Avoiding the Quagmire

Offering solutions to the problems of Object/Relational-Mapping

by Ted Neward
May 21, 2007

For almost a full decade now, the Java community has struggled with the Object/Relational impedance mismatch, a fundamental problem for those who would seek to store rich domain objects into a relational database. While this problem is not unique to Java—C++ developers wrestled with the problem long before Java was available, and .NET developers are just beginning to discover the depths of the issues involved—it is in Java that this particular battle has been fought most publicly and extensively. No less than four "standards" have been offered up as solutions (EJB Entity Beans, Java Data Objects, Java Persistence Architecture, and the aborted data-mapping solution proposed for Java6), and countless open-source projects have all attempted their own solutions, the most famous of which include Hibernate and Castor JDO. .NET developers are finding similar kinds of solutions, both from Microsoft (in the form of LINQ and language enhancements to C# and VB) and from the open-source community (in the form of NHibernate) and commercial sector.

As discussed in the summation of the essay "The Vietnam of Computer Science" (http://www.odbms.org/about_news_20070212.html), there are several potential solutions regarding the potential quagmire of the object/relational problem:

1. **Abandoment.** Developers give up on objects entirely and return to programming in a model that's more relational-friendly. Most developers will agree that this is a radical and unattractive approach, particularly given the success of object-oriented languages over the last two decades, as evidenced by the millions, if not billions, of lines of object-oriented source code executing quite successfully today.
2. **Manual mapping.** Developers issue SQL queries against the relational database, harvest the results, mapping them to objects within the domain model through hand-written code. While this offers the greatest amount of control over the mapping of relational tables and the domain model, it also represents the greatest amount of work

and time required, leaving a sour taste. It does, however, provide the greatest degree of "loose coupling" between the domain model and the database, a desired attribute in those situations where the object developer does not control—and has little influence on—the relational schema, a common scenario in many IT shops where multiple languages are used against the same database.

3. **Acceptance of O/R-M limitations.** Developers simply accept that O/R-M cannot "go the last mile" to completely eliminate the impedance mismatch from the developer's perspective, and use O/R-M tools for that percentage—be it 50%, 75%, 95%, whatever—of their persistence solution to which it can be easily applied. Developers will need to have a good understanding of their O/R-M tool to do this, however, because they will need to make sure that their "out-of-band" direct queries against the database do not confuse or corrupt any sort of caching scheme the O/R-M tool is doing internally. This implies a greater educational investment on the part of developers, who frequently turn to O/R-M as a way to *avoid* having to learn too much about how the mapping between objects and relational database is handled.
4. **Integration of relational concepts into languages.** Developers use languages that integrate relational concepts directly, thus eliminating the impedance mismatch. Relational queries return sets, which are then used directly or indirectly within the language or framework, given first-class status from the language. While attractive, unfortunately this option is the least approachable as of this writing, as none of the current tools and technologies in widespread use (C++, C#, Java) provide any practical support today. Future tools and technologies, such as Microsoft's LINQ project, which looks to integrate new language enhancements and supporting framework classes to the C# and VB languages (scheduled for release in their next-generation Visual Studio product, code-named "Orcas"), may provide viable options for developers seeking this option, but for now, this remains a "future" option for all practical purposes.
5. **Integration of relational concepts into frameworks.** Developers can choose library classes or frameworks—such as Microsoft's DataSet, Ruby's ActiveRecord or Java's RowSet—that provide a thin wrapper over relational concepts (tables and rows and columns), thus minimizing the mapping between objects and relations by choosing relations as a natural storage mechanism for data even within the programming language. Such an approach brings relations closer to

the programming language, rather than historical approaches which attempt to bring objects closer to the database. While some development shops have found this approach somewhat successful, debate rages over its usefulness and applicability.

