Airplanes
Simulation and Training Systems
Huntsville, Alabama

ABSTRACT

The existence of so-called reusable modules does not necessarily indicate a correct or even desirable design. However, the design of a reusable Ada module implies an organized attempt to design a reusable Ada simulation. It is this organized approach that must be addressed before a reusable library can be developed. The approach must deal with certain questions:
(1) What are the goals that guide the development and use of the reusable library?
(2) What is the structure of the reusable software?
(3) What is the structure of the reusable simulation?
(4) What are the priorities for the development of software for the library?
An attempt at developing a reusable library without answering these questions is futile. Trade-offs must be performed between such goals as readability, maintainability, genericism, speed, and efficiency. The structure of the reusable softare will facilitate the efficient implementation of the software into a new project. A reusable library is of little use if the software will not fit the mold of the overall simulation. And finally, the plan for the development of software must be prioritized to allow developement based on importance and usefulness. This paper addresses the requirements and the design approach of such a user based reusable simulation software library.

INTRODUCTION

So many times a lump of software is gathered together because it performs a common function of many possible future simulations and that lump is called reusable. Later on down the road, it is conceded that "a few minor alterations" were required to make the code usable in the required application. The advent of the Ada language makes the possibility of truly reusable code real. However, even more than Ada is required. What is required is planning, discipline, and dedication. Planning is required to develop reusable software goals, software structure, simulation structure, and priorities of model development. Discipline is required by the developers to stick to those goals during the development of the reusable software and simulation structures. Dedication is required of the corporation (and user) to continue the investment throughout the development cycle to attain the final goal of a library of reusable code for the user's simulation needs.

Striving to obtain reusable code is a natural ambition (to save time and money). However, the attainment of reusable code requires effort. The days of haphazardly developing software systems must come to an end because the user (ultimately funded by the taxpayer) demands and deserves an organized approach in software development. It is this organized approach for the development of a library of Ada simulation modules, from the setting of software goals to the prioritizing of system development, that is addressed in this paper.

GOALS

What are the goals that guide the development and use of the reusable library? There has long been a misconception that the use of Ada will automatically result in a vast library of reusable code. This is simply untrue. Bad and nonreusable code can be written in any language. The introduction of Ada simply provided some features that, when properly used, can produce highly reusable code. But these features are not enough. Guidelines must be established for both the development and use of reusable code. The goals that guide the development are possibly the most important consideration in the design of a reusable simulation library.

A goal is simply defined as the object toward which an endeavor is directed. In this instance the endeavor is the development of a reusable simulation library. Without goals the library would flounder with code that may be too specific, too generalized, based on incorrect formats, or too costly. Without goals there is a danger that the perceived benefit from reusing code will always outweigh the problems associated with "lifting" the code from the old program. This may in fact not be true if the interactions between the liftable code and the rest of the program are excessive. Based on these facts, the need for goals is very obvious. Goals are needed in both the development of the reusable code and the establishment and use of the library itself. The goals selected will effect the nature of the software developed; but reusable software itself must be a goal of the manufacturer if it is ever to occur. Some of the goals worth discussing are maintainability, efficiency, scope, and portability.

Maintainability

The true cost of a simulation is only realized during the full life cycle of the software. The initial cost of development, including hardware, is minor when compared to the cost of maintaining the software. This fact must be taken in to account when developing a reusable library. Reusable software is not only maintained for the original program for which it was developed, but also for every other ensuing project. If the reusable software is not maintainable then the developmental cost savings derived from using the software in the first place becomes insignificant. It is apparent that the foremost goal of software in a reusable library is that it must be maintainable. Here Ada plays a big role. With the emergence of Ada, the ability to write truly readable/understandable code is now possible. It is this understanding that will increase both the reusability and the maintainability of the code. But this readability or understandability of code is not just a luxury. For too long the user has paid the price for code that was explained by so-called documentation. Reliance on documentation is a farce. The only documentation that can be trusted is that of self-documenting code. In essence all code is self-documenting, because software designers will always assume the document is incorrect and "go to the source". Ada provides the capability for self-documenting code, and therefore well designed Ada is maintainable. Code that can be considered self-documenting must be maintainable, because maintainable code is built for the long haul.

