# SOME DESIGN CONSTRAINTS REQUIRED FOR THE ASSEMBLY OF SOFTWARE COMPONENTS: THE INCORPORATION OF ATOMIC ABSTRACT TYPES INTO GENERICALLY STRUCTURED ABSTRACT TYPES

Charles S. Johnson

## ABSTRACT

It is nearly axiomatic, that to take the greatest advantage of the useful features available in a development system, and to avoid the negative interactions of those features, requires the exercise of a design methodology which constrains their use. A major design support feature of the Ada language is abstraction: for data, functions, processes, resources and system elements in general. Atomic abstract types can be created in packages defining those private types and all of the overloaded operators, functions and hidden data required for their use in an application. Generically structured abstract types can be created in generic packages defining those structured private types (i.e. lists, trees), as buildups from the user-defined data types which are input as parameters. A study is made of the design constraints required for software incorporating either atomic or generically structured abstract types, if the integration of software components based on them is to be subsequently performed. The impact of these techniques on the reusability of software and the creation of project-specific software support environments is also discussed.

## INTRODUCTION

The reusability of Ada software developed in support environments will be wholly dependent upon the quality of those environments. The ability of programmers that are relatively inexperienced in Ada to generate reusable software will be enhanced by an environment rich in already reusable software components, which act as models for good design. In an analogy to a factory, components which are tooled to fit can be easily assembled. Atomic abstract types define objects which represent the discrete phenomena that are the subjects of the system development. Generically structured abstract types organize the objects of the system in a manner representing the relationships between those objects. If atomic and generically structured abstract types are defined according to some general design goals and constraints, then the subsequent assembly of these software components is made considerably easier.

## BRIEF BACKGROUND

Kennedy Space Center/ Engineering Development/ Digital Electronics Engineering Division is in the process of prototyping distributed systems supporting I & T applications, particularly the Space Station Operations Language (SSOL) System, which is the I & T subset of the User Interface Language

(UIL) for the Space Station. The discussions in this paper were developed from the results of systems designed and developed in Ada to demonstrate the general feasibility of creating software support environments which maximized the reusability of software components. The Ada environment used was that of VAX Ada under VAX/VMS.

## OBJECT DEFINITION IN ADA

The design and development of software components that meet the needs of the user community can be viewed largely as an effort to define and refine the definition of abstract objects and their associated operations in computer systems. The definition of objects in these systems is akin to a simulation effort. There is a direct correlation between the effectiveness of programs and the fidelity with which objects in those programs simulate the behavior of the external phenomena they are intended to represent. For example, an element in a scheduler queue, representing a process awaiting execution, must reflect the correct state of the process (priority, blocked for I/O, etc.) for the scheduler to function properly. The element must be distinguishable from other elements and not lose identity or integrity during operations.

As in simulation efforts, the goals and objectives for defining an object in a system should be specified at the outset. The system functional requirements should drive the process, while the scope of the system concept constrains development to areas that are productive.

## SIMPLE TYPES

An object is characterized by it's attributes and the operations which mediate it's interactions with other objects in the environment. In the Ada language, the process of object definition begins with selection of base type or the creation of a composite type.

Objects whose behavior is simple enough to be modeled by a numeric value, can be represented by subtypes or derived types, of numeric or discrete types. The subtype definitions can include range constraints, and in the case of non-discrete numeric types (UNIVERSAL_REAL, UNIVERSAL_FIXED), they can include limits of precision for the representation. Declared objects of subtypes are, however, compatible with their base type and subtypes of the base type, which can allow erroneous combinations by operations allowable in the base type (adding MINUTES to HOURS, for example).

If the allowable operations of these base types are unsuitable, they can be restricted by the use of a derived type, which inherits the operations of the base type, but only for declared objects of the derived type (incompatible with the base type). This can yield dimensional errors, however, for multiplies and divides of objects of the same type (FT * FT = FT, instead of FT squared). In these cases subtypes and derived

types are too simple in behavior to correctly represent the objects in applications, and composite types must be used. [1]

## COMPOSITE TYPES

Objects which are characterized by collections of components or attributes are defined in Ada by the use of composite types: arrays and records or by access types which designate composite types. Objects which are collections of compatible components are represented by arrays, whereas objects which have various kinds of attributes are represented by records or access types designating records. Objects with attributes, and which have complex interactions with other objects in the system, would seem to be the more useful, although these are the most complex to define.