6. **Integration of object concepts into the database.** Instead of seeking to bring relations closer to the language, developers choose the opposite approach, that of bringing the objects entirely into the database, using an Object-Oriented DataBase Management System (OODBMS) for storage of their objects, rather than the traditional RDBMS. Using an OODBMS, such as db4object's db4o product, or Versant's Object Database, or Objectivity's ObjectSpace[1], developers store objects directly, rather than taking them apart into constituent fields and storing those fields into columns and rows, thus eliminating the need for a "mapping" layer entirely. While using an OODBMS will not completely eliminate all of the problems described in the "Vietnam" essay, it does address some of the more egregious problems, and thus frequently provides the developer a better chance of avoiding the quagmire and allowing them to focus more clearly on the problem at hand.

The remainder of this paper discusses the impact of choosing this final approach, using db4o as the database.

The object/relational impedance mismatch

For a more detailed discussion of the object/relational impedance mismatch, readers are encouraged to consult the original essay; in this paper, we examine the issues raised in that paper, and the various ways by which the db4o OODBMS implementation seeks to solve, or at least minimize, the dangers. Recall, however, that the main argument of that essay was that object/relational mapping faces the Law of Diminishing Returns: that early successes lead to a commitment to O/R-M that, if left unchecked and unexamined, leads to greater and greater "pain" as the various O/R-M issues begin to surface. Thus, a large part of the goal of an OODBMS-based persistence scheme will be to minimize those issues down the road, but more importantly, offer developers a greater flexibility under certain scenarios.

Many people, including Chris Date[2], have suggested that there is less of an impedance mismatch between a relational model and the object world than is

[1] See http://www.odbms.org/vendors.html for a complete list of products

[2] Introduction to Database Systems, 8th Edition, pp 813-886

commonly believed. Date, for example, suggests that objects are simply relations with associated behavior attached, and type inheritance is described in a manner similar to what we see in classic object languages and in any event, becomes an implementation detail in the relational database. (To those who would point out that no "object-relational" database provides any such support whatsoever, presumably Date would also be quick to point out that no commercial RDBMS implements relational theory correctly, either.) In some respects, this view of the impedance mismatch is true, but more practically, the mismatch still stands: the differences between an object approach and the relational theory are somewhat stark. For example, associativity in relational theory and object models stand entirely 180 degrees to one another: in a relation, associations are held by the thing referred, whereas in an object, associations are held by the referrer (via a reference or pointer). This single difference, for example, will result in some significant differences between how each reacts and present some core problems when trying to bridge the gap between the two.

The object-to-table mapping problem

One of the first, and most easily visible, problems with an object/relational mapping tool is that of the mapping itself. Deciding how tables and classes will align against one another can be a difficult prospect. Although not an insurmountable problem, as tools like Hibernate or JPA demonstrate, establishing good mappings between tables and classes can still be tricky, all the more so because frequently the mapping starts out very simply, with a one-to-one alignment between columns of a table and fields of a class, leaving developers to think that all mappings will be this straightforward.

In an OODBMS, there is no mapping present, because the OODBMS infers the schema directly from the class in question. In the case of db4o, when the object is sent to the database, the db4o engine uses VM metadata similar to that revealed by Reflection (java.lang.reflect.* in the JVM, System.Reflection in the CLR) to examine the object's fields, and makes on-the-spot decisions about its storage at that point. This avoids the need for a post-compilation "schema generation" step to create the relational schema, or, in the more frequent case where the relational schema is fixed, the pre-compilation code-generation step to infer objects out of the schema which then developers are constrained around.

Naturally, this also means that an OODBMS has an intrinsic understanding of two of the mapping problems that pose the greatest dilemma to the object developer: inheritance and associations. In the case of inheritance, the OODBMS understands natively the relationship between parent and child classes, and frequently that relationship is stored firsthand as part of the stored object; this means that no matter how deep the inheritance hierarchy, no costly joins and artificial associations must be made (in the case of table-per-class mapping) or extraneous column fields kept (in the case of table-per-concrete-class or table-per-hierarchy mappings).

The student of relational theory may, at this point, be curious as to where relational integrity constraints, one of the great strengths of the relational model, are kept in an OODBMS. In fact, an OODBMS expresses no integrity constraints directly, except those that are defined in the class hierarchy itself; remember, in an OODBMS, the schema is the object's class definition, not a standalone artifact, so any integrity constraints desired must be expressed directly inside of the class definition. This has the added advantage of keeping the integrity constraints consistent throughout both the storage and working memory tiers, something that requires some kind of housekeeping, either manual or automated[3], in an O/R-M scenario.

