ROMAN-9X: A TECHNIQUE FOR REPRESENTING OBJECT MODELS IN Ada 9X NOTATION

Gary J. Cernosek
Fastrak Training Inc.
2224 Bay Area Blvd., M/C 6-W
Houston, TX 77058
(713) 280-4768
cernosek@source.asset.com

Abstract

An object model describes the objects and classes in a system and their relationships among one another. When the desired implementation language is Ada 83, the software designer cannot directly implement the inheritance relationships of the model since the language does not provide sufficient features for doing so. Ada 9X will provide, among other things, direct support for single inheritance, making the task of representing object models much easier. The purpose of this paper is to present a technique for Representing Object Models in Ada 9X Notation (ROMAN-9X). An example object model with single inheritance is used to illustrate the technique, demonstrating how tagged types can be used in conjunction with child library units. A set of standard naming conventions is also presented as part of a canonical model to use for representing classes and single inheritance hierarchies in general.

1. Introduction

Many object-oriented development methods have been defined and are now in wide-spread use [e.g., Rumbaugh, et al, 1991; Coad and Yourdon, 1991a,b; Shlaer and Mellor, 1992; Booch, 1991; Wirfs-Brock, et al, 1990]. Although the terminology still varies somewhat across different methods, each method includes the notion of an *object model* as one of its artifacts. An object model describes the objects and classes in a system and their relationships among one another [Rumbaugh, et al, 1991]. While the notation used to build object models does vary more noticeably across different methods, the resulting descriptions are largely independent of any particular programming language or database implementation.

Several authors have developed techniques for representing object models in the Ada language [e.g., Atkinson, 1991; Booch, 1983, 87, 91; Coad and Yourdon, 1991b; Meyer, 1988; Rumbaugh, et al, 1991; Seidewitz and Stark, 1992].

Ada has indeed been a key technology in spurring the evolution and growth of Object-Oriented Design (OOD) methods over the past ten years. The language directly supports and enforces the principles of abstraction, information hiding, and encapsulation, all of which are key to any "object-based" programming language.

However, the current version of Ada, known as *Ada 83*, lacks direct support for inheritance and dynamic polymorphism, two additional features required for any language to be considered a true "object-oriented" programming language (OOPL) [Cardelli and Wegner, 1985]. This deficiency has forced Ada software designers to develop special implementation techniques that contrive the representation of these aspects of an object model. The absence of inheritance alone has become most impacting to the Ada community since the ability to generalize and specialize classes is now an integral part of most OO analysis and design methods. Consequently, the many strengths of Ada as a whole have been forsaken by the overriding interest in using a more true OOPL, the most notable being C++.

The revision to Ada 83, currently referred to as *Ada 9X*, includes features that directly support single inheritance and dynamic polymorphism. These features promote Ada to the status of a true OOPL. The revised language goes further to improve on its already rich set of design structuring mechanisms, making Ada even more suitable for large-scale OOD efforts. The new features of Ada 9X, combined with those already well established in Ada 83, will enable software designers to represent object models in new and more effective ways.

The purpose of this paper is to present a technique for Representing Object Models in Ada 9X Notation (ROMAN-9X). The paper builds on well-established OOD methods that have evolved with Ada 83 over the past ten years. An example object model with single inheritance is used to illustrate the technique, demonstrating how *tagged types* can be used in conjunction with *child library units*. A set of standard naming conventions is also proposed as part of a canonical model to use for representing classes and single inheritance hierarchies in general.

©1993 ACM 0-89791-621-2/93/0009--0385 1.50

2. Technical Foundation

A great deal of object-oriented literature exists from which to base a paper such as this. Our introduction cited several leading works in the field. However, our paper draws primarily on three particular resources: *Object-Oriented Modeling and Design* [Rumbaugh, *et al*, 1991], *Object-Oriented Design With Applications* [Booch, 1991], and *Object-Oriented Reuse, Concurrency, and Distribution - An Ada-Based Approach* [Atkinson, 1991]. All three texts address OO concepts and techniques for representing them in Ada 83. While being very complementary in many respects, each source provides discriminating strengths in selective areas.

Rumbaugh *et al* are particularly attentive to requirements analysis, an area still lacking in many OO methods. Their *Object Modeling Technique* (OMT) also provides for a simple graphical notation with which to work, and has in fact been adopted for this paper. Most notable from Booch is his presentation on OO concepts and the depth at which he addresses object-oriented design (proper). Booch also brings forth his legacy of previous works that address OOD with Ada 83 [Booch, 1983, 87]. The Atkinson book provides an even stronger treatment of mapping OO concepts into Ada 83. Atkinson's involvement with the DRAGOON language [Di Maio *et al*, 1989] and his recent (though unpublished) work with Ada 9X are largely responsible for the naming and structuring techniques used in this paper.

At the time of writing (July 1993), the definition of Ada 9X itself remains in the process of ANSI and ISO standardization. This paper is based on the latest documentation available. The *Ada 9X Mapping Documents* [Intermetrics, 1991, 92] and *Introducing Ada 9X* [Barnes, 1993] served as our initial tutorials into the language. Drafts of the *Ada 9X Reference Manual* [Intermetrics, 1993] were just becoming available but were not consulted in detail. Note, however, that all Ada 9X source code presented in this paper has been compiled by one of the testing compilers in use with the Ada 9X Project. Therefore, our code examples should be free of syntax errors, notwithstanding further changes in the language definition.

Finally, since Ada 9X is very upward-compatible with Ada 83, many features of the two languages are identical in both syntax and semantics. For such features, we will often reference "Ada" without qualification. For features that are new or modified for Ada 9X, we will specify the particular version of the language.

3. Reference Object Model

Figure 1 presents the object model that will be used as the basis for illustrating our technique throughout this paper.

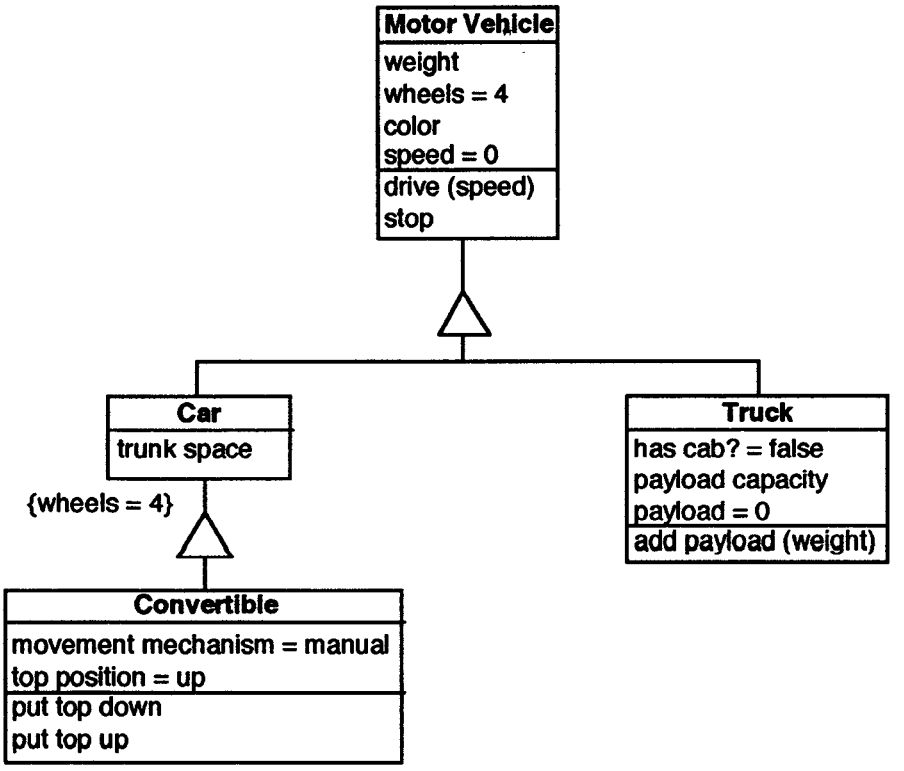

Figure 1. Reference object model.

The reference model depicts a single inheritance hierarchy of different kinds of motor vehicles. Note that we adopt Rumbaugh's Object Modeling Technique for our notation. We now highlight the features of OMT that will be addressed in this paper, italicizing key terminology.

