Surviving Client/Server: ODBMS In Practice, Part 1

by Steve Troxell

Last month we started to look at object oriented database systems from a conceptual point of view. This month, we're going to roll up our sleeves and take an actual ODBMS product through some basic operations to get a feel for how working with these systems differs from more traditional database systems. We're going to use the Jasmine 1.1 object oriented database system, which was recently released by Computer Associates. A free Jasmine Developer CD can be requested through their website at www.cai.com/products/jasmine.htm.

Before we get on with what this article is about, let me point out what it's not about. This is not a product review of Jasmine, nor is it a step-by-step tutorial for using Jasmine. It is a detailed look at how we would set up an object oriented database, with Jasmine being the tool of choice. That being said, let's dig in.

As is the case with nearly any technology, we have some new terms to learn before we can really know our way around. Fortunately most of these terms can be considered to be new names for familiar concepts such as databases, tables, columns, and rows. However, there are subtle differences in capabilities, otherwise we wouldn't need to give them new names. Figure 1 summarizes the ODBMS terminology and their closest RDBMS parallels.

The central database unit in Jasmine is the 'class', which is essentially an object oriented version of a table. In practice, we might use the term 'class definition' to refer to the metadata describing the structure of the data held by the class, and the term 'class' when referring to the container of the data (what we would normally think of as a table). Where a relational database would have a table containing several rows of data organized into columns of discrete values, we would have a *class* containing *instances* of data organized into *properties*.

The term 'class' is also used to describe what we would think of as a datatype, like integer or string. Because of the general nature of object oriented systems, a class describes the characteristics of a 'thing', whether that 'thing' be a simple value such as an integer, or a complex collection of data such as a table. Remember when I said there are subtle differences in the concepts and strict parallels with traditional terms is generally not possible? Well, here you go. To avoid confusion, I'll refer to classes that represent tables in a database as 'data container classes'.

All classes in the Jasmine system are derived from the Jasmine class hierarchy as shown in Figure 2. Like the Delphi VCL, everything in the system has a common ancestor, `Object`. Data container classes always derive from the `Composite` class. You can see that datatype classes as well as data container classes share the same class ancestry, so they are interchangeable in some respects. For example, the return type of a method function could be a simple datatype or an entire instance from a data container.

All new classes are derived from an existing class. Classes above a given class in the hierarchy are said to be *superclasses* of the given class. For example, `Atomic` is a superclass of `Numeric`, and `Literal` is a superclass of `Atomic`. But `Literal` is also a superclass of `Numeric`. A class derived from another class is called a *subclass* and inherits all the properties and methods of all the superclasses above it.

Class Families

Jasmine uses the concept of a class family to group related class definitions. For instance, all of the class definitions for Jasmine's Fashion Boutique tutorial are kept in a class family called `CAStore` (Computer Associates Store). Class families can be roughly thought of as databases, and they are used as such in query references. But they are more accurately thought of as metadata catalogs, or schemas. For example, two class families can be merged to create a new class family with all the class definitions of both original families.

This is how Jasmine implements support for multimedia classes. You'll notice that the object hierarchy shown in Figure 2 does not indicate any support for multimedia datatypes. That's because Jasmine implements all the multimedia functionality in a separate class family called `mediaCF`, rather than burden every database with the extensive multimedia classes if they aren't needed. If you wanted to develop a database supporting

➤ *Figure 1*

ODBMS Term	RDBMS Equivalent
Class Family	Database (roughly)
Class	Table or Datatype
Property	Column
Instance	Row
Method	Stored procedure or Trigger

```
defineClass Person
  super: Composite
{
  instance:
    String     FirstName mandatory:;
    String     LastName mandatory:;
    String     Address;
    String     City;
    String[2]  State;
    String[10] ZipCode;
    Date       DateOfBirth;
};
```

➤ *Listing 1*

multimedia datatypes, you would merge the `mediaCF` class family into your own class family to add multimedia capabilities. This also allows third party vendors to develop any number of specialized class families which you could then merge into your own architecture.

