Bridging the Object-Relational

Modern applications are built using two very different technologies: object-oriented programming for business logic; and relational databases for data storage. Object-oriented programming is a key technology for implementing complex systems, providing benefits of reusability, robustness, and maintainability. Relational databases are repositories for persistent data. ORM (object-relational mapping) is a bridge between the two that allows applications to access relational data in an object-oriented way.

ORM is a specialization of the general concept of object persistence. In his book *Object-oriented Analysis and Design with Applications*, IBM Fellow Grady Booch claimed that persistence, defined as “data that outlives the program,” is a minor element of an object model.¹ In modern applications, however, the amount of effort devoted to persistence can dominate the cost of a project, and using ORM tools can significantly reduce this cost.

Other technologies can be used to access relational data from object-oriented programs, but these technologies generally do not exploit the object behavior of the programming language. To exploit object behavior fully,

database-access technology should support separation of concerns, information hiding, inheritance, change detection, uniquing, and database independence.

Separation of concerns is the process of breaking up a program into logical parts that have little or no overlap. In database programming, several parts should be separated: business methods that find domain objects and operate on them; methods in those domain objects that might cause internal state changes and call methods in associated objects; generation of database commands to implement inserts, queries, updates, and deletes of database rows and columns; and methods for database transaction demarcation.

Information hiding is an implementation strategy for separation of concerns that reduces complexity and cost by defining behavior in terms of interfaces that are implemented by specific classes. The classes implement the behavior separate from the behavior of the callers. Changing one side does not necessarily require changes in the other.

Inheritance allows for reuse of code where common behavior is defined once for a number of related classes and unique behavior is implemented only in classes where the behavior is different. Thus, the behavior of a class might be the same or different from the behavior of

Divide

ORM technologies can simplify data access,
but be aware of the challenges that come with
introducing this new layer of abstraction.

Bridging the Object-Relational Divide

a subclass or superclass, and independent of the behavior of the caller.

Change detection keeps track of changes made to domain objects that are used in a database transaction so that at the end of the transaction, changes are applied to the database.

Uniquing is a property of database interaction in which a single domain instance corresponds to a database row, regardless of how the user acquired the object: via querying the database, navigating a reference from one instance to another, or finding a distinct domain instance by providing its primary identity. Without uniquing, changes made to one domain instance will not be seen by other domain instances representing the same database row; this might result in database corruption.

Database independence allows use of a common API and domain model to operate with various databases without changing the application view of the database.

The earliest—and still very popular—technologies for accessing relational databases use APIs to transmit SQL statements to the server and return the results of executing the statements back to the application. It is left up to the application to use the results directly or to create data structures that represent the query results, and to copy the query results to these data structures.

Data structures that directly model query results cannot model relationships that exist in the database; therefore, associations among instances of the data structures are poorly represented. Examples of this style of access include ODBC (Open Database Connectivity) and JDBC (Java Database Connectivity) interfaces. These interfaces allow for limited separation of concerns, but most often business logic is mixed with database programming. They do not support information hiding, inheritance, change detection, uniquing, or database independence.

More capable technologies provide methods in the interface to copy the query results to user-specified data structures. The user provides all of the SQL statements annotated so they correspond to the data structures. The SQL statements created by the user include all queries, plus insert, update, and delete. These technologies handle much of the error-prone and time-consuming jobs of analyzing the results of queries but still leave many of the difficult tasks to the user. One example of this style

of access is iBATIS, which supports information hiding and a limited separation of concerns but not inheritance, change detection, uniquing, or database independence.

With ORM, data stored in relational databases is represented to the application as objects in the native object programming language. Programmers map domain object model classes to relational tables and use an API implemented by a persistence provider to access the database. Queries against the database are expressed in terms of the domain object model. The provider generates SQL statements directly from the domain model. Martin Fowler has called this approach DataMapper.²

ORM techniques and products are available for many object-oriented languages, including Java, C++, C#, Python, Smalltalk, Ruby, and Groovy. The techniques and programming paradigms for ORM apply to many of the products that support these languages. (In this article, examples are given in Java.)

ARCHITECTURE OF ORM

The application's view of ORM consists of two major parts: the persistence API and the domain classes. In Java, the API is typically one of the Java Community Process standards—Java Persistence API, Enterprise JavaBeans, or Java Data Objects—or nonstandard yet popular APIs such as TopLink or Hibernate.

