developer.*

Best Practices
For Object/Relational Mapping and Persistence APIs

By Mario Van Damme

Over the last decade there has been a lot of effort put into *object/relational mapping*, which refers to techniques for resolving the mismatches between the object-oriented world, with its encapsulation of data and behavior, and the relational world, with its tables and columns. There's not only a difference in terms of data types (and complexity of these data types) but also in terms of relationship types. The object world has a variety of relationships (aggregation, composition, association, inheritance) which cannot be mapped directly to the database world.

The general topic of object/relational (O/R) mapping can be divided into two areas of concern: the mapping itself and the persistence API. The persistence API not only acts as an indirection layer for the database, but also hides the mechanics of mapping the objects to the database tables. A good persistence API should do this while not constraining the object modeling in terms of data types and relationships.

In this article I will begin with a discussion of home-grown vs. off-the-shelf persistence solutions, including areas to consider when deciding between the two, and advice for choosing the best off-the-shelf solution to meet your needs. I will also share suggestions and advice from my own experiences with O/R mapping and persistence APIs. It is not my intention to explain all of the background details of these topics, but to focus on "best practices."

Home-Grown Mapping

I have been directly involved with both home-grown (custom built) and commercial-off-the-shelf (COTS) mappers, and have also observed several other home-grown mapper implementations, each approaching the subject in a specific way. In my experience, the primary drawback of "rolling your own" persistence mapping implementation is that limited resources do not allow for enough time to think everything through, to improve the framework over time, and to backtrack if you realize that design changes are needed.

Because an O/R mapper is a generic piece of software, it is typically hard to explicitly list which aspects are of the most importance to you (and if you build your own, you will *not* be able to focus on all of them). I do not mean to say that you could not envision a good design, but that it would take a lot of time and effort to fully implement a solution that meets all of your needs.

I observed the following limitations in just one home-grown O/R mapper:

- It provided no support for association relationships; only containment relationships were supported. This was a serious constraint when defining the object model.
- It offered no support for transactions.
- It only supported one RDBMS.
- The API was not type safe, which caused a lot of errors that could only be detected at run time.
- Testing of the O/R mapper was underestimated. Not only were there a lot of possible paths through the code, there were also stability and performance issues to be considered.
- The designer had to create and maintain the object model by editing a plain text file without any special editor. I know the saying “a fool with a tool is still a fool,” but not being able to visualize one’s own object model is like walking around in the dark—you don’t see where you are heading and you have difficulties in pointing the others in the correct direction. In this case, the object model, which was like a red ribbon winding through the whole architecture, could not be clearly communicated to the team.

I hope these examples are enough to help you to avoid stepping into a home-grown solution. Of course, a home grown O/R mapping project would be fun to do from a development point of view, but a COTS (commercial-off-the-shelf) O/R mapper will be cheaper—unless you consider O/R mapping as one of the core competencies of your business.

Selecting a COTS O/R Mapper

Below I list some criteria you might want to consider when selecting an O/R mapper.

- Consider whether the tool will restrict your modeling freedom too much. For example, many tools don’t support relationships on abstract classes. A workaround for this is to duplicate relationships on concrete classes, which is less 'OO', but works for these tools.

- Consider whether the O/R mapper that allows you to model visually (preferably using UML).
- If UML is important to you, ensure that you can either import UML into the O/R mapper, or that you can export UML from the O/R mapper.
- Take a close look at the programming model the O/R mapper imposes and see whether it is compatible with the things you want to get out of it.
- Look at the range of mapping possibilities to ensure that the kinds of relationships you envision between your objects and tables will be supported. Typically, most O/R mappers support a large range of features, but not every mapper supports every type of relationship.
- Assess the performance, even if you think you do not have a lot of performance demands. Testing the performance can also give you a chance to learn and assess the API.
- If you prefer or need to start with an existing database schema and then map objects onto it, assess whether the O/R mapper supports the database system you want to use.

Selecting a Persistence API

The O/R mapping features are only part of the story. The other part of the story is the selection of a good API for persisting objects, and this part has a lot more visibility to your development team than the O/R mapping part. While the O/R mapping functionality will only be exposed to a few team members who are dedicated to maintaining the persistence layer, the persistence API defines the interface that the whole development team will use.

Persistence APIs can be divided into two categories: transparent and non-transparent.

