Oscillating Between Objects and Relational: The Impedance Mismatch

By
Dan Shusman

Oscillating Between Objects and Relational:
The Impedance Mismatch

Introduction

Impedance mismatch is a term commonly used to describe the problem of an object-oriented (OO) application housing its data in legacy relational databases (RDBMS). C++ programmers have dealt with it for years, and it is now a familiar problem to Java and other OO programmers.

Impedance mismatch arises from the inherent lack of affinity between the object and relational models. Problems associated with the impedance mismatch include class hierarchies binding to relational schemas (mapping object classes to relational tables), ID generation, concurrency, as well as other problems described below.

The impact of these issues is tied specifically to the blending of OO application and relational schema. But the ramifications are clear in terms of time-to-market**,** costs of design, development, and quality assurance, compromised code maintainability and extensibility, and the sizing and topology of the hardware required to ensure expected response and throughput times.

Given the increasing prevalence of the OO RDBMS impedance mismatch—and its corollary, the mismatch between SQL-based applications and object databases (OODBMS)—an examination of approaches to resolving the resulting problems is both timely and worthwhile.

Object Development Languages

Many long-popular technologies such as C++, Microsoft's Visual Basic, Borland's Delphi, the now-maturing Java language, and a host of open source languages provide an object environment in which to implement business logic and user interfaces. To greater or lesser degrees, these OO environments implement encapsulation, polymorphism, and inheritance. The benefits of their proper use in application development and evolution are well-known.

But Where to Put the Data?

Object languages, like all programming languages, need to bind to a data store if persistence is required and that store is often a database. The three most common data models are relational, object, and post-relational, a.k.a. transactional multidimensional.

The Relational and Object Database Models: Fundamental Differences

It would be useful to contrast the basic differences of the RDBMS and OODBMS models and the approaches to programming in each.

Simply stated, tables in the relational model contain information (columns) that organize the information in rows. Complex data structures can require many tables. Relationships among tables (one-to-one, one-to-many, and many-to-many) are based on foreign keys.

Business logic operations are applied from sources outside the table, for example, through the use of embedded SQL or static, pre-coded stored procedures or triggers. To build an effective and efficient application in the relational model, the developer must have a comprehensive knowledge of the tables, any relationships among them, and of these external logic components.

In contrast, classes in the object model are self-contained entities. In common with relational tables, they contain their information (properties). But a significant difference is that related data (so-called embedded classes and collections) can be stored within the "container" class rather than as a separate tables requiring a "foreign key"-type construct.

Another significant difference is that business logic is not applied externally in the object model. Instead, a class implements methods that contain code for operating on the class's properties. Methods provide interfaces through which they are called and thereby the application developer is buffered from the complexities of the schema.

Not Another Invoice Example!

An example of database design and coding in each of the models will demonstrate the differences previously described.

Consider an automobile registration application. An automobile has its data—make, model, trim line, year, VIN, etc. It has one or more owners, one or more drivers, a repair history, etc. An automobile must also be registered, scrapped, etc.

To represent an automobile's data in the object model, you would create Car, Person, Owner, Driver, and RepairHistory classes.

The Owner and Driver classes would inherit properties and methods from the Person class. You would extend those two as necessary with any unique properties and methods