This also means that refactoring an OODBMS is much more straightforward, as db4o can silently handle changes to classes even in those situations where production data for that type already exists. So, for example, adding a field to a class when existing objects (without that new field) are already present in the data store is not a problem; db4o simply "extends" the stored objects to include the new field, setting its value to be the default for that type (typically 0, false, or null, depending on the type) and returns the object when queried. Db4o can also go the other way, allowing classes to remove fields (and those field values will simply not be returned when the object is queried) without penalty, even going so far as to store the old values in case that field removal is ever undone[4].

Internally, the data is stored in similar techniques to that of the world's favorite RDBMSes, but this is all hidden from the db4o user—db4o never exposes its internal data storage scheme to the developer[5], and thus the

[3] It should be fair to point out that in most cases, an automated tool should be able to track these updates without requiring developer intercession; however, the old 80/20 rule does come into play here, in that roughly 20% of the mappings created will need hand-tuning and adjustment and therefore developer time.

[4] The db4o documentation has more on how and when db4o can track these old values; suffice it to say, db4o will *not* track these values indefinitely, as that would be a tremendous waste of storage and time.

[5] Although there is a Reflection-like API for examining the structure of stored objects, for those situations where a lower-level view is desired.

developer is left with the perspective that the class definitions she writes are the only schema definition used. (Ironically, or perhaps entirely appropriately, this is precisely in line with Date's idea of encapsulation from the actual details of data storage, a key cornerstone in his definition of a database.)

Besides an easier refactoring scenario, the OODBMS approach also creates a much tighter coupling between the executing objects and their persistence mechanism, which can be either good or bad, depending on the scenario. In certain situations, the RDBMS is used as a point of interoperability, where differing languages/platforms can exchange data, and thus the loosely-coupled nature of the RDBMS/SQL approach stands to be more useful. In situations such as "silo" applications where a single user interface accesses a single database (the traditional "baby webapp on top of a big database"), or the more leading-edge "service" implementations, however, all interaction will be through that user interface or service interface, and never against the database itself, thereby making persistence truly an implementation concern only. In these situations, an OODBMS back-end can be invaluable in defining and preserving a rich domain model, as now there are no entity definitions in two languages (Java/C# and SQL DDL) to be reconciled.

The dual schema problem

In many respects, the object-table mapping problem is reflective of the larger problem, that of the *dual-schema problem*, in that in a traditional object/relational world, two sets of entity definitions are in play: one defined by the programming language itself, the other by the relational model using SQL DDL. This sets up an inherent challenge, as now two sets of definitions must be kept up to date as the system grows and evolves, either by "slaving" one to the other (frequently seen by the use of code generation tactics, either from schema to classes or the other way around), or by editing/adjusting the two separately and hand-tuning the mapping between them as necessary. This creates a tension between the two, and frequently developers are forced to make sacrifices in the purity of both models in order to keep the two in sync with one another.

Again, in an OODBMS, the fact that the class definitions are the only schema present means that no such "dual schema" problem exists; the domain model need not be slaved to the storage definitions, and the storage definitions need not be twisted into strange formations just to support the storage of a rich domain model.

The schema-ownership conflict

One issue faced by many developers struggling with the object/relational problem is that relational databases now stand firmly within the borders of the IT kingdom, and her citizens are loath to allow foreigners to define how things are done there. In many companies, IT admins and database administrators have very specific ideas about “the way things should be”, and frequently their policies and procedures present obstacles to refactoring. In some companies, in fact, the definition of new database views, stored procedures and triggers are under the control of the DBAs, rather than the developers, which now means that not only does the organization need to sync up the two different code artifacts, but now this is done across two disparate groups entirely.

db4o offers a mixed blessing here: since the schema is owned by the developers, DBAs have nothing to “own”, and therefore the political conflict, in theory, doesn’t exist. In practice, when the DBA group hears that “mission-critical” data is being stored outside of their domain, they tend to go ballistic and raise all kinds of grief with upper management, where their complaints will likely be heard with sympathetic ears, since theirs is what’s familiar, and the db4o world is not. (Until db4o has at least a decade under its belt, this is likely to remain a problem for adopters, as with any other new technology. Even relational databases went through this, back in the day.)