An advantage of using a standard persistence API is that it allows projects to make a late deployment decision on both the database and the persistence provider. This can be a significant factor in many projects where the need for features that differentiate persistence providers is not obvious at the beginning of the project.

The persistence API allows the application programmer to perform all of the standard CRUD (create, read, update, delete) operations of the database. Since the application programmer does not access the rows and columns of the database directly, a shorthand notation describes the behavior. For example, “make an instance of a mapped domain persistent” is shorthand for “create a row or rows in the database that correspond exactly to the instance of the mapped domain class.” Likewise, “delete an instance of a domain class” means “delete the database row or rows that correspond exactly to the instance of the domain class.”

The API includes methods to make instances of a mapped domain class persistent, to retrieve an instance of a domain class by its primary identity, to find all instances of domain class and subclasses by a query expressed in terms of the domain class values, and to delete an instance of a domain class from the database. Updates are done in the context of a transaction, by retrieving an instance of a domain class and using domain methods to modify values of the instance.

The domain classes that represent the application's view of the data stored in the relational database are typically either written by hand or generated from a database schema using a tool. Typically declarative, the mapping relates the domain object model to the relational schema model and is defined before the application runs. Since the domain classes need not contain specific persistence behavior, the classes are often referred to as POJOs (plain old Java objects).

The absence of persistence code allows operation of the persistent classes independent of the persistence aspect. Much of the behavior of the domain classes can thus be tested without access to the database or to the persistence environment. This style of programming encourages separation of concerns and encapsulation.

Many tables map directly to domain classes, such as *Employee*, *Department*, *Customer*, *Order*, *LineItem*, *Contract*, *Claim*, *Product*, and so forth. Rows of these tables are mapped to instances of the domain classes. Columns of the tables are mapped to fields of the domain classes.

IMPEDANCE MISMATCH?

The so-called impedance mismatch between object and database schema has generated much discussion, for good reason. There are “deceptive similarities” between the two technologies, according to Scott Ambler.³ Lack of understanding the difference between the technologies can lead to bad design choices and project failure.

The data model in object languages such as Java is not exactly the same as in relational databases, so special care must be taken to prevent problems from arising. For example, the maximum lengths of character columns must be specified in the relational schema, but Java Strings are essentially unbounded.

Floating-point numbers may cause problems, too. Java implements IEEE floating-point numbers; relational databases often have a different representation. The effect is that not all values for floating-point numbers in Java can be stored in the database, and vice versa.

Even fixed-precision decimal numbers may pose problems when mapped to relational schema. Java supports

fixed-precision decimal numbers that are self-describing—that is, each value has a specific precision (number of digits) and scale (number of digits to the right of the decimal point). In the database, however, all numbers in the same column have the same precision and scale.

Mapping between the application domain classes and database schema is, by design, not isomorphic. On the domain model side, certain aspects of the object model are not mapped to the database, including behavior of the classes and class variables. On the relational database side, not all of the tables and columns in every schema in the database are represented in the domain model, nor are stored procedures. Additionally, domain models might have several valid mappings.

Representing a database schema with a domain object model is usually straightforward, especially if the schema is well normalized. With an entity-relationship model of the schema, each entity is mapped to a domain class. Each simple relationship, implemented as a foreign key in the schema, is mapped to a field of a reference type on one side and a field of a multivalued type on the other.

Many complex constructs in the entity-relationship model can be nicely mapped to object-model constructs. For example, a join table in which the columns are foreign keys to two different tables can be mapped to two classes in which each class contains a field, the type of which is a set of objects of the other class type.

On the other hand, storing an arbitrary domain object model in a relational database can be challenging. Models that represent relationships using abstract classes, deep

Bridging the Object-Relational Divide

inheritance, or interfaces are generally more difficult to store and retrieve with reasonable performance.

MAPPING

Inheritance. In the domain object model, inheritance is a relationship between two classes in which one class is a specialization of another. Figure 1 shows a human relations object model in which `FullTimeEmployee` and `PartTimeEmployee` are specializations of `Employee`, which in turn is a specialization of `Person`.