Objects with attributes interact with each other by the means of those attributes, under the control of the allowable operations of the objects. These interactions can produce modifications and deletions of the objects or creation of new and different kinds of objects. In Ada, the operations are defined as subprograms (functions and procedures) with parameters of the object type or subtype.

The operations which correspond to functions can be overloaded onto the set of computer math symbols for the given types. A function producing a scalar dot product from two input vectors could be given the name "*", for instance. At the same time, a function producing a vector cross product from two vectors, could also be named "*". The compiler would resolve these two operations from the type of the returned object. The compiler cannot, however, resolve these operations when the type of return is unknown. Vector products defined in this way could not then be embedded in longer equations, where they would generate intermediate results of indeterminate type.

## DIFFERENTIATION

There are different levels of definition for a system, it's objects and operations. Definition of the gross structure of a system can typically be generated, in a fairly simple manner, by the object-oriented or functional decomposition methods. Definition of the fine structure of the system involves different methods, which produce results of greater complexity. One proposed second-stage method is differentiation.

If the definition of an object is found to be too amorphous to yield the correct behavior, differentiation can produce separate and more distinct types of object. The differentiated types will tend to be closely coupled and capable of interacting with the same operations that the undifferentiated type allowed for interactions of objects of that type. Where they differ in behavior is that area of operations or attributes that required the split. This type of tightly coupled interaction between different types is produced automatically in subtypes of the same base type, through inheritance. Subtypes, however, are very

tightly coupled, and can only differ from each other in terms of ranges (numeric or discrete subtypes), numbers of components (constrained array subtypes) or discriminant values (constrained record subtypes).

If the differentiation is more extensive, requiring objects of differently structured base types, then all of the allowed interactions between the objects must be defined more laboriously. The rewards of this diligence, which are unique to Ada, are the isolation of system complexity to a package defining all the closely coupled interactions, while the programming using these types and operators can proceed at a higher level.

## OBJECT LIFE CYCLE

Definition of a system down to the fine structure produces a definition that is no longer intuitive, and requires some non-intuitive method for it's verification. The life cycle of an object may prove to be useful in providing a path to follow, in the analysis of complex objects.

All objects have their own life cycle, however brief, in the system environment. They are created and deleted by an operation or system event, either explicitly or implicitly. During their life they interact with other system objects, with results dictated by the appropriate operations.

The verification of the results of object definition can be performed by a "walkthrough" of the object life cycle. During this process, the defined attributes and operations of the object can be evaluated in the light of the events it experiences: creation, interactions and demise. If, under these circumstances and within the scope of the requirements, the abstract object behaves similarly to the phenomena which it is intended to represent, then the object with it's attributes and operations can be expected to reliably support the development of applications concerning that phenomena.

The Ada language features which directly support the definition of objects are packages and private types. Packages contain the definition of the object and allowable operations, which are visible, and the implementation, which is hidden. Private types further close the window of visibility, allowing only higher-level or interface attributes of the object definition to be visible

## TWO CLASSES OF ABSTRACT TYPES

For the purposes of the assembly of software components, there appear to be two broad classes of private types. The types which support the definition of objects as discussed above are called, only for the purpose of distinction, atomic abstract types. These types represent the discrete phenomena which are the subjects of system development, and are defined in packages as private types. They have the indivisible property of atoms, and can be incorporated into the second class of types: the generically structured abstract types.

Generically structured abstract types are managed by generic software components (packages or subprograms), and are built-up from application-defined types which are contained as components of the generic structure. These structured abstract types organize the objects of the system in a manner representing the relationships between objects, and they shall be discussed first.

## GENERICALLY STRUCTURED ABSTRACT TYPES

These structures, built-up from application-defined atomic abstract types and managed by generic packages, support the basic organization of the elements of the system. The organization of objects in a structure is a representation of the relationships between those objects, which can be either static or dynamic in nature.

