Reladomo's Internal Architecture

July 1st 2006

Table of Contents

1. Overview 1
2. Object-Relational Mapping 1
    2.1. Object Lifecycle Management 1
    2.2. Code Generation 2
    2.3. Objects in Multiple Databases or Desks 3
    2.4. Tier independence 3
    2.5. Caching 3
    2.6. Transaction Management 4
    2.7. Object Chaining 5
    2.8. Flexible Relationships 5
3. Object Query 5
4. The List Object 5
5. Nullable Attributes 6
6. Appendix: Class Diagram 7

1. Overview

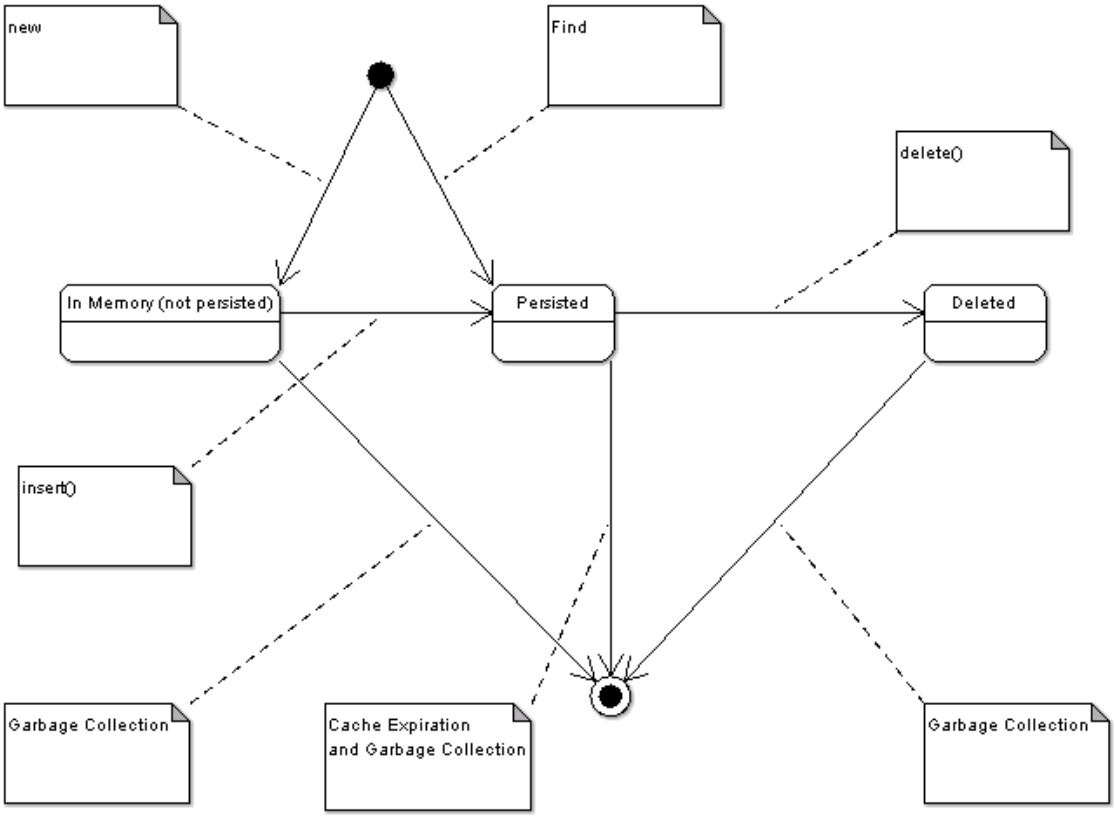
- Provide meta-data driven object-relational (OR) mapping
- Handle objects of the same type residing in multiple databases
- Support 2-tier and 3-tier modes of operation
- Optimize caching behavior using different caching strategies: full cache, soft cache, and query cache
- Object oriented query mechanism
- Use flexible relationships: object queries to define object relationships
- Manage transactions
- Support chaining of objects: bitemporal, business only, audit only

2. Object-Relational Mapping

2.1. Object Lifecycle Management

Reladomo will map database tables to Java objects using XML based meta-data. It will manage full object lifecycle including object creation, modification, and deletion as shown in Figure 1, “Object Life Cycle”.

Figure 1. Object Life Cycle



2.2. Code Generation

Users will be required to create one XML file for every Reladomo object. The file will have the meta-data for the object. The meta-data file will include the following:

- List of object attributes. For each attribute the meta-data will define:
  - Its type
  - Mapping to the database column (if any)
  - Relationship to another class (if any)
- List of transactional methods

A template driven code generator will generate all the necessary classes for Reladomo's use and generate empty classes that will be modified by users to implement business logic. The initially generated empty classes will subclass Reladomo specific classes. This approach helps users add their own logic to automatically generated classes without worrying about losing their changes when they run the code generator again. Reladomo will gain from the pre-generated classes by not incurring the cost of java introspection. The code generation will be available as an Ant task for convenient integration with our build process. The generated classes are shown in Section 6, “Appendix: Class Diagram” .