The critical point to remember about class families is that subclasses can only be created from classes in the same family. One exception is that all data container classes must descend from `Composite`. When you create a class family, all your top level classes will descend from `Composite`, which obviously is not in the class family you just created.

Jasmine allows you to set a default class family so you don't have to constantly specify the family name. However, in some contexts you may have to qualify a class name with its family name, for example `mediaCF::MMSoundFile`. All Jasmine system classes shown in Figure 2 are defined in the class family `systemCF`. System classes never need to be qualified unless there is another class with the same name in the default class family.

Object Query Language

Relational databases typically rely on SQL as the database query language. Object databases employ a similar language with an object oriented flair. The general term for this language is Object Query Language (OQL) whereas Jasmine's specific implementation is called Object Database Query Language (ODQL).

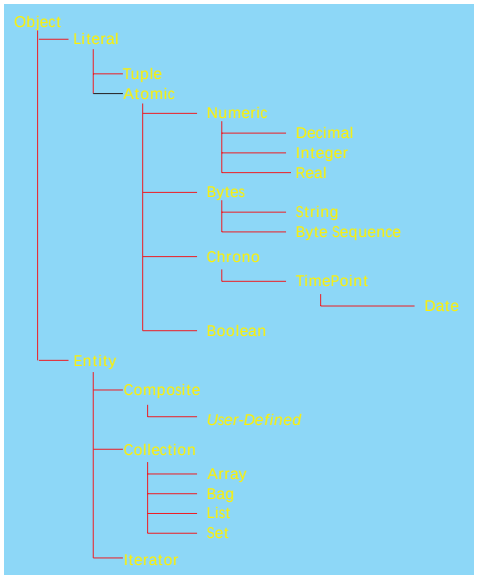
ODQL is structured very much like C++. ODQL statements and object names are case sensitive. This includes class, property, and method names. You can define variables and have flow control statements in ODQL, and all statements must be terminated with a semi-colon.

ODQL statements can be executed interactively with a command line utility provided with Jasmine called `codqlie`. This is roughly equivalent to a command line ISQL utility provided with a relational database. ODQL statements can also be placed in text files and run as scripts via the `codqlie` utility. It is also possible to send ODQL statements to the server from a client application and have the client process the results. In this instalment, we will see lots of examples of using ODQL to define database structures, and next month we'll get into more details about using ODQL for data access and manipulation.

Creating A Class

There are two steps to creating a Jasmine class: defining the class and building it. Listing 1 shows how we use the ODQL command `defineClass` to create a class called `Person`. The first clause in our `defineClass` command identifies the immediate superclass of our class. All data classes must ultimately descend from the system class `Composite` (refer back to Figure 2). So we identify `Composite` as the superclass of `Person`, which means `Person` inherits all the characteristics of `Composite`. Then we list off the properties for this class along with their datatypes. The convention is to list the datatype first, followed by the property name, followed by any qualifiers. The keyword `instance` means the properties that follow are instance-level properties, as opposed to class-level properties which we will get to in a moment.

Values within class properties can be constrained, similar to SQL tables. The `mandatory` constraint means a value is required for that property when an instance is created (similar to `NOT NULL`). If no value is explicitly given for a property, then its value is `NIL`. `NIL` functions very much like `NULL` in a relational database. The `mandatory` constraint means that an error is generated if an attempt is made to assign `NIL` to the property. We can also specify unique constraints and provide default values.



➤ *Figure 2*

The `String` datatype functions similarly to SQL's `VarChar` datatype. We can indicate a maximum length as shown for the `State` and `ZipCode` fields, or by not specifying a length, the space used by the value is limited by the system maximum of 65,536.

Once a class is defined, it must be built before it can be used or before any instances can be created for it. Before a class is built, its definition can be redefined at will without the need to delete the existing definition. Also, references to other classes do not require that those classes exist or are even defined yet. However, before a class can be built, all referenced classes must already have been defined, but not necessarily built. To build a class, we simply call the `buildClass` ODQL command and name the class, like this: `buildClass ExampleCF::Person;`.