Each box in the diagram models a *class* of objects identified in our example problem domain. Within each box, the centered and bolded identifier designates the *class name.* Partitioned in each box are sections for the class's *features,* which are divided into a set of *attributes* and a set of *operations.* Some attributes are defined to have *initial* or *default values.* A *constraint* is shown in braces next to the Car class denoting that all Car instances must have four wheels. Some of the operations are defined with *arguments* to be exchanged during *message passing.*

The triangles attached to the lines connecting the classes denote *inheritance* relationships. The classes Car and Truck are called *subclasses* of the Motor Vehicle *superclass.* Since Motor Vehicle has no superclass of its own, it is referred to as the *root class* of the *inheritance hierarchy.*

Figure 2 depicts several actual objects denoted by the rounded boxes. An *object* is an instance of a class; the word *instance* can be used interchangeably with object in this context [Booch, 1991]. When necessary, Rumbaugh uses the term *object instance* to distinguish it from *object class.*

Each instance identifies its corresponding class name in bold type and in parentheses at the top of the box. A name is given to the instance just below the class name. All attributes defined for each instance are given initial values to be provided upon object creation. Note the effect of inheritance: Instances of subclasses implicitly take on all of the attributes defined by the subclass itself, plus those defined by its superclass(es). While operations are also inherited by subclasses in the same manner, they are not shown on instance diagrams.

The remaining sections of this paper illustrate and explain how our reference object model is represented in Ada 9X notation using the ROMAN-9X technique.

4. The Technique

We present ROMAN-9X in three stages. We first demonstrate how a root-level class can be implemented with a specialized abstract data type. We then illustrate how ordinary *clients* (users) of root classes create objects and send messages as a result of the technique. Lastly we show how subclasses and their clients are represented, introducing some of the new OOP features of Ada 9X in the process. Throughout our discussions, we will continue to italicize key terminology upon first usage.

4.1 Representing Root Classes

Figure 3 begins our example by showing the complete package specification that represents our Motor Vehicle root class. Upon examination, we see that code must be generated to represent both explicit and implicit features of an object class. The following subsections explain how each of these features maps into our example Ada 9X code.

4.1.1 Mapping Object Class Names

First note that Ada 9X does not introduce a new "class" construct, per se. In fact, we represent a root class from an object model in basically the same way as we would in Ada 83: by defining a private type in a package. Such a package is, of course, the Ada implementation of an Abstract Data Type (ADT) and is the basis for many OOD methods targeted to Ada 83.

Our technique maps the object class name "Motor Vehicle" directly to a corresponding library package name to enforce a "class = module" semantics. An ADT package structured in this way is thus referred to as a *class package.* While all class packages are considered to be ADTs, not all ADTs will necessarily be class packages; a class package must be traceable to a corresponding object class.

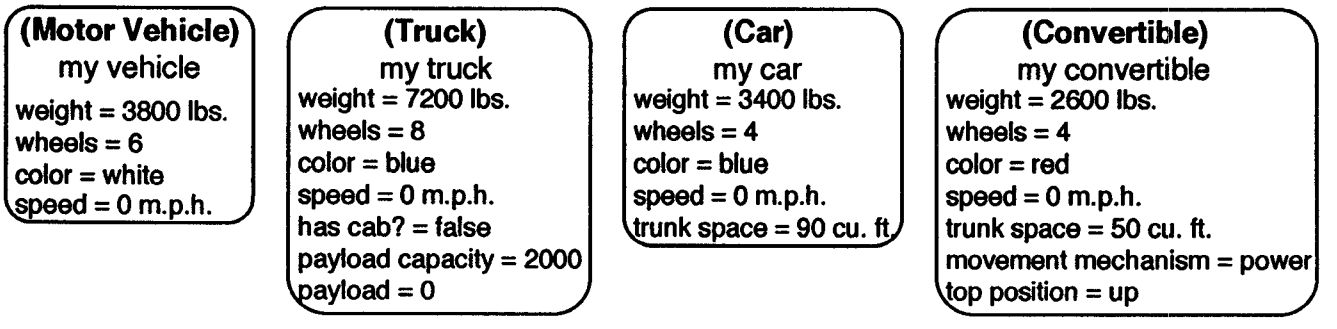

Figure 2. Example instances of classes from the reference object model.

```
package Motor_Vehicle is

   -- Standard class type declaration for a root class
   type Object is tagged limited private;

   -- User-defined attribute types
   type Vehicle_Weight is range 0..20_000; -- lbs.
   type Wheel_Count is range 2..18;
   type Colors is (White, Red, Blue, Black);
   type Velocity is range 0..100; -- m.p.h.

   -- Constructor operation
   procedure Create (Instance : in out Object;
                     Weight : in Vehicle_Weight;
                     Color  : in Colors;
                     Wheels : in Wheel_Count := 4);

   -- Modifier operations
   procedure Drive (Instance: in out Object; Speed: in Velocity);
   procedure Stop  (Instance: in out Object);

   -- Selector operations
   function Speed (Instance: in Object) return Velocity;

private

   type Object is tagged
      record
         Weight : Vehicle_Weight;
         Wheels : Wheel_Count;
         Color  : Colors;
         Speed  : Velocity := 0;
      end record;

end Motor_Vehicle;
```

Figure 3. Representation of an object model root class.

The private type in a class package represents the type implicitly defined by the corresponding object class. Hence, we refer to the private type as the *class type* [Meyer, 1988; Atkinson, 1991]. We name this type "Object" in our example, and in fact will do so for every class package. Client references to different class types will never be ambiguous since such references will always be of the form:

class_package_name.Object

This convention eliminates the need to contrive distinct names for both a class package and its private type, a practice that often results in two names artificially distinguished by either pluralization or abbreviation. Thus, our technique enforces a "class = type" semantics as well.

Some further discussion on terminology is in order. While many object model terms map unambiguously into corresponding Ada 9X terms, some do not. The most notable in this paper is the term "class." In an object model, a class defines a set of objects that share a common structure and behavior. In both Ada 83 and 9X, a *type* takes on this meaning, defining a set of values and a set of operations on those values. A *private type* hides the representation of a type and thus enforces a cohesion between the values and operations. Ada 9X then defines a *type class* to be a "set of types" whose members include a root type and all of the types derived directly and indirectly from that root type.[1] The Ada 9X notion of type class corresponds more to an "inheritance hierarchy" in object modeling terminology.

For the purposes of this paper, unqualified use of the term "class" will refer to a class from an object model. The term

[1] Ada 9X introduces a new attribute T'CLASS to denote the entire set of types rooted at the type T.

"class package" will be used to denote our Ada 9X representation of such a class. We will use Rumbaugh's notion of "object class" when the context is not clear.

4.1.2 Mapping Attribute Features

The full representation for the class type will always be implemented in the private part of the package as a record type. The names used for attributes in the object model map directly to the components of this record structure. These record components define the state of objects declared of the class type. Note that each attribute must be declared with an associated type of its own. If a type name is defined for an attribute in the object model, then that name is used as the attribute type name. Otherwise, an appropriate type name must be conceived.

While Ada's predefined types are available to use, our technique generally takes advantage of the language's rich data typing facilities and declares appropriate user-defined types for attributes. Since such types are also needed to specify the parameter and result profiles of operations, user-defined attribute types are necessarily declared in the visible part of the class package. Note that as in [Atkinson, 1991], we make the distinction between a "class type," which denotes the private type associated with a class package, and an ordinary "data type," which is used to define the structure of attributes and other general forms of data.

Our technique provides two means by which initial or default values on attributes may be represented. When all instances of a class are to be created with the same initial value for a particular attribute, that attribute is given a default value in its corresponding record component declaration. In Figure 3, we see that all instances of Motor Vehicles are to begin with their `Speed` attribute initialized to 0. However, for an attribute whose initial value may vary from instance to instance, the initialization must be explicitly performed by a *constructor* operation. The attributes `Weight`, `Color`, and `Wheels` fall into this category. Note that a default initialization may still be provided for such attributes by simply specifying the default value in the constructor's parameter list. This is done with the `Wheels` attribute in our example. We discuss more on constructor and other operations in our next subsection.