In practice, this means that one of three things will occur:

1. the db4o-adopting developer group will need to fight the political battle to retain ownership of the entire db4o footprint (which may in fact be counterproductive—developers don’t need, or even want, to be the production support team for the db4o database),
2. the db4o-adopting developer group will surrender the runtime footprint for db4o to the IT staff; at times, however, this leads to complaints from the IT staff that the same kind of rich toolset they’re used to in the RDBMS world doesn’t exist around db4o (some of which is true, some of which is simply that object databases are just “different” than what they’re used to), or
3. the db4o-adopting developer group will use db4o as a “staging point” for storing data, from which the data is then imported into a traditional RDBMS

Of course, the fourth option, converting/educating the IT staff to OODBMS technology is always possible, but this is by far and away the least likely option to succeed, particularly given the history of the OODBMS industry's attempts to do exactly that.

Again, this being a political problem, there are always non-technical solutions to this (and non-technical impediments), but if the problem can be avoided entirely, most times it's best to take that approach.

One issue that cannot be avoided, however, is that of reporting: frequently, the IT staff create reports (using tools like Crystal Reports) against the RDBMS, and in a db4o-based scenario this isn't practical or feasible, since no popular tools exist for reporting from an OODBMS. In practice, then, this implies that developers would be "on the hook" to write code to generate reports themselves, or that the data from the db4o data store would need to be exported to an RDBMS for reporting, a scenario not unlike the relationship between OLAP and data warehouse databases in the relational world. Db4o's Replication tools can assist with the latter approach, but any "real-time" reporting against the db4o database still needs to be done using a developer-authored interface.

Entity identity issues

Part of the impedance mismatch centers around the fundamental discordance about identity: in a relational model, identity is given by the state of the entity, typically a subset of that state to which we give the more recognizable name "primary key". In an object system, however, identity is given by an artificial identifier, known as an "OID", or object identifier. Typically, in Java or other object language, the OID is given simply by the object's location in memory (the "this" pointer, as it was known in C++). This notion of identity-via-OID is also the reason for the distinction between Java's "==" operator and "equals" method, as it turns out—the latter tests for equivalence, the former for identity, which is whether or not two references point in fact to the same object.

This presents a problem to an O/R-M system, since now we have two different, and conflicting, systems of identity, which in turn confuses caching systems (particularly those inside of an O/R-M implementation). An OODBMS persistence implementation helps avoid some of this confusion, since its system of identity is based around OIDs just as the in-memory object representation does. However, one problem an OODBMS frequently faces is that of keeping straight the connection between in-memory object and OID-attached object, and db4o is no exception here. It's not impossible

—the db4o API provides a static method to check to see if an object's been attached to an OID. This distinction of "attached" vs "unattached" is subtle, however, and some developers may find it more subtle than managing the distinct forms of identity in an O/R-M.

The partial-object problem and load-time paradox

In a SQL query, developers write precisely which data elements they want retrieved as part of a query, meaning developers can return as much or as little of a table (or collection of tables) as desired. In an object-oriented language, however, this isn't possible—developers cannot work with "pieces" of an object, or individual fields extracted into a new object type. This creates a "partial object" problem, in that the atomic unit of an object database is the object itself, which frequently is more data than necessary.

To the original O/R-M implementors, this problem was solved by creating "on-demand" load triggers, such that if only a few fields of an object were explicitly requested, only those fields would be populated in the object, the object as a whole returned, and if the developer asked for a field not populated in the returned object, a "demand-trigger" would be tripped and the object would go fetch the field in question from the datastore. This typically implied a remote network call (either a JDBC query or some other RPC-like call), with all the negative performance implications that brings. The dangers of this approach were fully exposed by the backlash against the EJB Entity Bean implementation, where each of these retrieval operations were themselves protected (individually) by a full two-phase commit transaction, leading to a general revolt against EJB as a whole that persists to this day, and the generalized advice that clients of an EJB system should never access an entity bean directly, but only through session beans and DTOs, which clearly destroys any attempt to build a rich domain object model and shield developers from the details of persistence.