Subclasses

It seems pretty obvious that the `Person` class is intended to be a base class for different types of people like customers, employees, managers, contractors, etc. Let's create our first subclass for employees using the ODQL shown

in Listing 2. Since we are descending from Person, that becomes the superclass for Employee, and we then inherit all the properties from Person. The only additional properties we will add are an employee number, a department reference and a list of hobbies for the employee (we are a particularly nosey company).

But wait. Our class definition shows not three properties but four, including one called NextEmployeeNo. The types of properties we've been dealing with are instance-level properties, which means that separate property values are held for each instance. NextEmployeeNo is an example of a class-level property.

Class-Level Properties

A class-level property is accessed just like any other property, but its value does not change from instance to instance. Values for class-level properties are fixed for every instance in the class. Note carefully that I said they were *fixed*, not *constant*. Values in class-level properties can change, but when they do, they change for every instance in the class. Think of them as 'global variables' in the class that retain the same value no matter what instance you are examining at the moment. They are simply accessed as a property, but their values are not dependent on nor affected by the values in the instance.

In this case, we are holding an integer value to use as the next employee number when we create a new employee. Note that this is in no way an automatic function of the class. We will have to provide code to increment NextEmployeeNo and assign it to EmployeeNo when we add a new instance. Normally, we would do this by defining a method for the class, and we would always use that method when adding new employees.

Multi-Valued Properties

The declaration of the Hobbies property in Listing 2 looks a little strange. The requirement is to store zero or more names of hobbies the employee is interested in. In an SQL table we could provide a single varchar column with a comma-separated list of hobbies. Or we could create a separate EmployeeHobbies table containing an employee number and a varchar field for the hobby name. There would be a one-to-many relationship between Employee and EmployeeHobbies. If we were really going to town on this, we would define a third table called Hobbies which would have a hobby number and a hobby name, and EmployeeHobbies would be reduced to an intersection table containing employee number and hobby number links.

In our class definition we have simply made a multi-valued property which holds more than one value for a single instance of the class. This is something that is not possible with SQL tables. The Employee.Hobbies property is declared with the Bag keyword. A bag is a collection of values of the datatype given within the angled brackets, in this case a collection of strings. In Delphi terms, this is similar in concept to the Params property of the TDatabase class. Strictly speaking Params is a single instance of the TStrings class though, not a true collection of strings.

Looking back at the Jasmine class hierarchy shown in Figure 2, we see that the Bag class descends from the Collection class and that Bag, Array, List, and Set are specialized collections of other classes. We can more accurately compare Collection to Delphi's TList and say that it is a class that manages one or more instances of another class. However, TList handles a series of untyped pointers which we must explicitly typecast as their actual classes when we reference them. In Jasmine we can define precisely which class our collection manages and reference the instances as that class implicitly.

Class Relationships

The Employee.Dept property in Listing 2 shows how we would set up a reference to another class. Here the Dept property returns an instance of the as yet undefined class Department. Note that we're not getting back a department number or key value like we would expect in a SQL table, but we're getting back the entire department instance. If you think about it in terms of Delphi classes, if you access the Font property of a TEdit control, you are really accessing an instance of TFont from an instance of TEdit. The same concept applies here. The database maintains the reference by storing the object identifier (OID) for the Department instance that has been associated with a given Employee instance.

In an SQL table we would have probably stored a department number as an employee column. Then when we wanted to access all the employees in a given department, we would just filter on the department number in the employee table. Using the department number in this way would give us a one to one relationship between Employee and Department and a one to many relationship between Department and Employee.