Efficiency

Before someone rises up in arms it must be noted that the reusable software to be developed for the library must be efficient. The requirements for reusable software components are significantly more demanding that those for conventional flight simulation software design. This is due to the fact that the software components must be generic in order to respond to differing requirements from simulator to simulator while at the same time being efficient enough to meet real time requirements. It is always in the programmers' best interest to design and code his software efficiently regarding both speed and memory usage. This goal must be observed in a reusable library. But this goal must not take precedence over the goal of maintainability; there is a lot of truth in the saying "Memory is cheap". The development and maintenance of the software will greatly out weigh the cost of more memory or a faster computer. This is an area in which the user will have to be willing to "bite the bullet". The cost of the hardware may be a bit more in order to develop or reuse software but in the long run it will save the user money. Although efficiency ranks second as a design goal (to maintainability) now, as computers become faster and faster and memory becomes bigger and cheaper , it will lose importance. But for now it ranks second only to maintainability.

Scope

All too often it seems that reusable code is too general, is too specialized, or just handles the wrong type of data structure. The greatest cost savings from reusability are realized with the least amount of modification. A reusable software module can end up being more expensive to use than original development if the amount of modification becomes extensive. Both of these problems can result from a reusable library that is not of a proper scope. If the library is a repository of all the reusable software that the customer has, then soon the library will be too big and the software too generic. Especially in the beginning, for a reusable library to work it will have to have a limited scope. In this case the library needs to be limited to a collection of software directed at real-time flight simulation. However, there is another concern at play here, namely the management of a large number of reusable modules. As the user accumulates more and more software, the library can quickly become overwhelming. If it costs more to retrieve and understand a component than it does to develop it, then the goal of reusability is lost. To this end, the library must be set up with a limit of both files and number of routines that serve the same purpose. It would be much better to improve on the old modules than to add new ones.

To avoid the overwhelming effect of many models in the library with a variety of model abstractions, the library must evolve a standard interface definition for each item models support. All models must support the standard interface and report on the fidelity of their support. The models would be catalogued for the user by their fidelity. For example, consider a library which contains an item "aircraft bank angles", with standard interface inputs of rudder, ailerons, pitch, spoilers, elevators, flaps, and airspeed. The library might present a summary catalogue such as :

Model	Rudder	Aileron	Pitch	Spoiler	Flaps	Airspeed
A	Full	N/A	N/A	N/A	N/A	N/A
B	Full	Inc	Inc	N/A	N/A	N/A
C	Full	Cont	Cont	Inc	Inc	Inc
D	Full	N/A	N/A	N/A	N/A	N/A

N/A = Not Applicable, Full = Fully Simulated,
Cont = Continuous Position Considered,
Inc = Incremental Position Modeled

A user could select a model which considers the aspects of the bank angle with which his simulation is concerned. Introduction of a new model which considered fuel load center of gravity would require update of the catalogue such as :

Model	Rudder	Aileron	Pitch	Spoiler	Flaps	Airspeed	Fuel CG
A	Full	N/A	N/A	N/A	N/A	N/A	N/A
B	Full	Inc	Inc	N/A	N/A	N/A	N/A
C	Full	Cont	Cont	Inc	Inc	Inc	N/A
D	Full	N/A	N/A	N/A	N/A	N/A	N/A
E	Inc	N/A	N/A	Cont	N/A	N/A	Full

The specification of each model would include holes for each interface, with the ones not implemented of type null (private). Therefore, the user could easily experiment with several closely related models before final selection.

This limit on scope will help solve one of the biggest obstacles to a reusable library, which is programmer acceptance. There is a natural reluctance on the part of programmers to accept responsibility for someone else's code. A reduced scope will make the library smaller, available to hold more specific designs, and make it easier to use and accept.

Portability

We live in an ever changing world of technology. The prospect of using the same computer, compiler, or operating system on all simulator projects is zero. This fact requires that the reusable software be portable or that the nonreusable components be isolated from the rest of the software. The reusable component can not be tied to a certain computer or system. This restriction will not sit well with everyone but must be a firm requirement from the user. Without it, the prospect of putting software into the library that can be of some use on other contracts is diminished. The obvious dependencies are tying into an operating system or pragma Interface to assembly or C. But there are also more subtle ones such as using the predefined types that may be present. For example, on one compiler integers may be declared as Long_Integer, Integer, and Short_Integer while another might be Integer, Short_Integer, and Tiny_Integer. Not only are the names different but the representations may differ between 64, 32 16, and 8 bits. Such things must be avoided if the library is to contain portable software.