4.1.3 Mapping Operation Features

Operation names in an object model map directly to subprograms declared in the visible part of the corresponding class package. In Figure 3, we see that the operations `drive` and `stop` are represented as Ada procedures. By virtue of the private type, normal clients of a class package must call on subprograms to access or manipulate the components of the object record.

Note that we choose to implement class types as *limited* private. The non-limited form would allow the predefined assignment and equality operations to be performed between two instances of the type. However, such operations may not always be meaningful from an application perspective. The limited form requires that even assignment and equality operations be explicitly defined in the class package (assuming that such operations are indeed needed). This approach ensures that the behavior of corresponding objects is precisely that which is intended by the class package designer.

An important characteristic of an object model is that a class implicitly defines the target object of each operation. It is understood that a call to an operation will be preceded by a specific object instance name. Thus, when a parameter and result profile is specified for an operation, a target object argument is never included. In Ada, however, a class package is not a type itself but merely a container for one. A class type declares only the attributes for the object class; the operations must be declared in the same class package, but are still syntactically separated from the class type. This requires us to explicitly include a target object parameter for all operations on the class type[2].

Therefore, each operation represented using ROMAN-9X will contain at least one argument in its parameter profile, that being the one indicating which particular object is to be acted upon by the operation's *method* (implementation). By convention, we will always place this argument first in the parameter list, and we will always name the formal parameter "Instance." Note that the Instance argument is analogous to the pseudovariable *self* used in Smalltalk methods and to the implicit argument *this* used in C++ methods.

By convention, we organize the operations of a class into several of the categories suggested in [Booch, 1991], using comments as shown in Figure 3. A class may define *constructors* to initialize objects, *modifiers* to change the state of objects, and *selectors* to simply read the values of an object's attributes.[3]

Note that an object model may not identify all of the operations that are needed for a complete class representation. For example, our reference object model specifies only two (modifier) operations. The constructors

---

[2] Any of several other Ada features could be used to represent classes that operate on an implicit target object, including generic abstract state machines, task types, and, from Ada 9X, protected record types. However, none of these constructs readily support inheritance and other OOP features. This issue is addressed in detail in [Atkinson and Weller, 1993].

[3] Note that [Booch, 1991] also identifies *destructors* and *iterators* as categories of operations, but we do not address the representation of such operations in this paper.

Object Model Element	ROMAN-9X Mapping Rule to Ada 9X Code
Class name	Class package name
(implied class type)	`type Object is tagged limited private;`
Attribute names	Object record component names
Attribute type names	Object record component type names (public type declarations will generally be needed for user-defined attribute types)
Initial and default attribute values	1) Initial values on object record components, and/or 2) Parameters (w/optional defaults) on constructor operations
Operation names	Class package subprogram names
(implied target object for operations)	`Instance : in [out] Object`
Operation argument names	Corresponding subprogram parameter names (use attribute names and corresponding types when appropriate)
(implied constructor operations)	`procedure Create (Instance: in out Object; ...);`
Operation return type names	Corresponding function return type names (use attribute type names when appropriate)
(implied direct selector operations for reading attributes)	`function <attr-name> (Instance: in Object)` `return <attr-type>;`

Table 1. Summary of mapping technique for representing root classes in Ada 9X.

and selectors for an object class are often suppressed from an object model. Details of parameter and result profiles may also be omitted. Deferring such information is very common and quite often appropriate, especially during the early stages of a system's lifecycle. However, the class package must ultimately incorporate all of the various kinds of operations needed to complete the representation.

As stated earlier, an initial value for an attribute may be defined along with its record component declaration. However, since the record type is private, any initialization to be performed by a client must take place via a constructor operation. In these cases, a constructor must be specified in the class package regardless of whether its object class calls for one or not. By convention, all constructor operations will be named "Create" and will be implemented as procedures.[4]

We also discussed earlier how attribute names from an object class map directly to components in the corresponding object record. These same names are used as formal parameter names whenever an attribute needs to be included in an operation's parameter profile. These attribute names can also be used whenever simple selector operations are needed, as shown for the `Speed` attribute in Figure 3. Note that our technique does not call for automatically generating selector functions for each attribute (although doing so would be very straight-forward and systematic). We introduce selectors in this paper only when our client examples demonstrate the need.

The only aspect of our Motor Vehicle ADT that differs from one which might be implemented in Ada 83 is the designation of *tagged* for the private type declaration. Ada 9X introduces tagged types as its means for extending derived types with additional components, paving the way for a single inheritance mechanism. Tagged types also instruct the compiler to automatically encode type information into objects declared from them, thus providing the means to implement dynamic binding of methods to messages. We will discuss the effects of tagged types later in more detail when we address the representation of inheritance.

The naming conventions described in this section are intended to help map object models more directly and systematically to their code representations. Adhering to these conventions significantly reduces the number of unique identifiers that must otherwise be conceived. Table 1 summarizes the mapping technique and naming conventions used in ROMAN-9X.

[4] We will discuss why procedures are used over functions in Section 4.2.

```
package body Motor_Vehicle is

   procedure Create (Instance : in out Object;
                     Weight : in Vehicle_Weight;
                     Color  : in Colors;
                     Wheels : in Wheel_Count := 4) is
   begin
      Instance.Weight := Weight;
      Instance.Color  := Color;
      Instance.Wheels := Wheels;
      -- Speed attribute has already been initialized to the default value
      -- specified in the object record declaration.
   end Create;

   procedure Drive (Instance: in out Object; Speed: in Velocity) is
   begin
      Instance.Speed := Speed;
   end Drive;

   procedure Stop  (Instance: in out Object) is
   begin
      Instance.Speed := 0;
   end Stop;

   function Speed (Instance: in Object) return Velocity is
   begin
      return Instance.Speed;
   end Speed;

end Motor_Vehicle;
```

Figure 4. Implementation of a root class body.

While a full treatment of object-oriented programming in Ada 9X is out of the scope of this paper, it is critical that the basic implementation of classes be presented and understood. Figure 4 presents the body of the Motor Vehicle class package where we see the method for each of its operations implemented. We will discuss more on class implementations later when we look at how subclasses specialize inherited operations.

4.2 Representing Root Class Clientship

Clientship refers to the fundamental relationship between objects and classes [Atkinson, 1991]. A *client* provides the place where object instances are created and where operations are invoked on those instances. The class from which an object instance is declared is called the *server* for that object. Clients may be considered the "users" of server classes and are thus responsible for the "application code" of an object-oriented system.

In Ada, an "object" refers to anything that has value associated with it. In most cases, an Ada object is an instance of some type. Our technique further constrains the notion of an "object" to being an instance of a private type. Objects are never declared within their own class package, but rather in some other program unit that depends on that class package (which could itself be another class package). Thus, a client may be represented in Ada by any library unit that specifies a class package in its context clause.

Figure 5 presents a simple program that serves as a client for the root class package previously shown. We see that the 'with' statement establishes visibility to the class package, affording the client access to the public declarations of package. The object `My_Vehicle` is declared as an instance of the Motor Vehicle class type. This object declaration performs three important functions:

1. Allocates the memory needed for an instance of the class type,
2. Initializes each attribute of the object record that has an initial value specified, and
3. Establishes a reference for the object (a variable name in this case).

```
with Motor_Vehicle, Text_IO;
procedure Motor_Vehicle_Client is

   My_Vehicle: Motor_Vehicle.Object; -- Any attributes declared with
                                     -- defaults now have those values.
   Current_Speed : Motor_Vehicle.Velocity := 0;

begin

   Motor_Vehicle.Create (Instance => My_Vehicle,
                         Weight   => 3800,
                         Wheels   => 6,
                         Color    => Motor_Vehicle.White);
      -- The vehicle's speed was initialized to a value of 0 m.p.h. upon
      -- declaration of the object above.

   -- Begin driving at 55 m.p.h.
   Motor_Vehicle.Drive (My_Vehicle, Speed => 55);

   -- Print out current speed
   Current_Speed := Motor_Vehicle.Speed(My_Vehicle);
   Text_IO.Put ("The current speed is");
   Text_IO.Put ( Motor_Vehicle.Velocity'IMAGE(Current_Speed) );
   Text_IO.Put_Line (" m.p.h.");

   -- Stop the vehicle
   Motor_Vehicle.Stop (My_Vehicle);

end Motor_Vehicle_Client;
```

Figure 5. Client code utilizing a root class package.

