AFIT/EN-TR-93-6

Air Force Institute of Technology

An Examination of Two Ada Language Object-Oriented Databases

Karl S. Mathias, Capt, USAF
Mark A. Roth, Maj, USAF

8 July 1993

DEPARTMENT OF THE AIR FORCE
AIR UNIVERSITY
AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

UNCLASSIFIED

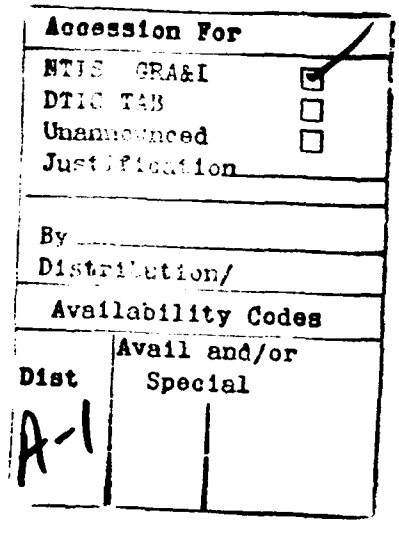
AFIT/EN-TR-93-6

Air Force Institute of Technology

An Examination of Two Ada Language Object-Oriented Databases

Karl S. Mathias, Capt, USAF
Mark A. Roth, Maj, USAF

8 July 1993



Approved for public release; distribution unlimited

An Examination of Two Ada Language Object-Oriented Databases

Karl S. Mathias
Mark A. Roth[‡]

Abstract

Many object-oriented databases marketed today target themselves to either C++ or Lisp as a host language. This presents problems to Department of Defense contractors who need to utilize Ada for development. This paper examines two Ada object-oriented databases: Classic-Ada with Persistence and Science Applications International Corporation's OODBMS developed for the US Air Force's Dental Data System.

1 Introduction

The popularity of object-oriented database management systems (OODBMS) continues to grow as their usefulness in object-oriented programming is realized. They integrate both data and methods into objects while providing persistence and concurrency. This makes them naturally attractive to users of object-oriented programming languages such as C++, Lisp, and derivatives of Smalltalk. Major OODBMS such as ONTOS, ObjectStore, O2, and Itasca all use C++ as their base language.

This paper examines two OODBMS that use Ada as the host language. The first, Classic-Ada with Persistence, extends the Ada language to allow class hierarchies, dynamic object instantiation, and persistence. The second is a set of Ada packages developed by Science Applications International Corporation's (SAIC) as part of an Air Force contract to build the Dental Data System (DDS). This set of packages defines an OODBMS that allows class hierarchies, dynamic object instantiation, persistence, and dynamic creation of object methods. While both meet many of the requirements of an OODBMS, their approaches are quite different.

[‡]The authors are with the Department of Electrical and Computer Engineering (AFIT/ENG), Air Force Institute of Technology, 2950 P ST, Wright-Patterson AFB, OH 45433-7765.

2 Approach

There has been a lot of discussion about what features a database management system must support before it is considered an OODBMS. One of the primary questions to be determined about the Ada databases in question are: 1) What features do they support, and 2) Are these features sufficient to consider them OODBMSs?

There have two important papers written on this subject. In the "Object-Oriented Database System Manifesto" [1], the authors specify several areas that an OODBMS must support: complex objects, unique object identifiers, object encapsulation, support for types or classes, inheritance, late binding, computational completeness, extensibility, persistence, storage management, concurrency, and the ability to make ad hoc queries. The "Third-Generation Data Base System Manifesto" [3] also calls for rules in the engine (triggers, constraints), collections of objects, updatable views of objects, and support for SQL.

Rather than picking one set of requirements and basing an examination upon them, this paper uses a combined approach. Barry [2] compiled a checklist of attributes that an OODBMS may consist of. While not mandating that an OODBMS have all or any of these attributes, he provides several main categories of features. The categories examined here are:

- *Object Model.* What types of objects are supported? What type of support is there for object methods? Is polymorphism supported? How are objects encapsulated? How are types and classes supported? Is inheritance supported? How are relationships between objects defined?
- *Schema Development.* How are schema's defined? What support exists for changing the schema? Can it be changed dynamically? What support exists for changing the methods of a class?
- *Architecture.* What is the implementation of the database server? Does it support multiple clients? Where are methods executed, the server or the client? Does it support rules?

- *Transaction Properties.* Are transactions atomic? What degree of consistency is maintained? Are long transactions supported? Are nested transactions supported?