A project is only as good as its requirements. The development of a reusable software library for flight simulation will only be as good as its goals and how well they are enforced. The need for such a library is evident but an unprepared attempt might set back the possibility of such a library a long time. Resuable software will be more expensive to develop and must be planned from the onset of development. Without pursued goals a lot of effort can go to waste.

Tradeoffs

All of these goals will not be achieved together in one implementation. Once it has been decided on the goals that are needed, then trade-offs have to be made to resolve the conflicts. The major problem with prioritizing goals is the "We've done it this way for 20 years" syndrome. The goals have to be based on the current costs and problems with software and on the current state of technology. A good example of this is the debate between efficiency and maintainability. It is a given that simulation code must be efficient both in speed and memory usage; but if it costs more during the life cycle of a program to maintain the software, then it makes sense to write maintainable code even if it results in a need for faster hardware or more memory. This prioritizing of goals must be consistent throughout the software in the reusable library. The manufacture's tradeoff selection must be expressed and enforced throughout software life cycle. This may be difficult, but it will insure the success of the library and its code. Each of the goals discussed must be evaluated and prioritized along with others such as genericism, size, etc.

STRUCTURAL MODEL

What is the ideal structure of reusable software? To many this may sound like a moot point, or something that will fall out after each module is produced. But, in the discussion of a reusable library or any project for that matter, the concept of a structural model can mean the difference between success and failure. The structure of the reusable software is what will make the use of the library possible.

It is clear that every project has one or many structural models. Problems arise because the design process starts before a single structural model is clarified or even specified. As a result, several structural models may be found in the final product. These arise from:

(a) Individual engineers with their own ideas about "how to do things on this project" before a consensus is reached.

(b) A consensus that does not take the actual structure that develops into account. For example, allowing certain modules to develop outside of the consensus, for whatever reason.

(c) No attempt at a central structure.

The purpose of the structural model is to ease or eliminate the two main problems associated with the implementation of a reusable software model. These problems are best stated by Michell D. Lubars in "Code Reusability in the Large Versus Code Reusability in the Small" as follows:

(a) The proper interaction between the reusable code and the rest of the program must be set up. As in performing an organ transplant, all the life-lines and vessels must be made to line up correctly between the transplanted code and the host program.

(b) The reusable code may have to be modified to precisely fit the new application. The code may solve a slightly different case, or may be too general or specialized as originally coded. In either case, some changes may have to be made before the reusable code is suitable for reuse.

Definition of a Structural Model

A structural model is a paradigm that indicates the form of the general solution of the software. It is a set of constraints on the solution space for the project (e.g., how data will be passed).

The structural model specifies:

(a) The kinds of entities that will exist in the design (How do you package?);

(b) How the real world is mapped into the software entities (What's in a package?); and

(c) The communication between entities (How do packages communicate?).

The structural model chosen represents the point of convergence for tradeoffs between maintainability and performance, quality and efficiency. As such, different projects may have different structural models.

The structural model forms the basis for implementing the design method. Requirements specifications are instantiations of the structural model: they constrain the identification of objects, how objects are grouped together, and how objects communicate.

Benefits of a Structural Model

Without a formal structural model up front, each engineer has his own model in his heart. This results in:

(a) Wasted work from the development of parallel structural models;

(b) "Analysis paralysis" caused by too many available choices;

(c) Separate designs items that don't play together.

The structural model makes the solution space more manageable in the following areas:

(a) Training of designers and implementors is based on the structural model (the way to do things on this project and why);

(b) Internal design reviews and reviews with the user do not have to explain (for example) the interface for each system: the interface is a given, based on the structural model. Further, the structural model reveals the quality of the software early in the process and before anything is specified in detail;

(c) More powerful testing and configuration management tools can be developed once the structural model is settled; and

(d) The structural model provides a basis for knowledge about the work to be done and the resources required.

The structural model reduces development costs:

(a) Once the structural model is done, it is reusable over the entire project life cycle. Having constrained the solution space of the software, designers never need to do this work again.

(b) Tools developed in-house and purchased from outside vendors can be selected on the basis of the structural model. They will be the right tools, and they will be applicable system-wide.