Note that any object record component declared without a default will have an undefined value upon object declaration.[5] While syntactically allowed, any operations attempted on objects with undefined attribute values are technically erroneous and thus subject to unpredictable behavior. As discussed earlier for these cases, the class designer must provide a constructor for client initialization of such attributes. Of course, we could have our technique simply require that all object record components be given a default initial value. However, choosing a default value can in many cases be quite arbitrary, and in fact, sometimes be outright misleading. So as a general rule (not enforceable by the compiler), each attribute defined in an object class must either be given an initial value in its object record declaration, or be included in the parameter profile of a constructor operation.

Unfortunately, Ada 9X does not provide for automatic constructor invocations.[6] Thus, clients are themselves responsible for ensuring that a constructor is called before any other operation. It would be ideal to perform the initialization immediately upon declaring the object. Normally this could be done by specifying the constructor operation as a function subprogram and calling it as such to initialize the object. However, since we declare class types as limited private, assignment is not allowed, and thus, immediate initialization at object declaration time (even via a function call) is not possible. This is why we must declare all constructor operations as *procedure* subprograms.

Once an object instance is created and initialized, we can send *messages* to it. In Ada, messages are sent to objects simply by calling on the normal operations defined for the object's type in the corresponding class package. Since the subprograms to be called are declared in a library unit, references to them are preceded by their enclosing package

[5] Ada's only exception to this is that *access* objects are automatically initialized to `null`.

[6] Ada 9X does address the notion of initialization and finalization, but only for task and protected objects.

name. Our naming conventions have thus resulted in a message passing syntax that takes on the form:

```
class_name.operation_name (target_instance,
                           other_parameters);
```

Note the effect of the explicit target object: Each message sent must explicitly indicate which particular object is to be acted upon. See Figure 5 for examples of message passing.

We need to make a few remarks regarding the identity semantics of objects. In our Motor Vehicle Client, the object declaration for `My_Vehicle` allocates the actual memory that is to be used for the instance. Objects such as this are said to exhibit *copy semantics* in that a copy of the actual object record structure is created upon each object declaration. We could alternatively include an access type declaration in the class package that designates the class type:

```
type Reference is access Object;
```

Instances of this access type (which may be referred to as *instance variables*) would then be used to reference instances of the actual class type. The resulting *reference semantics* enables the use of dynamic memory allocation and other OOP features.

The meaning of assignment and equality is obviously quite different when using reference semantics. An assignment between two instance variables creates an *alias*, resulting in two references to the same object. The equality operation tests to see if two instance variables are indeed referring to the same object. Using both copy and reference semantics within the same context can be quite confusing if not properly managed.

Further exploration of reference semantics can become quite complex very quickly. So for simplicity, all objects created in this paper will exhibit copy semantics.

4.3 Representing Subclasses and Inheritance

A *subclass* is a specialized version of its *superclass* (also called its *parent class*). *Inheritance* is the mechanism that allows us to create a subclass by specifying only the differences between it and its superclass. One of the key benefits of creating subclasses via inheritance is that the original superclass need not be modified in the process. This allows software components to be very reusable and extendable.

As was briefly mentioned earlier, *tagged types* provide the means for representing single inheritance in Ada 9X. However, in order to properly modularize each tagged type, we will introduce another new feature of the language, *child library units*. Figure 6 illustrates how a subclass package can be constructed by declaring a new tagged type within a child library unit.

Note how the package name is expanded for child library units, depicting the parent-child hierarchy. Child units offer significant control over the visibility between packages, and are in fact quite instrumental to our technique. However, we will first address the inner workings of our subclass package and explore the visibility characteristics of child units later in detail.

4.3.1 Inheriting Superclass Features

Ada 9X supports inheritance through the facility of *derived types*. A derived type has the effect of making a "copy" of another type (called the *parent type*). We see in Figure 6 that the Truck package derives its class type from that of the Motor Vehicle. Thus, the same record structure defined for Motor Vehicles is implicitly redeclared for Trucks, but as a distinct type of its own.

One of Ada's fundamental notions is that a type defines both a set of values and a set of operations. Thus, when a type is "copied" via derivation, the operations already defined for the parent type are copied as well. That is, they are implicitly redeclared and made type compatible with the derived type. In our subclass example, all of the operations on Motor Vehicles (Create, Drive, Stop, and Speed) automatically become callable by Truck clients, as if they were all respecified at the place of the type derivation.

Of course, simply creating a "copy" of a superclass package in this way is of little use. The main purpose of inheritance is to *specialize* the features defined by a superclass. The subclass that results is in every way as capable as its parent, but generally even more so. Specialization is be accomplished by extending or modifying inherited features. The next few subsections discuss how ROMAN-9X represents various kinds of class specialization.

4.3.2 Adding New Attributes

One of the most common applications of inheritance is to extend the set of attributes defined for an object class. Simple type derivation has always been available in Ada 83 but has had little usefulness because of its inability to add new components to the parent data structure. Ada 9X has solved this problem with a natural extension to the standard type derivation scheme.

If a type is derived from a *tagged* type, then the derived type may readily "extend" the parent structure with additional components. Note that a tagged type must be a record type (or a private type whose implementation is a record type). Any type derived from a tagged type implicitly becomes a tagged type as well.

```
package Motor_Vehicle.Truck is     -- with Motor_Vehicle; is implicit

   -- Derived type declaration to inherit Motor Vehicle features
   type Object is new Motor_Vehicle.Object with private;

      -- All operations defined in the Motor Vehicle superclass have been
      -- inherited and are thus implicitly declared in this subclass package.
      -- These inherited operations may be directly invoked by clients of the
      -- Truck class on instances of this new class type.

   -- New attribute types
   subtype Payloads is Vehicle_Weight range 0..5_000; -- lbs.

   -- Specialized constructor for Trucks (becomes overloaded with inherited
   -- Create procedure)
   procedure Create (Instance : in out Object;            -- type declaration:
                     Weight           : in Vehicle_Weight;    -- from parent
                     Color            : in Colors;            -- from parent
                     Payload_Capacity : in Payloads;          -- from child
                     Has_Cab          : in Boolean := False;  -- predefined
                     Wheels           : in Wheel_Count := 4); -- from parent

      -- Note that two new attributes have been added as extra parameters.  Also
      -- note that visibility to inherited attribute types is direct.

   -- New modifiers
   procedure Add_Payload (Instance : in out Object; Weight : in Payloads);

   -- New selectors
   function Payload (Instance : in Object) return Payloads;

private

   type Object is new Motor_Vehicle.Object with
      record
         Has_Cab          : Boolean;
         Payload_Capacity : Payloads;
         Payload          : Payloads := 0;
      end record;

   -- Now instances of Motor_Vehicle.Truck.Object will have 7 attributes,
   -- the 4 that are defined for any motor vehicle, plus these 3.

end Motor_Vehicle.Truck;
```

Figure 6. Representation of an object model subclass.

In order to enforce the same encapsulation as that provided for the Motor Vehicle class package, we defer the actual type extension to the private part of the package. There we see that we need only to specify the new attributes for Trucks, knowing that those inherited by the type derivation will be implicitly created upon object declaration. Thus, instances of the Truck class type will contain the original four attributes defined for Motor Vehicles, plus the three new ones added here for Trucks.

4.3.3 Adding New Operations

New operations defined in a subclass enrich the behavior already inherited from its parent. For example, our reference object model requires a special operation for adding a payload to a truck. We thus declare a corresponding subprogram in our subclass package. While not specified in the object model, the package also declares a new selector operation for reading the newly added payload attribute

value. These new operations combine with those inherited to form a composite behavior that is (virtually) structured in one place.

4.3.4 Specializing Inherited Operations

Sometimes a subclass does not need to actually add a new operation but to rather provide a more specialized version of an inherited one. For example, the Create operation for Motor Vehicles has been inherited by the Truck class package and is therefore directly callable by Truck clients without respecification. However, three new attributes have been added for Trucks, two of which need client initialization. We could simply define another constructor, one that initializes only the new attributes. But this solution would require a client to make multiple constructor calls to complete an object initialization.