- *Persistence Transparency.* Are the Ada data structures themselves persistent, or do they have to be loaded into persistent database structures? Must the user explicitly tell the database to save an object or is this done automatically?

- *Concurrency Control.* What form of concurrency control is used?

3 Classic-Ada with Persistence

Classic-Ada is a product of Software Productivity Solutions Incorporated (SPS). The basic Classic-Ada package is a preprocessor that extends the Ada language to allow class constructs. Objects may be dynamically instantiated, and a message passing mechanism alleviates the problems associated with late binding in Ada.

Classic-Ada with Persistence is an extension that allows class structures to be defined as persistent. Objects instantiated from persistent classes maintain their state between application executions. Two packages are provided that provide functions for managing the database and persistent objects.

3.1 Object Model

Classic-Ada augments the package structure by introducing classes into Ada. A class may contain instance variables and instance methods. Single inheritance is supported, and classes may override their parent class' methods (polymorphism).

Objects are instantiations of classes. Dynamic instantiation is supported. Methods in objects are invoked by sending messages to them—the object determines at run time which of its methods should execute the message. According to SPS, this gives support to late binding, though it is unclear how an object could adjust its response to a given message.

Objects reference each other with the use of a 32-bit object id. Though not defined as a limited private type, SPS recommends that it be treated as such and supplies relational operations to test it with. The id remains valid across application executions so long as the target object is persistent. The id is not valid across executions for non-persistent types. There is no support for inverse relationships.

```
persistent class TEST_CLASS is
  superclass TEST_CLASS_PARENT;

  subtype INDEX_TYPE is 1..20 of NATURAL;
  subtype VALUE_TYPE is 1..99 of NATURAL;

  method CREATE(NEW_TEST: out OBJECT_ID);

  instance method GET_VALUE
    (INDEX : in INDEX_TYPE;
     VALUE : out VALUE_TYPE);
  instance method PUT_VALUE
    (INDEX : in INDEX_TYPE;
     VALUE : in VALUE_TYPE);

end TEST_CLASS;
```

Figure 1: Classic-Ada Class Definition

An example of a class definition using inheritance is shown in Figure 1. The keyword `persistent` indicates that objects of this class will be persistent. The `superclass` keyword causes TEST_CLASS to inherit all the methods and instance variables of TEST_CLASS_PARENT. A class method is defined that creates objects. These objects will have the methods GET_VALUE and PUT_VALUE available to them.

Figure 2 illustrates how a corresponding body would be written. The class contains one instance variable, TEST_ARRAY. The CREATE method instantiates a new object of this class and returns its object identifier. The other two methods allow access to the array defined by TEST_ARRAY.

Invocation of the methods is performed as shown in Figure 3. First, an object is created by calling the CREATE function and saving the object identifier. Messages are then sent to the objects via the identifier. The messages and parameters correspond to the declarations of PUT_VALUE and GET_VALUE.

3.2 Schema Development

A schema in Classic-Ada is essentially the class structure as defined by the program. Once this structure has been fixed, it cannot be modified without losing all persistent data. According to the Classic-Ada manual:

> "To ensure consistent views of persistent objects, Classic-Ada with Persistence requires that

```
persistent class body TEST_CLASS is
  type TEST_ARRAY_TYPE is array(INDEX_TYPE'FIRST..
                                INDEX_TYPE'LAST) of VALUE_TYPE;

  TEST_ARRAY : instance TEST_ARRAY_TYPE;

  method CREATE(NEW_TEST : out OBJECT_ID) is
  begin
    NEW_TEST := instantiate;
  end CREATE;

  instance method GET_VALUE(INDEX : in INDEX_TYPE;
                            VALUE : out VALUE_TYPE) is
  begin
    VALUE := TEST_ARRAY(INDEX);
  end GET_VALUE;

  instance method PUT_VALUE(INDEX : in INDEX_TYPE;
                            VALUE : in VALUE_TYPE) is
  begin
    TEST_ARRAY(INDEX) := VALUE;
  end PUT_VALUE;
end TEST_CLASS;
```

Figure 2: Classic-Ada Class Body

```
.
.
TEST_OBJECT : OBJECT_ID;
TEST_VALUE  : VALUE_TYPE;
.
.
begin
  .
  .
  TEST_CLASS.CREATE(TEST_OBJECT);
  send(TEST_OBJECT, PUT_VALUE,
       INDEX => 1, VALUE => 10);
  send(TEST_OBJECT, GET_VALUE,
       INDEX => 1, VALUE => TEST_VALUE);
  .
  .
```

Figure 3: Classic-Ada Method Invocation

all applications accessing a common persistent object base are generated from a single state of the class library. This state is marked by the date and time when the Classic_Executive was generated." [5]

3.3 Architecture

Each application in Classic-Ada opens the database and is responsible for maintaining its integrity. Classic-Ada does not support concurrent access, so only one application can open a database at any given time. Because of this no-server implementation, all methods are executed by each application.