The specification of a generic package is parameter driven. The generic formal parameters of a generic package are the basis and controlling factor in the reusability of the package. The use of generic software has implications, however, for the design of atomic abstract types which are later to be used in an instantiation of that software. The benefits of reusability can only be fully realized if the design of atomic abstract types follows distinct lines.

Taking an example of a generic sorting routine, it can readily be seen that the reusability of the routine is dependent upon the initial typing of the generic formal parameters and the matching rule for generic formal parameters. If the parameter is typed as simply private, then the maximum reusability is achieved, because it will match nearly everything (except discriminant or limited private types). However, if the parameter is typed as a real (digits <>) or integer (range <>), the operations that are consistent with those types will be available to the internals of the generic, but at the expense of only allowing those types as parameters.

It should be noted here, that although a generic formal parameter of the limited private type would extend the generality of the generic software component, it is not useful due to the lack of both assignment and compare for equality within the generic. Without assignment, components of the structure cannot be set, or initialized to any value.

The concept of generic programming turns private types and visibility inside out. In the case of a generic package, the structure of a type passed as a formal parameter is not visible to the package which manipulates it.

In the support of generic structures, typically all that is needed is the assignment function ":=", the compare for equality function "=", and an ordering function ">". The assignment and compare functions are available with type private parameters, and the ordering function ">" can be passed as another formal parameter. With no other details or operations, structures like lists, queues, indexes, and hierarchical tree structures containing objects of the generic formal parameter type can be defined and maintained by the package.

## ATOMIC ABSTRACT TYPES

The atomic abstract types are the components which fit into the generically-structured abstract types, during the assembly of software components. As such they must be crafted to fit easily into the generic structure.

As has been noted, generic formal parameters of the maximum range of applicability are those of the private type. The problem then is to design atomic abstract types that match the simple model of the private type: assignability and comparability.

Discriminant types, although very useful on their own for the development of objects with constraining attributes, are fairly disfficut to use in conjunction with generic software. Very quickly it is found that, to match a discriminant type with a generic formal parameter, the types for each individual constraint must first be passed as generic formal parameters. Then the discriminant type must be passed with it's constraints. Unconstrained types are not allowed. Generic formal parameters of this combination should be fairly difficult to match with any type other than the type initially matched, making for extremely reduced reusability.

Access types, which are the foundation of the dynamic structure of generically structured abstract types, are of little use in constructing atomic abstract types. They perform the assignability function more or less according to the simple model of private types, however they do not create a copy of the designated object (object pointed to), but instead copy the access object value (pointer address) onto the new object. This creates a shared object, with a certain loss of object identity, and could cause integrity problems inside the generic structure which incorporates the access object as a component.

The ordering function ">", used to order the elements of a generic structure (index, tree), can be defined by overloading the ">" function for the access object, to create a function comparing the designated objects values (for a string access type, the ">" would compare the designated strings).

The compare function "=" is another matter, however. It exists for access types, but compares the values of the access objects to see if they designate the same designated object. The "=" can only be overloaded if the abstract access type is declared as limited private instead of private. When this is done, however, the assignment operation ":=" is lost (and cannot be overloaded), which is needed for internally manipulating the generic structure inside the generic package.

Embedding the access type in a non-discriminant record would not change the reference nature of the contained object, and the problem of compares.

Embedding discriminant types, however, is very successful. As long as the constraint is not needed for data validity, this technique can hide the discriminant type within a non-discriminant record. The non-discriminant record will match a generic formal parameter of type private. This allows, for

instance, a variable string (unconstrained array type), to be contained within a non-discriminant record, and passed to generic procedures easily.

## DESIGN GOALS AND CONSTRAINTS FOR ATOMIC ABSTRACTION

In the process of feasibility prototyping for the generation of application independent software support environments, the following design goals and constraints were found to yield, for packages supporting atomic abstract types, the maximum in abstraction, flexibility, and potential for generic structure incorporation:

1. Package-defined atomic objects being declared in the application software should, where possible, be defined as abstract types, that is, made private.

2. If the operations of an object are analogous to those of standard objects already in the system, overload the same names for the operations. This enhances readability and learnability of the application software support environment. Do not, however, overload names with non-analogous functions.

3. The functions performed by the operations of an object should be intuitive. The action performed by an operation should be predictable from the context of the application software.