Therefore, we declare a specialized constructor, one that will perform all of the required initialization in a single call. Note how the constructor operation has been respecified with the same subprogram name, but with two additional parameters needed for initializing Truck instances. This technique is an example of *overloading* an inherited operation. Such overloading is very common in Ada and can be used for any operation so long as the compiler can unambiguously distinguish one call from another. This distinction is made by declaring each overloaded operation with a unique parameter and result profile. So long as two profiles differ by type, number, or order, the profiles are considered unique. Note that differences in formal parameter name, mode, subtype indication, or default value cannot be used as the basis for overloading operations.

A subclass package will always end up overloading its inherited constructors whenever it needs to provide for a different client initialization protocol. In the case of the Truck subclass, we need to initialize the newly added attributes, and thus must add two new parameters to the original constructor. We will later see that there are other situations where a specialized constructor is needed as well.

It is important to realize that overloading an inherited operation does not modify or hide the original operation in any way. In the example shown in Figure 6, both versions of Create are visible to Truck clients—one resulting from the type derivation, and one from the overloading. Client calls made to "Create" are (in this case) statically resolved by the compiler, and are thus unambiguous.

However, a problem arises in our example if we specify a default value for each of the newly added attributes. In this case, a call to Create is ambiguous because the compiler cannot determine whether the client wants to call the specialized Create with its defaults, or the inherited constructor that does not have the extra parameters in the first place. Fortunately, this error is detectable by the compiler. However, it still presents an issue in our technique. We could change the name of the operation, but that would be inconsistent with our naming convention. Alternatively, we could simply live with the restriction that disallows placement of defaults on all additional parameters of an overloaded inherited operation.

But the real problem here is that we really do not want Truck clients to call the inherited constructor in the first place. Doing so would result in an incomplete initialization of the Truck attributes and a generally unpredictable subsequent behavior. So while we fully intend for clients to call on the specialized version of the constructor, simply overloading the inherited operation cannot prohibit the client from invoking the inherited version.

Had the compiler found the new Create operation declared with the same parameter profile, the new operation would have *overridden* the inherited one, masking its visibility from Truck clients. A subclass may choose to override an inherited operation whenever it needs to redefine the method with a different (but hopefully related) implementation. But in this case, such overriding would not allow for initializing the new attributes, and thus does not present a solution.

What we really need in cases such as this is to employ *method restriction* [Atkinson, 1991]. This would enable us to completely remove the inherited Create operation from the subclass package protocol. In doing this, the specialized constructor would no longer be overloaded with the inherited one, and thus would be the only constructor presented to Truck clients. Removing the inherited version would also remove the limitation on new parameter defaults described earlier. Method restriction is thus very desirable in cases like this where it is erroneous to call on certain inherited operations.

Unfortunately, Ada 9X does not provide for removing inherited operations, so care must be taken on the part of subclass clients to call on the correct constructor. Alternatively, there is a way that one could approximate the effect of method restriction: We would still overload the inherited constructor as before, but would override it as well. Instead of implementing the overridden version as a real constructor, it would act more as an "error trap" operation. Its body would simply raise an exception, such as "Constructor_Error," indicating that the wrong constructor was called.

4.3.5 Delegating to Parent Operations

Figure 7 presents the body of our subclass package and illustrates an important technique for implementing specialized operations. We show how the Create operation *delegates* part of the construction job to the Motor Vehicle parent class. A special type conversion for tagged types is performed, called a *view conversion*, whereby the subclass

instance is made type compatible with the superclass type. The parameters initializing the inherited attributes are simply passed down through the delegation. New attributes are then locally initialized. It is very common for a specialized constructor (either overloaded or overridden) to call upon its parent class to initialize inherited attributes. In fact, doing so avoids redundant code development and preserves the semantics of constructors that may be further specialized down the inheritance hierarchy.

The reader might wonder why we chose not to simply invoke the inherited version of the parent's Create operation for the delegation. After all, it is directly visible and would certainly result in a simpler syntax. But a problem arises if we later decide to override the inherited Create procedure. Doing so would mask the parent constructor and divert the delegation to the locally declared version, which might or might not be appropriate. But the real danger lies in the case where the inherited constructor is solely overridden and not overloaded. In this case, an infinitely recursive call would result. This issue is quite significant because *such a problem cannot be detected by the compiler.* Therefore, we choose to explicitly delegate to the superclass package to ensure that the parent constructor will always be called regardless of the combination of overloading and overriding that might exist.

```
package body Motor_Vehicle.Truck is

   procedure Create (Instance : in out Object;
                     Weight           : in Vehicle_Weight;
                     Color            : in Colors;
                     Payload_Capacity : in Payloads;
                     Has_Cab          : in Boolean := False;
                     Wheels           : in Wheel_Count := 4) is
   begin

      -- Delegate the initialization of inherited attributes to the parent type's
      -- version of the Create operation
      Motor_Vehicle.Create (
              Instance => Motor_Vehicle.Object(Instance),
         -- The above type conversion is a "view" conversion and is necessary
         -- since the inherited Create operation is NOT being called here.
              Weight   => Weight,
              Wheels   => Wheels,
              Color    => Color);

      -- Initialize the new attributes
      Instance.Payload_Capacity := Payload_Capacity;
      Instance.Has_Cab          := Has_Cab;

         -- The attributes Payload (defined for the Truck class) and Speed
         -- (inherited from the Motor Vehicle superclass) have already been
         -- initialized to their default values as specified in their respective
         -- object record declarations.
   end Create;

   procedure Add_Payload (Instance : in out Object; Weight : in Payloads) is
   begin
      Instance.Payload := Weight;
   end Add_Payload;

   function Payload (Instance : in Object) return Payloads is
   begin
      return Instance.Payload;
   end Payload;

end Motor_Vehicle.Truck;
```

Figure 7. Implementation of a subclass package.

4.3.6 Extending Clientship

The previous subsections described how a subclass package derives its class type from a tagged type to inherit and specialize the features of an existing superclass package. We now turn our attention to the effects of using child library units to modularize our type derivations.

The first identifier in the expanded package name indicates the root class of the hierarchy, while subsequent names indicate subclasses. Since each subclass package is declared in this way, each effectively documents its ancestry in its very identification. Although this expanded notation can become unwieldy for lower-level subclasses, Ada's 'use' clause and 'renames' statement can help considerably when referencing such units.

The key feature of a child library unit rests in the extended visibility that it has into its parent unit. Child units are "logically nested" inside their parent's specification, which means that they have implicit and direct visibility to the public declarations made in the parent. What is even more significant from an OOP point of view is that *the private part and body of a child unit have direct visibility to the private part of its parent.* This privileged visibility allows a child unit to easily access, and in fact, "extend" its parent's capabilities. Child library units thus provide another form of "programming by extension" [Barnes, 1993].

Our technique relies on this extended visibility because it allows the type derivations to be nicely modularized into their own (separately compilable) library units. Without child unit visibility, subclass methods would not be able to access the attributes inherited from their parent's private object structure. While our Truck subclass example does not demonstrate the need to do so, such visibility will often be needed.

As another direct consequence of the extended visibility, the attribute types defined for a parent class package are effectively "inherited" by subclass packages as well. In Figures 6, we see that the specification for the Truck's Create procedure references the Motor Vehicle attribute types by their simple names. This sharing of data types is most significant since previous approaches with Ada 83 would often declare *subtype aliases* of attribute types to propagate their visibility, or would have the designer place such types into a *common types package.* While such techniques may still have a role in ROMAN-9X, child library units present a significant improvement in that they directly address the module visibility implications of inheritance relationships.

Technically, a child library unit may be considered a client of its parent unit in certain contexts. A child unit can certainly create instances of its parent's class type and perform associated operations on those instances. Or, as in the case of our Truck constructor, a child unit may simply call a parent operation on an existing instance. However, the primary reason to use a child unit is to extend a parent's capabilities to subclass clients while offering some additional or refined features in the process. Thus, we refer to a child library unit as an *extending client.* In contrast, a client that has visibility to only the public part of a class package is referred to as an *ordinary client.* Without qualification, clientship is understood to be of the ordinary form.