There are several ways to map this domain object model to relational schema. A single-table inheritance strategy maps all classes in the inheritance hierarchy to a single table that contains a column for each of the fields in any of the classes. In this mapping, slots of the table

contain null or default values when the row represents an instance of a class that did not contain a field for that column.

For the single-table strategy to work, the database must contain information that allows the mapping to determine to which class any particular row is mapped. The mapping typically contains the name of a discriminator column and the values of that column that map to each of the possible classes.

For example, the `PERSON` table (shown in figure 2) might contain a `DISCRIMINATOR` column whose values are S, E, F, and P, depending on whether the row maps to an instance of `Person`, `Employee`, `FullTimeEmployee`, or `PartTimeEmployee`, respectively. When retrieving rows from the `PERSON` table, the provider would always retrieve the value of the `DISCRIMINATOR` column in order to instantiate the correct class.

A table-per-class inheritance strategy (figure 3) maps each class to its own table and maps each field to a column in the mapped table. All of the data for a single instance would have the same primary-key column value in each of the tables, and the schema would declare foreign-key relationships among the tables. When querying these tables, the provider constructs join queries to determine to which class a row maps. For example, to find all employees hired after a specific date, the query would join the `PERSON`, `EMPLOYEE`, `PART_TIME_EMPLOYEE`, and `FULL_TIME_EMPLOYEE` tables to determine whether the rows were mapped to `Person`, `PartTimeEmployee`, or `FullTimeEmployee`.

A table-per-concrete-class inheritance strategy (figure 4) maps each nonabstract class to its own table and maps each field to a column in the mapped table. This strategy reduces the number of tables needed to represent the domain classes.

A mixed inheritance strategy might use a combination of strategies. For example, it might map the fields from *Person* and *Employee* to one table that includes a discriminator column; map the fields from *FullTimeEmployee* to a second table; and map the fields from *PartTimeEmployee* to a third table.

The benefits of single-table inheritance mapping are simplicity and performance. To query for instances of any of the classes in the hierarchy, only a single table needs to be used. The disadvantage of single-table inheritance is that it might require many columns that are used by only a few class mappings. This inefficiency might be an issue with very wide inheritance hierarchies.

The benefits of other inheritance mapping strategies are that the schema can be completely normalized. The primary disadvantage of inheritance mapping strategies that do not include a discriminator column is performance. Most queries for instances of subclasses require an outer join of the tables in the hierarchy, and these queries are notoriously difficult to optimize.

Relationships. In relational schemas, relationships are modeled via foreign keys, defined as a constraint on the values that columns are allowed to contain, based on values contained in rows of other tables. This constraint is enforced by the database. As shown in figure 5, an *EMPLOYEE* table might contain a *DEPT* column that contains the depart-

ment number of the department to which the employee belongs. The *DEPT* column in the *EMPLOYEE* table has a constraint declared on it so that the value of the *DEPT* column of any row of the *EMPLOYEE* table must be a value that is also contained in the primary-key column of the *DEPARTMENT* table.

A foreign-key constraint can be mapped to multiple relationship fields in the domain object model. For example, the *DEPT* column maps to two relationship fields in the object model: the *Employee* class contains a reference to *Department*; and the *Department* class contains a set of references to *Employee*.

If the foreign-key column is unique, then only one row in the table relates to the same row in the other table. In figure 6, an *INSURANCE* table has an *EMPLOYEE* column

Bridging the Divide

that relates a row in the INSURANCE table with a row in the EMPLOYEE table, and there is only one row in the INSURANCE table for any row in the EMPLOYEE table. This foreign-key column maps to the object model via a reference in each domain class that refers to the other.

In a normalized relational schema, relationships in which many rows of one table are related to many rows of another table are represented by join tables, in which each row of the join table contains a foreign key to each of the related tables. If multiple rows of an EMPLOYEE table are related to multiple rows of a PROJECT table, a third table is needed to maintain a normalized schema

(figure 7). The EMPLOYEE_PROJECT table contains two columns: one column with a foreign key to the EMPLOYEE table and one column with a foreign key to the PROJECT table. Thus, join tables with two foreign keys map to relationships of domain classes that have multivalued fields on both sides.