4. The outcome or result of operations of an object should be intuitive. The kind of object produced by operators, for example, should be predictable from the context of the application software.

5. Maximize the completeness of the application interface to the atomic type defined in the package. Give the application developer all of the operations required to manipulate and combine objects, in an easy-to-use yet well controlled manner.

6. Maximize the potential use of reusable software incorporating the abstract atomic type into generically structured types. This can be accomplished by defining types that perform simply under the operations of assignment and comparison (not discriminant types or access types, which follow a more complex model).

## DESIGN GOALS AND CONSTRAINTS FOR GENERIC ABSTRACTION

In the process of feasibility prototyping for the generation of application independent software support environments, the following design goals and constraints were

found to yield the maximum in reusability and flexibility for packages managing generically structured abstract types:

1. Package-managed generic objects that are declared in the application software should, where possible, be defined as abstract types, that is, made private.

2. Maximize the generality of the package. This comes from the use of formal generic parameters, particularly for types, that match the widest variety of application input types (type private instead of digits <>, for example).

3. Maximize the usability of the application interface to the package. Extend, as far as possible into the application domain, access to the structures managed in the package, without violating the integrity of the internals, or the independence of the application from the generic software component (generality).

4. Maximize the completeness of the application interface to the package. Give the application developer all the operations required to access and manipulate the internal structures, in a package-controlled manner.

5. Support, if possible, multiple objects with the same package. This limits the need to re-instantiate the package several times within the same scope, for processing of multiple objects.

6. Design for flexibility: a single tool, suited to a wide range of applications, is more likely to be remembered, and used by developers.

7. Cover the infrequent failure modes. Most failures of algorithms and processing logic in programs occur at the extremes of their domain of applicability. Testing should cover the ends of ranges and the infrequent states of the application. If the software component is reusable, it will be used in a wider range of applications, and the infrequent failure modes will occur more frequently.

## PACKAGES SUPPORTING GENERICALLY STRUCTURED ABSTRACT TYPES

The index package, described as a list of elements ordered by another set of associated elements or keys, will be used as an example for a package supporting a generically structured abstract type. The index structure itself should be a private type. It should be defined in the package specification, not hidden, so that it can be declared as an object in the scope of the application. The package should be capable of accessing and managing several objects of type INDEX, so there should be a USE_INDEX function, which selects the appropriate object, and

sets a package-internal access object to the same value as that passed as the USE_INDEX parameter. Then there will be two access objects pointing to the index structure internals, one in the application scope, one in the package scope.

Since the access object in the application scope cannot be changed, neither can the access object in the package scope (unless there is a subsequent USE_INDEX call). They must stay aligned. This means that the INDEX access object cannot designate the head of the index-list, but must instead designate an access object that designates the head of the index-list. This is in case an insert must be made at the head of the index, and the access object that designates it must be modified.

The importance of having the index object in the scope of the application is in the flexibility of use of the object at the application level. The developer should be capable of passing the object as a parameter to subprograms developed at the higher level. If the object of type INDEX is hidden, this flexibility is not there.

The indication of success or failure of an operation (add/delete, search, etc.) should be available for the application, for the purpose of logical tests and conditional branching. It should be contained in the package scope, visible in the package specification, and it can be called STATUS. Values contained in status can be defined in the package specification to show conditions (END_OF_LIST, ELEMENT_NOT_FOUND, etc.).

## CURRENT-NODE POINTER FACILITY

One question about package operations that must be answered before the design phase is about the context-sensitivity of operations. Higher level operations, like those involved in command languages, are typically constrained to be context insensitive, on a line-by-line basis. This means that the interpreter of the command or function requires no information, other than that in the command, to interpret it completely. There is no contextual basis.

This can be effectively at a high level of application, but is difficult for the implementation of any complex functionality. For the package managing a complex structure, it is really necessary for the package to keep a contextual indication of the current position of the search through the structure in between calls. A USE_INDEX call to a new index would reset this position indicator, of course, as would any search, add/delete, or sequential positioning call. This prevents the need for a node search upon every call. This position indication variable can be called CURRENT.