3.4 Transaction Properties

The lack of a server greatly simplifies maintaining the integrity of the database. Since Classic-Ada keeps some of the database on disk and some in memory, all they require is that the database be closed before application termination. Unfortunately, failure to close the database

could leave it in an inconsistent state, and it is not clear from the documentation whether recovery is possible.

3.5 Persistence Transparency.

Classic-Ada offers total transparency to the programmer. After opening the database, the objects stored in it are available without any further system calls. There is no requirement to do an explicit save of an object, though SPS recommends closing and reopening the database periodically to ensure that objects buffered in memory get written to the disk.

3.6 Concurrency Control

As indicated previously, Classic-Ada does not support concurrent access to the database.

3.7 Summary

Classic-Ada provides extensions to the Ada language that support classes, inheritance, and dynamic instantiation of objects. It does not utilize a server and does not allow concurrent access to the database. Persistence of objects is maintained across executions, but the schema of the objects may not be altered without loss of data.

4 SAIC's OODBMS

The SAIC DDS was developed for the US Air Force to replace an existing COBOL-based system. As part of the contract, SAIC developed an object-oriented database system that was not tied to the DDS application. This OODBMS, written in Ada, is owned by the government and distributable within its agencies.

The database is supplied as a set of Ada packages that compile into the database server. Some of these packages may need to be modified depending on the application. In general, however, the user will write a client program that interfaces with the server via a messaging system.

SAIC approaches the task of object management in an entirely different manner from that of Classic-Ada. Rather than extending the language, they have built data structures that allow classes to be defined and expanded. Classes are considered objects, and, according to the documentation, every object has a class—including the class object. Classes are defined by using a schema file which shows the inheritance and methods. Additionally, each class must have its methods placed into a package and be linked into the database server. SAIC supplies a core set of classes and methods to handle objects, classes, methods, collections, arrays, integers, floats, strings, etc.

Modeled after Smalltalk, the OODBMS is self-describing, allowing applications to create more complex types dynamically. Methods for these types can also be created dynamically and stored directly into the database. It should be noted that classes and class methods can not be created dynamically. A class is a pre-defined object with its methods written in Ada. A complex type is an Ada data structure which can contain pointers to class instances and method objects—both of which are created at run-time.

The server consists of 40,000+ lines of code, and there is little documentation to explain its use. The information presented here was gained mostly by examining the database server and DDS application code directly.

4.1 Object Model

4.1.1 Classes

A class is defined in two pieces. First, a schema file indicates the hierarchy of classes in the system. Sections in this file defines a class, indicates who its parent class is, and declares the methods available in the class. The contents of this file are placed into the database by an initializer. After this initialization process, the server may be invoked to run on the database.

The second part of a class definition is an Ada package called its *resolver*. The resolver package contains all the methods defined in the schema file. A special front-end resolves messages by directing them to the correct method. If a message does not resolve properly, it is passed on to the parent resolver. If the message does not invoke a method in the class hierarchy, an exception is raised by the database.

Figure 4 shows a portion of the schema file provided with DDS. Each class is defined by an entry of the form **classnum-parentnum classname**. Thus, 24-23 Blob is class number 24 with parent class 23 (i.e., Blob is a subclass of SimpleObject). The OODBMS Programmer's Manual [4] refers to the object number as a resolver number since the resolver packages use it to route messages.

Methods for each class are declared by entries in the form **methodnum.numparams methodname param1 param2 ... param_n**. Methods preceded by an exclamation point (!) indicate a method that operates on the class, not an object. A typical class method is `!new` which creates an object of a given class.

```
.
.
  23-1 SimpleObject

       0.0  remove
       1.0  value
       2.1  value: aValue
       3.0  !new
       4.1  !remove: anObject

      24-23 Blob

           0.0  !new
           1.1  !new: anObject
           2.1  !remove: anObject
           3.1  = anObject
           4.0  value
           5.1  value: anObject
           6.0  remove
.
.
```

Figure 4: Fragment of a DDS Schema File

4.1.2 Objects

Objects are instances of the pre-defined classes. Each object can be given a unique object identifier to be used for establishing relationships. Ada records can be used to collect these objects into BLOBs (Binary Large OBjects). These BLOBs can then be stored into the database under the pre-defined BLOB class. Using this technique, complex types can be developed and stored.