Embedded. For improved encapsulation, it may be useful to model a collection of fields in a domain class as a separate class. This mapping concept is called *embedded*, because the columns for the embedded class are embedded in the same table as the mapped domain class.

As shown in figure 8A, an Employee class might contain

fields representing a home address. These are mapped to columns in the EMPLOYEE table. Figure 8B shows a different mapping for the same schema using an embedded Address class that is mapped to the same columns in the same table.

Dependent. A common pattern in domain object models is composition, in which the life cycle of one collection of instances is dependent on another. In the database, the tables that store the dependent instances are indistinguishable from other tables. The behavior of the domain model, however, requires that if the relationship from the owning instance is removed, then the rows corresponding to the dependent instances should also be removed.

In figure 9, the skills relationship is marked as dependent, and if a Skill is removed from the skills collection of the owning

Employee, the persistence provider will delete the corresponding row from the SKILL table in the database.

The preceding descriptions cover many of the common cases of mapping between domain classes and normalized database schema. More complex domain classes can also map to more elaborate database schema.

PERSISTENCE API

The ORM API provides interfaces that represent the runtime environment. The primary interface of the API

encapsulates the connection to the database, the query generator, and a cache of domain instances that have been referenced by the application. For this article, this interface is called *Session*. (In different APIs, this interface is called *EntityManager* or *PersistenceManager*, but regardless of the name, it serves the same purpose.)

In a multiuser environment, multiple *Session* instances are used to isolate users from each other. Each *Session* has its own connection to the database and its own cache of domain instances. The session cache implements uniqueness. If the API supports multiple threads operating on the same *Session*, all such threads share changes to the domain objects managed by the *Session*.

The application obtains a *Session* from a *SessionFactory*, which encapsulates the data source (connection factory) and ORM model. The *SessionFactory* also typically includes connection pooling and a second-level or global domain object cache, and it handles the interaction with the container. The *SessionFactory* is the bootstrap instance and is typically looked up from a system service such as JNDI (Java Naming and Directory Interface) or configured via XML or properties.

Database transactions are represented by a *Transaction* class, an instance of which the application obtains from the *Session* instance. There is only one active *Transaction* instance for a *Session* (a *Session* can serially begin and complete multiple transactions), and the different interfaces represent a separation of concerns—that is, the application components that begin and complete transactions are separate from application components that access domain instances.

Database queries are represented by a *Query* class, instances of which the application obtains from the *Session*. A *Query* instance encapsulates a specific query and

implements an *execute* method that allows the application to get different sets of data from the database based on different parameters.

During a unit of work, the application might have made many changes to domain instances, and at the end of the unit of work, the persistence provider applies changes to the database based on changes made to the mapped domain instances.

Bridging the Object-Relational Divide

Because of foreign-key constraints in the database schema, the order of operations corresponding to the changes in the domain model is important. For example, a new row must be inserted into the database before another new row is inserted or an existing row is updated to refer to the new row. Similarly, a column in an existing row referring to a row to be deleted must be set to null before the row is deleted.

Automatic ordering of operations in the database based on database constraints is a significant usability feature of ORM technologies compared with other technologies in which the programmer must perform the operations in the correct order.

Queries. One of the primary benefits of ORM is that queries are expressed in terms of the domain object model and not the relational schema. This becomes important if the domain model or queries are complex.

One of the challenges in designing the query language for ORM is to make it rich enough to allow most application queries to be implemented in the query language, but restricted to allow all object domain queries to be mapped directly to SQL.

For example, to find all instances of `Employee` where the weekly salary is greater than some parameter, the SQL query is:

```
SELECT * FROM EMPLOYEE WHERE WEEKLY_SALARY > ?
```

The corresponding query using the domain object model would be:

```
SELECT FROM FullTimeEmployee
WHERE weeklySalary > :salary
```

The provider generates the SQL from the domain model and the mapping. The “:” is a parameter marker that allows the application to bind parameters to the query by name.