(c) There will be less ad-hoc documentation. Documentation of the module interface method, for example, will appear only once, and is code, part of the product.

(d) Lower integration costs result from a clearly understood structural model. Engineers who fix one another's code are talking the same language because they begin with a defined set of premises.

(e) Parallel development within the company and with subcontractors will be cleaner. A structural model makes obvious both what requirements must be set for a subcontractor, and whether the subcontractor has met the requirements.

The Right Structural Model

What does the "right" kind of structural model look like for a particular project? At a minimum:

(a) It takes into account the complete domain of the problem:

(1) Visibility (e.g., record/replay, IOS),
(2) Inter-processor communications,
(3) Time and memory requirements,
(4) Frame balancing and processor balancing,
(5) Testing and debugging.

(b) It results in a maintainable software.

(c) It minimizes side effects.

(d) It permits systems to be developed independent of the source of the inputs and the destination of the outputs.

(e) It allows new implementations of systems to be plugged into existing specifications.

(f) It permits the management of data voids. Systems for which data is missing can be stubbed, and the interface specification defines what must be known later about the system.

Example Structural Model

The implementation of a structural model is expressed in the following example of an aerodynamic model. The structure is based on the use of Ada packages to encapsulate and separate reusable and non-reusable components in the aerodynamic model. Figure 1 graphically describes the aerodynamic model. The Aero Forces and Moments object is a package containing a main procedure. This procedure is the only interface with the host program. The interface is accomplished via the procedures' parameter lists. The object is then abstracted into six sub-objects consisting of all the build models. These sub-objects then access generic interpolation and zone routines (Figure 2). The routines access the aero data tables which are abstracted so to be easily replaced with each new aircraft's data. Thus the aerodynamic model's reusable units consist of the Aero Forces and Moments executive, the build routines, and the interpolation and zone generics. The application dependent data is abstracted away to become a separate package that is plugged into the model. The last part of the model is a package of types. Since only types are in this package, a global data "swamp" is avoided. This types package will be the basis for variables and logic in the reusable code. An example of its contents would include an enumeration type containing all possible coefficients incurred in fixed-wing aircraft. A subtype of this enumeration would contain the coefficients applicable to the particular application and would be modified with each use. The code is designed such that it operates on the contents of the subtype.

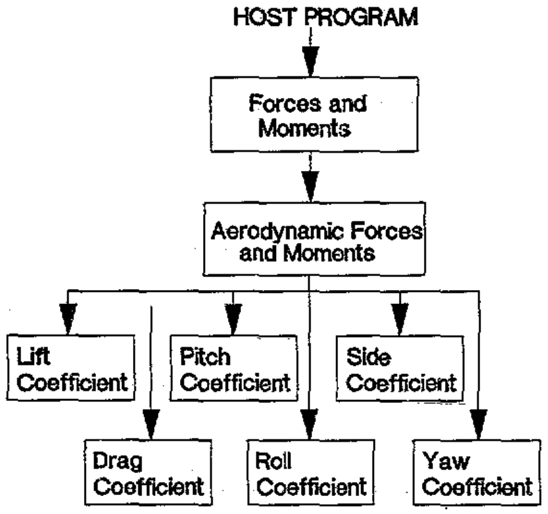

Figure 1

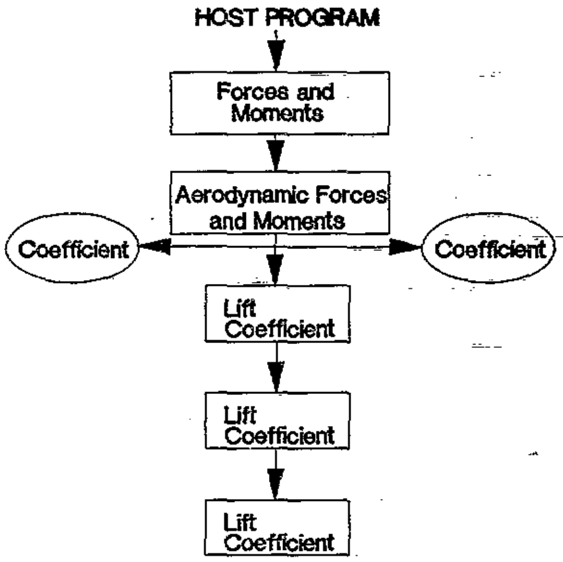

Figure 2

The reusable aerodynamic model must answer the questions posed in a structural model. The example answers the questions as follows:

(a) How you package? Based on an object-abstracted design and that all knowledge about the behavior of objects is strictly encapsulated within the objects themselves.