Figure 5 demonstrates how Ada converts a record into a BLOB object. A BLOB object is allocated with the `CB.Allocate_Blob` routine. The Ada record is then copied into the memory reserved for the BLOB. A pointer to this area is returned to the creator. Later, this pointer will be used to place the BLOB in the database.

4.1.3 Methods

Methods are defined in two ways—in the class resolver package, or by means of a multiple message method. In the first case, the method is an Ada subprogram compiled and linked into the server. It cannot be changed without recompiling and linking the entire system. Note that the resolver structure allows polymorphism since a method defined in a child class will be resolved before the same method in a parent class.

In the second case, a special database object called a MultipleMessageMethod is created. Essentially, this object can be dynamically "programmed" to issue a sequence of messages. This sequence of messages might involve getting a key from a dictionary object and then using it to return a BLOB object identifier from a sorted collection. In addition to the object messages, the MultipleMessageMethod can accept various control structures such as an if-then-else construct. This powerful device allows applications to dynamically write and modify their methods.

MultipleMessageMethods are created as shown in Figure 6. A Method_Builder package has been defined previously that makes the primitive calls to the MultipleMessageMethod object. The package has been renamed to MB in the figure.

The code begins by creating a MultipleMessageMethod object with the `New_Method` call. Subsequent calls store the steps needed to have the method retrieve an association from a Dictionary object called `Record_Dictionary`. The association returns the `OBJECT_ID` of the BLOB containing the base Ada record. A message is then sent to the corresponding object which returns a pointer to the value of the BLOB itself.

4.2 Schema Development

There are no tools available for changing the schema of a database without invalidating the data. In general, changing the application schema will require writing conversion routines to recover information. As discussed previously, stored methods may be changed at any time.

Different applications may use the same database so long as they share a common schema. The class schema is built into the database when it is initialized, so all applications will have it available. Application-defined complex types, however, will have to be declared by each application at run-time. Some application may use only subsets of this part of the schema.

4.3 Architecture

SAIC's system consists of one server and multiple clients. Each client communicates with the server by passing a shared memory segment that contains an object message. The server accesses the segment and attempts to resolve the message by following the class hierarchy discussed above.

Methods are executed solely in the server. Class methods are actually compiled as subprograms in the server. MultipleMessageMethod objects execute their methods

```
--
-- Package CB renames Communications_Buffer
-- Package DBT renames Database_Types
-- Package A renames Allocation
--
function Make_Blob (The_Record : A_Record) return CB.POINTER is

  -- Allocate the BLOB memory
  Mem_Pointer : CB.Pointer := CB.Allocate_Blob(A_Record'SIZE / DBT.Byte'SIZE);
  -- Create a BLOB to access the memory
  Blob_Ptr : DBT.Blob_Ptr := CB.Blob_Access(Mem_Pointer);

begin

  -- Copy the data in the Ada record to the BLOB buffer
  A.Copy_Block( The_Record'ADDRESS, Blob_Ptr(1)'ADDRESS, Blob_Ptr'LENGTH);

end Make_Blob;
```

Figure 5: Converting Records to BLOBs

when an "execute" message is received. Since the MultipleMessageMethod method for "execute" is itself a compiled subprogram in the server, all the processing for this dynamic method is performed in the server.

Figure 7 shows how the method defined in Figure 6 would be executed. The `The_Key` parameter is placed into the argument list defined by the `Args` array. A call to `DI.Send_Message` invokes the MultipleMessageMethod specified by the object identifier stored in `The_Method`. The "execute" method causes the MultipleMessageMethod to look up the object identifier of the BLOB and return it. The routine is then able to load an Ada record by copying information out of the BLOB.

4.4 Transaction Properties

Database transactions are initiated with a `Start_Transaction` call to the server and terminated with a `Commit` or `Rollback` call. Nested transactions are not allowed.

The server allows only one transaction to execute at a time. In this manner it maintains serializability at the cost of performance. This limitation disallows long transactions, and the documentation encourages the use of short transactions for good multi-user performance.

Only commited transactions are written to the database. Due to the non-concurrent nature of the database, this allows for a very simple recovery since only the current transaction will have been lost.

4.5 Persistence Transparency.

Persistence is not a transparent as in Classic-Ada. Each class determines in its create mechanism (usually a method called `new`) whether instances will use persistent memory. This persistent memory is supplied by a memory page manager package.

It becomes the responsibility of the programmer to copy Ada data structures into DDS database objects. The object method used to accomplish this copying results in the object being saved. So while the programmer doesn't explicitly have to tell the database to save the object, they must still copy information into database objects.