It is interesting to note that in Ada 9X, the responsibility for determining extended vs. ordinary clientship lies with the client. If, for whatever reason, the form of clientship needs be changed, the parent package would not require modification itself. This is important from an evolutionary development and change management perspective since it is critical to keep the higher-level classes in an inheritance hierarchy as stable as possible.[7]

Note that a subclass package could alternatively be implemented as an ordinary client. Such a client would still inherit all of its parent's features via the same type derivation scheme. The client could then add new features of its own as before. However, an ordinary client looses direct visibility to attribute types and must thus declare subtype aliases to propagate type visibility. More importantly, the implementation of this client could not access the private representation of the inherited attributes. This is not a problem if the implementation does not require such visibility, as is the case for our example. Some authors go further by stating that ordinary clientship should be the choice for representing subclasses in general. [Wild, 1992] goes as far to say:

> Child classes should be thought of as [ordinary] clients of their parent classes. This way of thinking promotes good encapsulation, and minimizes the number of hard dependencies that will exist between classes in a class system. Allowing classes to access the parent's instance data [results in a] degree of coupling too strong to be desirable.... All code that accesses the common data [needs to be] considered with a full understanding of the behavior of all of the other code that accesses the common data.

An ordinary clientship approach to inheritance does indeed better preserve and enforce the original encapsulation defined by parent classes. But we must examine the very essence of what a "subclass" is really suppose to model, that being the so-called "is-a" relationships in problem domain. An instance of a subclass should simultaneously be an instance of its superclass in every respect. In fact, as far as the

---

[7] In contrast, clientship visibility privileges in C++ is dictated by the server class declaration itself.

```
with Motor_Vehicle.Truck; -- with Motor_Vehicle; is implicit
with Text_IO;
procedure Truck_Client is

   My_Truck: Motor_Vehicle.Truck.Object; -- Any attributes declared with
                                         -- defaults now have those values.

   Current_Speed   : Motor_Vehicle.Velocity       := 0;
   Current_Payload : Motor_Vehicle.Truck.Payloads := 0;

begin

   Motor_Vehicle.Truck.Create (Instance         => My_Truck,
                               Weight           => 7200,
                               Wheels           => 8,
                               Color            => Motor_Vehicle.Black,
                               Payload_Capacity => 2000);

      -- The attributes Speed (from the Motor Vehicle class) and Payload
      -- (from the Truck class) were initialized in the object declaration
      -- above, according to their respective object record defaults.  The
      -- Attribute Has_Cab added for the Truck class has been defaulted to
      -- False, as is specified in the profile for the Create procedure.

   -- Begin driving at 50 m.p.h.
   Motor_Vehicle.Truck.Drive (My_Truck, Speed => 50); -- Inherited operation

   -- Print out current speed
   Current_Speed := Motor_Vehicle.Truck.Speed(My_Truck); -- Inherited operation
   Text_IO.Put ("The current speed is");
   Text_IO.Put ( Motor_Vehicle.Velocity'IMAGE(Current_Speed) );
   Text_IO.Put_Line (" m.p.h.");

   -- Stop at hardware store
   Motor_Vehicle.Truck.Stop(My_Truck);

   -- Pick up some lumber
   Motor_Vehicle.Truck.Add_Payload (My_Truck, Weight => 500);

   -- Print out a payload message
   Current_Payload := Motor_Vehicle.Truck.Payload(My_Truck);
   Text_IO.Put ("The current payload is ");
   Text_IO.Put ( Motor_Vehicle.Truck.Payloads'IMAGE(Current_Payload) );
   Text_IO.Put_Line (" lbs.");

   -- Head on back to the ranch
   Motor_Vehicle.Truck.Drive (My_Truck, Speed => 40);

   -- Stop the truck
   Motor_Vehicle.Truck.Stop (My_Truck);

end Truck_Client;
```

Figure 8. Client code utilizing a subclass package.

subclass instance is concerned, all of its features could have just as well been defined in the subclass to begin with.

So a counter-case can be made stating that true "is-a" relationships are best represented by an extending mechanism such as child library units. One can apply the "is-a" relationship test by presenting simple true-false statements, such as, "a Truck *is a* Motor Vehicle." As long as such statements make sense from an application domain perspective, the overtly strong coupling that results from using child library units is warranted. If, however, the "is-a" criteria cannot be satisfied, then the very notion of "subclass" should be conceded to ordinary clientship and implemented accordingly.

In any respect, a "pseudo-modularity" does result from structuring subclass packages in this way. But when presented with "true" subclasses, child library units used in conjunction with tagged types do allow single inheritance to be directly and very easily implemented in Ada 9X.

4.3.7 Representing Subclass Clientship

Figure 8 illustrates the effects that inheritance has on users of a subclass package. As before with the Motor Vehicle client, we include an explicit context clause specifying the server class. But note that a child library unit cannot be with'ed in by its simple name. Since a child unit is defined within a strict hierarchy, its context is dependent on its ancestry and must therefore be referenced by its fully expanded path name. Note that a single 'with' statement implicitly with's in each of the packages indicated in the expanded name.

Within the client code itself, the expanded name must also be used to specify exactly to which level in the hierarchy are references being made. While this syntax may at first appear to be a bit obtrusive, it does have the maintenance benefit of being very self-documenting, disclosing a more "honest" view of how packages in a class hierarchy are actually coupled to one another.

4.3.8 Constraining Inherited Attributes

We continue working through the inheritance hierarchy and show the representation of the Car class in Figure 9. The Car class is similar in form to that of the Truck class. However, an important difference is that the Car class package must represent the object model's constraint on the number of Car wheels allowed. One approach might be to define a subtype that tightens the value range allowed for the Wheels attribute on Cars. However, since the constraint is for exactly four wheels, we can do better. In the overloaded constructor operation, we have removed the parameter previously used for allowing clients to explicitly initialize the Wheels attribute.

Upon examining the implementation for the constructor in Figure 10, we see in the delegation that the constraint of four wheels is hard-wired into the corresponding actual parameter. This technique of varying the parameter profile of an inherited constructor can be used in general to constrain the values of selected inherited attributes, thus providing another means for specializing a class. But as discussed earlier, this overloading does not mask the inherited constructor from the client. Figure 11 demonstrates the client interface to the Car class. So long as we call the proper constructor, instances of the Car class will be constrained to having four wheels.

4.3.9 Representing Lower-Level Subclasses

We conclude the representation of our reference object model by implementing the third-level class in the inheritance hierarchy. Figure 12 illustrates how once the pattern for building subclasses is established, a new or adapted capability can be very quickly and reliably developed. As before, each parent name in the hierarchy must precede the new child unit name, and such a specification implicitly with's-in those units. Note that the class type is derived from that of its parent, which is itself a derived type. While the characteristic of "tagged" is explicitly specified only for the root class type, all types derived directly or indirectly from a tagged type are implicitly tagged. The body of the Convertible class package is straight-forward (Figure 13).

As we proceed down a child unit hierarchy, the lengthy references to child unit names can become overwhelming. One alternative, of course, is to employ the 'use' clause which establishes direct visibility to the server class package features. In Ada 9X, a use clause can be selectively applied to any level in a child unit's expanded name, allowing the degree of direct visibility to be better managed. However, a probably better alternative is to use a *renaming* approach, as is shown in Figure 14.

```
package Motor_Vehicle.Car is       -- with Motor_Vehicle is implicit

   -- Derived type declaration to inherit Motor Vehicle features
   type Object is new Motor_Vehicle.Object with private;

      -- All operations defined in the Motor Vehicle superclass have been
      -- inherited and are thus implicitly declared in this subclass
      -- package.  These inherited operations may be directly invoked by
      -- clients of the Truck class on instances of this new class type.

   -- New attribute types
   type Trunk_Space_Measure is range 0..150; -- cu. ft.

   -- Specialized constructor for Cars (becomes overloaded with inherited
   -- Create procedure)
   procedure Create (Instance : in out Object;
                     Weight       : in Vehicle_Weight;       -- Defined in parent
                     Color        : in Colors;               -- Defined in parent
                     Trunk_Space  : in Trunk_Space_Measure); -- Added for Cars

      -- Note that Wheels parameter has been removed; the implementation of Create
      -- ensures that all Car instances are constrained to having 4 wheels.

   -- (no new operations are defined for Cars)

private

   type Object is new Motor_Vehicle.Object with
      record
         Trunk_Space: Trunk_Space_Measure; -- New attribute unique to cars
      end record;

   -- Now instances of Motor_Vehicle.Car.Object will have 5 attributes, the 4 that
   -- are defined for any motor vehicle, plus this new one.

end Motor_Vehicle.Car;
```

Figure 9. Representing constraints on inherited attributes.