The root of this problem is that of network traversal—if the persistence implementation lives inside the same process as the client-accessing code, then the demand-trigger approach to the partial-object problem yields a comfortable solution, since the costs of making a disk I/O access (at worst) are far more manageable than those of making a network I/O trip. Both db4o and other relational database systems (HSQL and Derby being two of the popular Java options, Microsoft's SQL Server Express being another) offer "in-memory" or "local" options, where the database resides in that same process and thereby avoids the network. For most production systems, however, running the database in the same process as the application server or even on the same machine as the application server is an unacceptable solution due to firewall constraints or security concerns.

Here, the db4o system is just as vulnerable as any relational system or O/R-M tool: db4o offers no "partial object" solution, meaning that any object retrieved is retrieved in full, potentially retrieving data fields that are not needed at that point in the client code. More importantly, however, objects often hold references to other objects, and a naïve object database implementation will need to retrieve all of those "dependent" objects that are referenced by the fields of the retrieved object, leading potentially to a huge graph of objects retrieved in response to a simple first name/last name request on the Person class in the database. Db4o solves this problem via a concept they call "activation depth", whereby the db4o developer indicates how "deep" the graph should be retrieved when the object in question is retrieved from the database—an activation depth of 2 tells the db4o system to retrieve not only the object requested, but any objects it in turn references but no further. Objects beyond the activation depth are not populated at all, but their fields are (artificially) set to a null value. Developers can then, if necessary, activate those child objects by calling a db4o API, again activating more child objects from there if necessary or desired.

(Just as this paper was being finished, the db4o team committed a new feature whereby developers can register method callbacks that will be invoked as objects are added, removed, changed, and so on. Interested clients can then receive notifications about object updates, and thus keep local copies in sync with what's being stored in the database.)

A close corollary to the partial-object problem is the load-time paradox, in which the decision to retrieve data "eagerly" or "lazily" from the datastore is often not made on a class-by-class basis, but would rather be made in accordance with the context of the situation in the client code. Most O/R-Ms do not acknowledge this paradox, referring developers to their associated query language (or trusting in their demand-load triggers inside partial objects) to be more specific about what needs to be retrieved in each situation. As with the partial object problem, db4o offers no particular solution, but developers generally will set/reset activation depth in front of each query in order to control the amount of data being retrieved in each scenario, a single API method call in front of the query call itself.

The data-retrieval mechanism concern

Probably the most distinctive difference between an OODBMS-based approach (db4o in particular), and an O/R-M-based approach is in the data-retrieval mechanism. In a traditional relational scenario, developers use SQL to specify exactly the relational values to return, using the relational algebra and predicate calculus to specify exactly what should be retrieved, nothing more and nothing less. This provides a degree of economy in what is pulled across the network rarely matched in any other distributed system.

Object databases have historically proven less efficient at describing the data to be retrieved, despite a variety of approaches: Query-By-Example (QBE), Query Object APIs, and object query languages such as OQL, HQL, EJBQL, and so on. The problems associated with QBE, Query Object APIs, and query languages are not difficult to spot: a QBE approach severely restricts the kinds of queries that can be made against the database, and is usually used only for simple, single-object requests ("identity" requests), or for requests to retrieve a collection of objects that obey simple AND-ed predicates, such as "find all Persons where lastName is 'Powers' and occupation is 'Software Developer'". Query Object APIs provide a greater range of query capability, but are tedious and awkward to manipulate, requiring construction of numerous intermediate objects to hold the query together, such as "predicate" objects designed to hold the queried field and the predicate query value together. Developers generally tire of the obtuse Query Object API quickly, which then leads to the "query language" approach, a new language designed to allow for quick and easy generation of queries such as ODMG's Object Query Language (OQL), or Hibernate's Query Language (HQL), all of which look alarmingly similar to SQL. This in turn often leads developers full circle, asking themselves why they didn't just bother writing SQL in the first place, particularly when these query languages turn out to be missing certain features of the SQL language, such as ORDER BY or GROUP BY or DISTINCT or outer joins and so on.