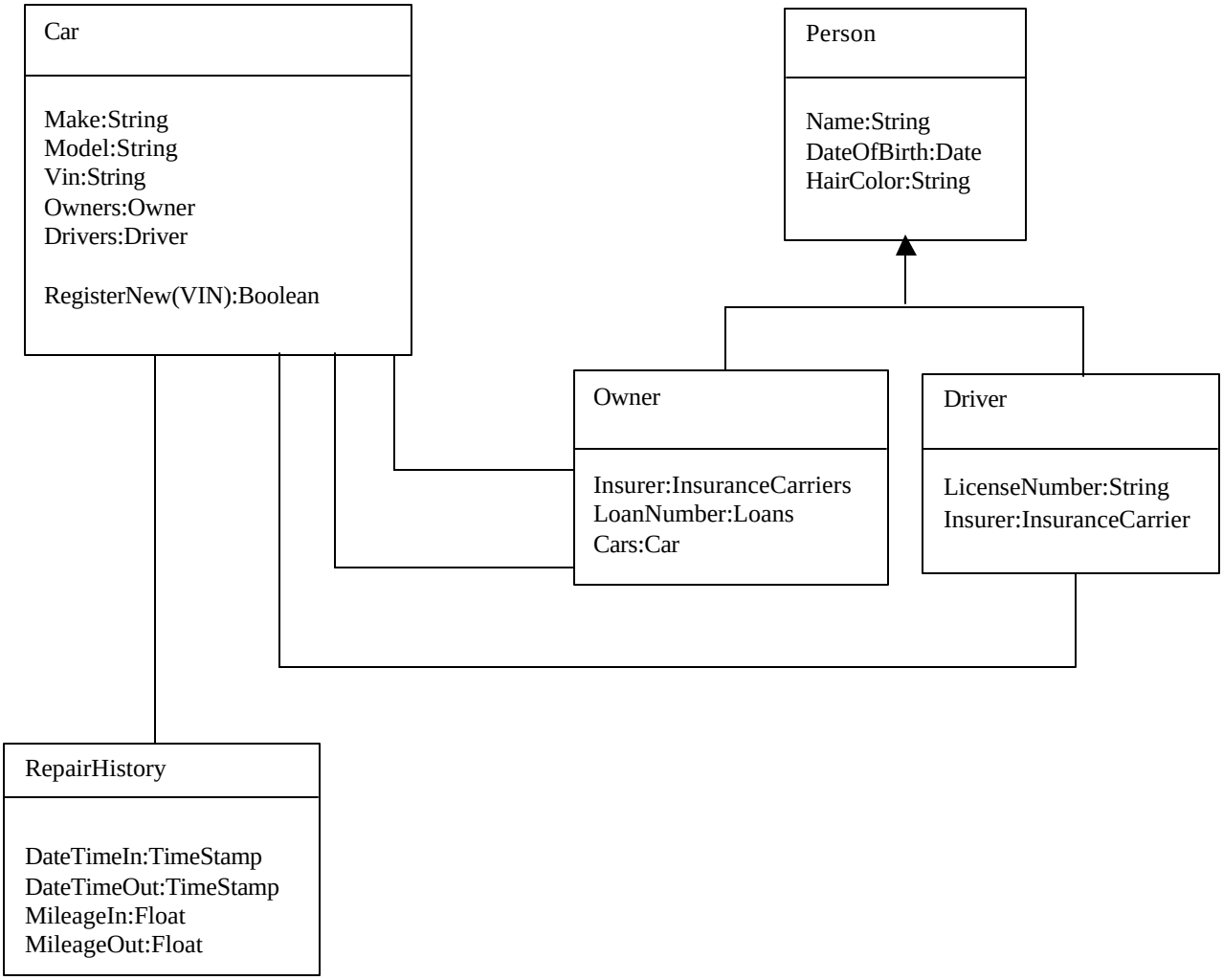
Owners, Drivers, and RepairHistory would be considered collections within the Car class—perhaps collections of references for the former two, and embedded instances for the latter.

To make this more realistic, let's specify that an Owner can own many Cars and a Car can have many Owners. To implement this many-to-many relationship the Owner class might contain a collection of references to Car. (As above, the Car already has a collection of Owners.) You might also implement a many-to-many relationship for Cars and Drivers.

Finally, let's specify a method for the Car class, RegisterNew(), called with the parameter VIN. All the work required to register a new car would be handled behind this interface.

These specifications are diagrammed in Figure 1.