In SQL, all the tables involved in the query are declared in the `FROM` clause and the joining conditions are expressly included in the `WHERE` clause. To find instances of `Employee` whose weekly salary is greater than some parameter and who work in a `Department` with some name, the SQL is a bit more complex:

```
SELECT E.* FROM EMPLOYEE E, DEPARTMENT D
WHERE E.WEEKLY_SALARY > ? AND D.NAME = ?
AND E.DEPARTMENT = D.ID
```

The corresponding query using the domain object model would be:

```
SELECT FROM FullTimeEmployee
WHERE weeklySalary > :salary && dept.name = :dept
```

NAVIGATING RELATIONSHIPS IN QUERIES

One of the biggest differences among ORM implementations is how navigation in queries is expressed. In JDOQL (Java Data Objects Query Language), each query filter is a Java Boolean expression. The Boolean collection methods such as `contains()`, `containsKey()`, and `containsValue()` are used to navigate multivalued relationship fields.

Other domain query languages use constructs that are more similar to SQL to navigate relationships. For example, in Java Persistence API, the query language uses special keywords such as `JOIN`, `IN`, and `OUTER` to navigate relationships.

To find instances of `Employee` where the weekly salary is greater than some parameter, who work in a `Department` with a specific name, and work on a `Project` with a specific name, the SQL is even more complex:

```
SELECT E.* FROM EMPLOYEE E, DEPARTMENT D, PROJECT P,
EMPLOYEE_PROJECT EP
WHERE E.WEEKLY_SALARY > ? AND D.NAME = ?
AND E.DEPARTMENT = D.ID AND E.ID = EP.EMPID
AND P.ID = EP.PROJID AND P.NAME = ?
```

The corresponding domain model query in JDOQL is:

```
SELECT FROM Employee WHERE weeklySalary > :salary
&& dept.name == :deptname && projects.contains(p)
&& p.name == :prjname
```

This example query uses the `Set.contains(Object)` method of the `Set<Project>` `projects` field to map to the join table in the relational model. The two tokens `:deptname` and `:prjname` are variables that are provided to the query at runtime.

In the Java Persistence API, the query would use an explicit join clause to navigate the employee-project relationship:

```
SELECT e FROM Employee e, JOIN e.projs as p
WHERE e.weeklySalary > :salary
AND e.dept.name = :deptname AND p.name = :prjname
```

PERFORMANCE ISSUES

The domain object model, the mapping, and the database schema can affect the performance that an application can obtain by using an ORM tool. Furthermore, the quality of the provider implementation is as important as the quality of the application program.

One factor in performance is how the persistence provider performs change detection. When the application uses the API to commit a transaction, the provider sends an update request to the database for each instance that was changed.

Some providers iterate the cache of instances and compare the current value to the value as it was retrieved from the database. Other providers dynamically track changes as the application modifies the instances. Applications that retrieve a large number of instances and change only a few can see a significant performance difference.

Another factor in performance is how to retrieve exactly the data needed for an application, no more and no less. A significant difference between a low-level API such as JDBC and a high-level API used by ORM is the control over the actual data that is retrieved from the database when the application executes. When using a domain model, the same classes can be reused in several applications, and different use-cases might require different columns to be retrieved.

For example, if an application use-case examines all Employees in a specific Department, a single query can retrieve a subset of columns from both the EMPLOYEE and DEPARTMENT tables. The challenge is for the application to inform the persistence provider which columns to retrieve and which part of the object model to instantiate for this use-case.

One common technique is to statically declare certain fields in the domain model as *eager fetch*, so that whenever an instance is retrieved, the related instances are also fetched. A highly optimized persistence provider can determine the exact SQL needed to accomplish the retrieval with the best performance.

A somewhat more dynamic alternative approach has the domain query itself specify the retrieval of a complete

collection of instances. In this case, the application would write a separate query for each use-case, specifying the navigational path to follow for the query.

A more general approach is to define access patterns that specify a collection of fields and navigational paths, and for the application to choose the access pattern needed for a particular use-case. For example, the application can define fields in the Employee class and fields in the Department class to be part of a fetch group named `empdept`, and at runtime tell the persistence provider to use fetch group `empdept`. Whenever a query is done for either Department or Employee, the `empdept` fetch group controls the exact columns and join conditions executed by the provider.

MULTITHREADING ISSUES

Aside from roll-your-own frameworks, programmers are likely to encounter multithreading in two areas: GUI applications and Ajax (asynchronous JavaScript and XML) applications.