Transparent Persistence APIs

A transparent persistence API hides the persistence completely. A transparent API does not need to have a lot of methods; a load and a save method is sufficient most of the time. Typically, a lot is defined declaratively instead of procedurally. Hibernate and JDO are examples of transparent persistence APIs. Let me clarify with an example:

An `Insurance` object can contain `0-n` `Warranty` objects. The client application updates an attribute of an `Insurance` object. Semantically a `Warranty` is contained by an `Insurance`, so when you update an `Insurance` it is possible that you implicitly update its `Warranties`. According to the requirements, this might be a correct design, but it could have a negative impact on performance, especially if I am not aware that the implicit update of the `Warranty` objects is happening. When I only have modified an attribute of the `Insurance` object, I should be able to limit the persistence manager to this functionality.

The beauty of this is the "magic" way in which the persistence manager knows what to do. The negative side is that the persistence manager is thinking in your place.

Non-Transparent Persistence APIs

A non-transparent persistence API has a lot less "magic" inside of it. When compared to a transparent persistence API, it has a rich API, offering a lot of control to the user of the API.

Consider a transparent persistence API with a single method `Persist()`. This persistence method does all the magic behind the scenes, like checking whether there are associations that potentially need to be persisted as well. Although this might sound attractive, when selecting a persistence API, ensure that you can optimize. When touching associations, it is possible that the associated objects haven't been persisted yet. What should the `Persist()` method do? Persist those objects first? I say that it is better to put the client in control.

To illustrate the power of a non-transparent persistence API, I'll use the SimpleORM API as an example. Consider an insurance class with two methods. The first method will explicitly load all of the children linked to this object:

```
insurance.getAllChildren(insurance.Warranties)
```

The second method will only list those items that were added to the object in memory and the ones that are already retrieved from the database:

```
insurance.getRetrievedChildren(insurance.Warranties)
```

The advantage here is that the user of the API can "see" what he is doing, and can make better decisions regarding performance costs. (Also during code reviews this visibility can make life a lot easier.) In contrast, due to the fact that a transparent persistence API has a very generic interface (e.g. `Save()` and `Load()`), it also creates the illusion that database actions are cheap.

The cost for the non-transparent API is that the interface is more complex than a transparent persistence API. However, if you are planning to write your own persistence API, I recommend a non-transparent persistence API.

It is not my intention to classify all transparent persistence APIs as "do not use." But if you are considering a transparent persistence API, I would advise you to assess its performance carefully.

In case you are not satisfied with the persistence API that is provided to you, you can also decide to wrap it such that it maps onto your needs. In the next section, I'll elaborate on some good reasons to wrap a COTS persistence API.

Wrapping a COTS O/R Mapper's Persistence API

After you have selected a COTS O/R mapper, you should also decide whether to use its persistence API directly or to wrap it. Good reasons for wrapping the O/R mapper's persistence API are:

- You want to add some extra logic (for example, field validation is not part of the O/R mapper, but you want it to be an inherent part of the persistent objects).
- Some O/R mappers generate code for the persistent objects, others expose a generic API. Typically, the generated persistent objects are type safe, while the generic API is not. Type safety is a good reason for wrapping the O/R mapper's persistence API.
- Apply the subsystem principle: you want to avoid a tight coupling with a specific O/R mapper. Therefore you treat it as a subsystem and work interface-based.
- You want to limit the features that your clients can use. For example, your mapper might support the use of direct SQL while you don't want to expose such a feature.
- You might want to expose certain services in a different way. For example, you might want to introduce a query object rather than to expose an OQL query interface.

Of course you need to place everything in perspective: if you are creating a throw away application, you don't need to worry about aspects such as maintainability, extensibility, and resilience. For strategic applications however (commercial software products, product families, core business applications, etc.) you really should spend some time evaluating the pros and cons of different approaches.

A bad reason to wrap a persistence API is that you think that you will be able to boost performance. In general, you won't be able to do this because the performance is inherent to the internal design of the COTS component. Only in rare occasions you will be able to turn the performance to your advantage by wrapping the COTS component.

A better way to approach performance is to invest some time in seeking a usage pattern that is optimal for your situation.

Wrapping and Object Management

Another way to discriminate between persistence APIs is the way objects are managed.

In one technique, the persistence manager is an object factory (using the factory pattern):

```
MyPersistenceManager mgr = MyPersistenceManager().Instance;

Insurance insuranceObj = mgr.CreateInsurance();

mgr.Persist(insuranceObj);
```

An advantage of this approach is that the manager always knows the current state of the object (new, retrieved, already saved). The manager can use this information to its advantage, resulting in good performance.