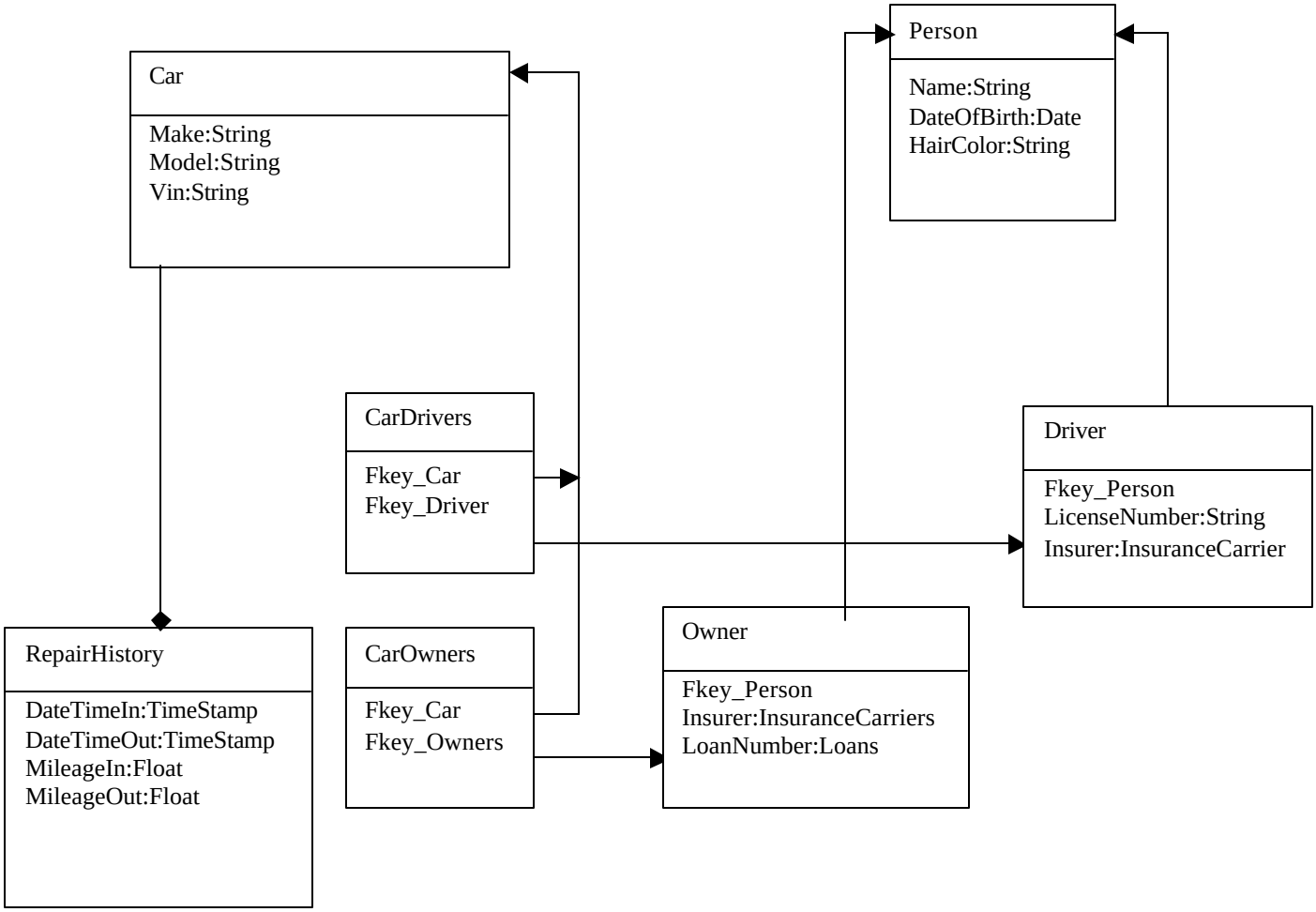
Figure 1



The Same Example in the Relational Model

To represent the example in the relational model, you would create several tables as shown in Figure 2.

Figure 2



Note the additional tables (CarDrivers and CarOwners) needed to maintain the many-to-many relationships between Owners, Drivers, and Cars.

Writing Some Code

Let's perform two activities.

Examples 1a and 1b show the use of Object and SQL programming, respectively, to report a Car's VIN and find all of its Owners.

Example 1a – Listing Car VINs and Their Owners: OO

```
objCar=Open Car(vin);
owner_count=objCar.Owners.Count()
For (i=0; i<owner_count; i++) {
        owner_name=objCar.Owners.Get(i).Name;
}
```

Example 1b – Listing Car VINs and Their Owners: SQL

```
Select Car.VIN,Person.Name
From Car, CarOwners, Owner, Person
Where CarOwners.Fkey_Car=Car.CarID
        And CarOwners.Fkey_Owner=Owner.OwnerID
        And Owner.Fkey_Person=Person.PersonID
        And Car.VIN=:vin
```

Examples 2a and 2b are pseudo-code showing the object and SQL approaches to registering a new Car.