(b) What is in a package? Simply the objects noted in Figure 1 and 2, their procedural interfaces, and all pertinent information about the object.

(c) How packages communicate? Provided through procedure calls. All information passing is performed up and down the model and never side to side. This form of communication provides information hiding, alleviates the numerous problems of global data, and furnishes the user with an accurate list of inputs and outputs.

STRUCTURE OF THE REUSABLE SIMULATION

The next major question which must be addressed is: "What is the structure of the simulation?" Before the question can be answered, what is meant by "structure of the simulation" must be defined. The simulation shell support (or executive function) controls the execution, initialization, reset, reposition, testing, and data interfacing of the entire simulation. It must incorporate general interfacing and packaging schemes which not only allow efficient operation of code, but also enforces the hiding of valuable and vulnerable information. It must integrate perfectly with the reusable software because it is the backbone of the reusable simulation. The lack of a reusable simulation structure will be the main deterrent in the development of an industry-wide library since simulation structure varies from company to company and program to program.

Requirements of the Simulation Structure

When contemplating the structure of a reusable simulation, as with all designs, the requirements must be well defined. The primary objectives must be held as goals with ardent fervor. Otherwise, you miss the point of 90% of your effort. Remember, the system will naturally gravitate along the path of least resistance and that almost always results in chaos. To define the requirements the following list of choices must be reviewed:

(a) A structure which affords either consistent interfaces or lets the interfaces "fall into place".

(b) A structure which allows independent compilation of packages during development and maintenance or which forces recompilation of the entire simulation with even the smallest coding change.

(c) A structure which allows interfacing with software tools or which does not allow variables to be accessed at all for debugging or development.

(d) A structure which allows information hiding or one which permits all data to be seen and/or used by all modules.

(e) A structure which predisposes itself to automatic documentation or which discourages any effort of self-documentation or automatic documentation.

The list could go on and on. It is obvious that the list of choices has been structured to lead the reader down a selected path of logical design requirements. What is the right design? There probably are many in Ada, but there are certain goals and approaches to goals that are worthwhile. These goals are the commonly accepted software engineering principles. These principles, as enumerated by Booch (Reference 1) are:

(a) Abstraction,
(b) Information hiding,
(c) Modularity,
(d) Localization,
(e) Uniformity,
(f) Completeness, and
(g) Confirmability.

Development of a Reusable Structure

In answering the question: "What is the structure of the simulation?". A popular method, and for good reason, is to use Object Oriented Design (OOD). Applying a reusable library to the problem involves a variation of OOD which includes these basic steps:

(a) Acquire several typical simulations in the domain,

(b) Dissect those "typical" simulations,

(c) Analyze the various functions, interfaces and strong and weak points for data and control flow,

(d) Perform group abstraction for interfacing and control, and

(e) Assemble into a strong, reusable design.

Steps (a) and (b) are fairly self-explanatory. During step (c) one must analyze the existing simulations for data and control flow. After thorough analysis and observations have been made as to which designs incorporated what desirable characteristics, the interfaces and flow control require abstraction into object-oriented groupings (beginning object-oriented design for step (d)).

Step (e) concerns the reusable design method for the simulation structure. As stated earlier, an object-oriented (object-abstracted) development is a desirable approach for reusability. However, that simple statement does not describe all of the design activity required. The process of implementing control flow is best approached from the top-down. It is for this reason the simulation top-level control module (let's call it the Master_Simulation_Executive or Master_Sim_Exec) was called the backbone of the reusable simulation structure. The Master_Sim_Exec will then control the execution of the next level (or tier) of executive or control modules. The design of lower tiers can continue as structural analyses are completed on the design criteria. As the design firms up on the lower levels, corrections or revisions will need to take place on the levels affected above. This iterative process leads to a more effective design in terms of data and control flow. Besides, how many times do engineers produce perfect code on the first try? The resulting tier-level structure will have a conical shape and look somewhat like Figure 3.