It would be possible in our object oriented database to obtain the OID of a particular department and scan the Employee class for matching OIDs to find the subset of employees for the department. But there is a better way. Listing 3 shows the class definition for Department. We define only three properties: a department name, a manager (returning an Employee instance for the department manager), and a collection of

➤ *Listing 2*

```
defineClass Employee
  super: Person
{
  class:
    Integer     NextEmployeeNo default: 1;
  instance:
    Integer     EmployeeNo mandatory:;
    Department  Dept;
    Bag<String> Hobbies;
};
```

employees assigned to the department.

The Staff property is a multi-valued property, or collection, containing references to all the employees in the department. We use the Bag keyword again and specify that this is a collection of employee instances. Remember that String is just a class in the Jasmine object hierarchy, as Employee is, so there's no conceptual difference between a collection of string instances and a collection of employee instances. Behind the scenes, Staff contains a list of OIDs for all the employees assigned to a given department. We could iterate through the Staff collection and gain access to the complete instances of each employee in the department. It is through this multi-valued property that we define the one-to-many relationship between Department and Employee.

Many To Many Relationships

Many to many relationships in data are not uncommon. Our employee hobbies example is one such case: an employee can have any number of hobbies and a hobby can be associated with any number of employees. Another more realistic example is a class for products and a class for suppliers. We may have more than one supplier for the same product, and our suppliers will generally be providing us with more than one product. In this case, a relational database will always require an extra table to do nothing but hold the keys that make the associations between the two entities. We would have a Products table, a Suppliers table, and a ProductsSuppliers table, probably containing nothing more than a product code and a supplier code.

In an object database, we would simply have a multi-valued property in the two main classes, referring to the associations in the other class. The need for a separate relationship class is eliminated. Listing 4 shows a simplified example.

➤ *Listing 4*

```
defineClass Product
  super: Composite
{
  instance:
    Integer       ProductCode;
    String        Name;
    Bag<Supplier> Suppliers;
};
defineClass Supplier
  super: Composite
{
  instance:
    Integer       SupplierCode;
    String        Name;
    Bag<Product>  Products;
};
```

Methods

Methods are to object databases what stored procedures and triggers are to relational databases, but methods offer much more flexibility and are more seamlessly integrated into the database architecture. A good deal of database functionality is built into the object hierarchy in the form of system methods. For example, the system class Composite (from which all user-defined data classes descend) implements a system method called delete() which deletes an instance from the class. In SQL we would use the DELETE statement with a WHERE clause that isolated the row we wanted to delete. In an object database, we would obtain the instance of interest and call its delete() method.

Like properties, methods can be instance-level or class-level. Instance-level methods are called from and typically operate on an instance of the class. We would define a class-level method to perform operations that do not require a specific instance of the class. For example, we might have a class called Orders containing customer orders. In this class we might define a method called ReadyToShip which returns a collection of all instances in Orders that are identified as ready to ship. The method does not operate on any particular member of the class, but rather on the class as a whole, so we would define it as a class-level method.

```
defineClass Department
  super: Composite
{
  instance:
    String          Name mandatory:;
    Employee        Manager;
    Bag<Employee> Staff;
};
```

➤ *Listing 3*

We can write our own methods for the classes we create using C, C++, or ODQL (or a mixture). Referring back to our Person class shown in Listing 1, let's define a method called Age which returns a person's age in years calculated from their date of birth and today's date. Listing 5 shows how we write an ODQL method attached to the Person class.

The addProcedure ODQL command adds a method definition to an existing class definition. The first argument is the class returned by the method, which in this case is an integer. If the method were to act as a procedure rather than a function, we would specify a return class of Void. Following the return class, we identify the interface to the method by its class name, level, method name, and parameter list. Here we are adding a method to the class Person, it is an instance-level method, its name is Age and it has no parameters.