CURRENT is of necessity an access object. If CURRENT is kept in the application scope, it must be passed in the subprogram interfaces of every operation. Also, being in the application scope, synchrony can be lost between USE_INDEX calls (pointing to the wrong INDEX designated structure).

If CURRENT is kept in the package, the package can track application context, and reset CURRENT upon USE_INDEX

invocation. Also, it should be hidden, because it would be difficult for the application to interpret it anyway.

With these design issues decided, a generic package for managing INDEX objects can be developed.

## REUSABILITY ISSUES

Reusability is generally discussed in terms of taking software written at other sites, and not necessarily on the same machine, and porting it for use in an application. There is a context here, which can be called inter-project reusability. This kind of reusability is based on two types of software development.

In the first type of reusable software, software components or interfaces to non-Ada components are produced for general application support areas, like DBMS, user interface software, graphics, communications, data reduction and others, even AI. These will certainly be necessary to include, as they are more expensive to develop than to buy. They will also be the most commercially available.

In the second type of reusable software, and with far less availability, software components are written targetting the application area of interest. These will probably be less of a fit to the specific application, with fewer packages to choose from.

In the I & T area, high performance software is hard to obtain, and will be in the future. This is due to the narrow market and the very high degree of system dependence of the applications developed. In application domains with parameters like those of I & T, the major gains in Ada reusability will be those derived from software designed and developed in the same project.

This kind of reusability can be called intra-project reusability, and comes from design by abstraction. High level software can be produced for specific application domains by the production of packages tailored to support those domains.

Packages implementing private types can be developed that support the objects and operations representing the phenomena which are the subject of system development. If these objects and operations simulate the behavior of those phenomena well (within the purposeful domain), then the applications developed using them will be higher level, and generally more effective and maintainable.

Generic packages can also be developed supporting the static and dynamic relationships between objects in the system. If these packages can be made flexible and with maximum reusability, then the objects of the system can be organized by instantiation of those packages, allowing the system relationships to be established on a high order level in a logical way.

The reuse of both sets of software can be enhanced by establishing design constraints on each, so that the software components of the system can be assembled with maximum

likelihood. The design goals and constraints on Ada software can not be effectively left as an afterthought.

## PROJECT-SPECIFIC SOFTWARE SUPPORT ENVIRONMENTS

The effectiveness and reusability of software generated by relatively inexperienced Ada programmers will be directly related to the project-specific software environment that exists when they first enter the project. It will always be found that it is easier, quicker and more reliable to construct anything from pre-fabricated components that fit together as well as Lego blocks do. Two things are required to build a good set of blocks.

First, the objects (the logical atoms and molecules of the system) and their operations must be represented well by packages supporting those atomic abstract types and all of their support functions. Secondly, the relations organizing the objects of the system must be supported with generic packages that are flexible and easy to use.

In the internals of both of these packages can be buried the hidden complexity of the system, and some of the system dependencies as well. In this way, technology insertion into the system can be accomplished directly, without negatively affecting the applications of the system. [2]

Finally, a good set of blocks is not sufficient to build a system. The builder has to know what he is building to be effective. There is no substitute for Requirements Analysis and Functional Decomposition using data flows and similar techniques to express what a system does in a manner traceable back to the original User Requirements. The development of Ada and the object-oriented design methodologies which Ada directly supports will eventually prove, however, to be a large step on the way to cracking the problem of what to do after the System Requirements are assigned to the top-level components of the system.

## ACKNOWLEDGEMENT

I gratefully acknowledge the support given by the Kennedy Space Center/ Engineering Development/ Systems Integration Branch in supplying the computer facilities for the feasibility studies that provided the basis of this work. I also thank my wife, Bronwen Chandler, for her support.

## REFERENCES

1. Johnson, C., 1986. "Some Design Constraints Required for the Assembly of Software Components: The Incorporation of Atomic Abstract Types into Generically Structured Abstract Types", Proceedings of the First International Conference On Ada* Programming Language Applications For The NASA Space Station, F.4.4.

2. Johnson, C., 1986. "Some Design Constraints Required for the Assembly of Software Components: The Incorporation of Atomic Abstract Types into Generically Structured Abstract Types", Proceedings of the First International Conference On Ada* Programming Language Applications For The NASA Space Station, B.4.3.