Once the basic simulation structure is defined, further reusability needs to be addressed in the design methodology. The reusable design methodology must address: maintainability, readability, testability (reliability and consistent test interfacing), consistency of documentation (for ability of reuse), capability of module stand-alone (modularity), consistency and control of interfaces, and clarity of design criteria application (requirements traceability).

It can be inferred from the above topics, a reusable simulation will exhibit many important design characteristics (i.e. - commonly accepted software engineering principles). A simple cross-reference of software engineering principles to the primary Ada feature or software tool which answers that need is shown below:

Software Engineering Principle	Ada Feature or Software Tool
Abstraction	Enumeration types and Packages
Information hiding	Packages and Separates
Modularity	Packages and Separates
Localization	Packages
Uniformity	Coding Standards and Style Guide
Completeness	Packaging Strategy
Confirmability	Parameter list interfacing

CONICAL SOFTWARE STRUCTURE

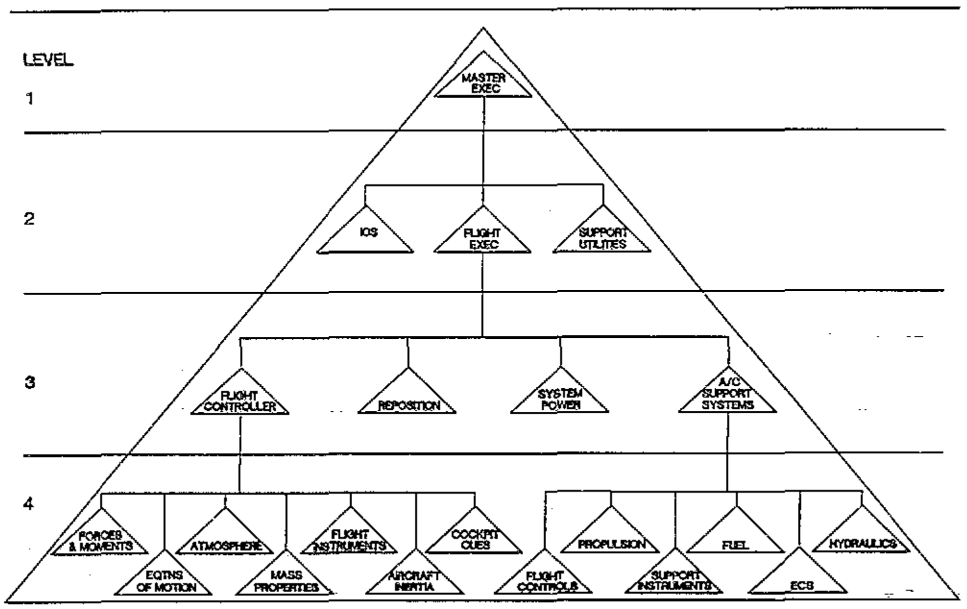

Figure 3

An obvious conclusion is that the effective use of packages for structuring the simulation can help tremendously towards the design of reusability within that simulation. Although packages are essential in a reusable simulation structure, they cannot meet all the needs required. If the Master_Sim_Exec is the backbone of the reusable structure, then the packaging strategy is the skeleton and the Ada features are the muscle and tissue holding the organs (the reusable modules) in place. Therefore, let's discuss the muscle and tissue of the Ada features that will contribute to the reusable design. First, for the protection of tender parts of the simulation anatomy, interfaces need to be well hidden. The use of records in procedure parameter lists allows both information hiding and "plug-in-ability" (modularity). Typical groupings of interfaces are shown in record type declarations. Second, to allow individual modules to stand alone (modularity), separates must be used whenever possible. By using separates, independent compilation of packages is made possible. Next, to allow requirements traceability, it is helpful if aircraft unique constants (for example, dimensions) are held in a central location such as a common upper level package specification. Also, for maintainability, a style guide must be enforced which necessitates a naming convention and coding structure that creates self-documenting code. Finally, all reusable modules require common testing and simulation control hooks. This is made possible with a strategy for passing data through parameter lists. The final product, according to Booch in Reference 2, should ideally exhibit the following characteristics:

(a) Maintainability,
(b) Efficiency,
(c) Reliability,
(d) Understandability,
(e) Sufficiency,
(f) Completeness, and
(g) Be primitive.