2.3. Objects in Multiple Databases or Desks

PARA system works on accounts residing on multiple desks. This means Reladomo should be able to handle objects of the same type stored in tables residing in different databases

Objects that may reside in different locations have a concept of a "source key." A source key is an object attribute that does not correspond to a column in a table, but rather codifies where the object came from. The source key becomes part of the object's identity. In effect, the object's primary key is composed of the primary key mapped to the database columns and the source key. The source key will be used in 3 places:

- Obtaining a connection.
- Finding objects: clearly, to find anything a source has to be specified because typically querying all sources is way too expensive.
- Getting the table name: this allows qualifying the table name which enables 1.5 phase commit when two databases reside on the same server (in which case, the connection will be the same, but the table name takes the form of database name.owner.table name, e.g. pnl_psprod.dbo.FROZEN_POSITION). Total number of physical connections to the database server can also be minimized by using this facility.

2.4. Tier independence

All features of Reladomo will be available for both 2-tier and 3-tier clients. Reladomo will have custom serialization for 3-tier clients. In both 2-tier and 3-tier mode, queries can supply custom hints for deep fetching of related objects.

2.5. Caching

Reladomo's goal is to let the business requirements drive the tradeoff between memory and speed based on the objects' usage. To do this Reladomo supports three different caching strategies: full cache, soft cache, and retrieval unit based cache.

2.5.1. Full Cache

Full cache is similar to Object Manager's static cache. This caching mode makes Reladomo load all the objects from database and keep them permanently in cache. All queries for these objects are returned from cache and all updates are write-through.

2.5.2. Soft Cache

Soft caching is implemented using Java soft references. Soft references are garbage collected by Java virtual machine if there is a memory crunch. This caching strategy therefore may result in objects expiring in cache at times. Queries that can be resolved by the cache (e.g. queries that pertain to unique indices) go to the cache first and then to the database if required objects are missing from cache.

A soft caching strategy can be further enhanced via a query caching mechanism. By keeping a list of recent queries, it will be possible to avoid hitting the database. This idea can be further enhanced by searching the query cache for supersets of the current query. For example, if the query cache contains the result for "all accounts in trial 123", then resolving a new query "all account in trial 123 that are clearance accounts" could be done by an in memory filter of the existing results. Determining query subset relationship is not always trivial, but in many cases quite possible. Query containment has not been implemented yet.

2.6. Transaction Management

Reladomo will classify all objects into either read-only or read-write. Read-only objects will help Reladomo do useful memory optimizations. Transactions apply only to read-write objects. The Java Transaction API (JTA) binds each transaction to a single thread. Reladomo objects implement a simplified form of Multi-Version Concurrency Control (MVCC) : an object participating in a transaction keeps two sets of data, one is the last committed dataset and the other is the transactional dataset. An object can therefore only participate in a single transaction at any one time. However, the same object may be accessed by any number of non-transactional threads for read operations. Upon commit, the transactional dataset is promoted to committed. The transactional dataset is only visible within the context of the transactional thread. Objects automatically enroll in a transaction when they are accessed by a transactional thread. To ensure transactional safety at the database level, when an object enrolls in a transaction, a database lock is acquired (select with holdlock).

When a running thread accesses an object, Reladomo looks at the thread to find its transaction context, and looks at the object's transaction state and continues processing as illustrated in Table 1, “Transactional Participation” :

Table 1. Transactional Participation

Thread Transaction State	Object Transaction State	Read Operation	Write Operation
N	N	Last committed state returned	Transparently creates a transaction and writes the change to the database
Y	N	Enroll the object in transaction Perform the requested operation	Same as Read
Y	Y (different than thread)	Waits until the object transaction is complete, then same as above	Same as Read
Y	Y (same as thread)	Return last value set in the transaction's context	Set the value on the transactional dataset
N	Y	Last committed state returned	Waits until the object transaction is complete. Transparently creates a transaction and writes the change to the database (same as the N/N case)

Reladomo discourages the users from creating their own transaction demarcation and instead encourages using methods in the persistent object for the demarcation. The Reladomo xml schema provides `transactionalMethodSignature` element to create transactional methods. This approach help in better object orientation because the same object is now responsible for managing the data and its behavior. Today our persistent objects simply hold data and the business logic tends to be buried in UI code and other places. For each transaction method listed in the meta-data, users must create an implementation method in their persistent classes. For example, if there is a transaction method named `adjust` in the meta-data for Position then Position.java must have a method named `adjustImpl`. This method should only have the business logic for adjustment and shouldn't open or commit transaction; transaction will be controlled from the generated Reladomo super-class.

Method level transaction demarcation also allows for intelligent and automatic transaction retries. A database will typically issue a special error code when a deadlock causes a transaction to be rolled back. Reladomo will detect this special error code and retry the transaction by rerunning the entire business

logic encapsulated within that transaction. Additionally, retries can be performed when Reladomo detects a deadlock directly.

For transactions that cannot be easily described as method calls for a business object, a command pattern may be used to encapsulate the transaction.