The body of the method is then defined within braces. The ODQL statements within the method body are preceded with the $ symbol. The first line defines a variable called Today whose datatype is the system class Date. The second line sets the value of Today to be the current date set on the system clock by calling Date's class-level method getCurrent(). The third line uses the date stored in Today and calls the instance-level method difference() to compute the number of years between today's date and the date in the

➤ *Listing 5*

```
addProcedure Integer Person::instance:Age()
{
  $Date Today;
  $Today = Date.getCurrent();
  $return(Today.difference(self.DateOfBirth, YEAR));
};
```

DateOfBirth property. Within methods, we can use the self variable to refer to the instance object that the method is operating on. In our case, self refers to the Person instance which we are using to call the Age method. This calculation is then passed into the ODQL return statement which is used to end execution of the method and set the return value.

Using Methods To Access Legacy Data

Method code could also include C or C++ code and as such could access anything within the system that could be accessed through a C/C++ program. For instance, we would make Windows API calls or even interface to the client API for a completely different database system, providing a gateway to legacy databases through a single database architecture. Jasmine even goes so far as to provide specialized classes for accessing SQL databases so that you can use SQL in your methods and queries to process data within existing SQL databases. This is a significant advantage for object databases since you can build bridges to the legacy systems in the object database, and your client applications would access the legacy data as though it were class instances in the object database. With this multi-tiered approach, client applications would only be concerned with connectivity to the object database. The object database server would be the only point requiring direct connectivity to the legacy databases.

The Trigger Effect

There are no triggers *per se* in an object database. Think about what a trigger is used for in a relational database. A trigger defines some action to take place when a row is added, modified, or deleted. For example, if a row was deleted in a master table, you could write a delete trigger to automatically delete associated rows in a child table. If the values in certain columns were changed, you could write an update trigger to make appropriate adjustments in other related tables.

In an object database, we delete an instance by calling its delete() method. If we want to take some special action when an instance was deleted (like a cascading delete), we would use the polymorphic characteristic of objects and reimplement the delete() method for our class. Jasmine calls this 'refining a method' while in Delphi we've come to call it a 'overriding a method.' To override a method we simply define a method in our subclass with exactly the same name, return class, and parameters as the method in the superclass we are overriding.

Listing 6 shows an example of a delete() method override for our Employee class. When we delete an employee we also want to delete all associated instances of performance reviews in the Review class. The Review class contains a method called DeleteReviewsForEmployee which does this for us, we just need to call it when we delete an employee.

The first line calls the class-level method Review.DeleteReviewsForEmployee. We're assuming this method accepts an instance of Employee as its parameter so it can figure out for itself which employee we're talking about. Next we need to call the delete() method of the superclass to make sure all the normal delete operations get performed. The second line does this and includes some special syntax to make sure we get the right delete() method.

If we had just said self.delete(), we would be recursively calling this same method over and over. What we really want is to call the implementation of delete() that exists in the immediate superclass, Person. We can qualify the method name with the name of the class containing the specific method implementation we want. For example, self.Person::delete(). Or we can qualify it with the special keyword super as we've done here to mean 'take the delete() method from my superclass, whatever the class happens to be named'. In Delphi terms, this is equivalent to the inherited keyword.

Methods in an object oriented database are all considered virtual. That is, whenever we execute a method for a given instance, the correct method implementation in the class hierarchy is always used depending on which class the instance was created for. Remember that all instances in a given subclass are also present in every superclass and can be accessed from any of those superclasses. For example, let's suppose we have different implementations of delete() in Person and Employee. If we delete an instance in Person that happens to exist in Employee, then the Employee.delete() method gets executed even though we are accessing the Person class.

Conclusion

What I hoped I have shown you is that setting up an object oriented database has some similarities to a traditional relational database and some similarities to the Delphi object architecture. Both of these concepts should be very familiar to us and, therefore, working with an object oriented database should not be a monumental re-education effort. Now that we've got a good handle on defining the structures in an object database, next month we'll look at accessing the data and how to plug it into a Delphi application.

➤ *Listing 6*

```
addProcedure Void Employee::instance:delete()
{
  $Review.DeleteReviewsForEmployee(self);
  $self.super::delete();
  $return();
};
```

Steve Troxell is a software engineer for Ultimate Software Group in the USA. He can be reached at Steve_Troxell@USGroup.com