4.6 Concurrency Control

As noted previously, the system allows only one transaction to execute at a time. Other transactions are blocked and must wait until the executing transaction completes.

4.7 Summary

The DDS system provides static classes that may be changed or added to by creating new server packages.

```
.
.
-- Create a method to get a record.
-- Returned pointer is a blob pointer.
--
-- Method name:
--    "get record"
--
-- Arguments:
--    "$1"      (The key used to locate the record)

-- Create a new method and place it in the database
aMethod := MB.New_Method ("get record");
Record_Object (aMethod);

-- Program the method with a sequence of messages to send
-- Start by sending a message to the record dictionary asking for
--   the object associated with the key stored in $1
MB.Message (Record_Dictionary, "at:", "$1");

-- Ask this object (a BLOB) to return a pointer to its value
MB.Message ("!","value");

-- Tell the method to resolve these messages based on the current
--   database schema of objects.  While not required, it removes the
--   need for interpreted resolution of messages and gives a large
--   performance boost.
MB.Resolve;
.
.
```

Figure 6: Creation of a MultipleMessageMethod in DDS

By using Ada record structures, applications can dynamically create complex data types and store them as objects. Methods may also be dynamically created and associated with objects via object identifiers. Only one database server is allowed, and it may only execute one transaction at a time. All methods are executed by the server.

5 Conclusions

Classic-Ada with Persistence has only marginal uses as an OODBMS. It provides excellent object-oriented programming extensions to Ada. This is offset, however, by the complete lack of a multi-user database system. Classic-Ada with Persistence would only be adequate for small single-user systems that require limited application database support.

The SAIC OODBMS holds a lot of promise. It provides extremely powerful methods to dynamically create complex types and methods. The base classes are easily extended by creating new Ada packages with method resolvers. The entire system appears to have been built with extensibility in mind.

On the down side, the SAIC OODBMS has very primitive concurrency control. Allowing only one transaction to execute disallows long transactions. This will become a severe handicap for large databases where searches will require more and more time. A good solution would be to implement a server that allows multiple transactions using a versioning scheme.

The SAIC OODBMS should be adequate for small to medium size applications that do not anticipate long transactions or large numbers of users. Until the con-

```
function Get_Record (The_Key : in Key_String_Type) return A_Record is

   The_Blob : DBT.Blob_Ptr;
   The_Record: A_Record;
   Args : DI.Arg_List(1..1);

begin

   -- Place key in the argument list and send an execute method to the
   --   get record method.  The variable The_Method holds the Object_Id
   --   of this method.  The_Method is initialized external to this
   --   routine.
   Args(1) := CB.Allocate_String (The_Key);
   The_Blob := CB.Blob_Access (
                  DI.Send_Message (The_Method, "execute", Args(1 .. 1)));

   -- Copy the data into an Ada record
   A.Copy_Block(The_Blob(1)'ADDRESS,The_Record'ADDRESS,The_Blob'LENGTH);

   return The_Record;
end Get_Record;
```

Figure 7: Execution of a MultipleMessageMethod

currency mechanism is updated, however, it will not be well suited to large systems.

References

[1] Atkinson, Malcolm and others. "The Object-Oriented Database System Manifesto," *Deductive and Object-oriented Databases* (1990).

[2] Barry, Douglas K. "OODBMS Feature Listing," *Object Magazine* (January-February, 1993).

[3] for Advanced DBMS Function, The Committee. *Third-Generation Data Base System Manifesto.* Technical Report Memorandum No. UCB/ERL M90/28, University of California, Berkely, 1990.

[4] Science Applications International, Inc. *DDS OODBMS Programmer's Reference Manual*, 1992.

[5] Software Productivity Solutions, Inc. *Classic-Ada User's Manual*, 1989.

8 July 1993 Technical Report

An Examination of Two Ada Language Object-Oriented Databases

Karl S. Mathias, Capt. USAF
Mark A. Roth, Maj. USAF

Air Force Institute of Technology, WPAFB OH 45433-7765

AFIT/EN-TR-93-6

Many object-oriented databases marketed today target themselves to either C++ or Lisp as a host language. This presents problems to Department of Defense contractors who need to utilize Ada for development. This paper examines two Ada object-oriented databases: Classic-Ada with Persistence and Science Applications International Corporation's OODBMS developed for the US Air Force's Dental Data System.

Ada, Object-Oriented Databases 9

UNCLASSIFIED UNCLASSIFIED UNCLASSIFIED UL