Example 2a – Registering a New Car: OO

```
// somehow we acquired the car's VIN, make, model and list of owners
// use a class method rather than an instance method  to validate
rc = Car.IsValidVin(vin)
//  if rc indicates an error, logic to reject goes here
// now assume an owners[i] array and this is where the logic for validating them goes
// instantiate a new car
objCar = New Car;
// assign the properties
objCar.VIN = vin;
objCar.Model=model;
objCar.Make=make;
// now register the car
objCar.RegisterNew(vin)
```

```
// assign the Owners
For (i = 1; i < count; i++) {
        obj.Car.Owners.SetAt[i] = owner[i]
}
// make it persistent
objCar.Save()
```

Example 2b – Registering a New Car: SQL

```
// somehow we acquired the car's VIN, make, model and list of owners
//validation of information is external to the table
If (vin == "") ! (vin=0){
        Exception("VIN required")
}
//Other fields would be validated here
//
&sql(Insert Into Car(Make, Model, VIN) Values(:make, :model, :vin)
// recover the new assigned Car ID
&sql(Select CarId Into :car_id from Car Where Car.VIN=:vin)
// and assign the owners to the Car
For (i = 1; i < count; i++) {
        &sql(Insert Into CarOwners(CarKey,OwnerKey) Values(:car_id,:owner[i])
}
```

An Impedance Mismatch When Manipulating the Database

An object programmer working in a pure OO environment might approach the development by writing the code represented by Examples 1a and 2a. An SQL programmer's approach using an RDBMS is outlined in Examples 1b and 2b.

An object programmer working with an RDBMS must somehow blend Examples 1a and 1b with those parts of Example 2b that touch the database. To put it another way, the persistence methods of each class—(Open(), Save(), Delete(), and so forth—must now be coded in SQL (or in another RDBMS technology, e.g., stored procedures) to set and fetch data as needed. For instance, in the Car example,

- The Open() method would execute an SQL SELECT query that recovers the columns of the Car table and binds their data to the attributes in the Car class.
- The Count() method used to iterate overall Owners would require code like this:
  Select Count(*) From CarOwners Where CarOwners.Fkey_Car=Car.CarID AND Car.VIN=:vin)

- The Get(index) method that examines each Owner instance to recover the Name would require code that would populate a 'result set' with a query similar to:

```
Select Person.Name
        From Car, CarOwners, Owner, Person
Where CarOwners.Fkey_Car=Car.CarID
        And CarOwners.Fkey_Owner=Owner.OwnerID
        And Owner.Fkey_Person=Person.PersonID
```

It Doesn't Stop There: More Subtle Complexities of the Impedance Mismatch

Additional issues that must be considered when attempting to blend an OO application with an RDBMS include:

- *Associating the classes with one or more underlying tables*
  The task at hand is to associate the attributes of each class with a column in some table. A one-to-one correspondence of a class to a table is easy to grasp, but the most appropriate object model might require a class that spans a *number* of tables, taking a subset of columns from each. Or multiple classes might map to the *same* table and reinterpret the semantics of the table according to their respective needs (which might require a new column in the table to hold the class name responsible for the row.) The reinterpretation and possible manipulation of the relational schema to project data to the object model can rapidly dissociate the object and relational schemas.

- *ID generation – the Object ID and Inserts into the RDBMS*
  Object ID generation and the process by which unique keys are generated in the RDBMS by an object Save() method are of great importance. In a one-to-one association of a class and a table, often the best solution is to use the RDBMS engine to generate the next ID and that value becomes the object ID. But a class might span multiple tables and its Save() method might require multiple relational IDs that as a group, must be associated with a single object ID. This association must persist so that when the object is opened, the rows from multiple tables can be recovered to populate the instance. Therefore, an additional table has to be built and maintained to associate the object ID with the (multiple part) relational ID.

- *Validation and other checks*
  If the original SQL-based application used against the RDBMS has applied external validation to data, that code must be identified and moved into the object application.

- *Handcrafting persistence methods and order of operation*
  You must be very conscious of application-level data integrity. If you are implementing a Save()

method, the operations (Insert, Update, and Delete) might require a specific sequence to ensure application-level database integrity.

- *Attending to concurrency*
  Another influence on application-level data integrity is concurrency. You must ensure that the levels of Locking—from one to exclusive read/write—offered by the RDBMS are reflected in the persistence methods of the class.

- *Attending to schema evolution*
  Because of the object-relational mapping, care must be taken that the object schema and the relational schema remain coordinated as they evolve. The possible dissociation of the object schema from the original relational schema (discussed earlier) will complicate this.

- *Maintaining dual use*
  All of the previous issues have bearing on whether the original application and the newly crafted object application can operate simultaneously against the original RDBMS.

Remedies Offered by the Marketplace

There are four typical approaches to addressing the impedance mismatch issue.