2.7. Object Chaining

Reladomo will support chaining requirements as defined by PARA and TAMS applications today. The chaining requirements include bitemporal chaining, business only chaining and audit only chaining.

2.8. Flexible Relationships

Reladomo's relationships are defined as a set of rules one object needs to satisfy to become related to another. With this approach, common relationships like one-to-many, many-to-many, etc. become special cases. For example:

```
<Relationship name="orderItems" relatedObject="OrderItem" cardinality="one-
to-many">
    OrderItem.orderId = this.orderId
</Relationship>
```

defines a one-to-many relationship from Order to OrderItem. We can define yet another one-to-many relationship with some more qualifiers:

```
<Relationship name="unshippedOrderItems" relatedObject="OrderItem"
    cardinality="one-to-many">
    OrderItem.orderId = this.orderId and
        OrderItem.shipped = false
</Relationship>
```

3. Object Query

Reladomo query syntax follows an object oriented paradigm that is more intuitive than SQL like queries. Here is an example of a Reladomo query

```
OrderList result = new OrderList(
    OrderFinder.customerName().eq("Fred").and(
    OrderFinder.items().shipped().eq(false)));
```

This query finds all Orders that have a customer named Fred and have one or more unshipped items. For every Reladomo object there will be a generated Finder class that exposes the object's attribute on which we can invoke operations such as`eq`, `isNull`,`like`,`in`, etc. Operations can be joined together using the "and" boolean operation.

4. The List Object

For every Reladomo class, a list collection is also generated. The list is defined by a query (as shown above). Operations pertaining to a collection of objects (e.g. total market value calculation) can be implemented on the list object. The list object allows for several abstractions that are typically difficult when databases are mapped to objects:

- Mass delete: the list object will have a `delete()` method that deletes all the elements with a single database call. This is much more efficient compared to looping through the elements of the list and calling the database for each delete.

- Mass update: the list object will also have set methods for object attributes that will result in fewer database calls.

- Group functions: certain operations such as `count`, `sum` etc. could potentially be performed without having to resolve any objects at all.

- Union/intersect: lists may be unioned or intersected. This will result in a transparent union/intersection of the defining queries.

- Group relationship traversal: if we have a list of objects (e.g. a list of trials), and we need the related list of objects (e.g. all account belonging to those trials), the `get[related objects]()` method (e.g. `getAccounts()`) will return the correct list without having to loop through the trials. Again, the related list is constructed by manipulating the underlying query.

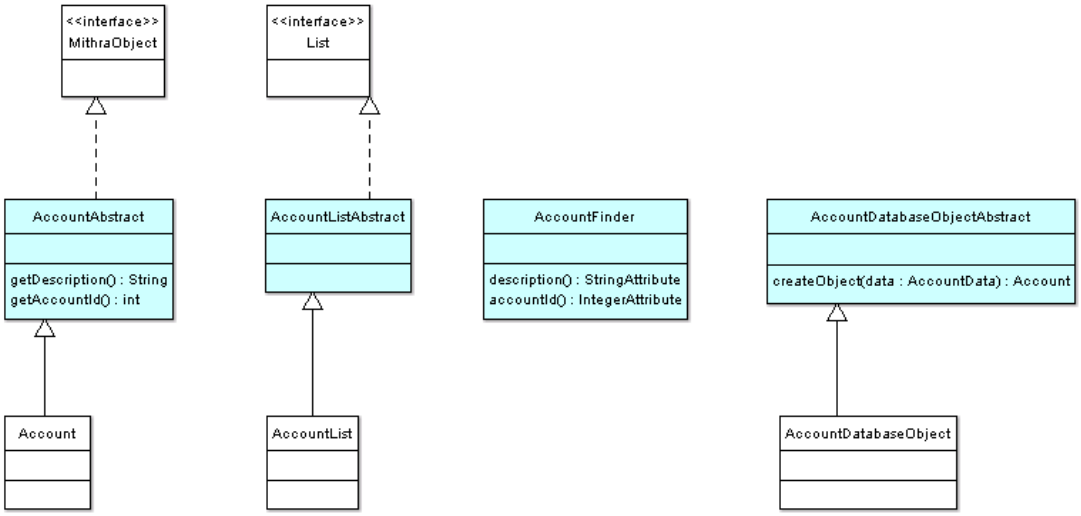
5. Nullable Attributes

Since all data in a database may be NULL (except for primary keys), Java primitive fields (int, float, etc) need special handling as they can't be null. In addition to the `get[attribute]` and `set[attribute]` methods, two other methods are provided: `is[attribute>Null]` and `set[attribute]Null`. For convenience, a default value can be supplied when an attribute is null.

Table 2.

Nullable	Default Value	Behavior
False	Must not be specified	• If the database does have a null value, throw an MithraDatabaseException • Attribute cannot be set to Null
True	Not specified	Throw MithraBusinessException if the `get[attritubte]` method is called and the attribute is primitive and it's null.
True	Specified	Return the specified default value when `get[attribute]` method is called and the attribute is null.

6. Appendix: Class Diagram



Sample class diagram. Classes in light blue are generated.