In other APIs, such as JDO, the objects are not created within the context of the persistence manager. The persistence manager takes an object in and determines what to do with it. In this JDO example, the manager (`mgr`) does not know the current state of the `insurance` object (unless perhaps it caches the information):

```
PersistenceManager mgr =
    persistencemanagerFactory.getPersistenceManager();

Insurance insurance = new Insurance();

insurance.SetPolicyNumber("2005001001-110");

mgr.makePersistent(insurance);
```

Okay, by now I hear you saying "All of this is very interesting, but why is it of my concern?" Two reasons:

First, consider object management when you want to wrap the persistence API. Performing the wrapping without knowledge of the pros and cons of different persistence API approaches would be very dangerous.

Second, consider the performance implications. In the case of the object factory mechanism, you can be quite sure of decent performance. In the latter mechanism, it's advisable to assess the performance by means of some unit tests or architectural prototypes.

Disconnected Objects

A typical disconnected object scenario starts with the retrieval of the object from the database in the server app, then streaming it to the client app, then in-memory modification on the client app, and finally streaming it back and storing it to the database again in the server app. The way the persistence framework handles disconnected objects has quite a big impact on the performance.

One way to improve the performance of disconnected object persistence is through *merging*, as in this JBoss example:

```
Insurance insurance = Util.deserialize(input);

// Changes are monitored as of now.
entityManager.merge(insurance);

insurance.setSubscriptionDate(d);

entityManager.persist(insurance);
```

In the above example, only the change of the subscription date is stored in the DB.

Another area of performance concern is keeping track of changes vs. figuring out at the time of persistence what changes have occurred. The latter technique is more dangerous for performance, especially when complete object trees are being saved. The algorithm to find out the changes has to be fast, or the original data needs to be cached in order to achieve a good performance.

O/R Mapping Best Practices

I have distilled the following “best practices” from my experiences:

- Don’t work against the O/R mapper’s persistence design. Rather, take the O/R mapper’s design principles as a constraint and exploit them. If you don’t, you’ll have to pay in terms of efficiency, performance, etc. Also, make sure that you know the basic concepts of the O/R mapper’s design.
- Wrap the O/R persistence API and treat it as a subsystem, such that you work interface-based, which eases the prospect of switching later. I don’t say that it won’t hurt to switch to another O/R mapper, but at least the pain can be isolated.
- Check the querying capabilities of the O/R mapper’s persistence interface. Especially, check whether aggregate functions can be used and whether you can query for raw values rather than plain objects. Objects can be too much of a good thing (object bloat) when, for instance, you just need a couple of values to fill a grid.
- Implement field metadata wisely. Generate field metadata (such as size, etc.) in-line in metadata classes instead of using reflection. This improves performance, and also makes it easy to debug the code.
- Put field validation at the level of the metadata. Make sure to expose your field-level business rules such that you don’t require a round-trip from your client to your server application in order to know whether or not the object is in a correct state to persist it. Tools such as SimpleORM suffer from this (typically field validation is foreseen internally, but not exposed).
- Be careful when calling a field a “mandatory” field. Typically, O/R mappers foresee the ability to tag certain fields as mandatory. In my experience, I have seen that this construct is used too much. Typically fields are mandatory dependent on the state of the object and such interdependencies can seldom be expressed. Therefore, the best practice is to limit the mandatory fields only to those fields that make the object incorrect within the application domain.
- When defining the persistence API, guard its consistency and ease of use. When specifying the interface of your persistence API, make sure that the parameters of each persistence method are consistent with the rest of the interface. As an example, the following dummy interface is not consistent because in one case an enumeration is used (`RelationshipName`), and in the other case a string:

```
SaveRelation(RelationshipName name, object value)
SaveAttributeToObject(string attributeName, object o)
```

I would suggest to either go for a type-safe approach or for a generic approach, but not to mix them in one interface. If you want both of them, then put the generic methods on a separate, more generic interface. Don't make the interface more complex than it should be.

- Make sure that the API can be used in a way that a client application can take advantage of its knowledge to optimize performance.

References

[Berglas, 2005] Berglas, Anthony. "SimpleORM White Paper: Simple Java Object Relational Mapping". `http://www.simpleorm.org/whitepaper.html`, May 2005.

[Fowler, 2004] Fowler, Martin. *Patterns of Enterprise Application Architecture*. Boston: Addison-Wesley, 2004.

###

Mario Van Damme is a software architect, working for quite a number of years in the medical industry, and prior to that in the insurance and banking industry. He can be contacted by e-mail at: `mvandamme ~AT~ sopragroup.com` and `mario.vandamme.mv ~AT~ belgacom.net`.