The db4o database uses an entirely new approach, called "native queries", in which the predicates used to query the database for objects to return from the query are described using the native language itself (Java or C#). This means that developers need not learn a new query language at all, since now all database interactivity is done using the language of their choice. Yet the query can be as expressive as they need it to be, since both Java and C# have the same rich set of comparison operators that SQL (or another query language) does:

```
List <Pilot> pilots = db.query(new Predicate<Pilot>() {
  public boolean match(Pilot pilot) {
    return pilot.getPoints() == 100;
  }
});
```

This approach has the added benefit, not present when using O/R-Ms (or SQL, for that matter), of being entirely type-safe: the compiler will verify that the fields used in the query are present on the object in question and that all comparisons are entirely type-safe, thus eliminating a source of potential errors and an entire class of unit tests necessary.

One concern developers voice when first seeing this feature is that this implies that every object in the database must be instantiated and passed into the comparison method, clearly a horrible performance hit if thousands—or millions—of objects reside in the database. Therein lies the beauty of the db4o native query feature, as the db4o system will do runtime Reflection and bytecode analysis to understand what the query is doing (what fields on the object are being compared, what values the fields are being compared against, what operations are being used to do the comparison, and so on), and converts that to its internal Query Object API (called SODA). While not all queries will be possible to translate this way, in general a significant percentage of developers' queries can be, leaving developers with an efficient-to-write and efficient-to-execute query approach. Developers can also be notified (via a runtime API call) when native queries cannot be optimized this way, thus providing them with an opportunity to "hand-tune" certain queries using the more fine-grained/low-level SODA query API.

Summary

At no point should this essay be seen as a general call to embrace the OODBMS over the relational database; any attempt to position a technology, *any* technology, as the one-size-fits-all solution to all problems faces dangers. While obviously applicable to the object database, this truism also works for the relational or even XML/hierarchical database, as well—it's a fair statement to say that "X technology" works well in a particular set of scenarios, less well in others, where "X" is either the relational or object database.

The traditional RDBMS sits at the center of a collective of hardware and software, acting as the central repository for storing mission-critical data, and as such, carries with it a heavy administrative load. Additionally, the traditional RDBMS server presents a sizable security concern, SQL injection attacks being just one of the possible ways an attacker can obtain data (from improperly constructed SQL queries—or HQL queries, or EJBQL queries…). An OODBMS avoids this injection attack entirely so long as it avoids the use of a query language style query, and native queries in db4o neatly avoid that problem while still providing the power of a query language.

While there's no reason that an object database cannot sit at the center of the IT universe next to an RDBMS, in many respects this misses part of the power and beauty of the object database, and db4o itself. Because the object database is much more closely coupled to the programming code, it makes for a much easier "edge" storage solution, pushing data out to the edges of the network, such as client desktop machines, small-device clients, and customer-facing web servers, all of which benefit strongly from the OODBMS's smaller administrative footprint requirements. In fact, db4o markets itself as a "zero-admin" tool, making it attractive for developers looking to create rich-client applications delivered over the Internet (via JavaWebStart or ClickOnce).

In the end, avoiding the quagmire so clearly identified by the name "Vietnam" is a goal that is equal parts technology and politics. For many, the problems of the O/R-M can be managed by strong developer discipline, deep knowledge of how the O/R-M works and the queries it generates against the relational database, and careful adherence to the rules of the O/R-M's caching infrastructure. For those who are seeking to truly "avoid SQL" and having to learn a new language and way of storing data, however, the OODBMS represents a powerful option that eliminates the mismatch entirely. And for those applications, services, or systems where persistence implementation is really just an implementation concern, why not take the path of least resistance, even if it is the road less traveled?

This paper has been made possible through the collaboration of Neward&Associates (www.tedneward.com), db4objects Inc. (www.db4o.com) and ODBMS.ORG (http://www.odbms.org/), the most up-to-date collection of free materials on object database technology on the Internet.