5. Conclusions

The purpose of this paper was to demonstrate how a simple object model might be represented in Ada 9X. Our technique is based on a sound foundation established by the many ADT-based approaches to OOD already in practice. However, with tagged types, we now have the ability to create *extendable ADTs*, incrementally enhancing the features defined for each successive abstraction. And with the special visibility afforded to child library units, we can now modularize each extended ADT and yet not suffer the implementation difficulties seen with ordinary ADTs. When used in conjunction with one another, tagged types and child library units allow single inheritance to be directly and very easily represented in Ada 9X.

A great deal of the ROMAN-9X technique is concerned with how to systematically map the various elements of an object model into compilable Ada 9X source code. Many of the elements map directly to corresponding Ada constructs, making the representation process very mechanical. This mapping provides the traceability that is needed when changing either the object model or the source code itself. And for the aspects of the code that are not directly traceable to the object model, we define naming conventions that eliminate the need to contrive superficial or otherwise arbitrary names. This practice allows different designs using the ROMAN-9X technique to maintain a high degree of uniformity and consistency.

The examples presented in this paper provide canonical forms for representing object models in general. By integrating the direct and indirect mapping rules, one could automate the process of code generation to a large degree. Thus, another goal for ROMAN-9X is to assist in the development of future Ada 9X CASE tools. Our technique

may be used to support *bi-directional change management* between object models and their associated source code.

Unfortunately, even with the best representation techniques, there will generally still remain other "design decisions" to be made—decisions that are not necessarily appropriate for capturing within the confines of the object model itself. The discussion portions of this paper serve to assist the designer in making these choices, addressing the options and trade-offs one faces when moving an abstract object model into fully compilable program source code.

Note that we do not consider ROMAN-9X to be another OOD "method" in and of itself, but rather a *technique* to be used as part of an overall object-oriented methodology. ROMAN-9X assumes that one has already gone through the analysis and design *process*, and has in fact produced an object model as a *product* of that effort. It is in the construction of the object model that a fully-defined methodology is most needed. The textbooks referenced by this paper represent several of the leading OO methods available today. ROMAN-9X may therefore be considered an additional technique for *completing* any of these methods, specifically in showing how elements of a graphical representation (an object model) transcend into a textual and more detailed notation (Ada 9X).

And finally, as a longer-term goal, we wish to use this opportunity to incite a new round of methodological discussions within the OO community at large. Even in its "object-based" form, Ada proved to be a leading technology in the development and evolution of object-oriented methods. Now that Ada has become a true object-oriented programming language, the past ten years methodology development stand to be revisited. We hope that ROMAN-9X will play a role in leading a "second generation" of OOD methods and in promoting the use of Ada 9X in general.

6. Further Study

Our ultimate goal for ROMAN-9X is to fully demonstrate how to build object-oriented software with Ada 9X. This paper focused on the fundamental task of representing classes and single inheritance. Other principles of object modeling and design that need to be addressed include:

- Aggregation
- Association
- Multiplicity of associations
- Abstract Classes
- Multiple Inheritance

```
package body Motor_Vehicle.Car is

   procedure Create (Instance : in out Object;
                     Weight     : in     Vehicle_Weight;
                     Color      : in     Colors;
                     Trunk_Space: in     Trunk_Space_Measure) is
   begin

      -- Delegate the initialization of inherited attributes to the parent type's
      -- version of the Create operation
      Motor_Vehicle.Create (
              Instance => Motor_Vehicle.Object(Instance),
                 -- The above type conversion is a "view" conversion and is
                 -- necessary since the inherited Create operation is NOT
                 -- being called here.
              Weight   => Weight,
              Wheels   => 4, -- Constrains all cars to having only 4 wheels
              Color    => Color);

      -- Initialize the new attributes
      Instance.Trunk_Space := Trunk_Space;

      -- The attribute Speed (inherited from the Motor Vehicle superclass) has
      -- already been initialized to its default value as specified in its
      -- respective object record declaration.

   end Create;

end Motor_Vehicle.Car;
```

Figure 10. Implementing subclass constraints on inherited attributes.

Several aspects of object-oriented programming need further attention, including:

- Class implementation with dynamic memory allocation
- Clientship with reference semantics
- Polymorphism with dynamic binding

Some additional features of Ada 9X needed to support these more advanced aspects of OOP include:

- Class-wide types and class-wide programming
- Dynamic dispatching of operations
- Access types to static memory
- Access-to-subprogram types
- Abstract subprograms

And finally, several general features of Ada 9X are worthy of further study, including:

- Generic formal derived types
- Generic formal tagged types
- Private child library units
- Named exception handling
- Protected record types

For software engineers constrained to using Ada 83, note that ROMAN-9X is directly related to ROMAN-83, an analogous technique designed for representing object models in Ada 83 notation. ROMAN-83 was developed with an awareness of the OO programming features being proposed in Ada 9X, and can thus be used for transitioning into the revised standard. Since our paper focused primarily on the new OO features of Ada 9X, we did not present the details of ROMAN-83, nor its differences with ROMAN-9X. Readers interested in ROMAN-83 should contact the author for more information.

```
with Motor_Vehicle.Car; -- with Motor_Vehicle; is implicit
with Text_IO;
procedure Car_Client is

   My_Car: Motor_Vehicle.Car.Object; -- Any attributes declared with
                                     -- defaults now have those values.

   Current_Speed : Motor_Vehicle.Velocity := 0; -- from Motor Vehicle class

begin

   Motor_Vehicle.Car.Create (Instance    => My_Car,
                             Weight      => 3400,
                             Color       => Motor_Vehicle.Blue,
                             Trunk_Space => 90);

   -- The attribute Speed (from the Motor Vehicle class) was initialized in the
   -- object declaration above, according to its respective object record default.
   -- The attribute Wheels has been removed from the parameter list and is
   -- internally constrained to 4.

   -- Begin driving at 60 m.p.h.
   Motor_Vehicle.Car.Drive (My_Car, Speed => 60); -- Inherited operation

   -- Print out current speed
   Current_Speed := Motor_Vehicle.Car.Speed(My_Car); -- Inherited operation
   Text_IO.Put ("The current speed is");
   Text_IO.Put ( Motor_Vehicle.Velocity'IMAGE(Current_Speed) );
   Text_IO.Put_Line (" m.p.h.");

   -- Stop the car
   Motor_Vehicle.Car.Stop (My_Car);

end Car_Client;
```

Figure 11. Limited clientship visibility to constrained attributes.

Acknowledgments

I would like to thank David Weller for providing me with the latest forms and versions of Ada 9X documentation. Ready access to such materials proved to be vital in my understanding of the new language. I also wish to thank Colin Atkinson for his review and consultation of the early drafts of the paper. Colin helped me constrain my presentation to a scope that was realizable for a paper such as this. Both Colin and David are largely responsible for many of the basic structuring and naming conventions used in ROMAN-9X and in its Ada 83 counterpart.

Special thanks go to Tucker Taft who was responsible for compiling my Ada 9X code examples. Presenting compiled source code lends to the credibility of any paper, but it was most significant here since there was no public compiler for Ada 9X available at the time of writing. Tucker also helped me semanticize the code, shedding light on some of the more subtle aspects of the technique.