PRIORITIES OF SYSTEMS SOFTWARE DEVELOPMENT

The evolution of a library of reusable Ada software will not come overnight or cheaply to the user; therefore, it would be best to prioritize the development of the systems. In other words, which models within the simulation "type" lend themselves best to the reusable design so that the design dollars can be spent wisely? Obviously, some systems within the realm of each simulation are totally unique; they are simply designed once and then "forgot about". However, other systems occur every time the "type" of simulation is done and the design is either the same every time or very similar. How are the models analyzed for reusability? Before prioritizing the design of the models, the simulation "type" and what systems it may entail.

Identify the Problem Domain

Certain types of simulations will actually require unique sets of systems to be modeled. As the type of simulation changes, so does the set of systems required. Some examples of training simulation types are listed below:

(a) Air Traffic Control Training,
(b) Nuclear Reactor Control Training,
(c) Battlefield Tank Crew Training,
(d) Fixed-Wing Aircraft Flight Training, and
(e) Rotary-Wing Aircraft Flight Training.

As can be seen from this set of examples, various simulations will need different sets of systems to be modeled. A battlefield tank crew trainer will need almost an entirely different set of systems than that of a nuclear reactor control trainer. Yet, even though the fixed-wing aircraft flight trainer will need some different systems than a rotary-wing aircraft flight trainer, many systems may be interchangeable or "reusable". This leads to a logical evolution of a library of reusable modules or systems represented by a set of modules which is oriented to the user. The user, for example, will not want the set of systems required for a nuclear reactor if he is going to simulate a helicopter. Nor, will a tank simulation customer want a contractor to start out with a library designed for an air traffic control simulation. In other words, identify the user's problem domain before acquiring or designing the reusable modules.

What Makes a System Model Reusable?

In order to figure out which systems lend themselves more to reusability than others, characteristics that make a system reusable must be identified. First, a system must reoccur in each (or at least most) of the simulations. Second, that system must perform the same basic function for each simulation. Third, the interfaces need to be similar (not necessarily the same). The method used to interface (records in parameter lists) helps to take care of inconsistency in interfaces. Using a fixed-wing aircraft flight trainer as an example, the systems listed below are typical of those in a fixed-wing aircraft flight trainer:

(a) Fuel_System
(b) Propulsion_System
(c) Environmental_Control_System
(d) Electrical_Power_System
(e) Navigation_System
(f) Cockpit_Instrument_Displays
(g) Flight_Dynamics
(h) Flight_Controls
(i) Mass_Properties
(j) Landing_Gear
(k) Atmosphere
(l) Hydraulics
(m) Ground_Station_Data
(n) Visual_Physical_and_Aural_Cues
(o) Instructor_Operator_Station

Typical simplified interfaces for two of these systems are:

(a) Mass_Properties

Inputs:

Fuel_Tank_1_Weight_Pounds
Fuel_Tank_2_Weight_Pounds
Fuel_Tank_3_Weight_Pounds

Outputs:

Aircraft_Gross_Weight_Pounds
Aircraft_X_CG_Percent_MAC
Aircraft_Y_CG_Percent_MAC
Aircraft_Z_CG_Percent_MAC
Aircraft_Ixx_Slugs_Feet_Squared
Aircraft_Iyy_Slugs_Feet_Squared
Aircraft_Izz_Slugs_Feet_Squared
Aircraft_Ixz_Slugs_Feet_Squared

(b) Atmosphere

Inputs:

Terrain_Height_Feet
Demanded_Altitude_Feet
True_Airspeed_Feet_Per_Second
Demanded_Wind_Profile
Demanded_Wind_Velocity
Demanded_Wind_Heading
Demanded_Temperature_Profile
Demanded_Icing

Outputs:

Height_Above_Terrain_Feet
Calibrated_Airspeed_Knots
Rate_of_Climb_Feet_Per_Second
Wind_Velocities_Feet_Per_Second
Dynamic_Pressure_Pounds_Per_Square_Foot
Total_Air_Temperature_Degrees_C
Pressure_Altitude_Feet

Looking at the interfaces more closely, it is obvious that the systems have been generalized and there is a need to look at a lower level. Most of these systems are actually built from several sub-systems. These sub-systems are where much of the in-depth analyses will occur. For example, internal systems to "Atmosphere" may include:

(a) Pressure
(b) Wind
(c) Temperature
(d) Icing

Since these objects are normally considered as the lowest level of module design, interfaces for these units will be the most difficult to design for consistency. It is for this reason that the interfaces between the sub-systems will be those which can make or break the reusability of the system. Let's look specifically at "Wind" interfaces. Typical "Wind" interfaces might be:

Inputs:

Reference_Airfield_Latitude
Reference_Airfield_Longitude
Direction_Cosine_Matrix
Pressure_Altitude_Feet
Aircraft_Height_Above_Terrain_Feet
Aircraft_Latitude
Aircraft_Longitude

Outputs:

Wind_Velocities
Wind_Accelerations

The procedure call in the package body may then look like this:

```
procedure compute_the_wind

  (EOM_Data_Outputs_To_Wind : in  EOM_To_Wind;
   Wind_Data_Outputs_To_EOM : out Wind_To_EOM);
```

where:

```
    EOM_Data_Outputs_To_Wind = inputs to "Wind"
    Wind_Data_Outputs_To_EOM = outputs to "Wind"
```

In such a case, control and test hooks are raised above this level. This use of input and output records in the procedure parameter lists would make the "Wind" model extremely reusable. To make interfacing provisions in the wind model to allow for instructor control, simply add the record:

```
        IOS_Control_To_Wind : in IOS_To_Wind;
```

Of course, this would then require functional changes within the "Wind" module. Taking "Mass_Properties" as an example; there would be less reusability if the model could only sum the weights of 3 fuel tanks. If the next simulation which came along had 5 fuel tanks, 4 external store locations and an internal cargo (which can be shifted to alter center of gravity), several (probably major) alterations to the code must occur. Of course, "Mass_Properties" is normally one of the easier systems to design for reusability; simply because most operations are summing functions and can be performed on a loop of an enumeration type.

It now becomes apparent that a system's degree of reusability will rely largely on the simplicity/complexity inherent to the system and on the capability to incorporate (using foresight) an adequate set of interfaces during the design of that system.

CONCLUSION

In the final analysis a reusable code library is both needed and desired, but it will not be easy to accomplish. For too long people have thought of reusability as a fortuitous by-product of a program. Reusable code is not black magic. It requires thought and hard work. Most of all reusable code will not be cheap, but it will save vast amounts of money in the long run. The developer and especially the user must recognize this and be willing to invest the time and money required to support this concept of reusability. Money alone will not result in the reusable library, it will require tough decisions dealing with goals, structures and priorities. As we address the problems, simulator design will become more reliable and maintainable and can be developed quicker and cheaper with the reusable components. The purpose of this paper was to remind the software community of the importance of a reusable software library and to introduce a plausible approach for such a development.

REFERENCES

(1) G. Booch, *Software Engineering with Ada, Second* Edition, Benjamin/Cummings, Menlo Park, California, 1987, p. 31.

(2) G. Booch, Software Components with Ada, Structures, Tools, and Subsystems, Benjamin/Cummings, Menlo Park, California, 1987, p. 34-35.

(3) M. D. Lubars, "Code Reusability In The Large Versus Code *Reusability In The Small*", *ACM SIGSOFT* Software Engineering Notes, vol. 11, no. 1 (January 1986) p. 21.

(4) R. Crispen, Personal Communication, 1988

(5) J. Hicks and L. Stuckey, "In Search of a Reusable Ada Aerodynamic Math Model", 10th Interservice/Industry Training Systems Conference Proceedings, 1988, p. 69-75.

ABOUT THE AUTHORS

Lynn D. Stuckey, Jr. is a software systems engineer with Boeing Military Airplanes in the Simulation and Training division. He has been responsible for software design, code, test, and integration on several Boeing simulation projects including the Ada Simulator Validation Program. He is currently involved in research and development activities on Ada software development. Mr. Stuckey holds a Bachelor of Science degree in Electrical Engineering from the University of Alabama, in Huntsville.

Alan J. Hicks is a systems engineer with the Simulation and Training Systems organization of Boeing Military Airplanes in Huntsville, Alabama. He has been responsible for implementation of design criteria, test and integration of the aerodynamic models in several Boeing S&TS simulator projects, including the Ada Simulator Validation Program. Mr. Hicks has 12 years of experience with Boeing in the field of aerodynamics and holds a Bachelor of Science degree in Aeronautical Engineering from Tri-State University, Angola, Indiana.