1. Use object facilities of a relational database
2. Use object-relational mapping tools
3. Adopt an object database
4. Adopt a post-relational database

1. Using Object Facilities of a Relational Database

While RDBMS vendors have made valiant efforts to add object capabilities to their engines, they have not really solved the problem. None is a true object implementation. In many ways, they suffer from the impedance mismatch themselves. Inheritance, polymorphism, and encapsulation are not concepts easily blended into the core technologies offered by these vendors. In the end, the developer is reduced to manipulating a quasi-object layer atop the still-existent RDBMS.

2. Using Object-Relational Mapping Tools

Many database vendors or third parties offer products that provide “object-relational mapping” software to associate application classes with underlying tables in an RDBMS. These tools might include a runtime caching component used to enhance the performance and integrity of database operations (inserts, updates, deletes, selects, and associated concurrency requirements), methods for unique ID generation, and generation of bindings (JavaBeans, COM objects) for use with supporting languages. Such tools can be

very effective in helping develop the code required to expose a relational schema to an object-based application. There are quite a few such packages that are well-regarded.

But object-relational mapping tools tend to shift responsibility back to the developer as problems become more complex. This is especially true when mapping a class across tables or mapping multiple classes to a single table**,** and when dealing with the resulting ID generation intricacies.

3. Adoption of an Object Database

Another approach to dealing with impedance mismatch is to adopt a true object database. The advantage to using these databases is their affinity for OO concepts.

Some considerations are:

- The market share for such databases is small
- Transactions posted against the RDBMS must be available to the OODBMS
- An impedance mismatch exists between SQL-based applications and object databases

Another consideration would be the example of an end user with sophisticated decision support (DS) tools on the desktop. Most likely those tools are composing SQL queries under the interface using ODBC to connect to targeted databases, therefore requiring a relational view of the object-based data. Some OODBMS vendors deliver relational-object mapping strategies deployed on a middle tier computer to overcome this mismatch.

4. Adoption of Caché, the Post-Relational Database

Post-relational (a.k.a., transactional multidimensional) databases such as Caché implement an associative array technology that can be projected to an object or relational model simultaneously and without intervening mapping tools or caching middle tiers.

Persistence methods such as Save() are projected directly to the object developer and are implemented by the database itself through native commands that manipulate the associative arrays. Simultaneously, these engines can project a relational view of the same (associative array) data exposed through ODBC and JDBC. The implementation of Insert and other SQL DDL, DML, and DCL commands result in the same native database commands that implement the object persistence methods. Since multidimensional engines are able to implement simultaneous and direct access by each projection, they minimize, if not obviate altogether, the issues of ID generation, concurrency, validation, etc.

Simultaneous and direct access by each projection (including management of concurrency) implies that an SQL-based application (VB, C++, or Delphi over ODBC) can operate at the same time as an object-based

application (VB, Java, or C++). Also, the relational projection would allow end users to employ their favorite SQL-based front-end tool for decision support.

Conclusion

We have examined the major approaches to resolving the impedance mismatch between OO languages and RDBMSs: trying to implement against the partial object support offered by some RDBMSs, employing an object-relational mapping tool, adopting an object database, or using a post-relational (transactional multidimensional) database.

Some of the questions to consider when evaluating such technologies include:

- Is the technology easily adopted (learning curve, etc.) and integrated for immediate need?
- Is the technology extensible enough that it can satisfy future application and database needs?
- Is the technology flexible enough to integrate newer technologies like XML, SOAP, and others?
- Does the technology allow the developer to implement code in the correct logical tier? In other words, will code that is better-executed on the database server have to be moved instead to an application server or other middle-tier strictly to accommodate the technology?
- Can the technology provide the scalability and performance needed to enable an organization to deploy it to the Web and/or enterprise-wide without undue impact on infrastructure and operations?
- Is the technology useful in resolving certain development issues only to create additional downstream run-time problems?

One thing is certain: The marketplace is quite clear that OO is the desired approach for new application development and evolution of legacy applications. Organizations with heavy investments in RDBMSs must decide whether they will take a short, medium, or long-term view in mitigating the effects of the impedance mismatch between the object and relational paradigms.

InterSystems Corporation
World Headquarters
One Memorial Drive
Cambridge, MA 02142
Tel: 1.617.621.0600
Fax: 1.617.494.1631

www.InterSystems.com