To maintain liveliness, GUI applications handle trivial requests in the main thread and delegate longer-running requests to one or more background threads. If using an MVC (model-view-controller) approach, then reading the value of a model object field in order to interact with the view object might appear to be a trivial request. If the field is persistent and not currently loaded, however, it becomes a database request. Other requests, such as querying the database by means of a dialog box, are clearly longer running and should be executed by a background thread. If a single Session is used for both tasks, then threading becomes an issue.

Similarly, when running Ajax applications with most Java Web frameworks, the same server session state is used for all of the requests from the same Web page. If a longer-running request is still executing when another request from the same page arrives, multiple threads are used to service the requests. If there is a single-persistence Session instance for the Web session, multithreading conflicts can occur.

Handling multithreading in an application is not trivial. Some of the considerations include whether to allow all threads to see the same view of the domain model and whether to commit all changes immediately or defer commit until some user action occurs (e.g., pushing a Save button). These decisions affect the application's transaction model.

To allow for the flexibility needed by applications, some persistence providers permit the programmer to specify the level of multithreading for the Session

Bridging the Object-Relational Divide

instance. Others require the application explicitly to acquire a Session per thread.

OPTIMISTIC AND PESSIMISTIC CONCURRENCY

Using a low-level API such as JDBC or ODBC gives the application programmer direct control over transactions. With these APIs, transactions are started and either committed or rolled back via APIs. There is no concept of optimistic transactions.

With ORM, transactions are coarser grained and controlled by APIs on the Transaction instance associated with a Session. Optimistic transactions are part of the model and include version fields in the domain model that correspond to version columns in the database. The application specifies either optimistic or pessimistic transactions, and the persistence provider automatically manages version columns for optimistic transactions.

Some persistence standards support optimistic concurrency only, with minimally defined APIs for locking; others support both optimistic and pessimistic concurrency models; and some products support more fine-grained policies for eagerly locking instances of classes upon retrieval. These finer-grained concurrency models are needed for certain high-concurrency applications in which any access to a persistent instance requires an aggressive lock to achieve high throughput.

DEADLOCK PREVENTION

Deadlocks occur when two or more transactions hold locks on resources that are needed by other transactions and none of the transactions can complete without canceling locks held by other transactions.

Low-level APIs such as JDBC and ODBC give the programmer complete control over the lock requests, but ORM persistence providers automatically generate and execute database requests based on coarser-grained operations such as commit or rollback. The order of operations of the database requests can accommodate foreign-key constraints and batching of SQL commands.

If an application is subject to deadlocks, persistence providers can often avoid the deadlocks by giving the application control over the order of execution of database operations that are not subject to foreign-key constraints. This requires that the persistence provider is

able to determine the order in which database operations should be applied, based on the order of changes made to the domain object model.

SUMMARY

ORM is a technology that provides access to relational data using the object-oriented paradigm. An ORM implementation consists of a language for mapping between the object and relational domains and an API for storing, querying, and modifying application objects. There are several standards with commercial and open source implementations, as well as some nonstandard products.

The ease of defining a mapping depends on the requirements of the application, its domain model, the existence of a legacy relational or object model, and the complexity of these models. ORM can provide significant improvements in programmer productivity, application quality, and maintainability. One of the most important ways this is achieved is through separation of concerns: separating the behavior of the domain object model from the access of the data from the database. Using a standard API allows the choice of implementation to be a late decision, providing more time for evaluation of alternative mapping and database technologies. Q

REFERENCES

1. Booch, G. 1991. *Object-oriented Analysis and Design with Applications*. Benjamin-Cummings Publishing.
2. Fowler, M. 2002. *Patterns of Enterprise Application Architecture*. Addison-Wesley.
3. Ambler, S. 2003. *Agile Database Techniques*. Wiley.

LOVE IT, HATE IT? LET US KNOW

feedback@acmqueue.com or www.acmqueue.com/forums

CRAIG RUSSELL is a senior staff engineer at Sun Microsystems. He is a member of the Apache Software Foundation, the chair of the Apache OpenJPA Project Management Committee, and a member of the Apache Incubator project responsible for bringing projects into Apache. He is specification lead for Java Data Objects (JSR 12 and 243) and leads the implementation team for its API and Technology Compatibility Kit.

© 2008 ACM 1542-7730/08/0500 \$5.00