I would also like to thank Keith Shillington for reviewing the paper in its final stages and for helping ensure its proper use of Ada's semantics. And finally, I wish to thank Eileen Quann and Fastrak Training Inc. for sponsoring the extra effort it took to produce this paper.

```
package Motor_Vehicle.Car.Convertible is -- with Motor_Vehicle,
                                         --      Motor_Vehicle.Car; is implicit

   -- Derived type declaration to inherit Motor Vehicle and Car features
   type Object is new Motor_Vehicle.Car.Object with private;

      -- The Drive, Stop, and Speed operations have been inherited from the Car
      -- class (which inherited them from the Motor Vehicle class) and are now
      -- implicitly declared in this subclass package.  These operations may now
      -- be invoked by clients of the Convertible class on instances of this new
      -- class type.

   -- New attribute types
   type Mechanisms is (Manual, Power);
   type Positions  is (Up, Down);

   -- Specialized constructor for Convertibles (becomes overloaded with inherited
   -- Create procedures from both Car and Motor Vehicle)
   procedure Create (Instance : in out Object;
                     Weight             : in Vehicle_Weight;       -- Defined
                     Color              : in Colors;               -- in
                     Trunk_Space        : in Trunk_Space_Measure;  -- parents
                     Movement_Mechanism : in Mechanisms := Manual);-- New for Conv

   -- New modifiers
   procedure Put_Top_Down (Instance : in out Object);
   procedure Put_Top_Up   (Instance : in out Object);

   -- New selectors
   function Top_Position (Instance : in Object) return Positions;

private

   type Object is new Motor_Vehicle.Object with
      record
         Movement_Mechanism : Mechanisms;
         Top_Position       : Positions := Up;
      end record;

      -- Now instances of Motor_Vehicle.Car.Convertible.Object will have 7
      -- attributes, the 4 that are defined for any motor vehicle, the one
      -- defined specifically for Cars, and these two.

end Motor_Vehicle.Car.Convertible;
```

Figure 12. Representing lower-level subclasses.

```
package body Motor_Vehicle.Car.Convertible is

   procedure Create (Instance : in out Object;
                     Weight             : in Vehicle_Weight;
                     Color              : in Colors;
                     Trunk_Space        : in Trunk_Space_Measure;
                     Movement_Mechanism : in Mechanisms := Manual) is
   begin

      -- Delegate the initialization of inherited attributes to the parent type's
      -- version of the Create operation
      Motor_Vehicle.Car.Create (
              Instance => Motor_Vehicle.Car.Object(Instance),
                 -- The above type conversion is a "view" conversion and is
                 -- necessary since the inherited Create operation is NOT
                 -- being called here.
              Weight      => Weight,
              Color       => Color,
              Trunk_Space => Trunk_Space);

         -- Note that the 4-wheel constraint imposed on Car instances is
         -- enforced by the delegation to the Create operation for Cars.

      -- Initialize the new attributes
      Instance.Movement_Mechanism := Movement_Mechanism;

      -- The attribute Speed (inherited from the Car superclass, which inherited
      -- it from its Motor Vehicle superclass) has already been initialized to its
      -- default value as specified in its respective object record declaration.
      -- The attribute Top_Position was initialized in the tagged type extension
      -- for this Convertible subclass.

   end Create;

   procedure Put_Top_Down (Instance : in out Object) is
   begin
      Instance.Top_Position := Down;
   end Put_Top_Down;

   procedure Put_Top_Up   (Instance : in out Object) is
   begin
      Instance.Top_Position := Up;
   end Put_Top_Up;

   function Top_Position (Instance : in Object) return Positions is
   begin
      return Instance.Top_Position;
   end Top_Position;

end Motor_Vehicle.Car.Convertible;
```

Figure 13. Implementation of lower-level subclasses.

```
with Motor_Vehicle.Car.Convertible; -- with Motor_Vehicle,
with Text_IO;                       --      Motor_Vehicle.Car; is implicit
procedure Convertible_Client is

   -- Rename the subclass package with its simple name for brevity
   package Convertible renames Motor_Vehicle.Car.Convertible;

   My_Convertible: Convertible.Object;
            -- Any attributes declared with defaults now have those values.

   Current_Speed        : Motor_Vehicle.Velocity := 0;
   Current_Top_Position : Convertible.Positions  := Convertible.Up;

begin

   Convertible.Create (Instance           => My_Car,
                       Weight             => 2600,
                       Color              => Motor_Vehicle.Red,
                       Trunk_Space        => 50,
                       Movement_Mechanism => Power);

   -- The attributes Speed (from the Motor Vehicle class) and Top_Position
   -- (from the Car class) were initialized in the object declaration above,
   -- according to their respective object record defaults.

   -- Put the convertible top down
   Convertible.Put_Top_Down (My_Convertible);

   -- Begin driving at 70 m.p.h.
   Convertible.Drive (My_Convertible, Speed => 70); -- Inherited operation

   -- Print out current speed and position of convertible top
   Current_Speed := Convertible.Speed(My_Convertible); -- Inherited operation
   Current_Top_Position := Convertible.Top_Position(My_Convertible);
   Text_IO.Put ("The current speed is");
   Text_IO.Put ( Motor_Vehicle.Velocity'IMAGE(Current_Speed) );
   Text_IO.Put (" m.p.h.");
   Text_IO.Put (" and the convertible top is ");
   Text_IO.Put ( Convertible.Positions'IMAGE(Current_Top_Position) );
   Text_IO.New_Line;

   -- Stop the convertible car
   Convertible.Stop (My_Convertible);

   -- Put the convertible top up
   Convertible.Put_Top_Up (My_Convertible);

end Convertible_Client;
```

Figure 14. Clientship of lower-level subclasses.

References

Atkinson, C., *Object-Oriented Reuse, Concurrency, and Distribution - An Ada-Based Approach*, ACM Press, Addison-Wesley, 1991.

Atkinson, C. and Weller, D., "Integrating Inheritance and Synchronization in Ada 9X," Tri-Ada '93 Conference Proceedings, 1993.

Barnes, J. *Introducing Ada 9X*, Intermetrics/Department of Defense, February 1993.

Booch, G., *Software Engineering With Ada*, Benjamin/Cummings, 1983.

Booch, G., *Software Components With Ada*, Benjamin/Cummings, 1987.

Booch, G., *Object-Oriented Design With Applications*, Benjamin/Cummings, 1991.

Cardelli, L. and Wegner, P., "On Understanding Types, Data Abstraction, and Polymorphism," ACM Computing Surveys, vol. 17(4), December 1985.

Coad, P. and Yourdon, E., *Object-Oriented Analysis*, Yourdon Press (2nd ed.), 1991.

Coad, P. and Yourdon, E., *Object-Oriented Design*, Yourdon Press, 1991.

Di Maio, A., Cardigno, C., Genolini, S., Destombes, C., and Atkinson, C., "DRAGOON: An Ada-based Object-Oriented Language for Concurrent, Real-Time, Distributed Systems," *Proc. Ada-Europe International Conference 1989*, The Ada Companion Series.

Intermetrics, *Ada 9X Mapping Document, Volume II: Mapping Specification*, U.S. Department of Defense, December 1991.

Intermetrics, *Ada 9X Mapping Document, Volume I: Mapping Rationale*, U.S. Department of Defense, March 1992.

Intermetrics, *Ada 9X Reference Manual*, Ada 9X Project Office, Version 3.0, 29 June 1993.

Meyer, B., *Object-Oriented Software Construction*, Prentice Hall, 1988.

Rumbaugh, J., Blaha, M., Premerlani, W., Fredrick, E., and Lorensen, W., *Object-Oriented Modeling and Design*, Prentice Hall, 1991.

Seidewitz, E. and Stark, M., *Principles of Object-Oriented Software Development with Ada*, Millennium Systems, Inc., 1992.

Shlaer, S. and Mellor, S., *Object Lifecycles: Modeling the World in States*, Yourdon Press, 1992.

Wild, F., "Experience Report - Creating Well Formed Class Inheritance Schemes in C++," OOPS Messenger, Addendum to the OOPSLA '92 Conference Proceedings, vol. 4(2), April 1993.

Wirfs-Brock, R., Wilkerson, B., and Wiener, L., *Designing Object-Oriented Software*, PTR Prentice Hall, 1990.

Biography

Gary Cernosek is a senior software engineer with Fastrak Training Inc. where he is responsible for developing and administering training courses in object technology. Gary has worked with object-oriented analysis and design methods since 1985 using Ada- and C-based languages. His primary assignment over the past year-and-a-half has been with CAE-Link Flight Simulation Co. in Houston, Texas where he is the lead instructor and mentor for the object-oriented development of realtime mission training simulators for NASA-JSC. Gary was also responsible for several software reuse initiatives while working at McDonnell Douglas through 1991 on both Space Shuttle and Space Station projects. Gary received a Bachelor of Science degree in electrical engineering from the University of Texas at Austin in 1983, and a Master of Science degree from the University of Houston at Clear Lake in 1988. His master's thesis was on the need for an object-oriented approach to requirements analysis. Gary is a member of the IEEE Computer Society and the ACM.